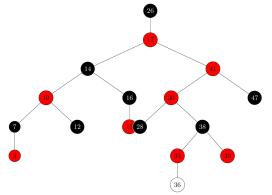


Review and Exam prep
Lecture 26

COSC 242 – Algorithms and Data Structures

Today’s outline

1. Exam details and preparation
2. Course overview
3. Asymptotic notation
4. Proofs
5. Divide and Conquer
6. Hash tables
7. Trees
8. Graphs and greedy algorithms
9. Dynamic programming
10. P and NP

Today's outline

1. Exam details and preparation
2. Course overview
3. Asymptotic notation
4. Proofs
5. Divide and Conquer
6. Hash tables
7. Trees
8. Graphs and greedy algorithms
9. Dynamic programming
10. P and NP

Exam details

- 3 hours in duration
- Worth 60% of final grade
- Written exam
- Closed book, no notes, no calculators
- No questions involve writing or reading code
- Includes two appendices
    1) RBT Insertion Fix-up code from L16, SL15 (comments removed)
    2) RBT Deletion Fix-up code from L17, SL18 (comments removed)

Exam suggestions

- If you have a final answer, circle or label it so you **make it very clear**.
- Bring an eraser or white-out.
- Use the separate rough work books. Cross our any rough working in your solution book.
- Put your student ID on your exam where indicated. It sounds silly, but every year at least one person forgets to do this.

Exam suggestions

- Attempt every question. Markers want to award as many marks as possible. If you leave a blank page, they can only ever award a zero, you leave them no “wriggle room”. If you’re close to a pass, then every mark counts.
- There is zero tolerance for “tricks”. E.g., don’t provide two different answers, leaving it ambiguous which is your “final” answer. It will be assumed you don’t know the answer and marked accordingly.

Exam strategies

Strategy 1 – A greedy approach to marks

- Think of the exam like the knapsack problem. You only have limited time, and you may not be able to answer everything. Begin by identifying how long each question will take you to answer, and the marks you think you'll earn. This gives you the metric: marks/time. Then attempt the questions in decreasing order of marks/time.

Strategy 2 – Sorted by difficulty or time

- Answer them from easiest to hardest, or quickest to longest.

Tip - if you get stuck on a problem, stop, and come back to it later. A break really helps, as it allows your brain to work on it in the background.

Exam preparation

- Re-do all the tutorials, without the solutions.
- Attempt exams from previous years. Even if you can’t answer all the questions, you’ll notice the types of questions that often appear.
- The exam cannot assess every aspect of the course. So, consider what material has and has not yet been assessed. Using intuition, material that has not yet been assessed is more likely to appear on the exam.

Today’s outline

1. Exam details and preparation
2. Course overview
3. Asymptotic notation
4. Proofs
5. Divide and Conquer
6. Hash tables
7. Trees
8. Graphs and greedy algorithms
9. Dynamic programming
10. P and NP

Concept overview (1 of 3)

Concept	Topic	Lectures	Tutorials
Asymptotic notation	O, Θ, Ω	3	2
	Proofs	4, 5	3
Proofs	Contradiction	4	3
	Induction	4, 5	3
Divide & Conquer	Merge sort	6	4
	Recurrence equations	7	4
	Quicksort	8	5
	Recursion tree	8	5

Concept overview (2 of 3)

Concept	Topic	Lectures	Tutorials
Hash tables	Definition, probing	9	5
	Hash functions	10	5
	Probs, Perfect hashing	11	6
Binary Search Trees	Definitions, proofs, insertion	12	6
	Search, traversal	13	7
	Min-max, pred-succ, deletion	14	7
Red-Black Trees	Properties, definition	15	8
	Rotation, insertion	16	8, 9
	Deletion	17	9
B-Trees	Definition, Insert, Delete	18	9

Concept overview (3 of 3)

Concept	Topic	Lectures	Tutorials
Graphs	Representation, BFS	19	10
	DFS, Sort, Path-finding	19, 20	10
	Weighted	21	11
Greedy algorithms	Dijkstra's, Prim's algorithms	21	11
	Greedy-vs-Dynamic, Mem, iter.	22	12
Dynamic programming	0-1 Knapsack	22	12
	Assembly line scheduling	23	12
	Longest common subsequence	24	
P-NP	Problems, HCP, TSP	25	

Today’s outline

1. Exam details and preparation
2. Course overview
3. Asymptotic notation
4. Proofs
5. Divide and Conquer
6. Hash tables
7. Trees
8. Graphs and greedy algorithms
9. Dynamic programming
10. P and NP

Defining $\boldsymbol{\Theta}$-notation (Big-theta)

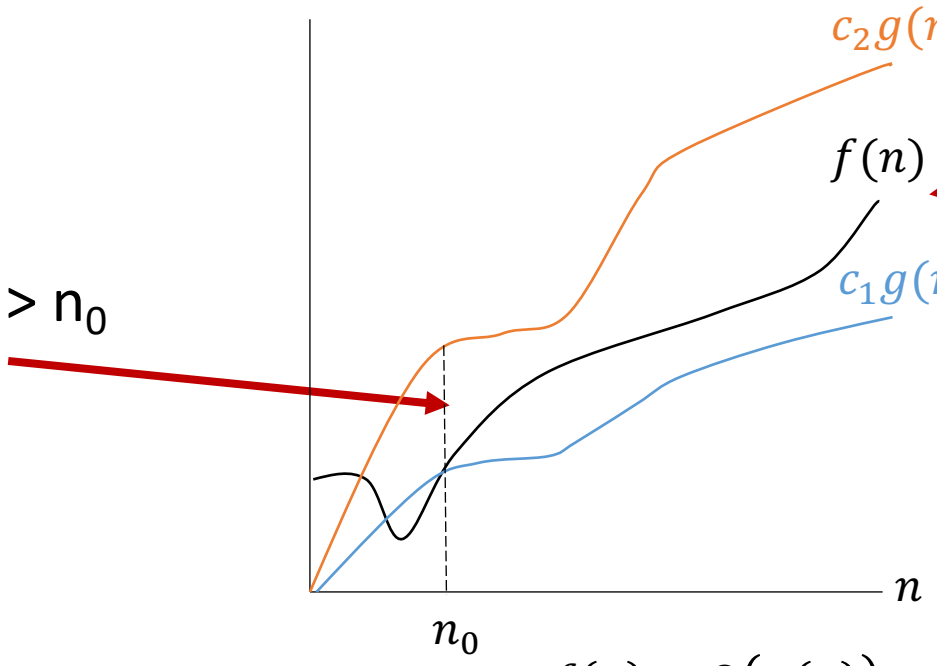
For a given function, $g(n)$, we denote by $\Theta\big(g(n)\big)$ the **set of functions***:

$\Theta\big(g(n)\big) =$ {f(n): there exists positive constants, c_1, c_2, and n_0, such that: $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$}

* By set of functions, we mean *many functions,* not just one function

Θ-notation

What this means is that as *n* gets large enough, the running time is **at least $c_1 g(n)$**, and **at most $c_2 g(n)$:**



$f(n) = \Theta(g(n))$

But only when n > n_0

Here's our function *f(n)*, sandwiched between $c_1 g(n)$ and $c_2 g(n)$

Typical problem

Show that $f(n) = n^2 + 2n + 1 = \Theta(n^2)$

How do we show this? Recall that:

$$\underset{\Omega(n)}{c_1 g(n)} \leq f(n) \leq \underset{O(n)}{c_2 g(n)}$$

This problem is therefore composed of two parts.

The first is showing the upper bound. We do this with the standard method (next slide). Then, show the lower bound (usually easy).

Big-oh Standard method

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

1. Choose $n_0 = 1$
2. Assuming $n > 1$, find c such that:

$$\frac{f(n)}{g(n)} \leq c \frac{g(n)}{g(n)} = c$$

Thus, when n > 1, $f(n) \leq c \cdot g(n)$.

When n > 1, implies: $1 < logn < n < nlogn < n^2 < n^3 < 2^n < n!$

This implication allows us to then optimally use:

3. Raise and simplify. That is, raise the numerator to simplify fraction.
4. Solve for c, and check your answer.

Typical problem – Part 1

First show that $f(n) = n^2 + 2n + 1 \leq O(n^2)$

Step 1:

Choosing $n_0 = 1$, then assuming n > 1

Step 2:

$$\frac{f(n)}{g(n)} = \frac{n^2 + 2n + 1}{n^2}$$

Continued…

Typical problem – Part 1

Show that $f(n) = n^2 + 2n + 1 \leq O(n^2)$

Step 3:

$$\frac{f(n)}{g(n)} = \frac{n^2 + 2n + 1}{n^2} \leq \frac{n^2 + 2n^2 + n^2}{n^2} = \frac{4n^2}{n^2} = 4$$

Set c = 4.

Step 4:

$n^2 + 2n + 1 \leq 4 \cdot n^2$ whenever n > 1

Typical problem – Part 2

Now show that $\Omega(n^2) \leq f(n) = n^2 + 2n + 1$

We have already shown it is O(n). From our relationship:

$$\underset{\Omega(n)}{c_1 g(n)} \leq f(n) \leq \underset{O(n)}{c_2 g(n)}$$

We only need to show $\Omega(n)$.

For lower bounds, we often don't need the standard method.

Continued…

Typical problem - Part 2

Show that $\Omega(n^2) \leq f(n) = n^2 + 2n + 1$

Step 1: Set c = 1

Step 2: Identify n_0 such that the inequality holds

$c \cdot n^2 \leq n^2 + 2n + 1$

$1 \cdot n^2 \leq n^2 + 2n + 1$

$n^2 \leq n^2 + 2n + 1$

Inequality holds when $n_0 = 1$, for all n > 1

E.g., $1^2 \leq 1^2 + 2 + 1 \leq 4$

Today’s outline

1. Exam details and preparation
2. Course overview
3. Asymptotic notation
4. Proofs
5. Divide and Conquer
6. Hash tables
7. Trees
8. Graphs and greedy algorithms
9. Dynamic programming
10. P and NP

Proof by contradiction

In proof by contradiction, our goal is to prove that the statement is True, by showing that it cannot be False.

To do this, we first *assume* that our hypothesis *A* is True, and that our conclusion *B* is False. We then show why this cannot happen.

This proof technique is valuable when the statement *NOT B* gives you useful information.

Proof by Contradiction with Big-Oh

Suppose you are asked: Is $n = O(n^2)$?

Recall that $f(n) \leq c \cdot g(n)$, it is easy to see that $n = O(n^2)$.

Just take $c = n_0 = 1$. That is, $1 \leq 1*1^2$

This is a simple case. But what about $n^2 = O(n)$?

Proof by contradiction: $\mathrm{n}^2 = \mathrm{O}(\mathrm{n})$?

Lets try and prove $\mathrm{n}^2 = \mathrm{O}(\mathrm{n})$ cannot hold by contradiction.

Then c, n_0 would have to exist such that $\mathrm{n}^2 \leq \mathrm{c} \cdot \mathrm{n}$ for all $\mathrm{n} \geq \mathrm{n}_0$

1. **Statement A (hypothesis)**: c, n, n_0 are real numbers ($\mathrm{c}, \mathrm{n}, n_0 \in \mathbb{R}$), such that $n > 0$, $c > 0$, and $n \geq n_0$
2. **Statement B (conclusion)**: $\mathrm{n}^2 = \mathrm{O}(\mathrm{n})$

Lets find a case that proves ¬B = False.

Proof by contradiction: $\mathrm{n}^2 = \mathrm{O}(\mathrm{n})$?

Rewriting our equation into standard notation:

1. $n^2 \leq c \cdot n$
2. $\frac{n^2}{n} \leq c$
3. $n \leq c$

We stated that c is a fixed value, while n can vary freely. So lets set n = c+1

4. $c + 1 \leq c$ → False

Conclusion: Our contradiction ¬B, or $\mathrm{n}^2 = \mathrm{O}(\mathrm{n})$, is shown to be False. Therefore, $\mathrm{n}^2 \neq \mathrm{O}(\mathrm{n})$, must be True. ■

Proof by induction

You should consider using induction when B has the form:

For every integer $n \geq 1$, “something happens”.

Where the **“something happens” is P(n)**, that depends on the integer n.

Formally, it’s used to prove statements of the form:

$$(\forall n \in \mathbb{N})\big(P(n)\big)$$

Steps of Induction

1. Base case: Verify that $P(1)$ is True.
2. Induction step: Use the assumption that $P(n)$ is True, to prove that $P(n+1)$ is True.

The hypothesis in Step 2, that our statement holds for a particular n, is called the **induction hypothesis**.

To prove the inductive step, we assume the induction hypothesis for n, and then use this assumption to prove that the statement holds for $\mathrm{n}+1$.

Proving Insertion sort

Insertion sort was: $0 + 1 + 2 + 3 + \cdots + \mathrm{n} - 1 = \frac{n(n-1)}{2} = \frac{n^2-n}{2}$.

We have a left hand side (LHS) and a right hand side (RHS):

$$l(n) = 0 + 1 + 2 + \cdots + n - 1$$

$$r(n) = \frac{n(n-1)}{2}$$

We need to prove that $\mathrm{l}(\mathrm{n}) = \mathrm{r}(\mathrm{n})$ for all n ≥ 1

Solve insertion sort with induction

$$l(n) = 0 + 1 + 2 + \cdots + n - 1$$
$$r(n) = \frac{n(n-1)}{2}$$

Base case: $l(1) = 0 = r(1) = \frac{1(1-1)}{2} = 0$

Inductive step: assume $l(n) = r(n)$, which is our $P(n)$ induction hypothesis, and use it to show that $l(n+1) = r(n+1)$

$$\begin{aligned} l(n+1) &= 0 + 1 + 2 + \cdots + n - 1 + n \\ &= l(n) + n = r(n) + n \\ r(n+1) &= (n+1)\big((n+1) - 1\big)/2 \\ &= (n+1)(n)/2 \\ &= (n^2 + n)/2 \\ &= (n^2 - n + 2n)/2 \\ &= (n^2 - n)/2 + 2n/2 \\ &= r(n) + n = l(n) + n \end{aligned}$$

Lets try and make this look like our RHS(n) $= \frac{n^2-n}{2}$

Conclusion: By P.M.I, the statement P(n) is True for n+1, and so the proof is complete. ■

Class challenge 1

Prove that $n^2 = O(2^n)$.

Class challenge 2

Prove that $2^n = O(n!)$, for some c, n_0, such that $2^n < n!$ for all $n > n_0$

Class challenge 3

Consider the set $X \in \mathbb{R}$, where

X = {3, 12, 33, 72, …}

Where X is given by $f(n) = n^3 + 2n$

Are all the elements of X divisible by 3?

Today’s outline

1. Exam details and preparation
2. Course overview
3. Asymptotic notation
4. Proofs
5. Divide and Conquer
6. Hash tables
7. Trees
8. Graphs and greedy algorithms
9. Dynamic programming
10. P and NP

Divide, then conquer. But first lets divide…

Divide-and-conquer processes the data structure X as:

1. **if** X is an atom **then**
2. process X directly
3. **else**
4. divide X into two or more smaller pieces
5. apply the algorithm to each piece “recursively”
6. combine the processed pieces (if necessary)

Merge sort = $\Theta(n \cdot \log n)$

To merge sort an array $A[0 .. n - 1]$ of keys, we repeatedly split A, and after getting to the bottom we rebuild by merging the pieces. To identify the pieces that must be split or patched together, we use indices *left* and *right*.

Mergesort(A, left, right) // sorts the keys in $A[left .. right]$

1. **if** $left \geq right$ **then**
2. stop since $A[left .. right]$ is sorted
3. **else**
4. $mid \leftarrow (left + right) / 2$
5. Mergesort(A, left, mid)
6. Mergesort(A, $mid + 1$, right)
7. Merge subarrays $A[left .. mid]$ and $A[mid + 1 .. right]$

Merge two arrays

We have two arrays X and Y each in sorted order. We want to build array Z containing all the keys of X and Y in sorted order. Length(X) = l, length(Y) = m:

1. initialise i, j, k to 0 (i, j, k index the arrays X, Y, Z)
2. **while** $i < l$ and $j < m$ **do**
3. **if** X[i] < Y[j] **then**
4. Z[k] ← X[i]
5. i ← i + 1
6. **else if** X[i] ≥ Y[j] **then**
7. Z[k] ← Y[j]
8. j ← j + 1
9. k ← k + 1
10. **if** $i \geq l$ then copy the end of Y to the end of Z
11. **else** copy the end of X to the end of Z

Analysing divide-and-conquer

When an algorithm contains a recursive call to itself, we can often describe its running time by a **recurrence equation** or **recurrence**.

This equation describes the overall running time on a problem of size n in terms of the running time on smaller inputs

We call these **recurrence equations** because the function name T recurs on the righthand side of the equation.

Recurrence equation for merge sort

The time complexity function for merge sort is:

$$T(1) = 1$$
$$T(n) = 2T(n/2) + n$$

It would be easier to compare this function to our landmark functions if we could find a simple non-recurrent formula defining the function T.

Solving is not always possible, but for merge sort we can.

