EECS3101 Design and Analysis of Algorithms

Lecture Notes

Fall 2025

Jackie Wang

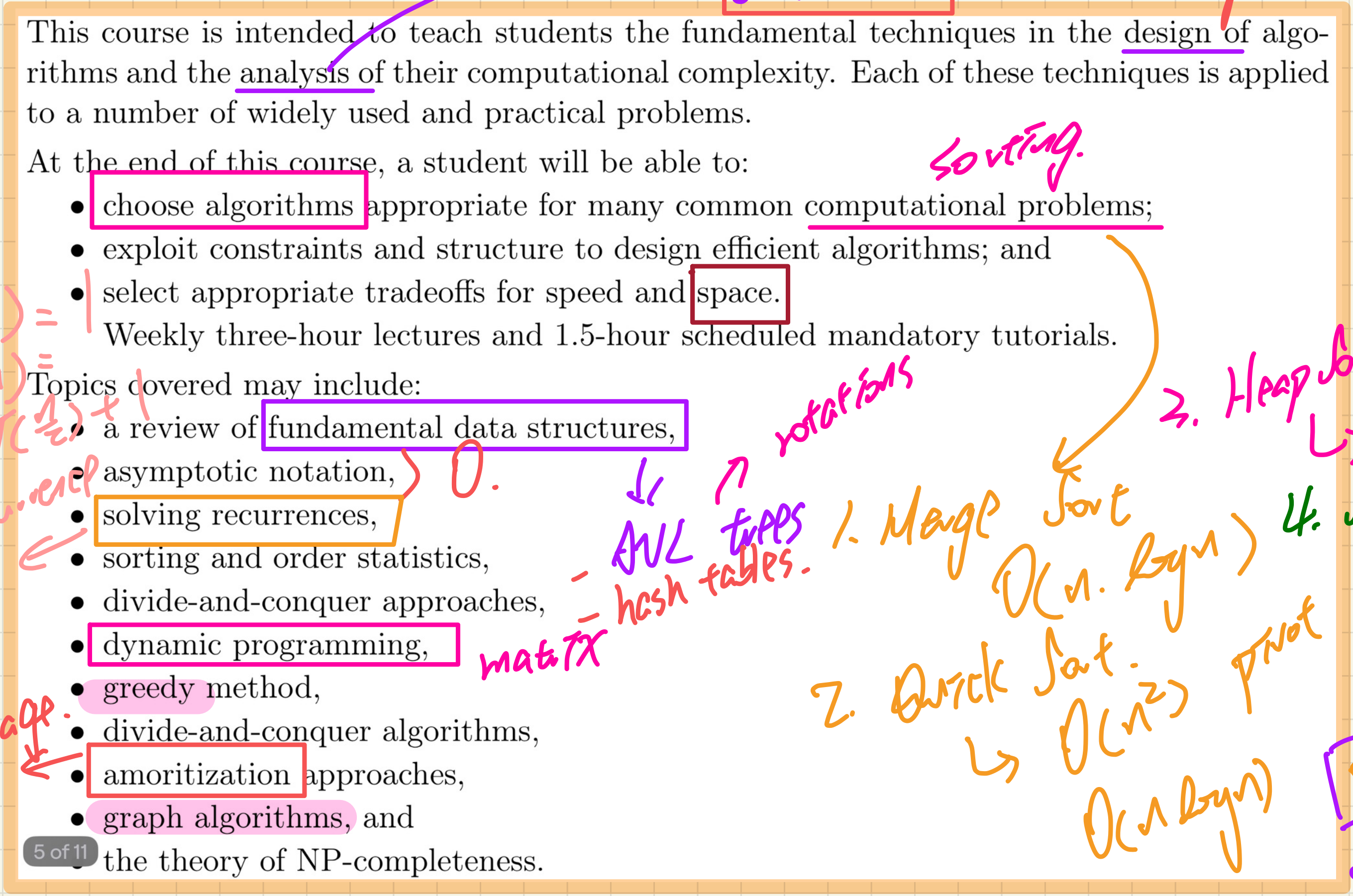
Lecture 1 - Sep 3

Syllabus & Introduction

Professional Engineers: Code of Ethics

Course Descriptions

1. Run time (O, Θ)
2. Correctness → for iterative algorithm → loop invariant.



This course is intended to teach students the fundamental techniques in the design of algorithms and the analysis of their computational complexity. Each of these techniques is applied to a number of widely used and practical problems.

At the end of this course, a student will be able to:

- choose algorithms appropriate for many common computational problems;
- exploit constraints and structure to design efficient algorithms; and
- select appropriate tradeoffs for speed and space.

Weekly three-hour lectures and 1.5-hour scheduled mandatory tutorials.

Topics covered may include:

- a review of fundamental data structures,
- asymptotic notation,
- solving recurrences,
- sorting and order statistics,
- divide-and-conquer approaches,
- dynamic programming,
- greedy method,
- divide-and-conquer algorithms,
- amoritization approaches,
- graph algorithms, and
- the theory of NP-completeness.

5 of 11

sorting.

O.

- AVL trees ↗ rotations
- hash tables.

matrix

$T(1) = 1$

$T(n) = 2 \cdot T(\frac{n}{2}) + 1$

recurrence rel

average

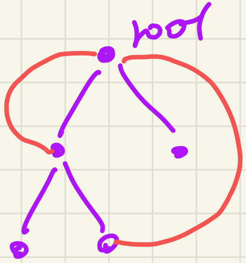
1. Merge sort $O(n \cdot \log n)$
2. Quick sort. $O(n^2)$ pivot ↳ $O(n \log n)$
3. Heap sort ↳ $O(N \cdot \log N)$
4. Selection / Insertion

far from median

pivot ≈ median ↓ efficient alg.

Graphs

1. extension to trees with cycles



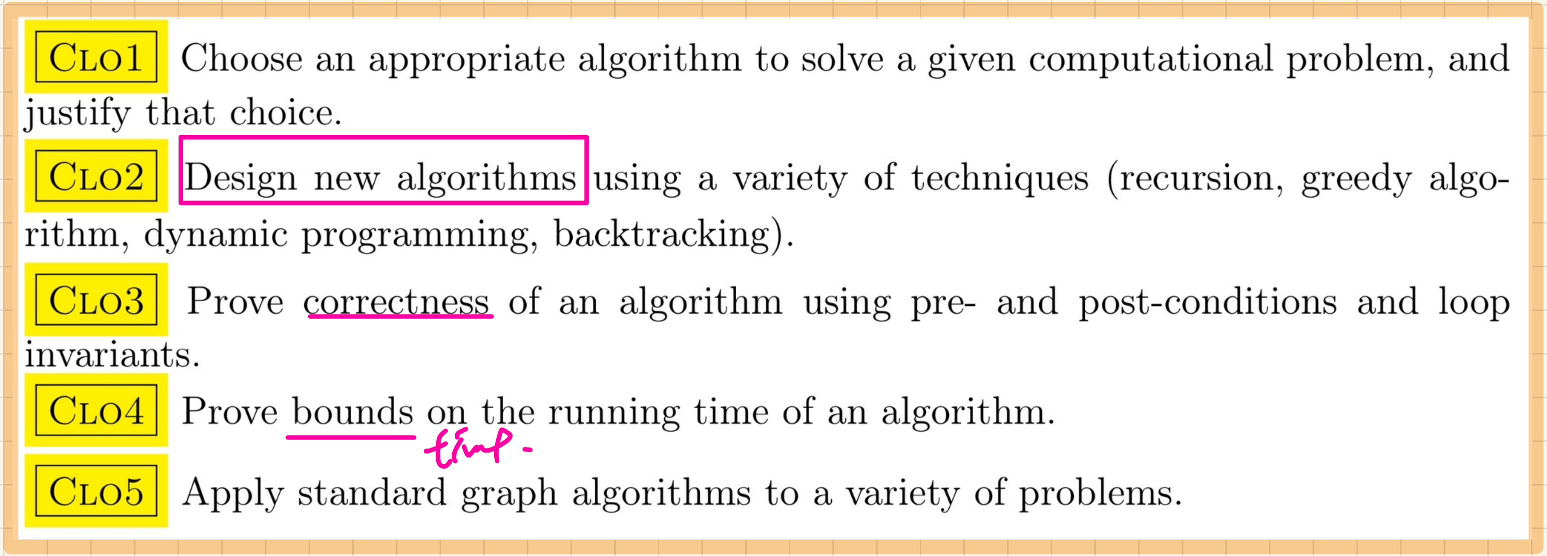
2. Implementations
   - ↳ edge list
   - ↳ adjacency list) ArrayList
   - ↳ adjacency matrix) 2D array.