Solve the merge sort recurrence

Base case: $T(1) = 1$

Function: $T(n) = 2T(n/2) + n$

Solution

k = 1

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

$$= 2[2\mathrm{T}(n/4) + n/2] + n$$

k = 2

$$= 4\mathrm{T}(n/4) + 2\mathrm{n}$$

$$= 4[2T(n/8) + n/4\,] + 2n$$

k = 3

$$= 8\mathrm{T}(n/8) + 3\mathrm{n}$$

$$= 8[2T(n/16) + n/8\,] + 3n$$

k = 4

$$= 16\mathrm{T}(n/16) + 4\mathrm{n}$$

$$= \mathbf{k}\mathrm{T}(\boldsymbol{n/k}) + \mathbf{n}\log \boldsymbol{k}$$

Workspace

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

$$= 2\mathrm{T}(n/4) + n/2$$

$$\mathrm{T}(n/4) = 2T(n/2/4) + n/4$$

$$= 2T(n/8) + n/4$$

$$\mathrm{T}(n/8) = 2T(n/2/8) + n/8$$

$$= 2T(n/16) + n/8$$

See L07 for induction solution

Quicksort implementation

```
QUICKSORT (A, p, r):
1 if p < r
2     q = PARTITION(A, p, r)
3     QUICKSORT(A, p, q-1)      // First half, from p up to partition point q
4     QUICKSORT(A, q+1, r)      // Second half, from q+1 up to end r
```

Initial call is `QUICKSORT(A, 1, n)`

Recall
A[p .. r]
p = first index
r = last index

Partitioning

The quicksort algorithm is deceptively simple. However, the key insight lies in the partitioning process.

```
PARTITION(A, p, r)
  x = A[r]
  i = p - 1
  for j = p to r - 1
     if A[j] ≤ x
           i = i + 1
           swap A[i] with A[j]
  swap A[i+1] with A[r]
  return i + 1
```

PARTITION always selects an element x = $A[r]$ as a ***pivot*** element around which to partition the subarray $A[p..r]$.

Quicksort: best case analysis

The best case occurs when partition produces two subarrays, each of which is no more than $n/2$. That is, completely balanced subarrays every time. Our function here is:

Base case: $T(1) = 1$

Thing to solve: $T(n) = T(n/2) + T(n/2)$

This is the same equation as mergesort (phew!)

In the best case, Quicksort is $O(n \log_2 n)$

Balanced partitioning

That's the worst case and best case. But what about the **average case**?

The average-case running time of quicksort is much closer to the best case than to the worst case.

Consider a pivot that always gives us a 9-to-1 (90%/10%) split of the data (hmm, that sounds bad…). In that case, the function is:

$$T(n) = T(9n/10) + T(n/10) + \theta(n)$$

This is tricky to solve with the substitution method, so we will use a **recursion tree** instead.

$T(1) = 1$

$T(n) = \underbrace{T(9n/10)}_{\text{Recursive 1}} + \underbrace{T(n/10)}_{\text{Recursive 2}} + \underbrace{cn}_{\text{Constant}}$

Tree depth	Tree	Row sum
0	n	c·n
1	$\frac{9}{10}n$ $\quad$ $\frac{1}{10}n$	c·n
2	$\frac{9^2}{10^2}n$ $\quad$ $\frac{9}{10^2}n$ $\quad$ $\frac{9}{10^2}n$ $\quad$ $\frac{1}{10^2}n$	c·n
3	$\frac{9^3}{10^3}n$ $\quad$ $\frac{9^2}{10^3}n$	c·n
⋮		
i		

$$\left(\frac{10^i}{9^i}\right) = n$$

$$\left(\frac{10}{9}\right)^i = n$$

$$\log_{\frac{10}{9}} n = i$$

$$\left(\frac{10^i}{1^i}\right) = n$$

$$\left(\frac{10}{1}\right)^i = n$$

$$\log_{10} n = i$$

Calculating our total work

Using this fact, our equation can be rewritten as:

$$\log_{\frac{10}{9}} n = \frac{\log_2 n}{\log_2 \frac{10}{9}} = c \cdot \log_2 n$$

Since $1/\log_2 \frac{10}{9}$ is a constant, we replace it with c. As long as this value is a constant, the log doesn't matter in asymptotic notation.

Again, we do 'n' work per level, so rewriting $\mathrm{O}\left(n \times \log_{\frac{10}{9}} n\right)$ we get:

$$T(n) = O(n \log_2 n)$$

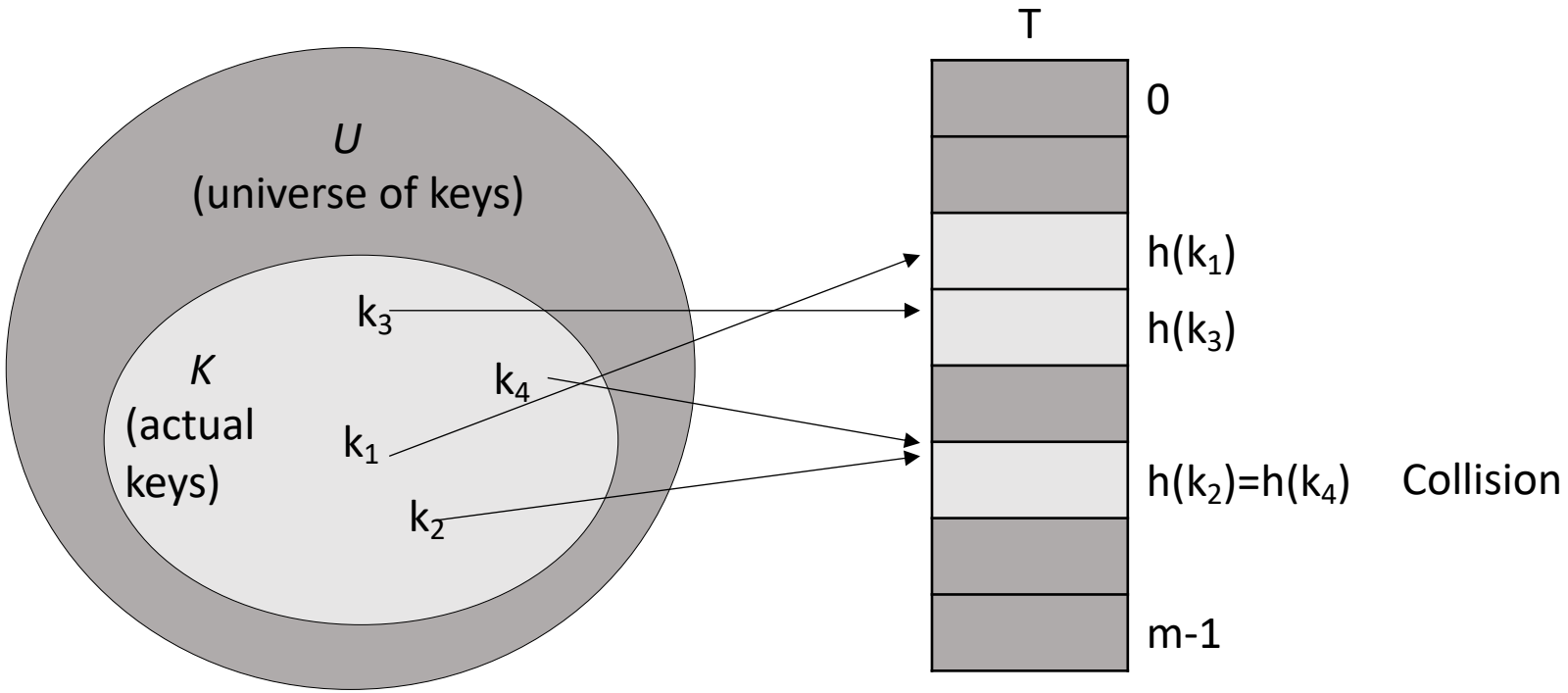
Today’s outline

1. Exam details and preparation
2. Course overview
3. Asymptotic notation
4. Proofs
5. Divide and Conquer
6. Hash tables
7. Trees
8. Graphs and greedy algorithms
9. Dynamic programming
10. P and NP

Hashing

The expected time to search for an element in a hash table is O(1), under some reasonable assumptions.

Worst-case search time is $\Theta(n)$, however.



Using hash function *h* to map keys to hash-table slots.
Keys k_2 and k_5 map to the same slot, and collide.

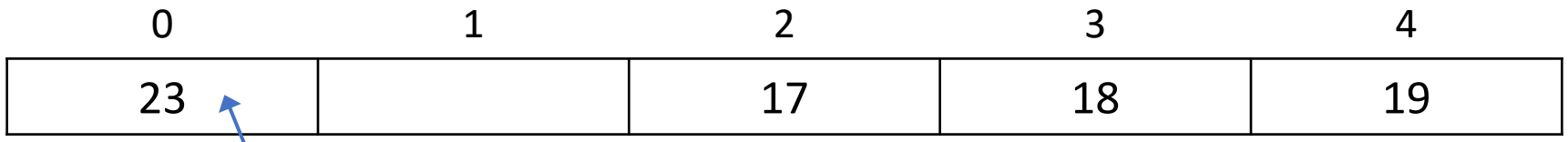
Linear probing

H(k, i) = (h(k) + i) % m

where *k* is the key, *i* is the number of collisions so far, and *m* is the array's capacity. The **home cell** is given by H(k, 0).

Linear probing example

n = {~~17~~, ~~19~~, ~~18~~, ~~23~~}, m = 5, using h(k, i) = (h(k) + i) % m



h(k, i) = (17 + 0) % 5 = 2

h(k, i) = (19 + 0) % 5 = 4

h(k, i) = (18 + 0) % 5 = 3

h(k, i) = (23 + 0) % 5 = 3, set i += 1
h(k, i) = (23 + 1) % 5 = 4, set i += 1
h(k, i) = (23 + 2) % 5 = 0

Done!

Double hashing

The drawback of linear and quadratic probing is that collision resolution strategies follow the same path from a collision point regardless of key value.

A better solution is **double hashing:**

$$h(k,i) = \big(h_1(k) + i \cdot h_2(k)\big)\%m$$

- *i* = the number of collisions so far
- h_1 = division hashing
- *m* = table size
- h_2 = *a secondary hash function*

h_2 is often something like $h_2(k) = 1 + k\%(m - 1)$.

Class challenge 4

Insert (12, 13, 43, 52, 72, 63) with h(k) = k%10, using first quadratic probing then double hashing.

quadratic

$H(k, i) = (k\%m + i^2) \% m$

Total collisions = 0

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

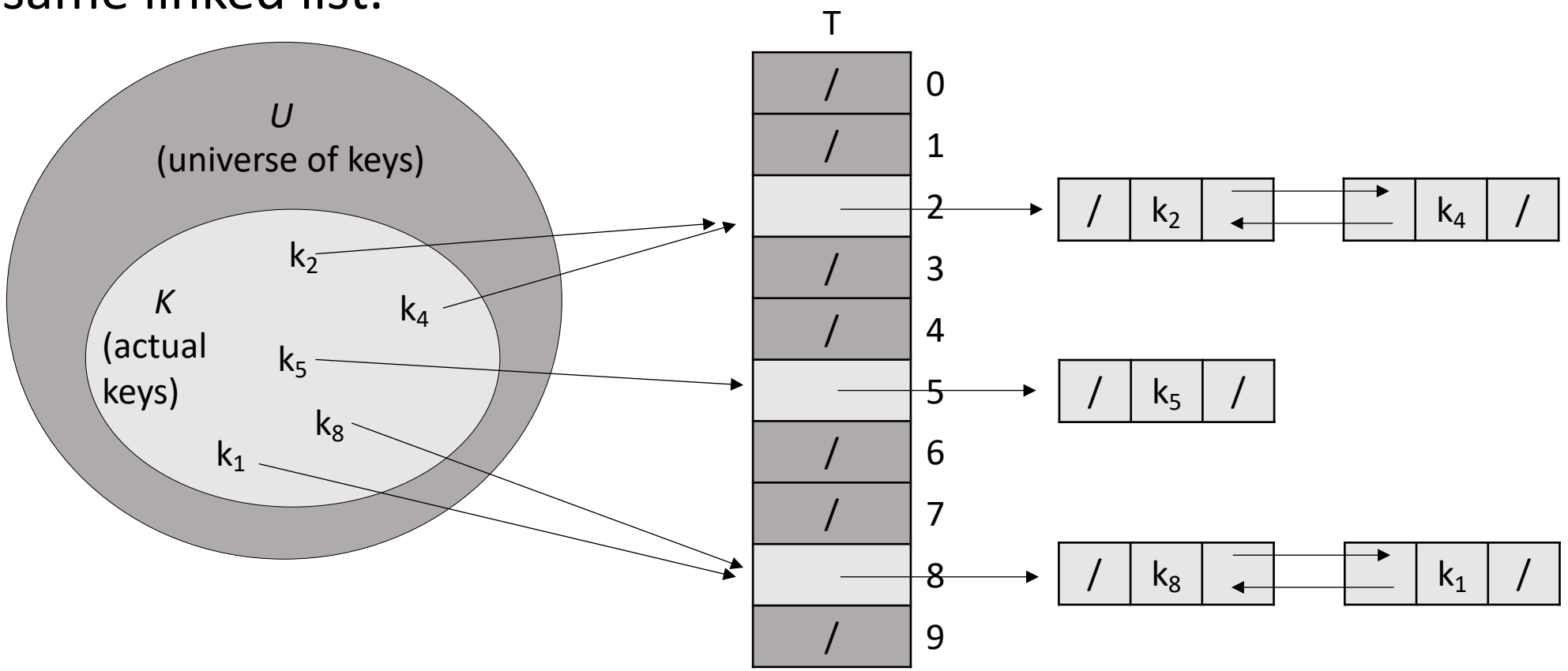
double

$H(k, i) = (k\%m + i(1+k\%(m-1)) \% m$

Total collisions = 0

Chaining

In chaining, we place all the elements that hash to the same slot into the same linked list.



A universal set of hash functions

Choose a prime number p big enough that every possible key *k* is < *p*, and choose your table size *m* < *p*.

Now make $h_{a,b}(k) = \big((ak + b) mod\ p\big) mod\ m$

The parameters *a* and *b* may take on integer values up to *p* - 1, but you must choose *a* > 0.

a and *b* are chosen randomly at program start up.

Perfect hashing

Main hash table: choose size $m = n$ where n is the number of data items, choose prime $p >$ the biggest key value. Choose a and b randomly to get primary hash function h(k) = ((ak + b)%p)%m.

Test h as follows:

1. For each key k, work out the home cell $h(k)$. Keep a count n_i for each cell i of how many keys hash to i.
2. Check whether space required is too big: is the sum of all the n_i^2 giving a total > $2n$? Then h is not good enough so repeat the process with new a, b for a new primary hash function.

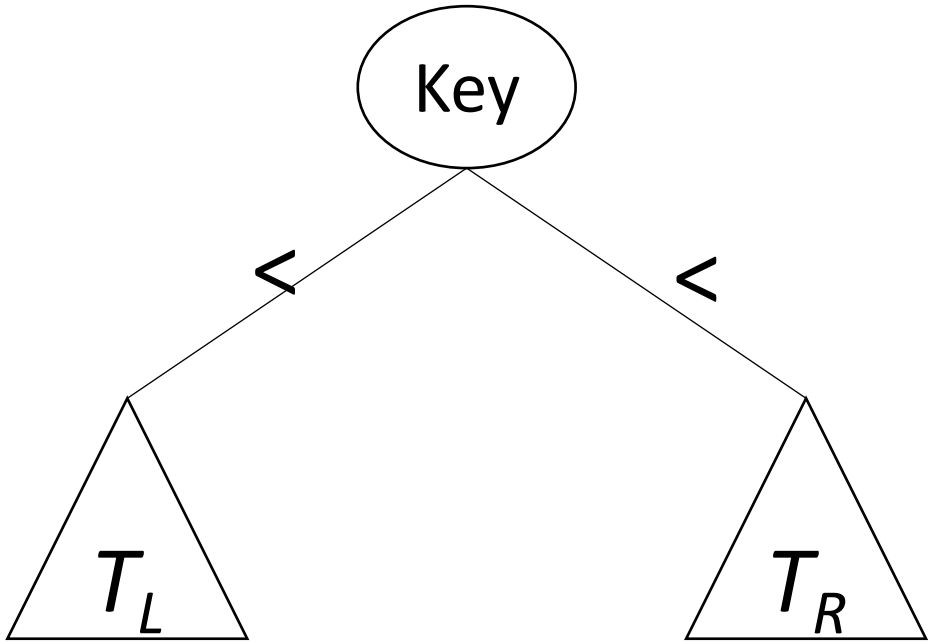
Today’s outline

1. Exam details and preparation
2. Course overview
3. Asymptotic notation
4. Proofs
5. Divide and Conquer
6. Hash tables
7. Trees
8. Graphs and greedy algorithms
9. Dynamic programming
10. P and NP

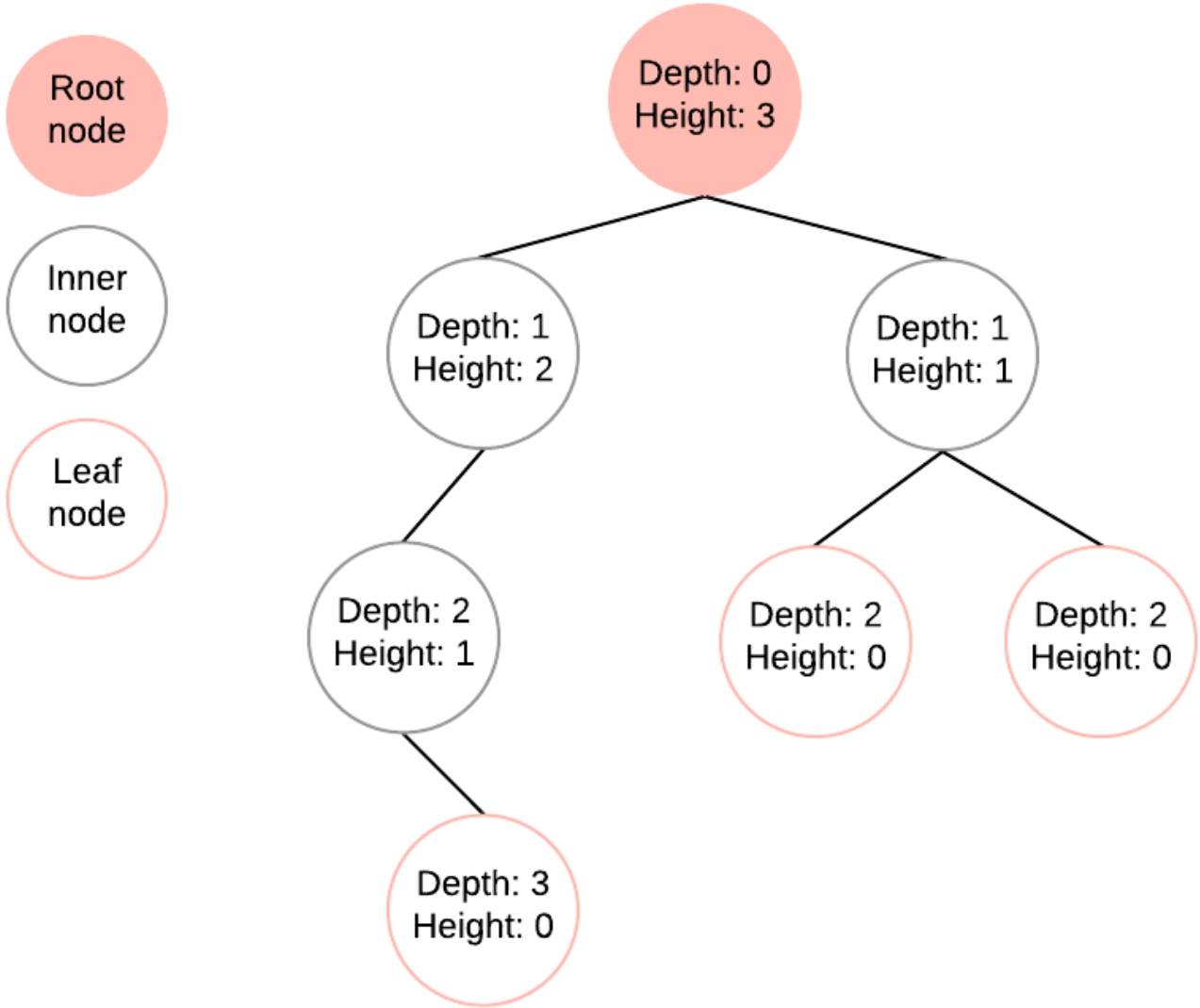
Binary search tree

A binary tree T has the BST-property if:

- nodes in T have a key field of ordinal type, so they can be ordered by <
- for each node N in T, N's key value is greater than all keys in its left subtree T_L and less than all keys in its right subtree T_R, and T_L and T_R are binary search trees.



Height vs Depth

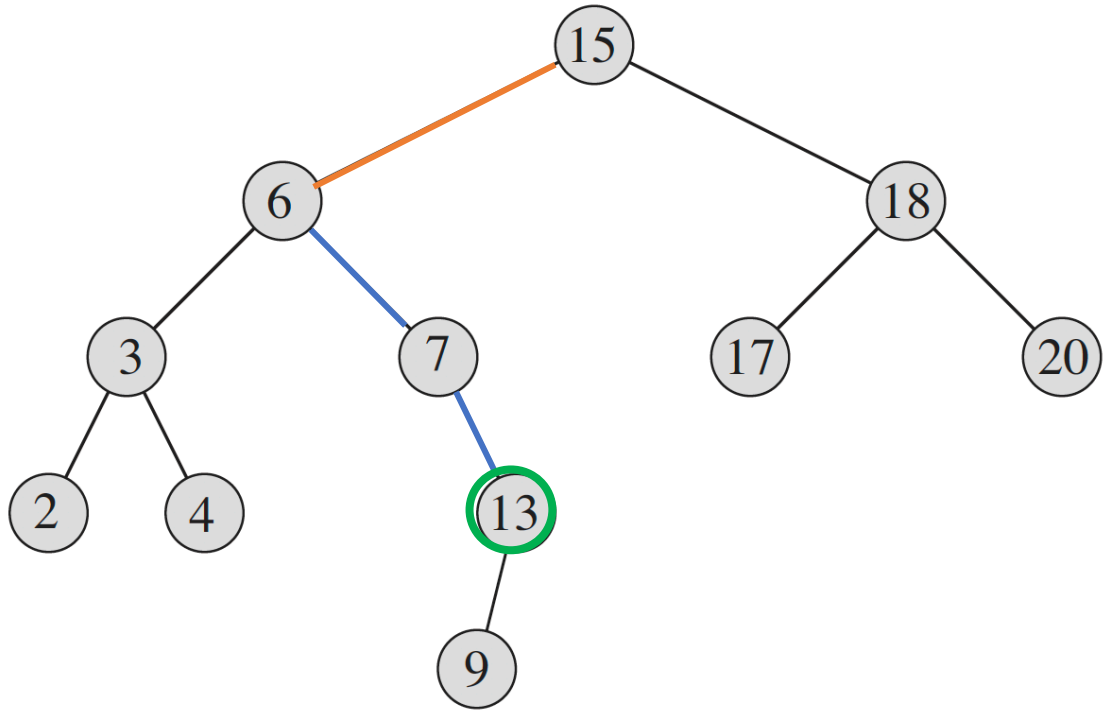


*The **height** of a BST is the number of edges from the root to the deepest leaf.*

*The **depth** of a node is the number of edges from that node to the root of the tree.*

Search example

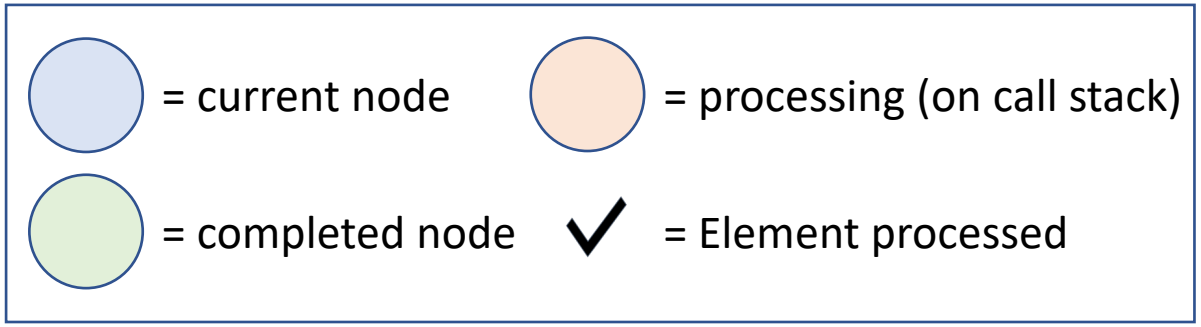
Trace the search path for Key = 13, indicating branch points in the pseudocode.

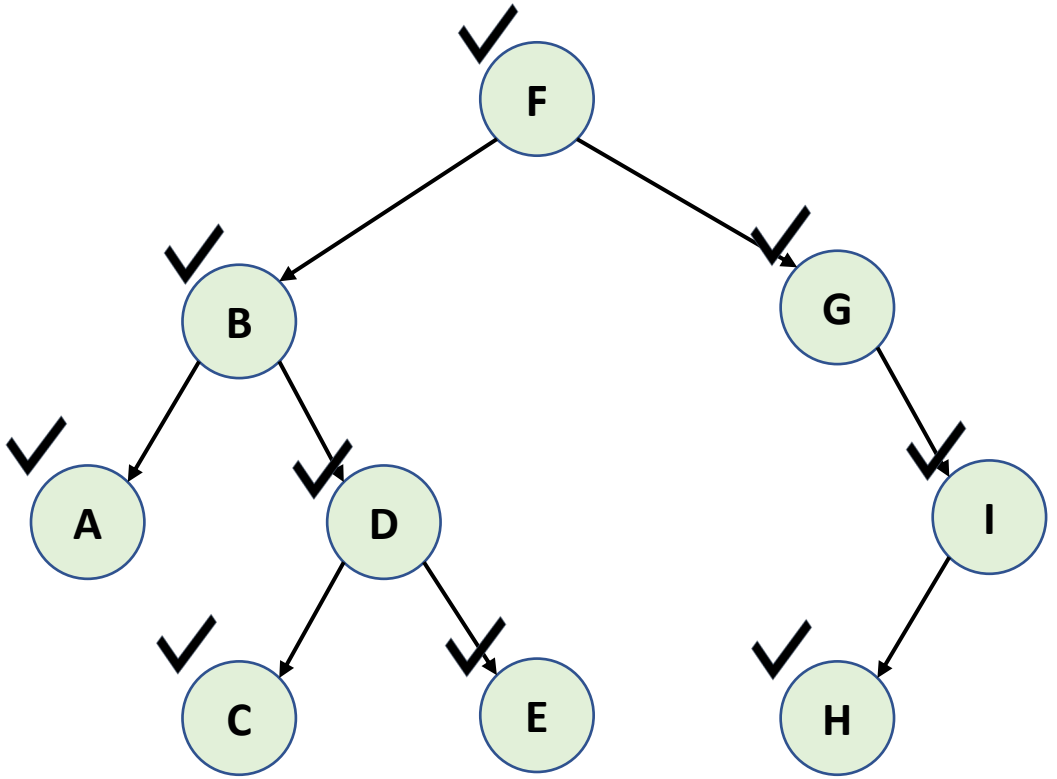



```
function BST_Search(BST T, KeyType key)
    if T == NIL then
        return Not Found
    else if key == T→key then
        return T
    else if key < T→key then
        return BST_Search(T→left, key)
    else
        return BST_Search(T→right, key)
    end if
end function
```

Inorder traversal

```
1:  function Inorder_traversal(BST T)
2:      if T ≠ NIL then
3:          Inorder_traversal(T→left)
4:          process(T)
5:          Inorder_traversal(T→right)
6:      end if
7:  end procedure
```

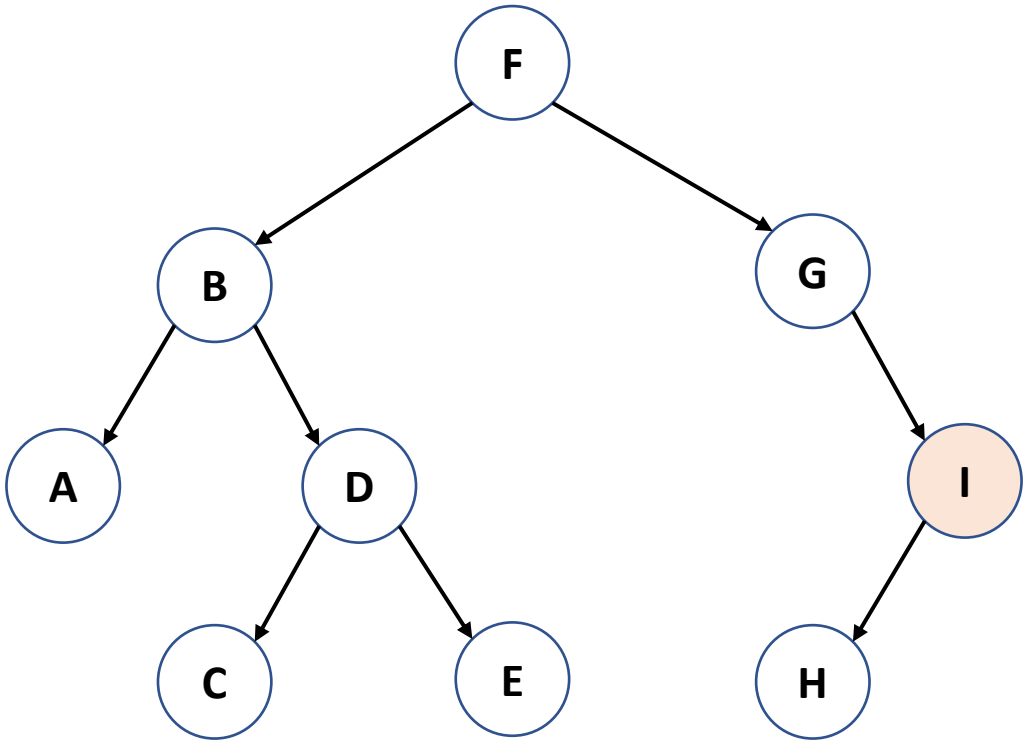




Process output: A B C D E F G H I

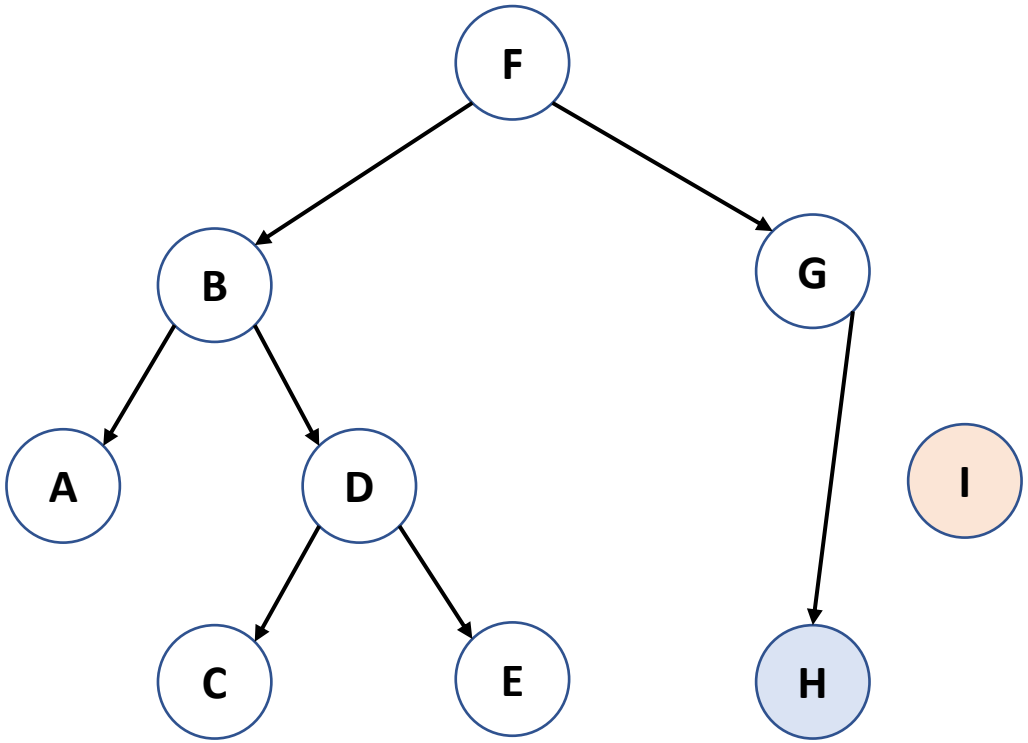
Delete ‘I’ (1 child)

```
1:   procedure BST_delete(T)
2:       if T→left == NIL and T→right == NIL then
3:           T→parent→[left or right] = NIL
4:           delete T
5:       else if T has one child then // splice out T
6:           T→parent→[left or right] = T→[left or right]
7:           delete T
8:       else if T has two children then
9:           BST_replace_with_successor(T)
10:      end if
11:  end procedure
```



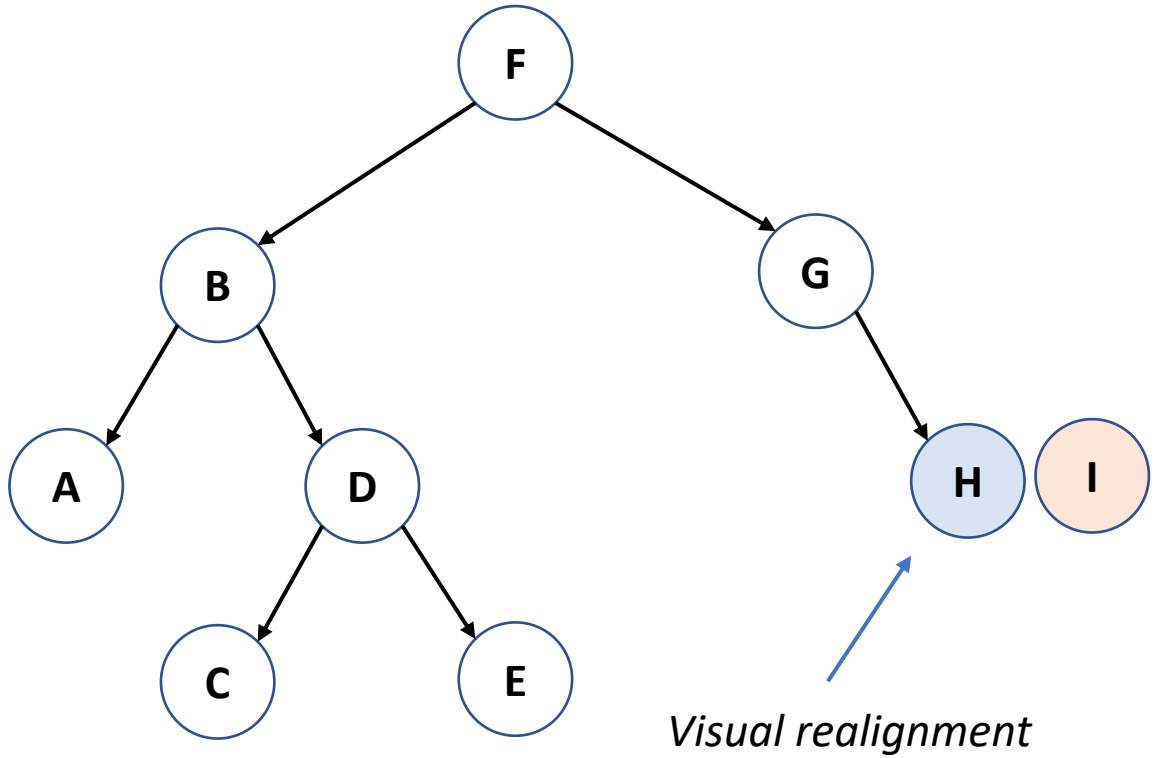
Delete ‘I’ (1 child)

```
1:   procedure BST_delete(T)
2:       if T→left == NIL and T→right == NIL then
3:           T→parent→[left or right] = NIL
4:           delete T
5:       else if T has one child then // splice out T
6:           T→parent→[left or right] = T→[left or right]
7:           delete T
8:       else if T has two children then
9:           BST_replace_with_successor(T)
10:      end if
11:  end procedure
```