3. algorithms on graphs (e.g., shortest path; minimum spanning tree; topological sort)

Queue (FIFO)

- ↳ array []
- ↳ resizing strategy
  - ↳ doubling 1000, 2000, 4000, ...
  - ↳ fixed increment 1000, 2000, 3000, ...

Course Learning Outcomes (CLOs)



CLO1 Choose an appropriate algorithm to solve a given computational problem, and justify that choice.

CLO2 Design new algorithms using a variety of techniques (recursion, greedy algorithm, dynamic programming, backtracking).

CLO3 Prove correctness of an algorithm using pre- and post-conditions and loop invariants.

CLO4 Prove bounds on the running time of an algorithm.

time -

CLO5 Apply standard graph algorithms to a variety of problems.

Lecture 2 - Sep 8

Introduction, *DbC*

Motivating Problems
Design by Contract
Clients vs. Suppliers

Announcements/Reminders

- First Class (Syllabus) recording & notes posted
- Today's class: notes template posted
- Exercises:
  - + Tutorial Week 1 (2D arrays)

A Searching Problem

not corresponding to array indices.

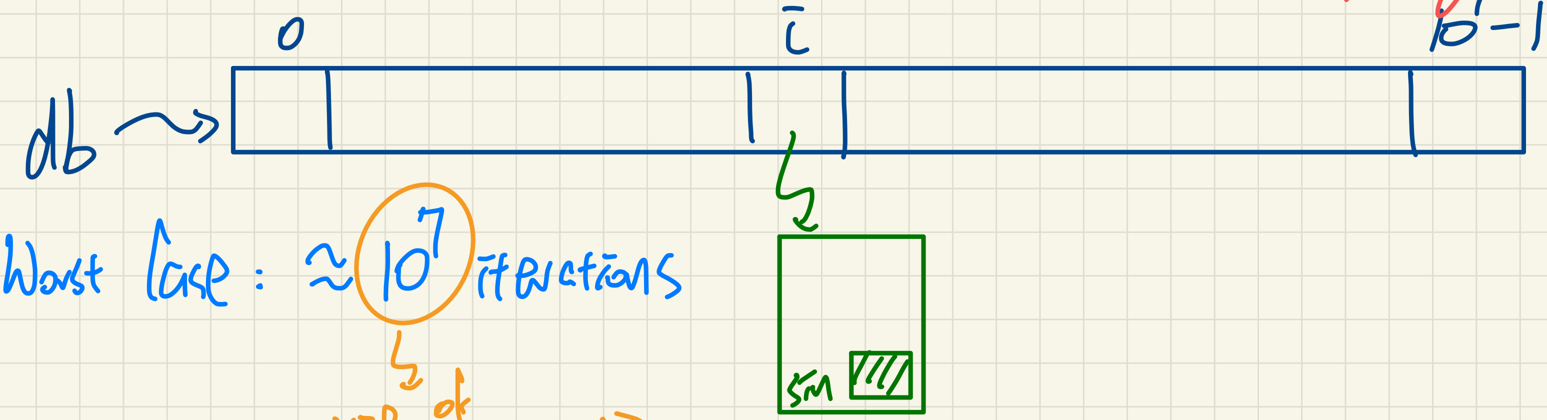
```
ResidentRecord find(int sin) {
  for(int i = 0; i < database.length; i ++) {
    if(database[i].sin == sin) {
      return database[i];
    }
  }
}
```

Hash Table

large data set

↳ collision

↳ linear probing



Worst case : ≈ 10^7 iterations

size of input $O(n)$

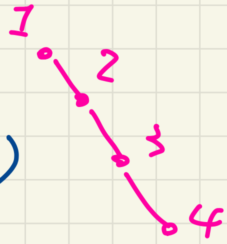
Inappropriate Solution

1. Store all records in an array (unsorted)

$O(n.\log n)$ 2. Sort the array

$O(\log n)$ 3. Binary Search on the array.

↳ less efficient than a linear search!



① Worst case: h is $O(n)$

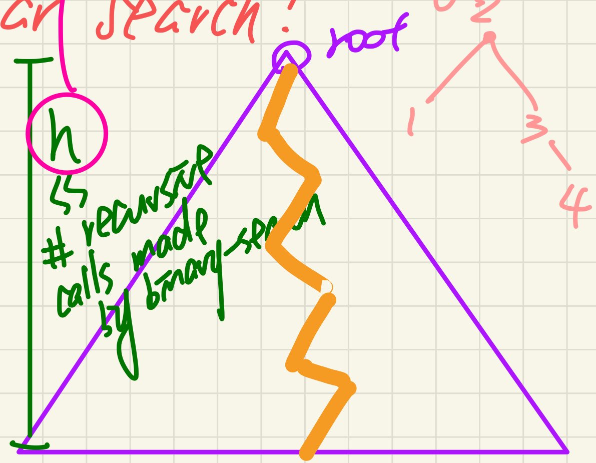
② best case: h is $O(\log n)$

Appropriate Solution (Tree).

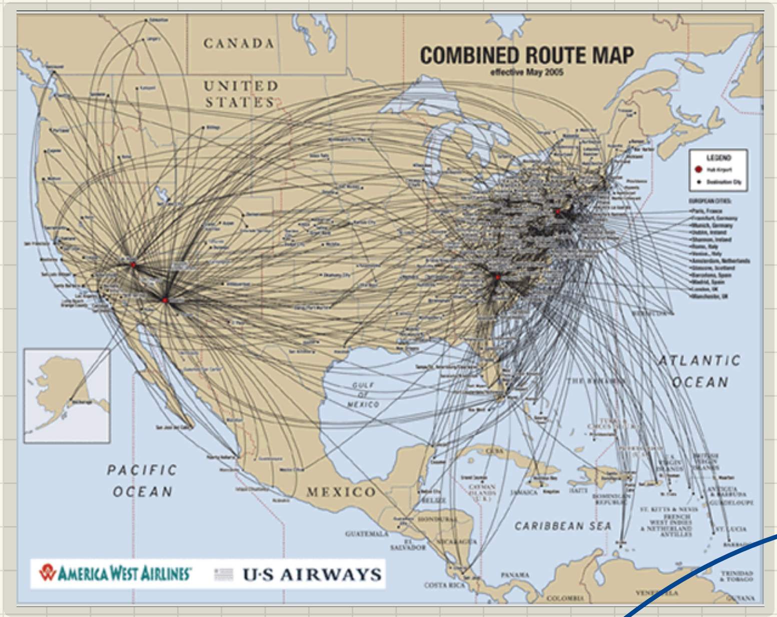
balanced binary search tree (BST)

↓ self-balancing search trees.

1. Heap.
2. AVL trees ↳ rotations



A Routing Problem



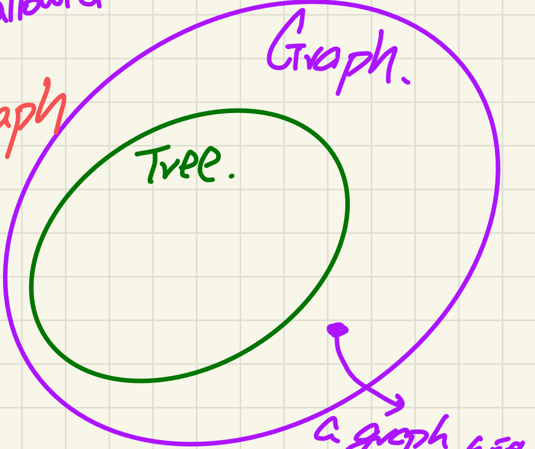
root

cycles ~~not~~ allowed

Tree ⇒ Graph

Graph ⇏ Tree

cycles very common.



a graph with a cycle.