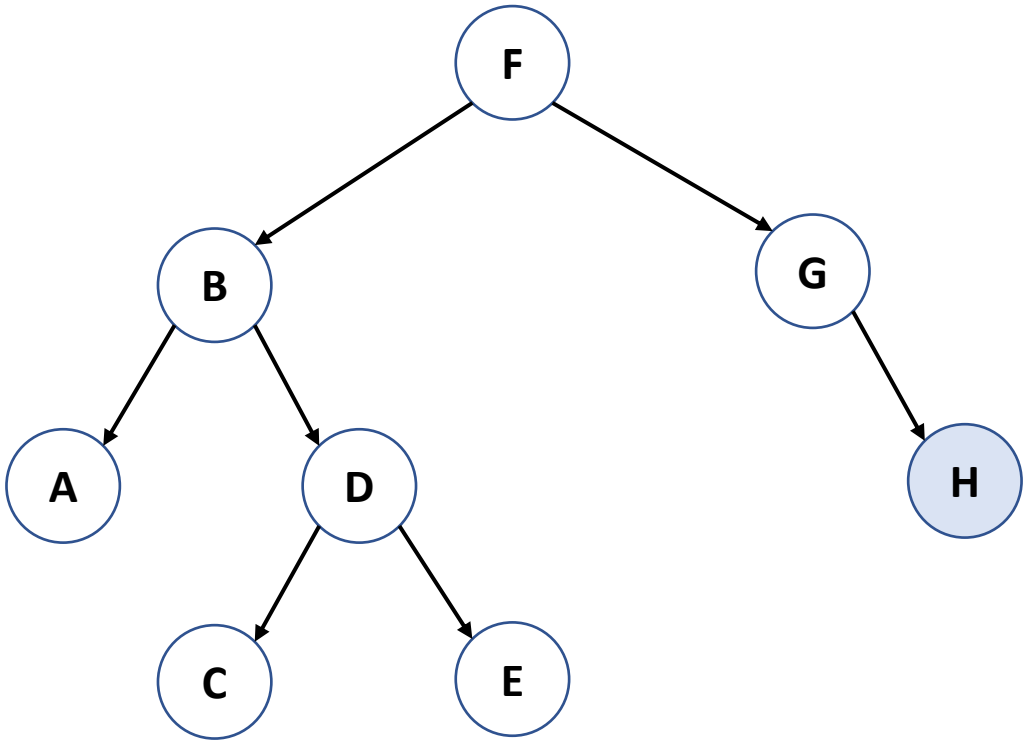
Delete ‘I’ (1 child)

```
1:   procedure BST_delete(T)
2:       if T→left == NIL and T→right == NIL then
3:           T→parent→[left or right] = NIL
4:           delete T
5:       else if T has one child then // splice out T
6:           T→parent→[left or right] = T→[left or right]
7:           delete T
8:       else if T has two children then
9:           BST_replace_with_successor(T)
10:      end if
11:  end procedure
```



Delete ‘I’ (1 child)

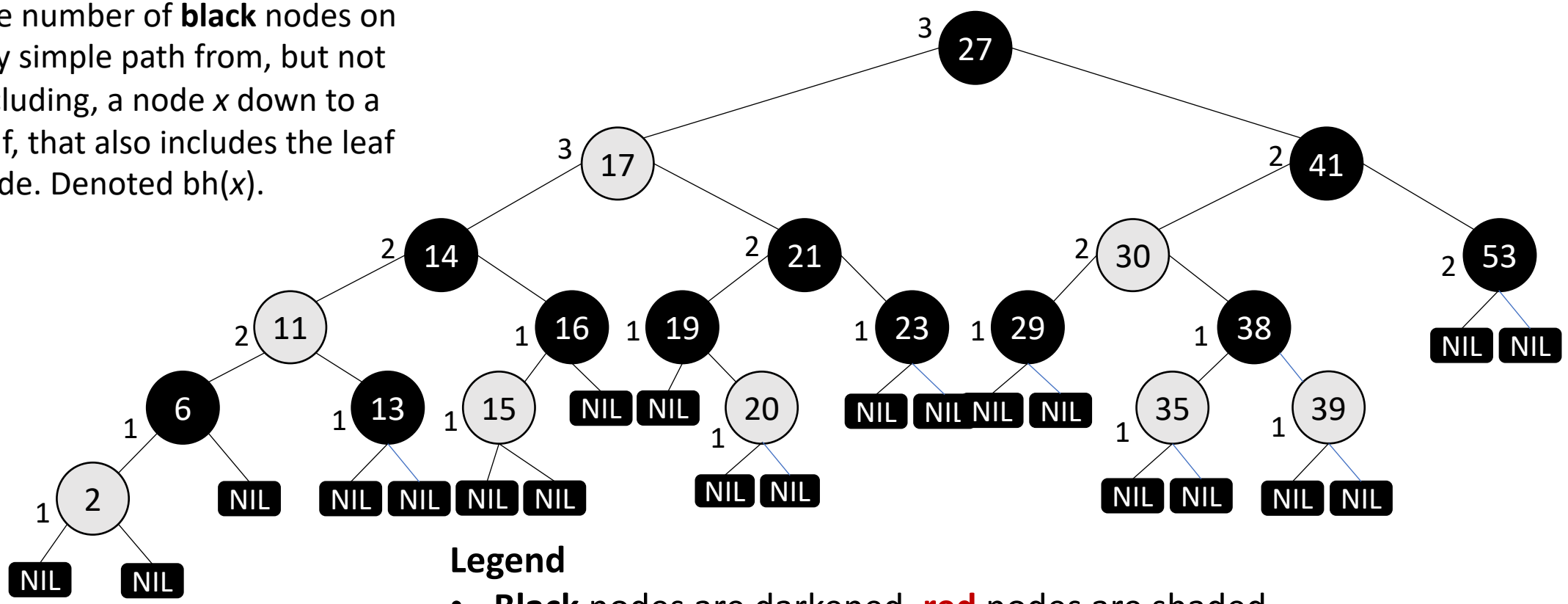
```
1:   procedure BST_delete(T)
2:       if T→left == NIL and T→right == NIL then
3:           T→parent→[left or right] = NIL
4:           delete T
5:       else if T has one child then // splice out T
6:           T→parent→[left or right] = T→[left or right]
7:           delete T
8:       else if T has two children then
9:           BST_replace_with_successor(T)
10:      end if
11:  end procedure
```



Visualising an RBT

1. Every node is either red or black.
2. The root is black.
3. Every leaf (nil/null) is black.
4. If a node is red, then both its children are black.
5. For each node, all paths from the node to leaves contain the same number of black nodes.

The number of **black** nodes on any simple path from, but not including, a node *x* down to a leaf, that also includes the leaf node. Denoted bh(*x*).



Legend

- **Black** nodes are darkened, **red** nodes are shaded.
- Every leaf (external nodes), are **black**, and shown as NIL
- Each node is marked with its "black height". E.g., black height of the root is 3.
- Leaves (external nodes) have a black height of 0 (unmarked).

RBT Properties

An RBT is a BST with the following properties:

1. Every node is either **red** or **black**.
2. The root is **black**.
3. Every leaf (nil/null) is **black**.
4. If a node is red, then both its children are **black***.
5. For each node, all paths from the node to leaves contain the same number of **black** nodes.

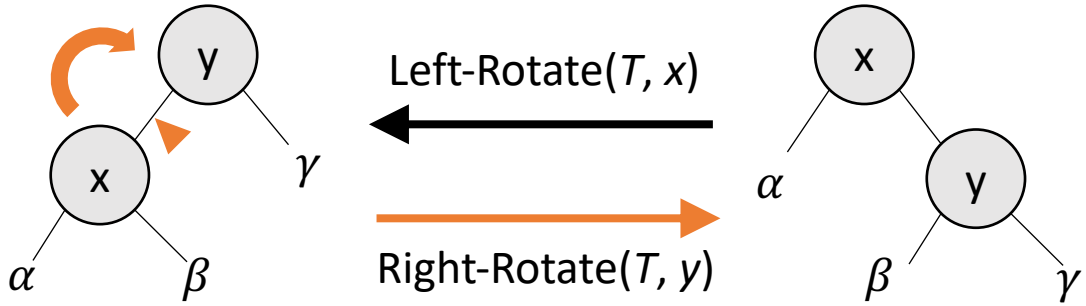
* This implies we cannot have consecutive red nodes.

Rotations

Rotations work by updating the pointer structure of the tree. When we do a right-rotation on node *y*, we assume that its left child *x* is not *T.nil*. *y* may be any node in the tree whose left child is not *T.nil*.

Right-rotation

- "pivots" around the edge from *x* to *y*.
- Makes *x* the new root of the subtree
- *y* becomes *x*'s right child
- *x*'s right child becomes *y*'s left child.



Insertion

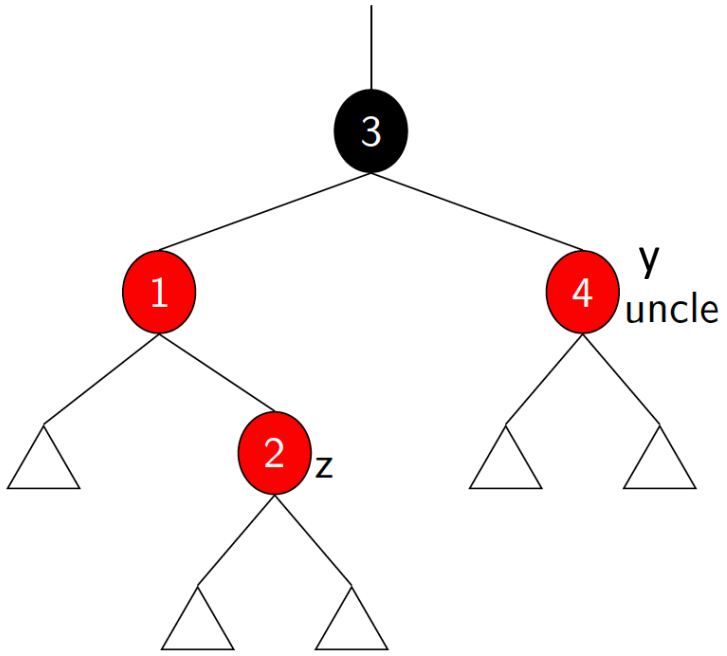
The basic algorithm for inserting a node into an RBT is:

```
1:   procedure RBT_Insert(T, z)
2:       BST_insert(T, z)
3:       z.colour = RED
4:       if z→parent == RED then      // Violation of property 4
5:           RBT_Insert_Fixup(T, z)
6:       end if
7:   end procedure
```

Insertion fixup

1. Label node inserted (z), and uncle (y)
2. Identify case fixup (1, 2, or 3) based on tree layout.

Initial:



Case 1: z's uncle, y, is **red**.

- L5-6: Colour z's parent and uncle black
- L7: Colour z's grandparent red
- L8: Z now points to z's grandparent

Case 2: z's uncle y is **black** and z is a right child.

- L11: z now points to z's parent
- L12: Left rotate on z (i.e. old z's parent)

Case 3: z's uncle y is black and z is a left child.

- L14: Colour z's parent black
- L15: Colour z's grandparent red
- L16: Right rotate on z's grandparent

RBT Deletion

At a conceptual level there are two stages

1. Delete the node
2. Execute a fix-up procedure (if deleted/spliced node was black).

Both stages require you to label nodes and perform an operation.

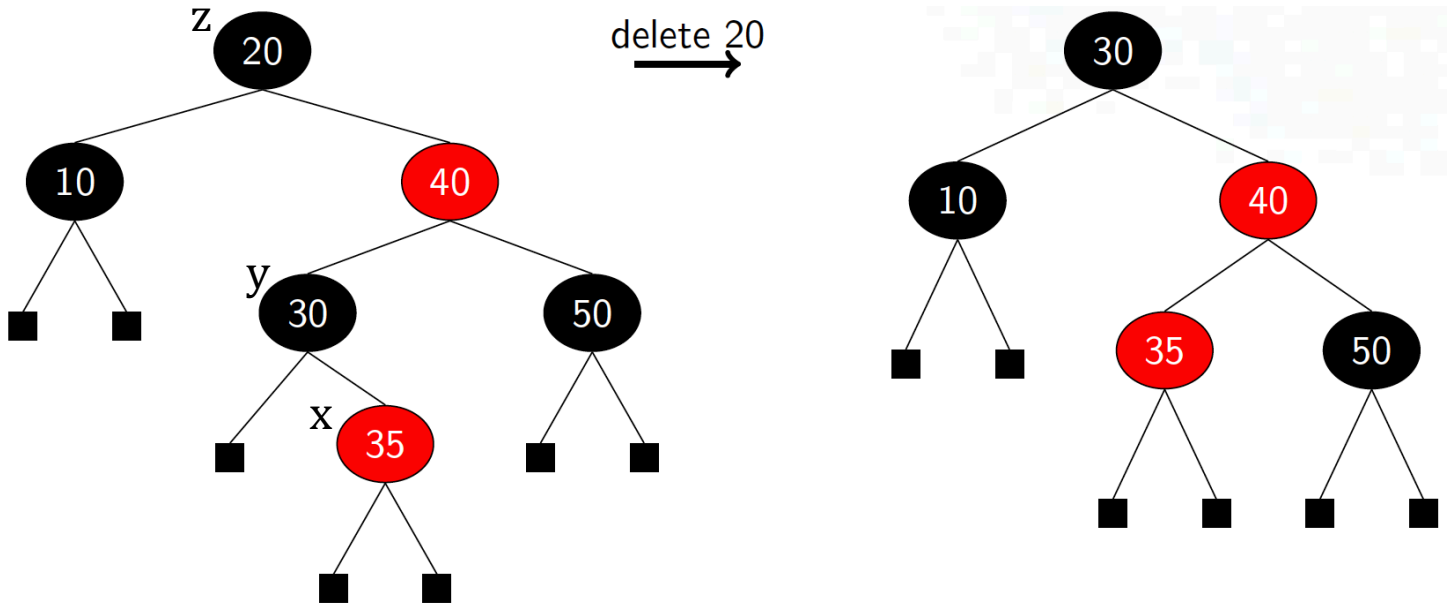
Stage 1 – Node deletion

To delete a node z, BST_delete recursively searches for z, and then:

1. If z has < 2 children, replace it by a child (possibly nil); or
2. If z has == 2 children, replace it by its successor

Some node, call it y, eventually gets spliced out.

It may be that $y = z$, or y may be z's successor.



Here z has two children, so y is z's successor, 30. y gets spliced out.
x replaces y

Stage 2 - Fixup

Fixup labels:

- z is the node to be deleted
- y is the node that gets spliced out (sometimes $y = z$ and sometimes y is z's successor)
- x is the child that replaced y
- w is the new sibling of x

For deletion fix up, case violation looks at **sibling**.
This is contrasts with insertion fix up, which looked at uncle.

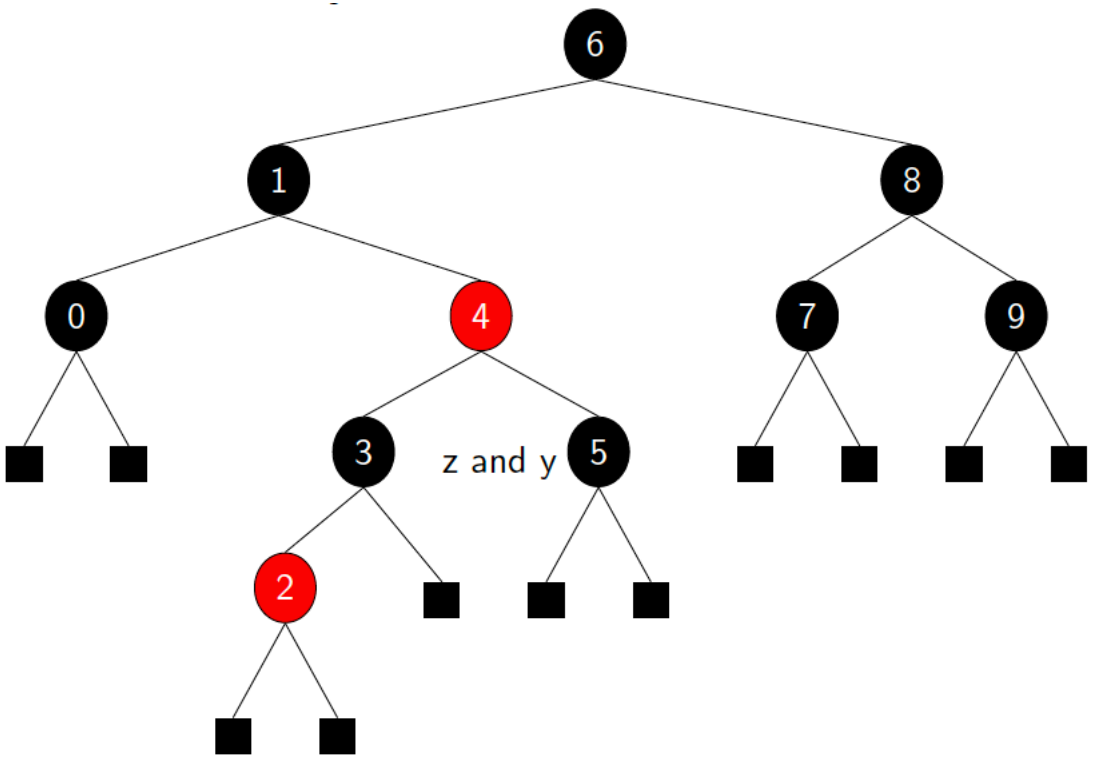
Stage 2 - Four cases to handle

1. x's sibling, w, is **red**. Fix then fall to case 2, 3, or 4.
2. w is **black** and has two black children. Fix then traverse up the tree. If fell through from case 1, terminate.
3. w is **black** and w's left child (or "inner" child) is **red** and right child ("outer") is **black**. Fix then fall to case 4.
4. w is **black** and w's right child ("outer") is **red**. Fix and terminate.

Here, "outer" and "inner" refer to w's child, and its position with respect to x.

Case 4 - Example

Delete 5

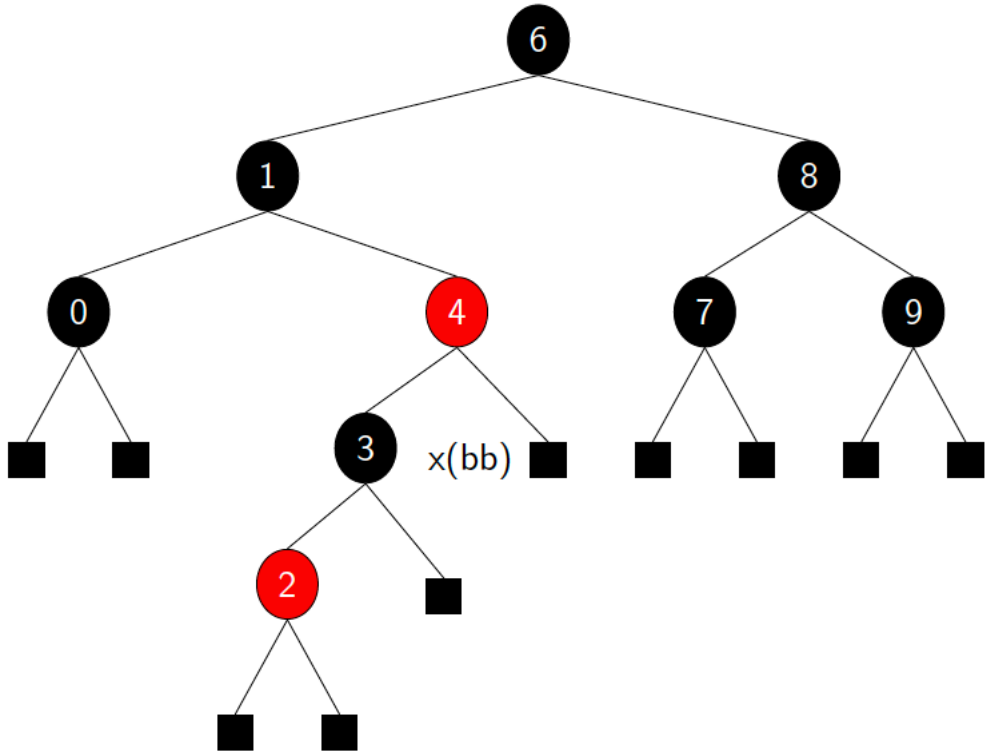


1. **Identify y and z.**
2. Delete z as for a BST
3. Identify w, w's children, and case
4. Fix any RBT violations

Y = spliced out node (removed from tree)
Z = Key value to remove
W = x's sibling

Case 4 - Example

Delete 5



1. Identify y and z.
2. **Delete z as for a BST**
3. Identify w, w's children, and case
4. Fix any RBT violations

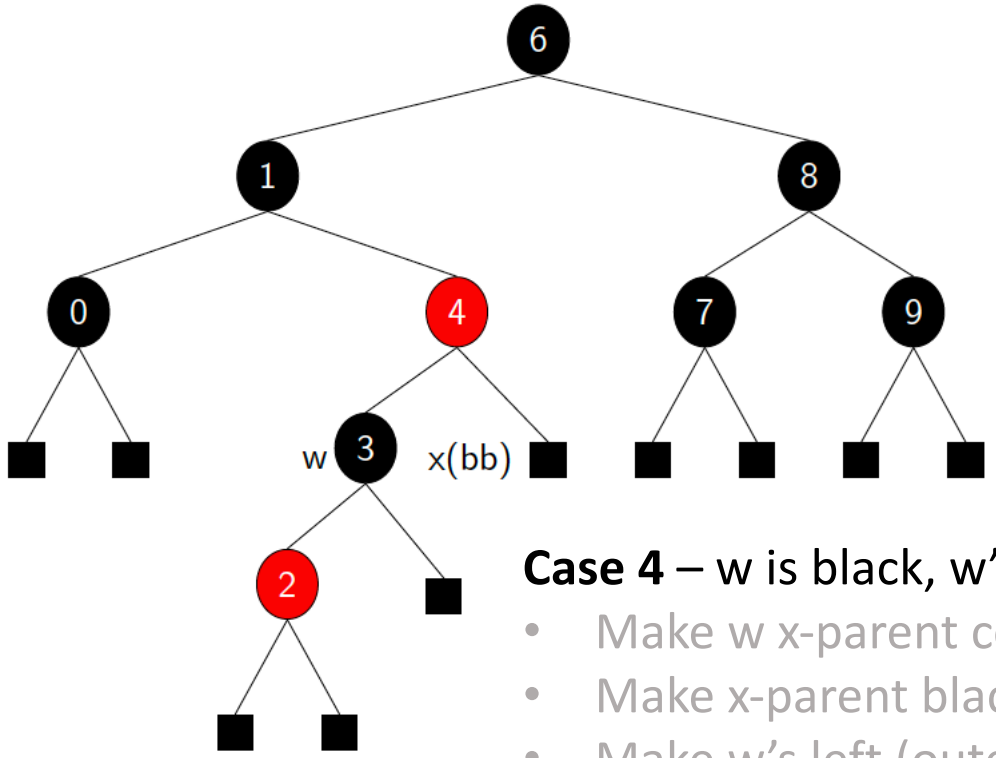
Y = spliced out node (removed from tree)
Z = Key value to remove
W = x's sibling

Case 4 - Example

1. Identify y and z.
2. Delete z as for a BST
3. **Identify w, w's children, and case**
4. Fix any RBT violations

Y = spliced out node (removed from tree)
Z = Key value to remove
W = x's sibling

Delete 5



Note: here we have x as a right-child. Therefore, we flip all left and rights. These are shown underlined.

Case 4 – w is black, w's left (outer) child is red

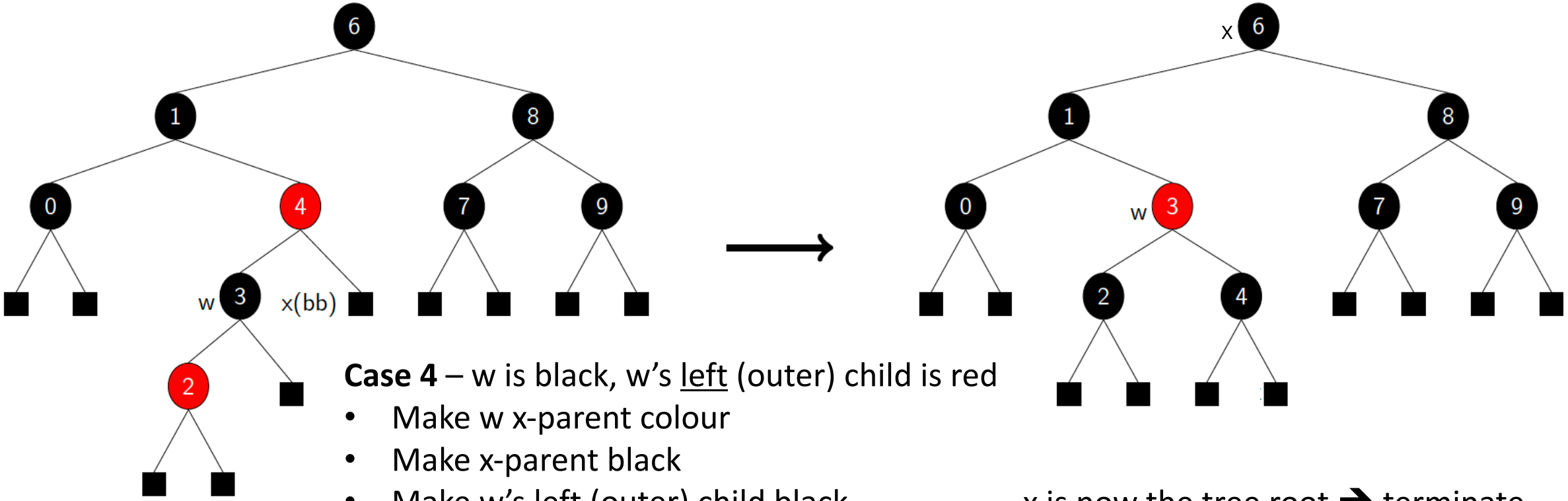
- Make w x-parent colour
- Make x-parent black
- Make w's left (outer) child black
- Right-rotate x-parent
- Make x the root

Case 4 - Example

1. Identify y and z.
2. Delete z as for a BST
3. Identify w, w's children, and case
4. **Fix any RBT violations**

Y = spliced out node (removed from tree)
Z = Key value to remove
W = x's sibling

Delete 5



Case 4 – w is black, w's left (outer) child is red

- Make w x-parent colour
- Make x-parent black
- Make w's left (outer) child black
- Right-rotate x-parent
- Make x the root

x is now the tree root ➔ terminate

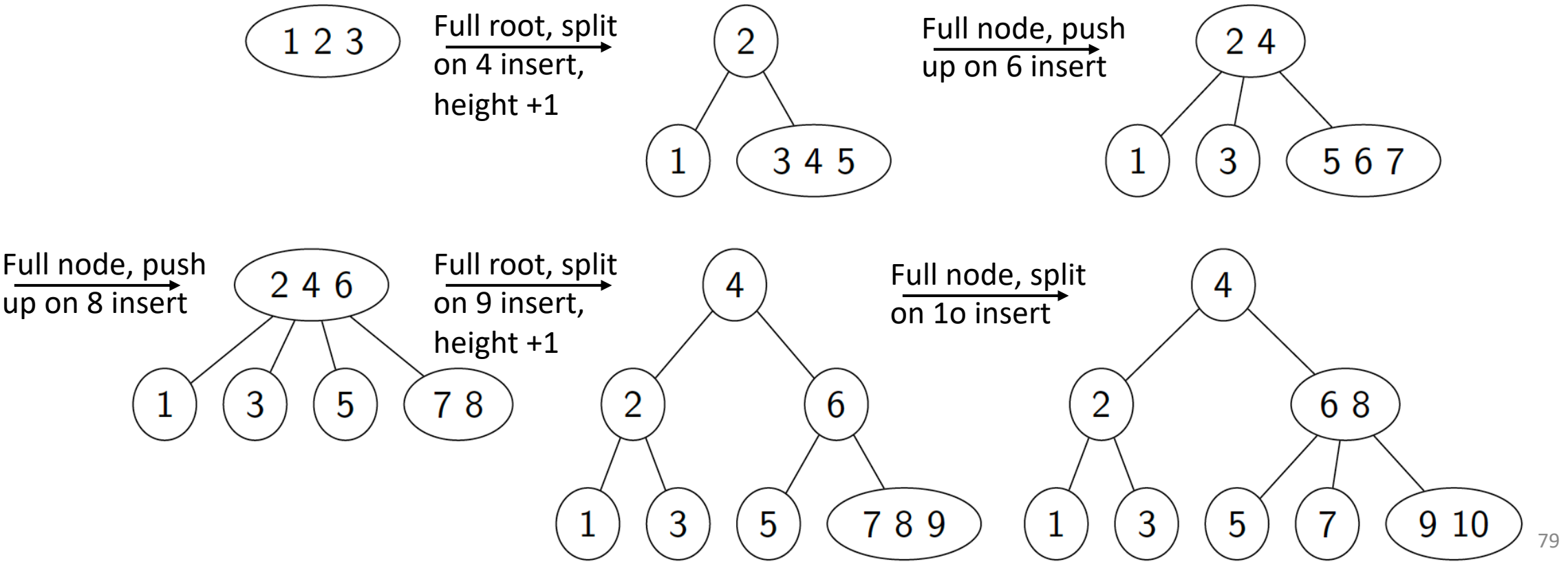
B-tree Definition

A B-tree of **minimum degree *t*** ≥ 2 has the following properties:

1. Every node *x* has the following attributes:
   a) *x.n* = the number of keys currently stored in node *x*
   b) Keys are stored in increasing order: $x.key_1 \leq x.key_2 \leq \ldots \leq x.key_n$
   c) *x.leaf*, a Boolean, is TRUE if *x* is a leaf, FALSE if *x* is an internal node.
2. All leaves have the same depth, which is the tree's height *h*
3. Nodes have upper and lower bounds on the number of keys.
   a) Every node other than the root has x.n ≥ $t - 1$ keys. Every internal node other than the root has ≥ *t* children.
   b) Every node may contain x.n ≤ $2t - 1$ keys. An internal node may have ≤ 2t children (*m* = *2t*). A node is full if it has x.n = $2t - 1$ keys.
4. The root has at least two children if it is not a leaf node.

Example: Insert 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

For t = 2, the maximum number of keys is 3 ($2t$-1) and max children m = 4.



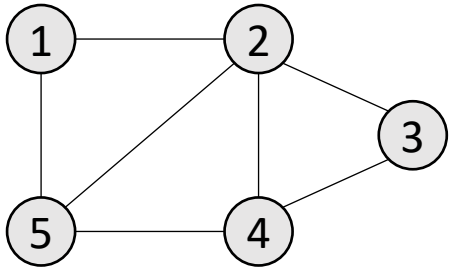
Deletion cases

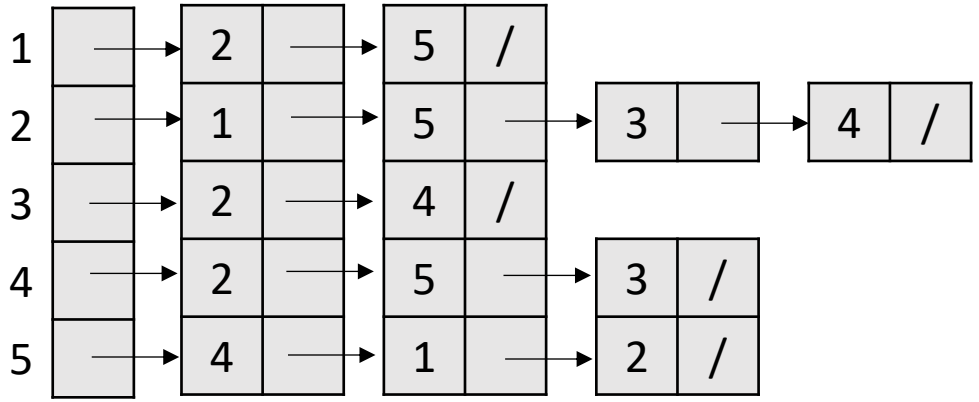
1. *x* is a leaf and contains the key (it will have at least *t* keys). This case is trivial – just delete the key.
2. *x* is an internal node and contains the key. There are 3 subcases:
   2a. predecessor child node has at least *t* keys
   2b. successor child node has at least *t* keys
   2c. neither predecessor nor successor child has *t* keys
3. *x* is an internal node, but doesn't contain the key. Find the child subtree of *x* that contains the key if it exists (call the child *c*). There are three subcases:
   3a. *c* has at least t keys. Simply recurse to c.
   3b. *c* has *t* - 1 keys and one of its siblings has *t* keys.
   3c. *c* and both siblings have *t* - 1 keys.