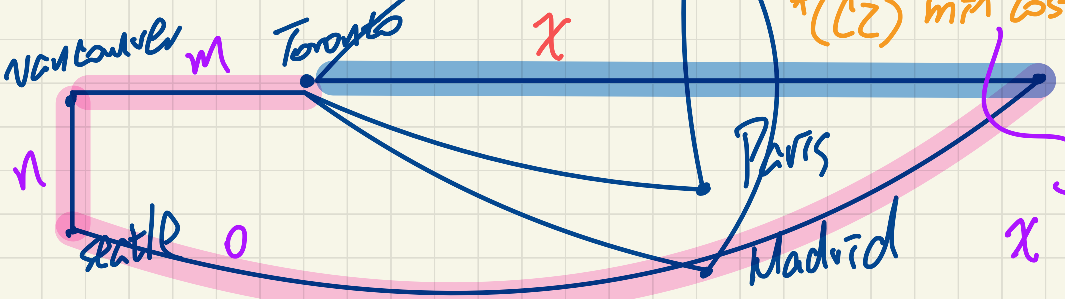
Tree ⊂ Graph

London

"shortest" path

(C1) min # transitions T-S

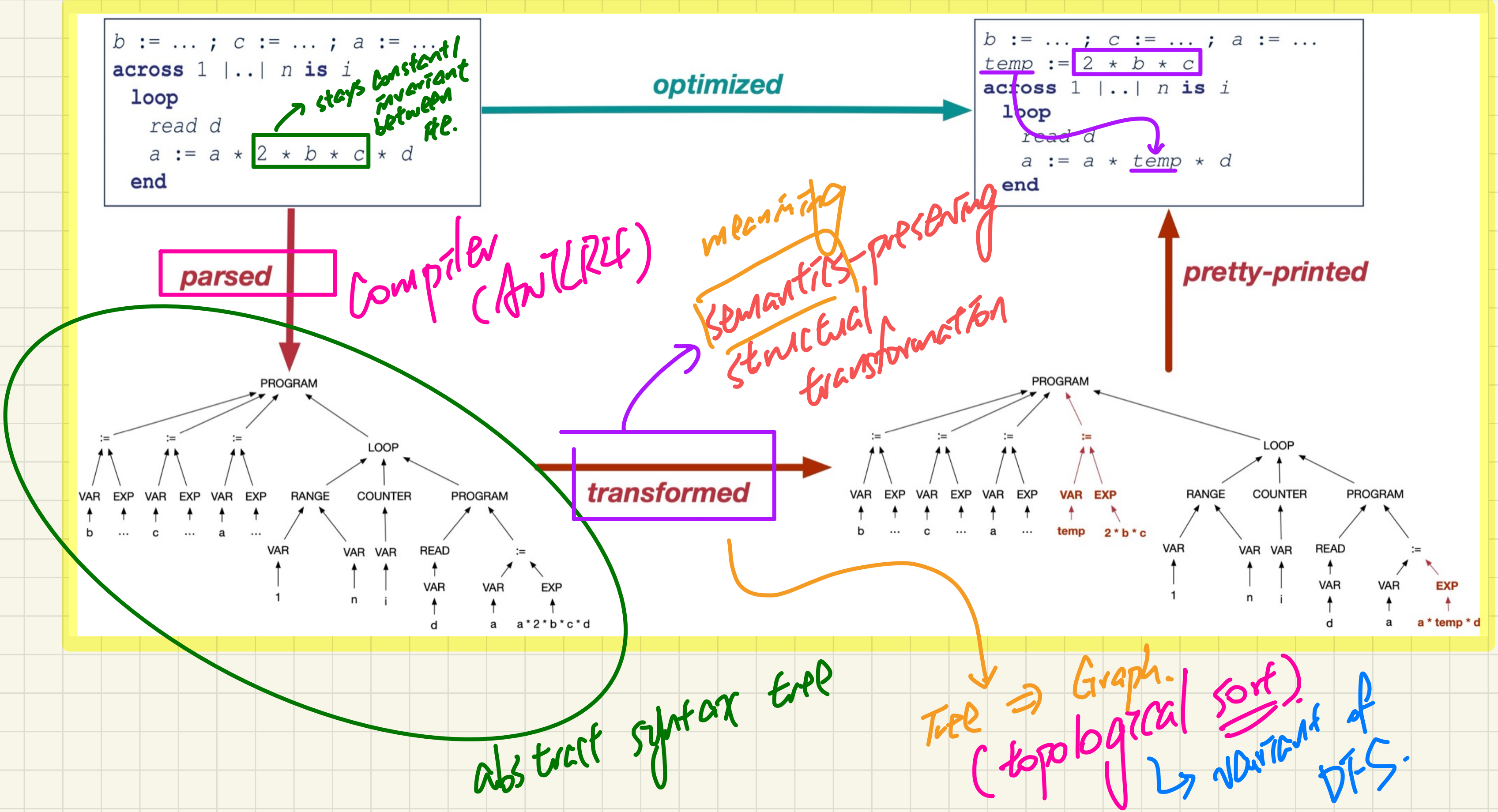
*(C2) min cost



x <? m + n + o

*amend the graph edges with weights

Program Optimization Problem



```
b := ... ; c := ... ; a := ...
across 1 |..| n is i
  loop
    read d
    a := a * 2 * b * c * d
  end
```

stays constant/invariant between ite.

```
b := ... ; c := ... ; a := ...
temp := 2 * b * c
across 1 |..| n is i
  loop
    read d
    a := a * temp * d
  end
```

Compiler (AnTLR4)

meaning

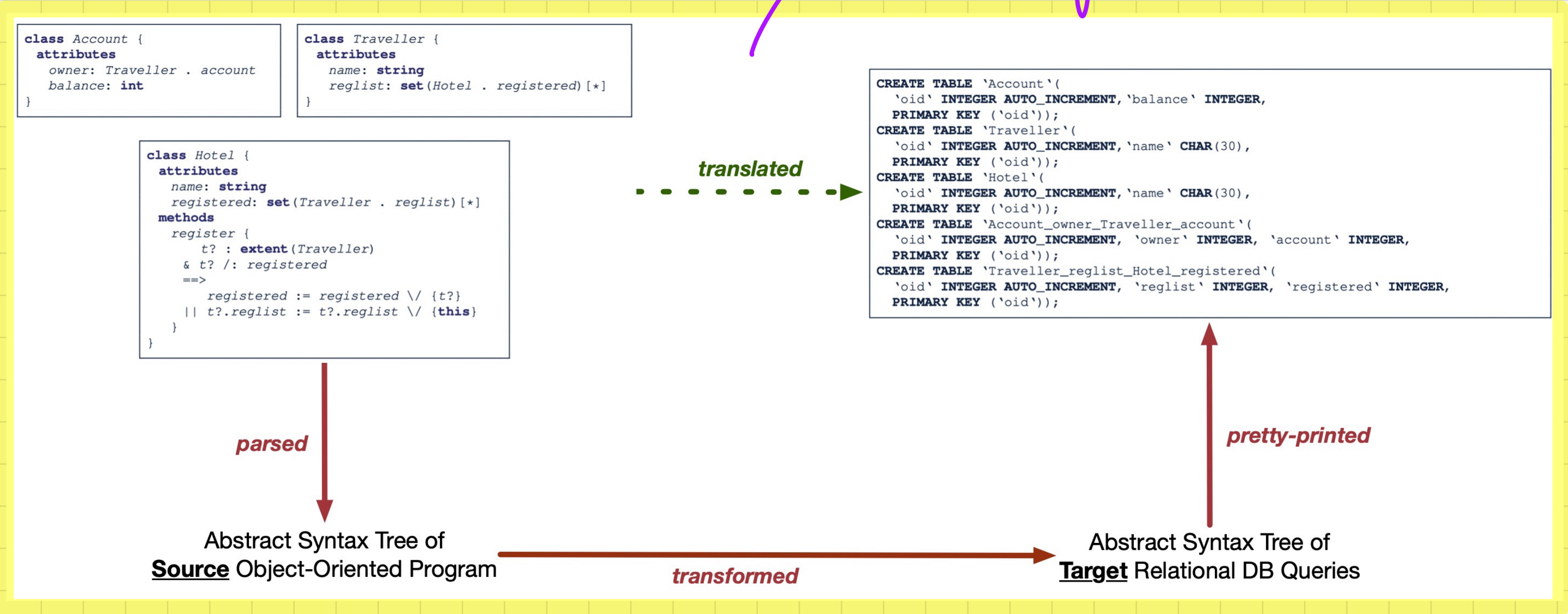
semantics-preserving structural transformation

abstract syntax tree

Tree ⇒ Graph. (topological sort) ↳ variant of DFS.