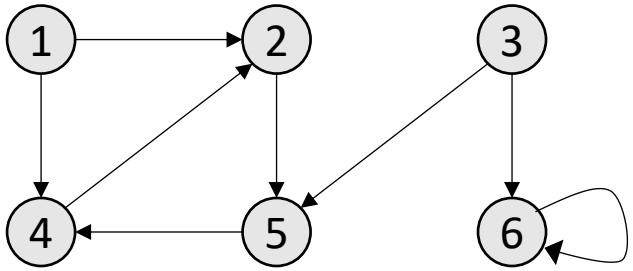
Today's outline

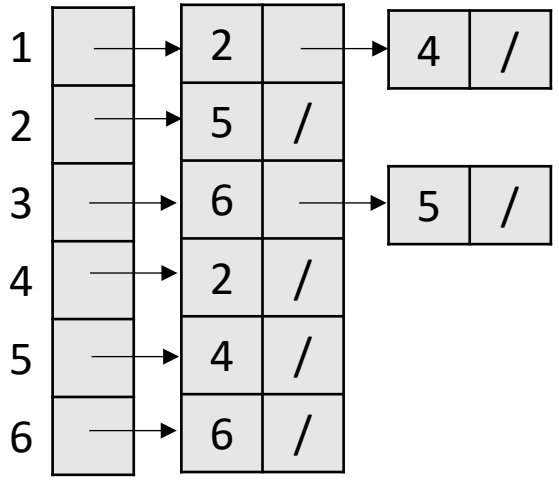
1. Exam details and preparation
2. Course overview
3. Asymptotic notation
4. Proofs
5. Divide and Conquer
6. Hash tables
7. Trees
8. Graphs and greedy algorithms
9. Dynamic programming
10. P and NP

Adjacency lists

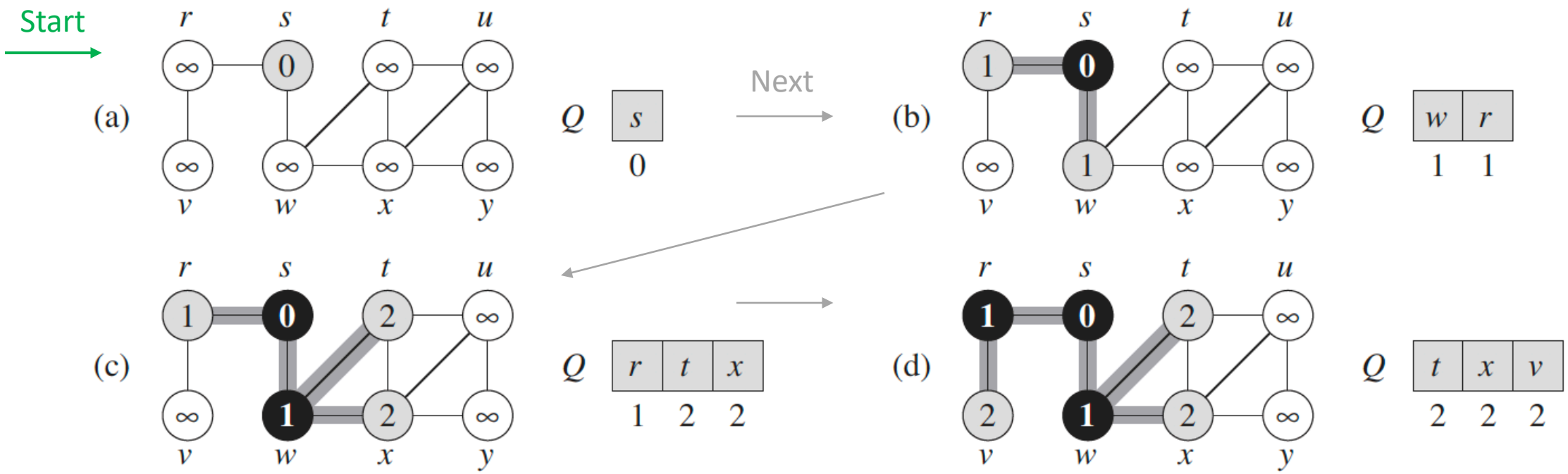






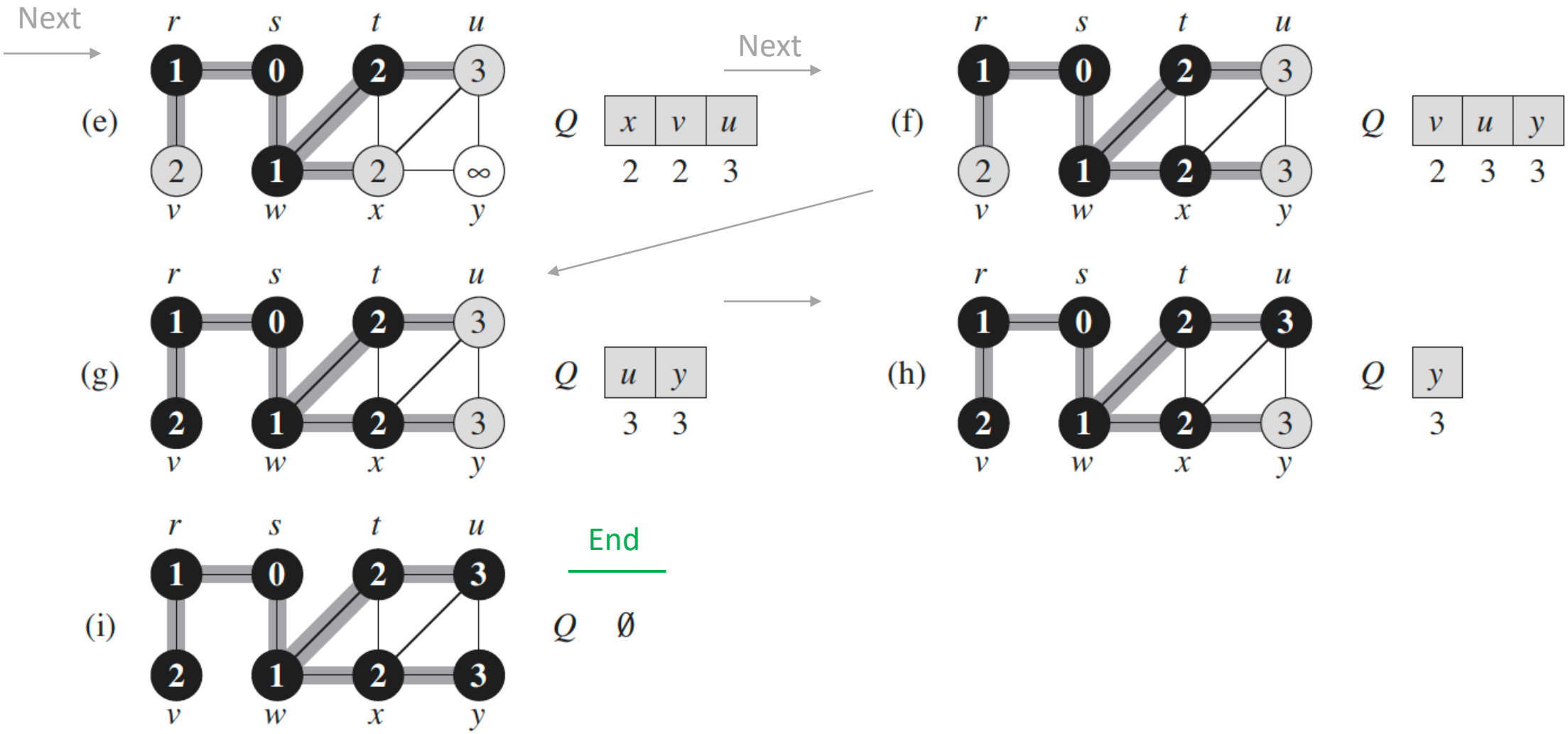


BFS Example



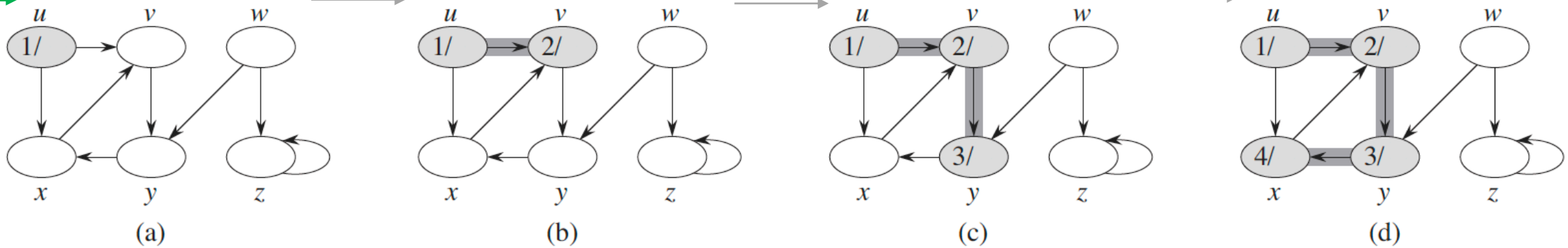
Q = Queue

BFS Example

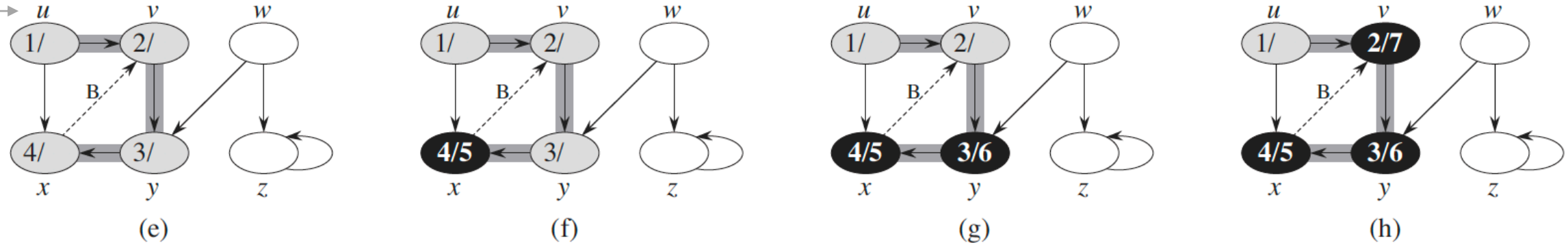

Next
(e)
r s t u
1 0 2 3
2 1 2 ∞
v w x y
Q x v u
2 2 3
Next
(f)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q v u y
2 3 3
(g)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q u y
3 3
(h)
r s t u
1 0 2 3
2 1 2 3
v w x y
Q y
3
(i)
r s t u
1 0 2 3
2 1 2 3
v w x y
End
Q Ø

DFS Example

Start



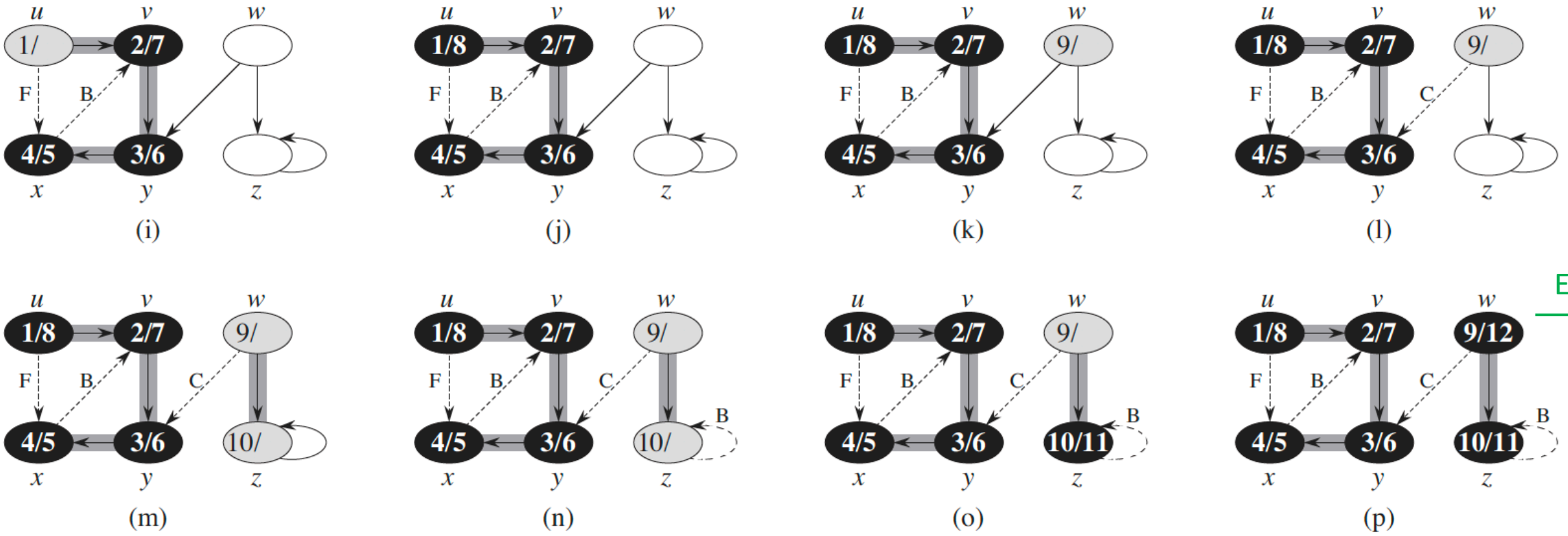
Next



(e) We rediscover vertex v, but it's not white, so this is a back edge.
(f) We've now explored all vertices in x, colour black, and recurse up to (g)

DFS Example

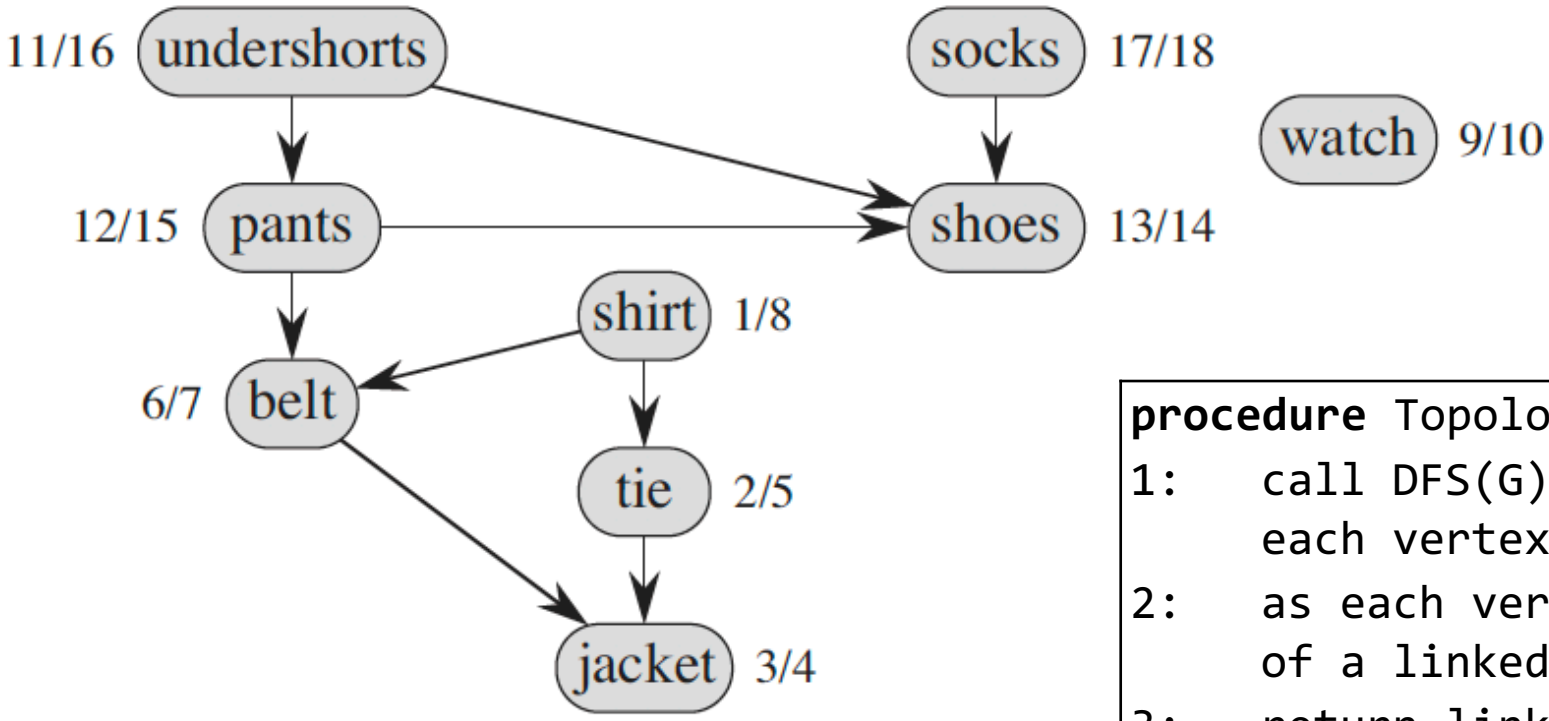
Next



End

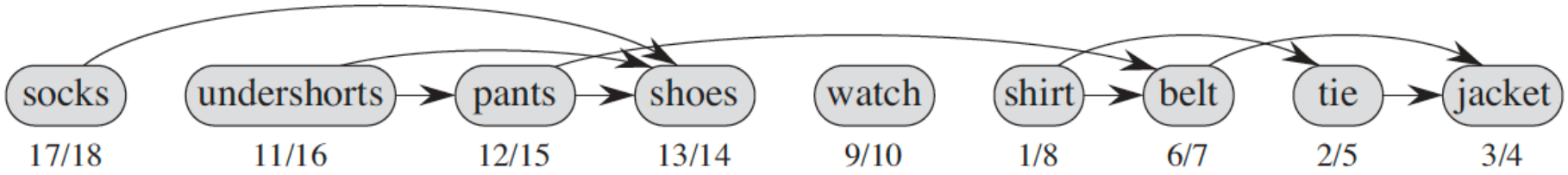
(j) Explored all vertices in u, colour black.
(k) Explore next white vertex in Graph (Liens 5-7 of DFS)
(l) Rediscover node y, but it's not white, so cross edge.
(p) All vertices fully explored, colour last vertex black.

Topological sort




```
procedure Topological-Sort(G)
1:    call DFS(G) to compute finishing times v.f for
      each vertex v
2:    as each vertex is finished, insert onto front
      of a linked list
3:    return linked list
```

List sorted by decreasing order of finish times. Edges retained.



Dijkstra's algorithm - Single procedure version

```
procedure Dijkstra(G,w,s)
1:    for each vertex v ∈ G.V                  // Initialise all vertices
2:        v.d = ∞
3:        v.π = NIL
4:    s.d = 0                                  // Set source distance to 0
5:    S = ∅                                    // Set of vertices with shortest-paths
6:    Q = G.V                                  // Set of unvisited vertices
7:    while Q ≠ ∅                              // While still vertices to explore
8:        u = Extract-Min(Q)                   // Get vertex with min shortest path d
9:        S = S ∪ {u}                          // Add to set with shortest paths
10:       for each vertex v ∈ G.Adj[u]         // For each adjacent vertex
11:           if v.d > u.d + w(u,v)            // Is u's adj vertex v.d longer than this path?
12:               v.d = u.d + w(u,v)           // Update v's distance to shorter value
13:               v.π = u                      // u is v's new predecessor
end procedure
```

Example: Dijkstra's algorithm

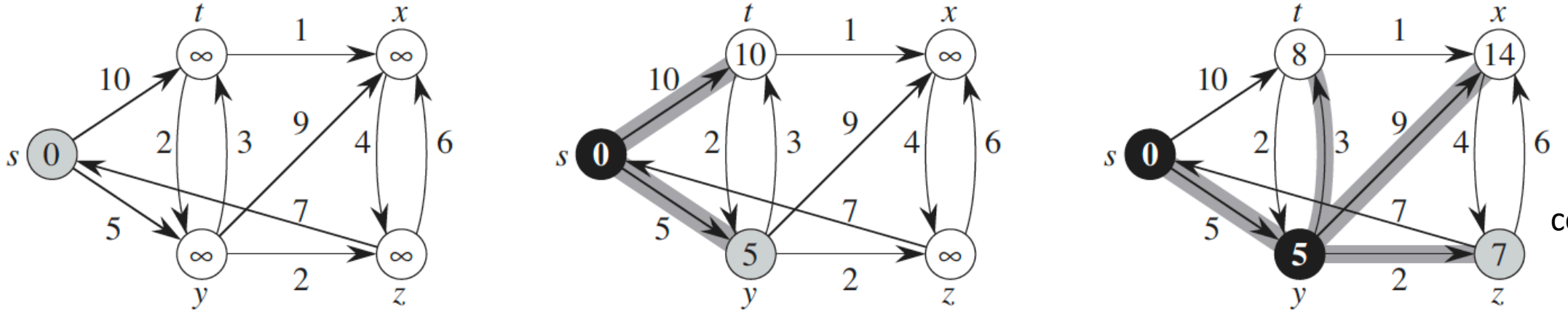
Line numbers from single procedure code

s is source vertex. Shortest-path estimates, *v.d*, appear within vertices. Shaded edges indicate predecessor values. Black vertices are in set *S*. White vertices are in min priority queue $Q = V - S$.

(a) Just before the while loop (Lines 4-8). Always begin at source *s*, as *s.d* = 0, Line 4.

(b) Add s to *S* (coloured black), and relax adjacent vertices *t* and *y*. *y* has the minimum *v.d* (shaded grey), and will be selected next.

(c) Add y to S, adjacent vertices *t*, *x*, and *z*. Note that *t* is updated again.



(a) (b) (c)

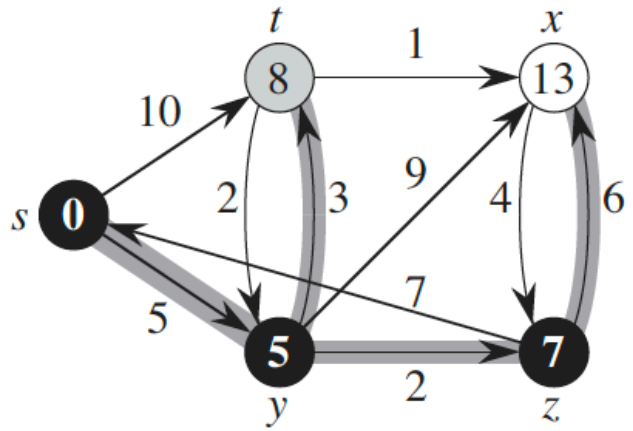
continued

Example: Dijkstra's algorithm

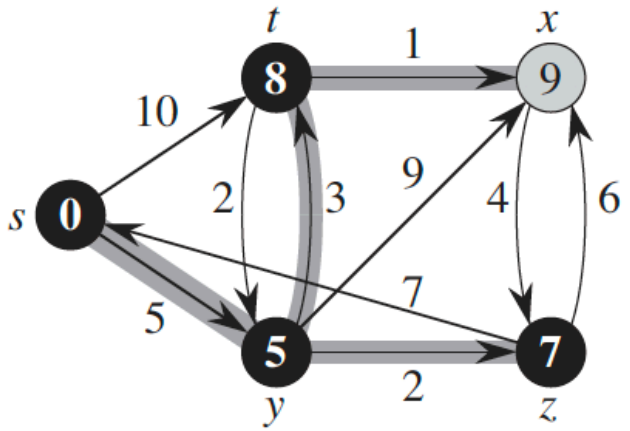
d) *z* is selected next. Update adjacent vertex *x*. *t* will be selected next.

e) Add *t* to S, update adjacent and final vertex *x*.

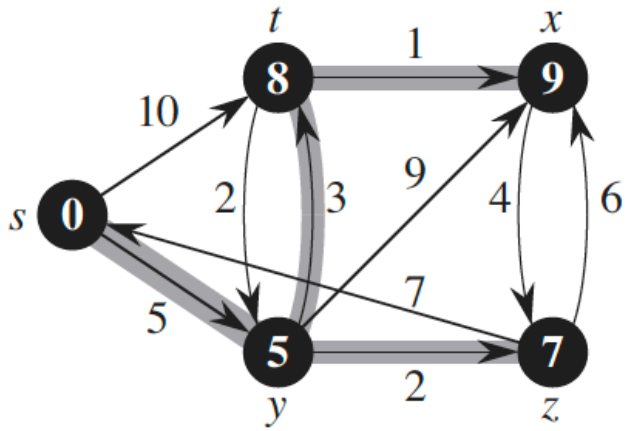
f) Add *x* to S. While loop (Line 7) will now terminate s all vertices have moved from *Q* to *S*.



(d)



(e)



(f)

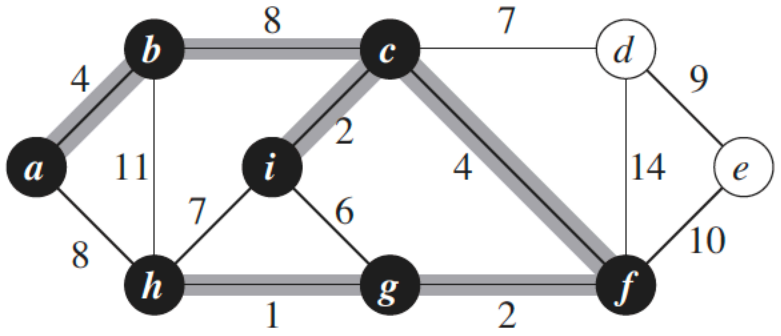
Prim's algorithm


```
procedure MST-Prim(G,w,r)                    // r = root of minimum spanning tree
1:    for each vertex u ∈ G.V                // Initialise graph
2:        u.key = ∞
3:        u.π = NIL
4:    r.key = 0
5:    Q = G.V                                // Set V – S (not in MST)
6:    while Q ≠ ∅
7:        u = Extract-Min(Q)                 // Get vertex on light-edge that crosses cut (.key)
8:        for each vertex v ∈ G.Adj[u]       // Update u's adjacent vertices not in MST
9:            if v ∈ Q and w(u,v) < v.key    // Update non-MST vertices with lower weight edge?
10:               v.π = u
11:               v.key = w(u,v)
end procedure
```

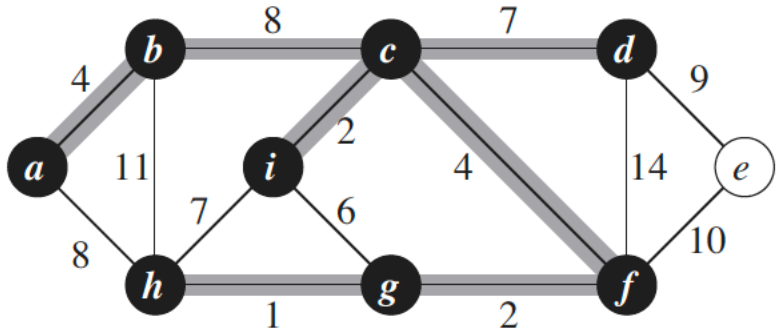
Example: Prim's algorithm

Root = vertex a

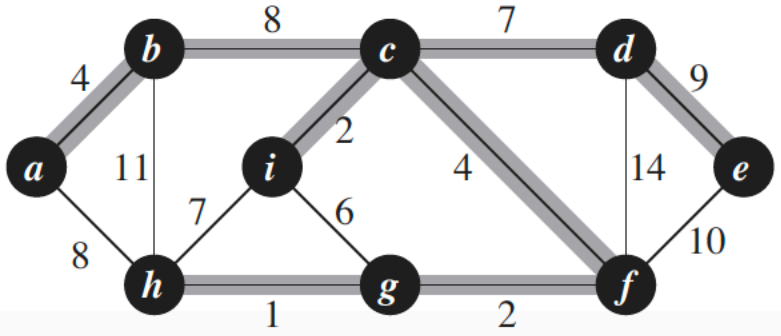
(g)



(h)



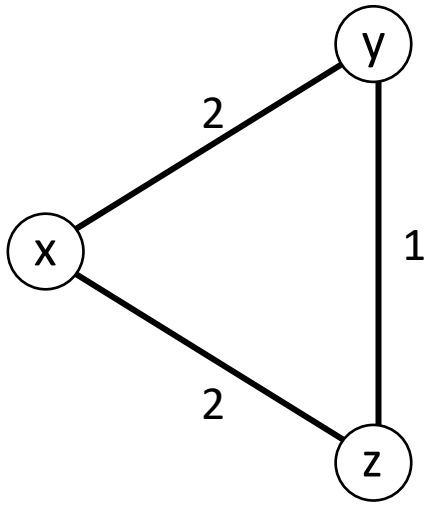
(i)



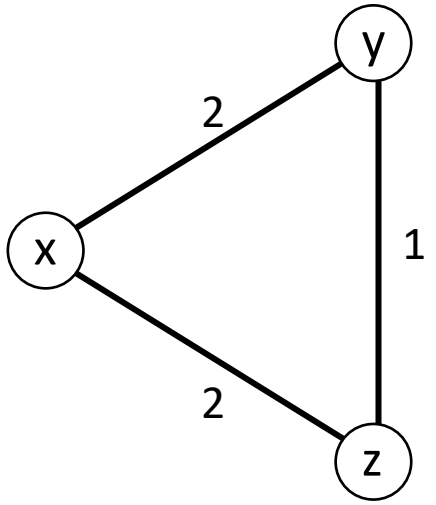
Finished

Class challenge 5

To illustrate the difference between Dijkstra's and Prim's algorithms, apply them respectively to the following graphs, starting at 'x'.



Dijkstra's



Prim's

Today’s outline

1. Exam details and preparation
2. Course overview
3. Asymptotic notation
4. Proofs
5. Divide and Conquer
6. Hash tables
7. Trees
8. Graphs and greedy algorithms
9. Dynamic programming
10. P and NP

Approach to dynamic programming

Greedy and dynamic programming approaches are used for optimisation problems.

Dynamic programming approach

1. Start with a greedy approach. If that fails, try dynamic.
2. Identify decision points in the problem, and determine if the problem has **optimal substructure**.
3. Define recursive solution
4. Improve it with memoisation or an iterative approach

0-1 Knapsack Recursive solution

We have made the following observations:

1) If there are no items in our set S_0, then the maximum value is 0.
2) If there is no space in our knapsack, then the maximum value is 0
3) If the k^{th} item can't fit in the knapsack, then the maximum is the same as the maximum for $k - 1$ items.
4) Otherwise, the maximum is either:
   - the maximum without the k^{th} item in the optimal set, in which case we have a new problem with $k - 1$ items and maximum weight w.
   - the maximum with the k^{th} item in the optimal set, in which case we have a new problem with $k - 1$ items and maximum weight $w - w_k$.

Recursive non-greedy solution

So we can define our optimum $V[k, w]$ recursively as:

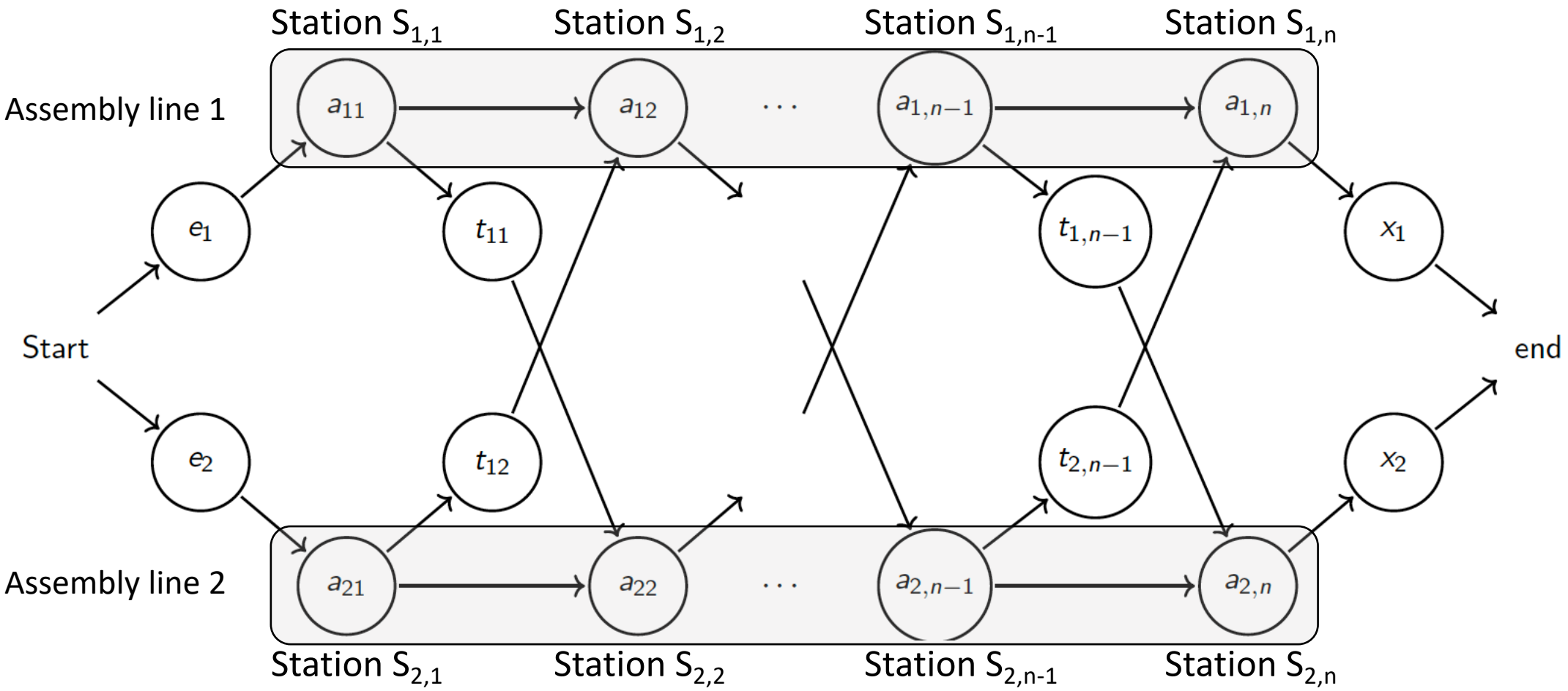
$$V[0, w] = 0$$
$$V[k, 0] = 0$$
$$V[k, w] = V[k-1, w] \; if \; w_k > w$$
$$V[k, w] = \max(V[k-1, w], v_k + V[k-1, w - w_k])$$

1) No items
2) No space
3) k^{th} item can't fit
4) Max of:
   a) Without k^{th}
   b) With k^{th}

Recursive top-down implementation (Brute force)

```
procedure RecursiveKnapsack(k, W, V, wmax) // Knap item, Weights, Values, max weight
1:    if k==0 or wmax ≤ 0 return 0, ∅        // No items OR no space (1 & 2)
2:    if W[k]>wmax                           // Can't fit k into knapsack (3)
3:        return RecursiveKnapsack(k-1,W,V,wmax)
4:                    // Check the maximum value (v1) without item k (4a)
5:    v1, items_not = RecursiveKnapsack(k-1,W,V,wmax)
6:                    // Check the maximum value (v2) with item k (4b)
7:    v2, items_do = RecursiveKnapsack(k-1,W,V,wmax-W[k])
8:    v2 = v2 + V[k]                         // Add value of current item
9:    items_do.add(k)                        // Add item k to list of take items
10:   if v2 > v1
11:       return v2, items_do                // Do use item k
11:   else
12:       return v1, items_not               // Don't use item k
end procedure
```

Assembly line



e_i = Entry time for line *i*

$a_{i,j}$ = Assembly time for Station *j*, on line *i*

$t_{i,j}$ = Transfer time away from line i, after station $S_{i,j}$

x_i = Exit time for vehicle to leave line i

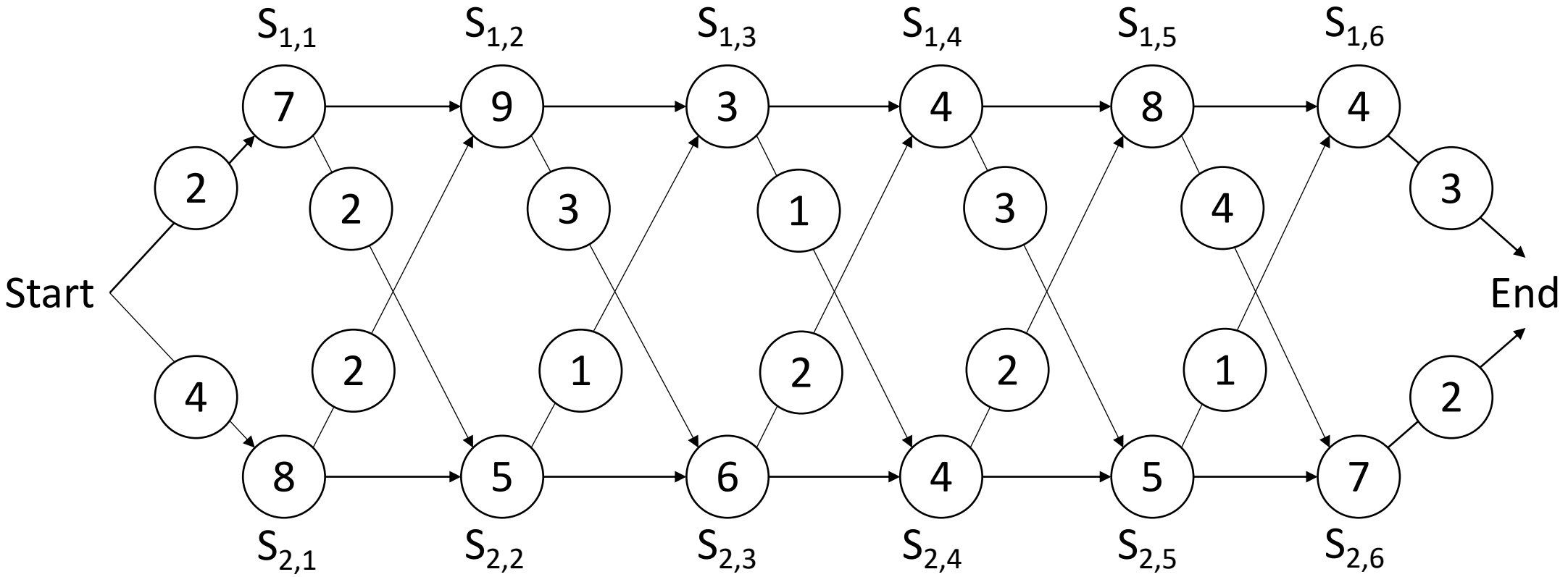
Developing a recursive solution

We can now define our final recursive equations:

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{If } j = 1, \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{If } j \geq 2. \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{If } j = 1, \\ \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{If } j \geq 2. \end{cases}$$

Class challenge 6



Longest common subsequence problem

Our theorem gives us a very clear optimal substructure problem from which we can write a naive recursive solution (brute force).