Program Translation Problem

object-relational bridge.



```
class Account {
  attributes
    owner: Traveller . account
    balance: int
}
```

```
class Traveller {
  attributes
    name: string
    reglist: set(Hotel . registered)[*]
}
```

```
class Hotel {
  attributes
    name: string
    registered: set(Traveller . reglist)[*]
  methods
    register {
        t? : extent(Traveller)
      & t? /: registered
      ==>
          registered := registered \/ {t?}
      || t?.reglist := t?.reglist \/ {this}
    }
}
```

```
CREATE TABLE ‘Account‘(
  ‘oid‘ INTEGER AUTO_INCREMENT,‘balance‘ INTEGER,
  PRIMARY KEY (‘oid‘));
CREATE TABLE ‘Traveller‘(
  ‘oid‘ INTEGER AUTO_INCREMENT,‘name‘ CHAR(30),
  PRIMARY KEY (‘oid‘));
CREATE TABLE ‘Hotel‘(
  ‘oid‘ INTEGER AUTO_INCREMENT,‘name‘ CHAR(30),
  PRIMARY KEY (‘oid‘));
CREATE TABLE ‘Account_owner_Traveller_account‘(
  ‘oid‘ INTEGER AUTO_INCREMENT, ‘owner‘ INTEGER, ‘account‘ INTEGER,
  PRIMARY KEY (‘oid‘));
CREATE TABLE ‘Traveller_reglist_Hotel_registered‘(
  ‘oid‘ INTEGER AUTO_INCREMENT, ‘reglist‘ INTEGER, ‘registered‘ INTEGER,
  PRIMARY KEY (‘oid‘));
```

translated
parsed
pretty-printed
Abstract Syntax Tree of **Source** Object-Oriented Program
transformed
Abstract Syntax Tree of **Target** Relational DB Queries

Design by Contract (DbC): Client vs. Supplier

2030: caller vs. callee

3101/3311: client vs. supplier

client ↳ e.g. microwave user

supplier ↳ e.g. microwave ↳ heat

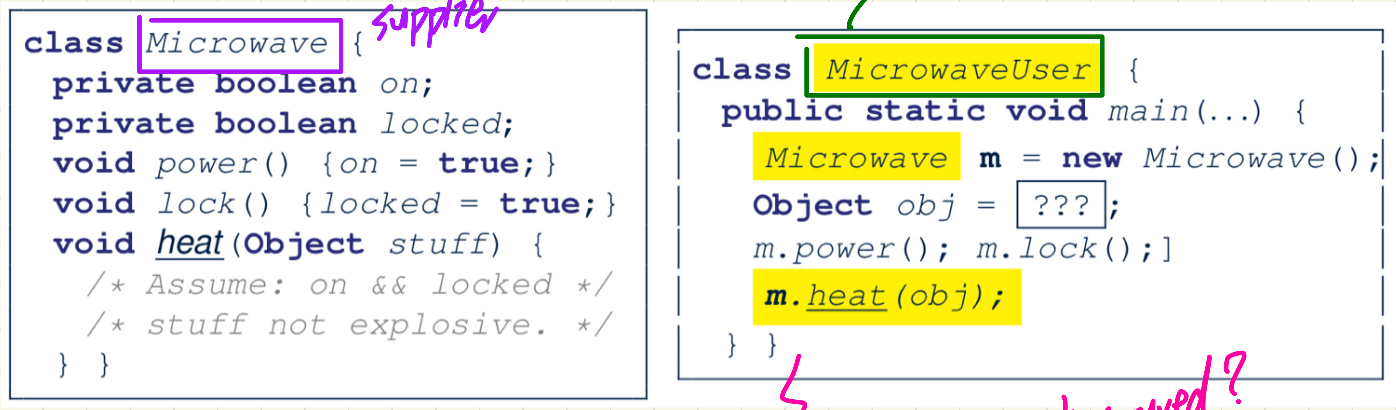
	benefits	obligations
client	e.g. (heat lunch box) obtain service	e.g. on, locked, non-explosive follow instructions
supplier	e.g. no need to create a magical microwave to work without power on. instructions followed	e.g. heat lunch box (given that instructions followed) provide service

binary search

`bSearch(int[] input, int k)`

	benefits	obligations
client/ user	find the item quickly.	input array sorted.
supplier/ implementor	only need to deal with sorted array.	recursive imp. done correctly.

Client vs. Supplier in OOP



```
class Microwave {
  private boolean on;
  private boolean locked;
  void power()  {on = true;}
  void lock()  {locked = true;}
  void heat(Object stuff)  {
    /* Assume: on && locked */
    /* stuff not explosive. */
  }  }
```

supplier

```
class MicrowaveUser  {
  public static void main(...)  {
    Microwave m = new Microwave();
    Object obj = ???;
    m.power(); m.lock();]
    m.heat(obj);
  } }
```

client

Is the contract honoured?

Lecture 3 - Sep 10

DbC, Modularity, ADTs, Asymptotic Analysis

DbC: Honouring the Contract

Modularity, ADTs

Asymptotic Upper Bound (Big-O)

Announcements/Reminders

- First Class (Syllabus) recording & notes posted
- Today's class: notes template posted
- Exercises:
  - + Tutorial Week 1 (2D arrays)
  - + Tutorial Week 2 this Friday (in person)

↳ 2D arrays

↳ exercises on big-O.

DbC: Contract Honoured?

clients vs. suppliers

supplier class

```
class Microwave {
  private boolean on;
  private boolean locked;
  void power() {on = true;}
  void lock() {locked = true;}
  void heat(Object stuff) {
    /* Assume: on && locked */
    /* stuff not explosive. */
  } }
```

supplier method

client class

```
class MicrowaveUser {
  public static void main(...) {
    Microwave m = new Microwave();
    Object obj = ???;
    m.power(); m.lock();]
    m.heat(obj);
  } }
```

① m.power(); m.lock(); ✓

② m.heat(obj);

this client may not honour the contract for heat.

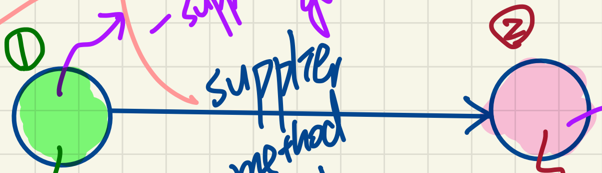
not necessarily honouring the contract : obj may be explosive

precond. of heat method violated.

method call ↳ point where client uses the supplier's service.

precondition

- client's obligation satisfied
- supplier's benefits guaranteed



supplier method call

pre-state before

post-state after

postcondition ↓ termination

client's benefit guaranteed

supplier's obligations fulfilled

partial correctness

↳ 1. assume alg. terminates

2. output is as expected

total correctness

↳ 1. termination guaranteed (should be proved) ↳ loop variant

2. output is as expected (loop invariant).