Let $l[i,j]$ be the length of an LCS of the sequences X_i and Y_j. If either prefix is zero, $i = 0\ or\ j = 0$, then the LCS has length 0. Then:

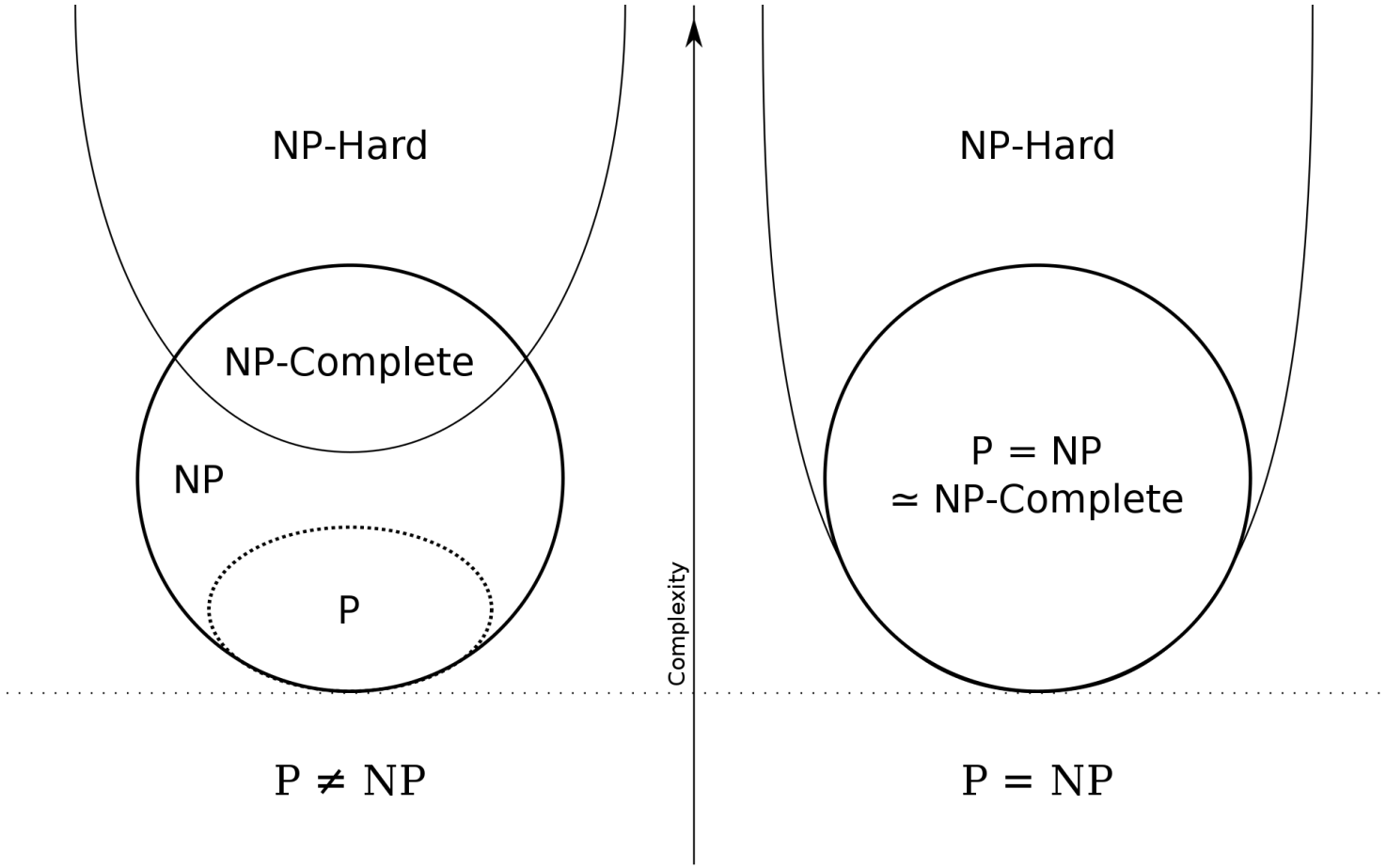
$$l[i,j]\begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0\ , \\ l[i-1,j-1]+1, & \text{If } i,j > 0 \text{ and } x_i = y_j\ , \\ \max(l[i,j-1], l[i-1,j]) & \text{If } i,j > 0 \text{ and } x_i \neq y_j\ . \end{cases}$$

1) Empty strings
2) Match char, or
3) Max of:
   a) X & Y-1
   b) X-1 and Y

Today’s outline

1. Exam details and preparation
2. Course overview
3. Asymptotic notation
4. Proofs
5. Divide and Conquer
6. Hash tables
7. Trees
8. Graphs and greedy algorithms
9. Dynamic programming
10. P and NP

P and NP


NP-Hard
NP-Complete
NP
P
Complexity
NP-Hard
P = NP
≃ NP-Complete
P ≠ NP
P = NP

Suggested reading

The lectures notes are the primary examinable material.

For each lecture, see the relevant suggested textbook sections.

Solutions

Class challenge 1

Prove that $n^2 \leq 2^n$

Base case: $4^2 = 16 \leq 2^4 = 16$

Induction hypothesis: $2^n \geq n^2$ for some n ≥ 4

Inductive step: $2^{n+1} \geq (n+1)^2 = n^2 + 2n + 1$

$$2^{n+1} = 2 \cdot 2^n$$
$$= 2^n + 2^n$$
$$\geq n^2 + n^2$$
$$\geq n^2 + n \cdot n$$
$$\geq n^2 + 4 \cdot n$$
$$\geq n^2 + 2 \cdot n + 2 \cdot n$$
$$\geq n^2 + 2 \cdot n + 8$$
$$\geq n^2 + 2 \cdot n + 1$$

Lets use our induction hypothesis here, that $2^n > n^2$

Lets use the second part of our induction hypothesis, n ≥ 4

Lets use the second part of our induction hypothesis, n ≥ 4

Conclusion: Since both the base case and the inductive step have been proved, the statement P(n) is True for n+1, for all n > 4. ■

Class challenge 2

Prove that $2^n \leq n!$ for all n ≥ 4

Base case: $2^4 = 16 \leq 4! = 4 * 3 * 2 * 1 = 24$

Induction hypothesis: $2^n \leq n!$ for some n ≥ 4

Inductive step: $2^{n+1} \leq (n+1)!$

$(n+1)2^n \leq (n+1)n!$

$(n+1)2^n \leq (n+1)!$

Lets multiple both sides of inequality by (n+1)

Since n + 1 > 2, then $(n+1)2^n > 2 \cdot 2^n = 2^{n+1}$

Therefore: $2^{n+1} \leq (n+1)!$ for all n ≥ 4

Conclusion: Since both the base case and the inductive step have been proved, the statement P(n) is True for n+1, for all n > 4. ■

Class challenge 3

Where X is given by $f(n) = n^3 + 2n$
Are all the elements of X divisible by 3?

Base case: $f(1) = 1^3 + 2 \cdot 1 = 3$ - True

Inductive step: $P(n+1)$: $f(n+1) = (n+1)^3 + 2 \cdot (n+1)$

$$f(n+1) = (n+1)^3 + 2 \cdot (n+1)$$
$$= (n+1)(n+1)(n+1) + 2n + 2$$
$$= (n^2 + 2n + 1)(n+1) + 2n + 2$$
$$= (n^3 + 3n^2 + 3n + 1) + 2n + 2$$
$$= n^3 + 2n + 3n^2 + 3n + 3$$
$$= f(n) + 3n^2 + 3n + 3$$
$$= f(n) + 3 \cdot (n^2 + n + 1)$$

Here's our f(n). Lets use the induction hypothesis, and assume that f(n) is True…

This term will always be divisible by 3.

Conclusion: Since both the base case and the inductive step have been proved, the statement P(n) is True for n+1, and so the proof is complete. ■

Class challenge 4

Insert (~~12~~, ~~13~~, ~~43~~, ~~52~~, ~~72~~, ~~63~~) with h(k) = k%10, using first quadratic probing then double hashing.

quadratic

H(63, 2) = (k%m + i^2) % m = 7

Total collisions = 8

0	
1	
2	12
3	13
4	43
5	72
6	52
7	63
8	
9	

0	52
1	43
2	12
3	13
4	72
5	63
6	
7	
8	
9	

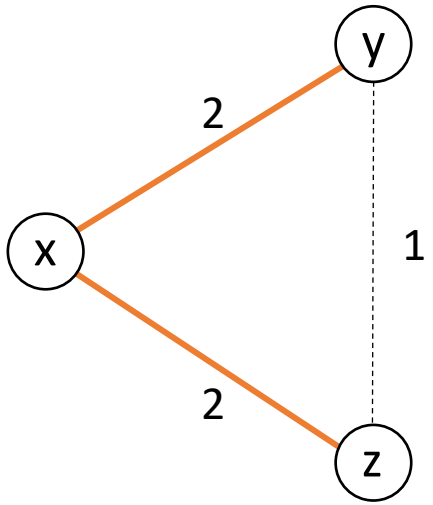
double

H(63, 2) = (k%m + i(1+k%(m-1)) % m = 5

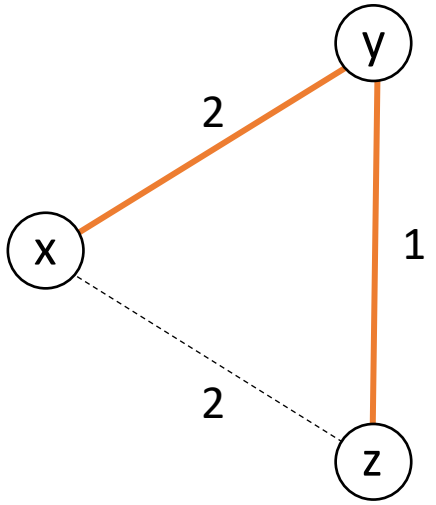
Total collisions = 6

Class challenge 5

To illustrate the difference between Dijkstra's and Prim's algorithms, apply them respectively to the following graphs.

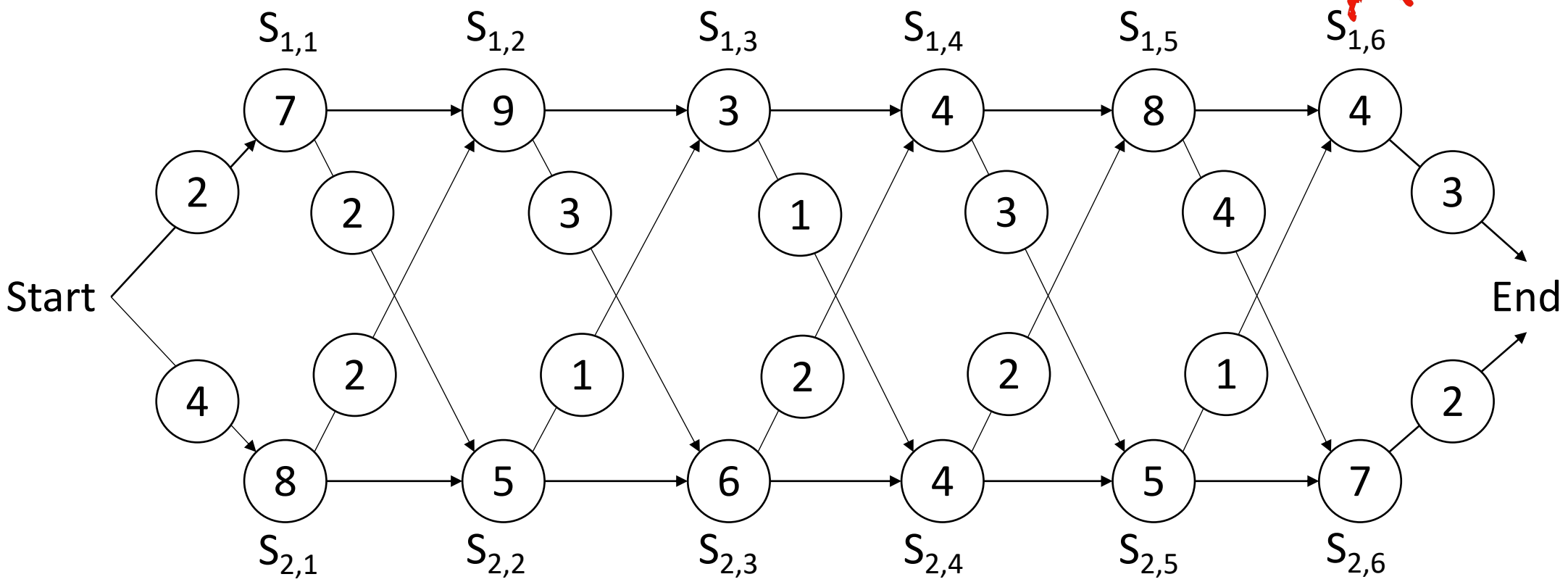


Dijkstra's



Prim's

Class challenge 6 (1 of 3)



$$f^* = \min(\mathbf{35} + \mathbf{3}, 37 + 2) = 38$$

We set L* to 1, our fastest exit station

j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

j	2	3	4	5	6
$L_1[j]$	1	2	1	1	2
$L_2[j]$	1	2	1	2	2

L* = 1

Class challenge 6 (2 of 3)

With $L_2[4] = 1$ (use $S_{1,3}$).

With $L_1[3] = 2$ (use $S_{2,2}$).

With $L_2[2] = 1$ (use $S_{1,1}$).

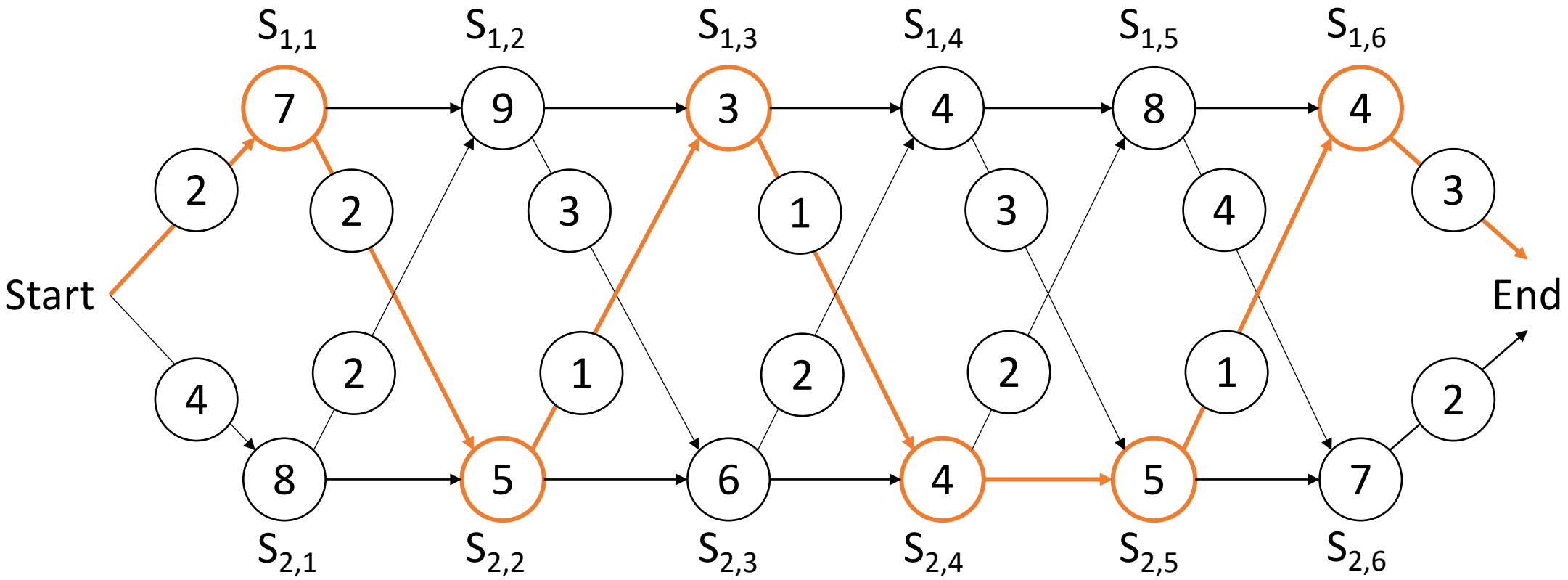
Therefore, our fastest path is:

$S_{1,1}, S_{2,2}, S_{1,3}, S_{2,4}, S_{2,5}, S_{1,6}$

j	2	3	4	5	6
$L_1[j]$	1	**2**	1	1	**2**
$L_2[j]$	**1**	2	**1**	**2**	2

L* = 1

Class challenge 6 (3 of 3)



Fastest path: $S_{1,1}$, $S_{2,2}$, $S_{1,3}$, $S_{2,4}$, $S_{2,5}$, $S_{1,6}$

Image attributions

This Photo by By Behnam Esfahbod, CC BY-SA 3.0

Disclaimer: Images and attribution text provided by PowerPoint search. The author has no connection with, nor endorses, the attributed parties and/or websites listed above.