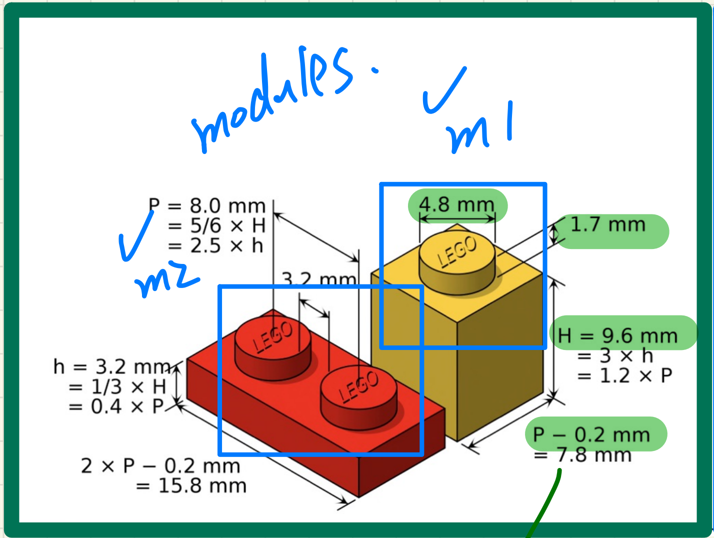
"good" design

↳ well-specified

precondition and postcondition

ob. of client ob. of supplier.

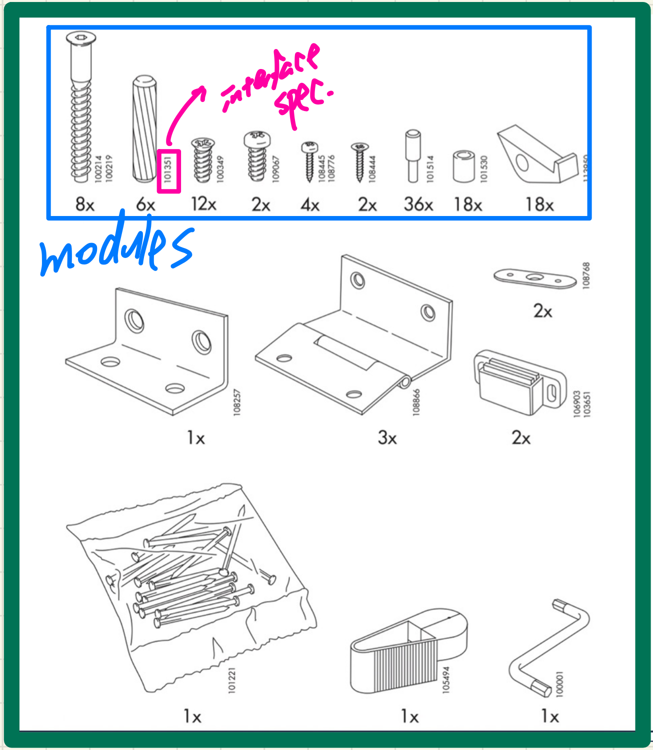
Modularity: Childhood Activities



Interface specifications

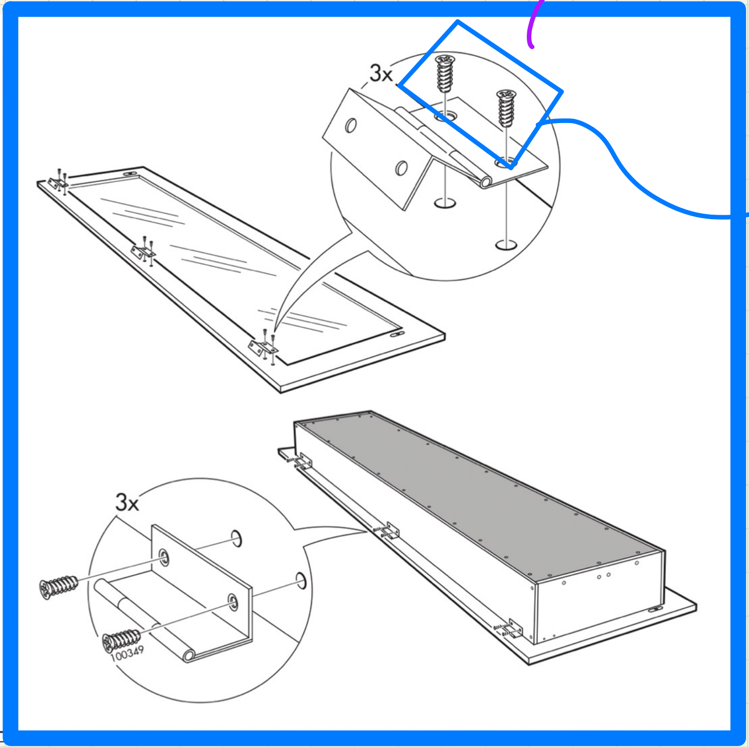
assembly of modules.
(reusable)

Modularity: Daily Constructions



interface spec.

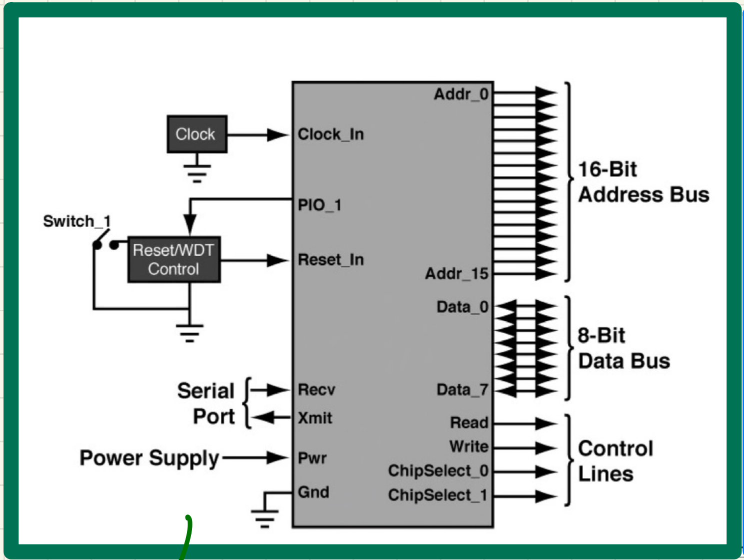
modules



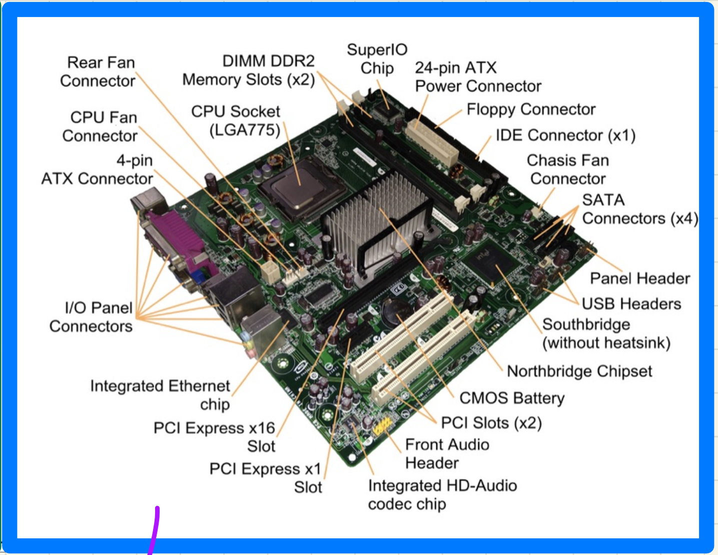
assembly

reusable modules

Modularity: Computer Architectures

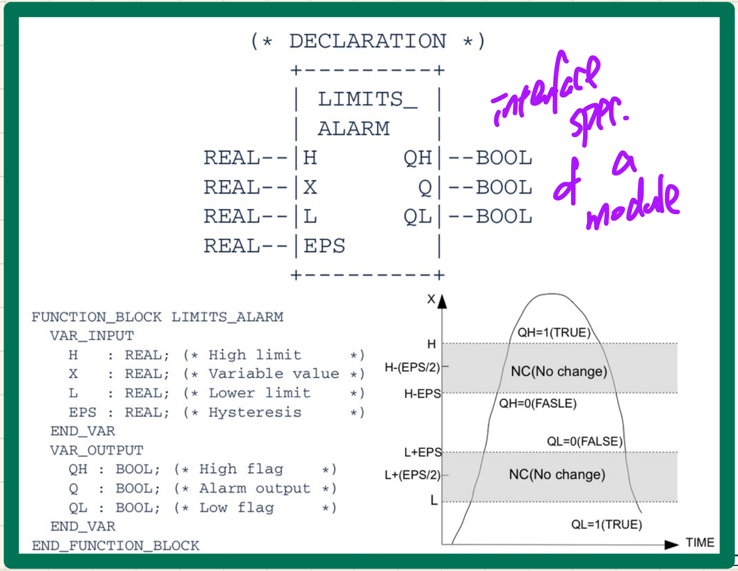


specification of module



assembly

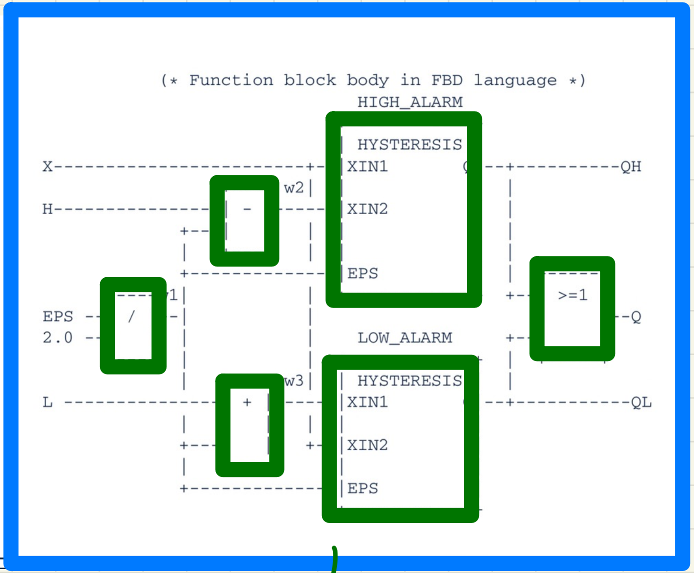
Modularity: System Developments



```
(* DECLARATION *)
          +----------+
          | LIMITS_  |
          | ALARM    |
   REAL--|H       QH|--BOOL
   REAL--|X        Q|--BOOL
   REAL--|L       QL|--BOOL
   REAL--|EPS       |
          +----------+
```

```
FUNCTION_BLOCK LIMITS_ALARM
  VAR_INPUT
    H   : REAL; (* High limit     *)
    X   : REAL; (* Variable value *)
    L   : REAL; (* Lower limit    *)
    EPS : REAL; (* Hysteresis     *)
  END_VAR
  VAR_OUTPUT
    QH : BOOL; (* High flag    *)
    Q  : BOOL; (* Alarm output *)
    QL : BOOL; (* Low flag     *)
  END_VAR
END_FUNCTION_BLOCK
```

interface spec. of a module



(* Function block body in FBD language *)

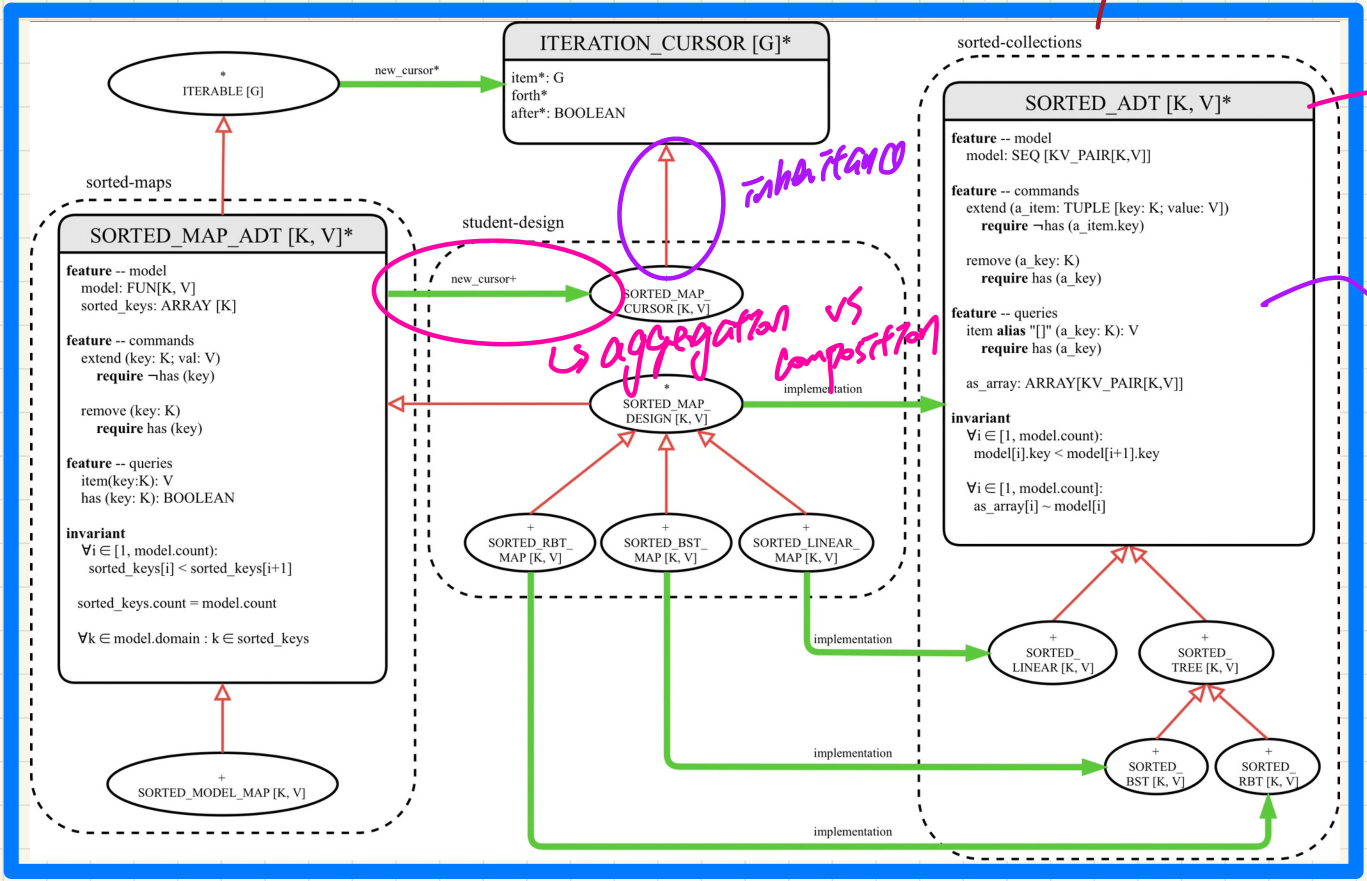
↳ assembly.

function blocks
↳ PLC (Programmable Logic Controlle)

Modularity: Software Design

Want to avoid

↳ superman module



ITERATION_CURSOR [G]*

item*: G
forth*
after*: BOOLEAN

* ITERABLE [G]

new_cursor*

sorted-maps

SORTED_MAP_ADT [K, V]*

feature -- model
model: FUN[K, V]
sorted_keys: ARRAY [K]

feature -- commands
extend (key: K; val: V)
require ¬has (key)

remove (key: K)
require has (key)

feature -- queries
item(key:K): V
has (key: K): BOOLEAN

invariant
∀i ∈ [1, model.count):
sorted_keys[i] < sorted_keys[i+1]

sorted_keys.count = model.count

∀k ∈ model.domain : k ∈ sorted_keys

+ SORTED_MODEL_MAP [K, V]

student-design

new_cursor+

SORTED_MAP_ CURSOR [K, V]

* SORTED_MAP_ DESIGN [K, V]

implementation

+ SORTED_RBT_ MAP [K, V]

+ SORTED_BST_ MAP [K, V]

+ SORTED_LINEAR_ MAP [K, V]

implementation

implementation

implementation

sorted-collections

SORTED_ADT [K, V]*

feature -- model
model: SEQ [KV_PAIR[K,V]]

feature -- commands
extend (a_item: TUPLE [key: K; value: V])
require ¬has (a_item.key)

remove (a_key: K)
require has (a_key)

feature -- queries
item **alias** "[]" (a_key: K): V
require has (a_key)

as_array: ARRAY[KV_PAIR[K,V]]

invariant
∀i ∈ [1, model.count):
model[i].key < model[i+1].key

∀i ∈ [1, model.count]:
as_array[i] ~ model[i]

+ SORTED_ LINEAR [K, V]

+ SORTED_ TREE [K, V]

+ SORTED_ BST [K, V]

+ SORTED_ RBT [K, V]

inheritance

↳ aggregation vs composition

↳ new module for new function.

maintain only the relevant module necessary.

abstract data type

module ≈ ADT

↳ 1. list of operations

2. for each operation

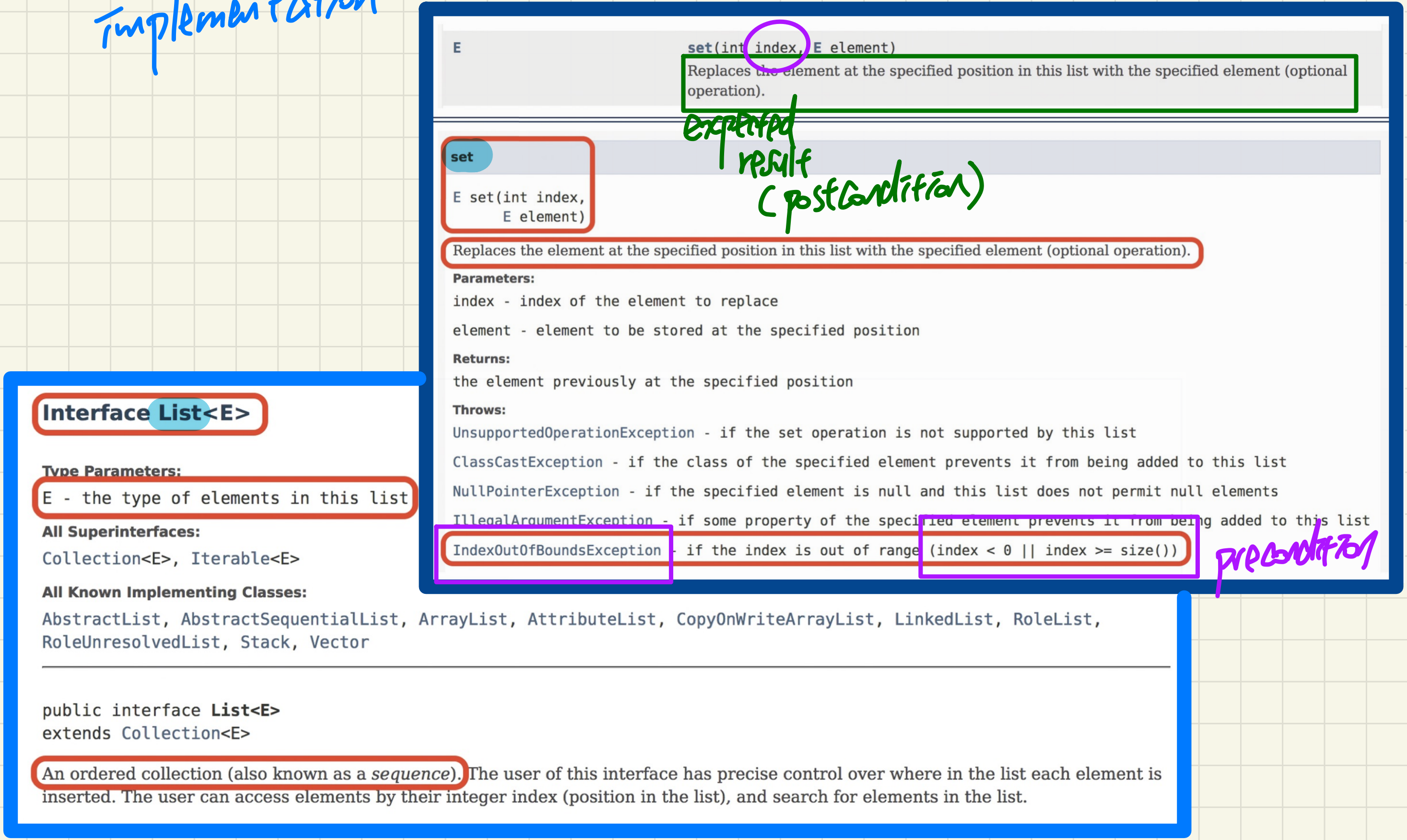
↳ precondition

↳ postcondition.

Java Classes: Abstract Data Types?

design

Implementation



E set(int index, E element)
Replaces the element at the specified position in this list with the specified element (optional operation).

expected result (postcondition)

set

```
E set(int index,
      E element)
```

Replaces the element at the specified position in this list with the specified element (optional operation).

Parameters:

`index - index of the element to replace`

`element - element to be stored at the specified position`

Returns:

`the element previously at the specified position`

Throws:

`UnsupportedOperationException - if the set operation is not supported by this list`

`ClassCastException - if the class of the specified element prevents it from being added to this list`

`NullPointerException - if the specified element is null and this list does not permit null elements`

`IllegalArgumentException - if some property of the specified element prevents it from being added to this list`

`IndexOutOfBoundsException - if the index is out of range (index < 0 || index >= size())`

precondition

Interface List<E>

Type Parameters:

`E - the type of elements in this list`

All Superinterfaces:

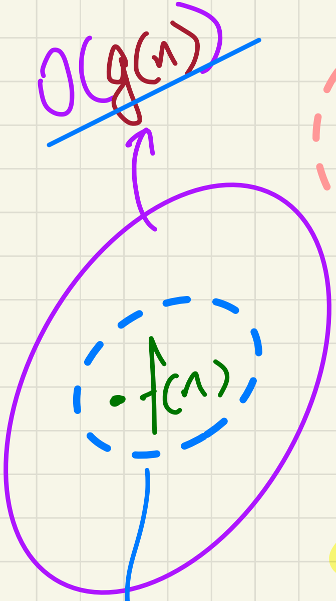
`Collection<E>, Iterable<E>`

All Known Implementing Classes:

`AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector`

```
public interface List<E>
extends Collection<E>
```

An ordered collection (also known as a *sequence*). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.



$f(n)$: Running Time (RT) function of some algorithm.

e.g. $f(\underset{\text{size of input}}{n}) = 7n - 2$

$g(n)$: refence function

e.g. $\boxed{g(n)} = n$ → further manipulated by some multiplicative constant c.

$c \cdot g(n)$

starting from some $n = \underline{\underline{n_0}}$

$f(n) \le c \cdot g(n)$ u.b.e.

$\boxed{f(n)}$ being in the family that it means can somehow be upper-bounded by $\boxed{g(n)}$

change the slope of $g(n)$.

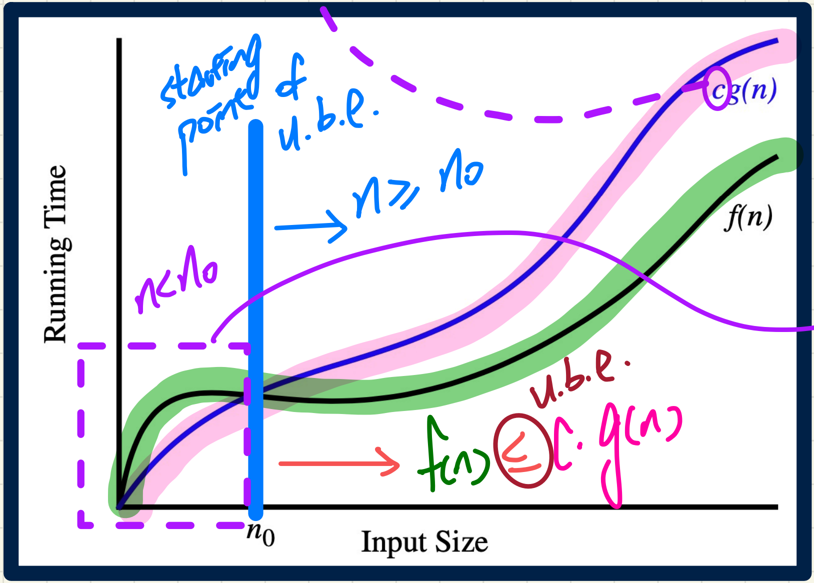
Asymptotic Upper Bound: Big-O

$f(n) \in O(g(n))$ if there are:

- A real constant $c > 0$ slope
- An integer constant $n_0 \geq 1$ starting point of u.b.e.

such that:

$$f(n) \leq c \cdot g(n) \quad \text{for } n \geq n_0$$



Example:

$f(n) = 8n + 5$

$g(n) = n$

Prove:

$f(n)$ is $O(\ g(n)\ )$

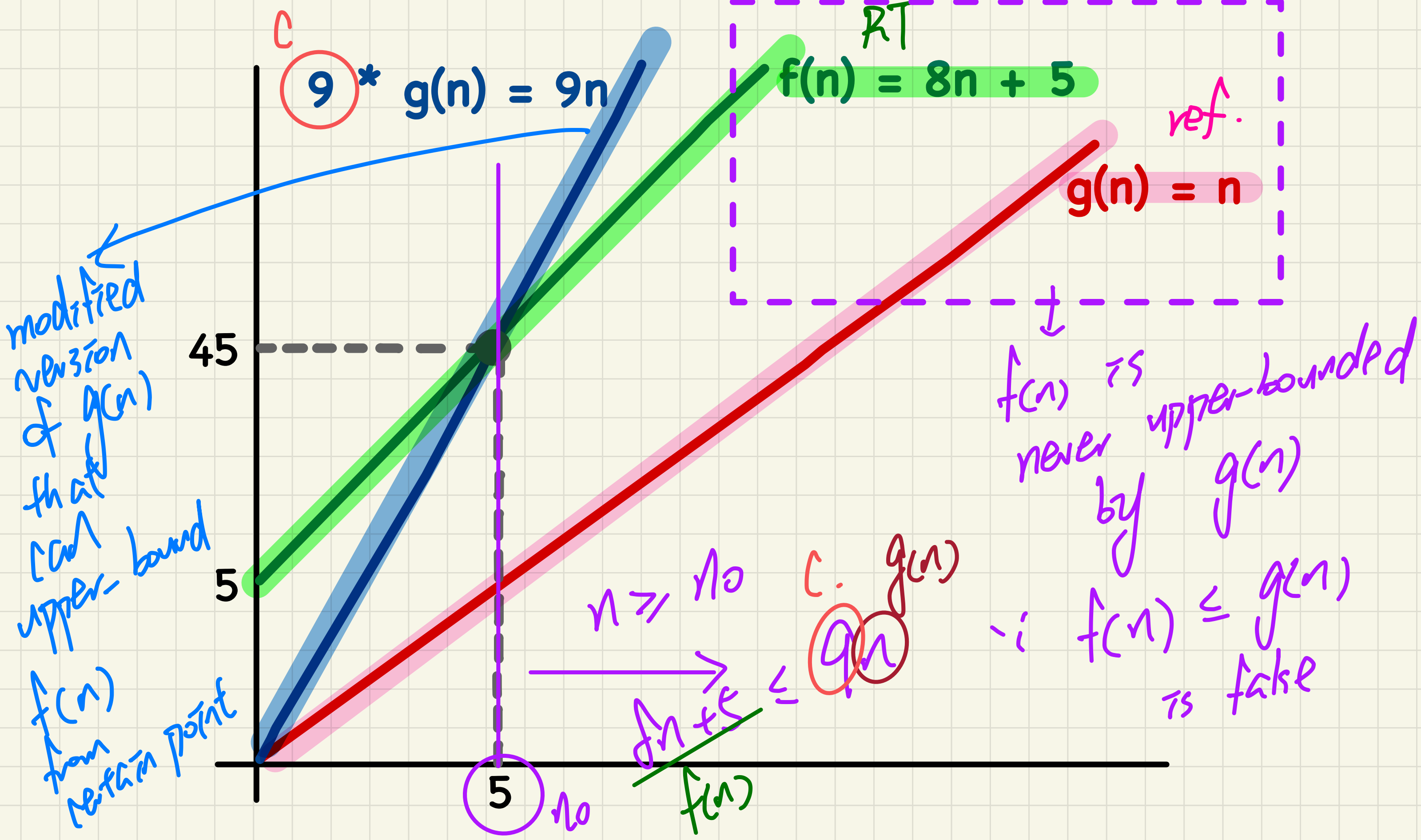
Choose c = 9.

What about n0?

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

↳ no u.b.e.

Asymptotic Upper Bound: Example



modified version of g(n) that can upper-bound f(n) from certain point

f(n) is never upper-bounded by g(n)

$n \geq n_0$

$8n + 5 \leq 9n$

∴ $f(n) \leq g(n)$ is false

Tutorials - Week 2 - Sep 12

Utilities using 2D-Arrays, Asymptotic Analysis

Row with Maximum Sum, isRectangle
Proving Asymptotic Upper Bounds
Deriving Asymptotic Upper Bounds

2D Array Algorithm: Row with Max Sum

-, 0, +

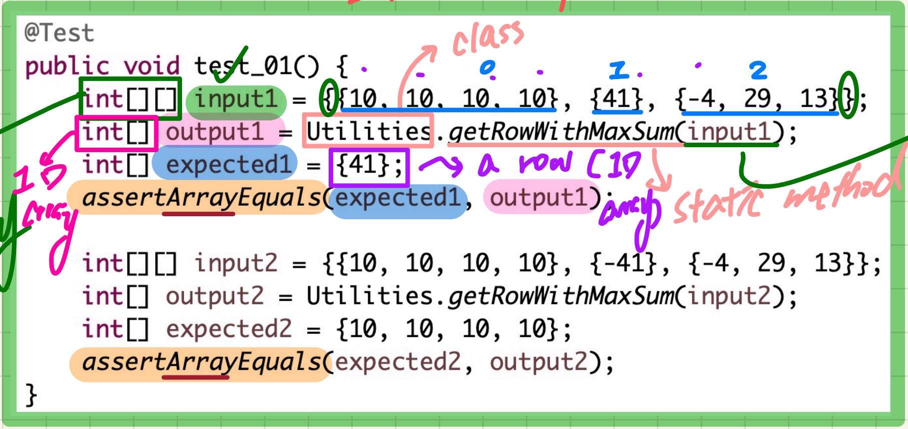
[][]

Problem: Given a 2D array a of integers, find out the row (i.e., a 1D array) with the maximum sum.

[ ]

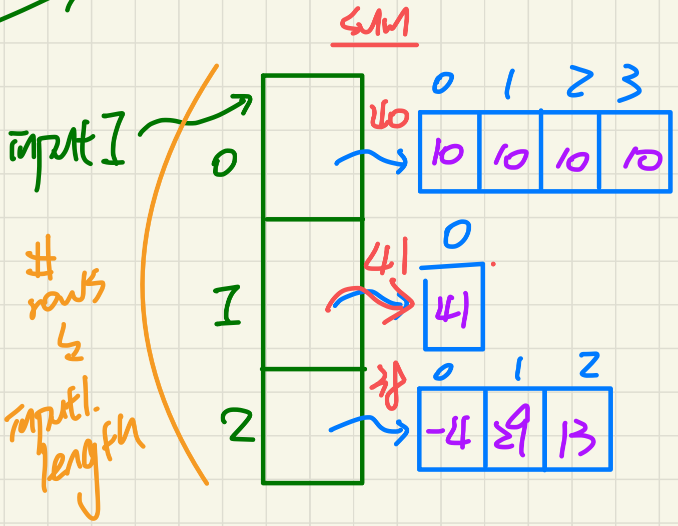
Assume: a is not empty, the row with max sum is unique.

↳ how to express in Java?



```
@Test
public void test_01() {
    int[][] input1 = {{10, 10, 10, 10}, {41}, {-4, 29, 13}};
    int[] output1 = Utilities.getRowWithMaxSum(input1);
    int[] expected1 = {41};
    assertArrayEquals(expected1, output1);

    int[][] input2 = {{10, 10, 10, 10}, {-41}, {-4, 29, 13}};
    int[] output2 = Utilities.getRowWithMaxSum(input2);
    int[] expected2 = {10, 10, 10, 10};
    assertArrayEquals(expected2, output2);
}
```

2D-array



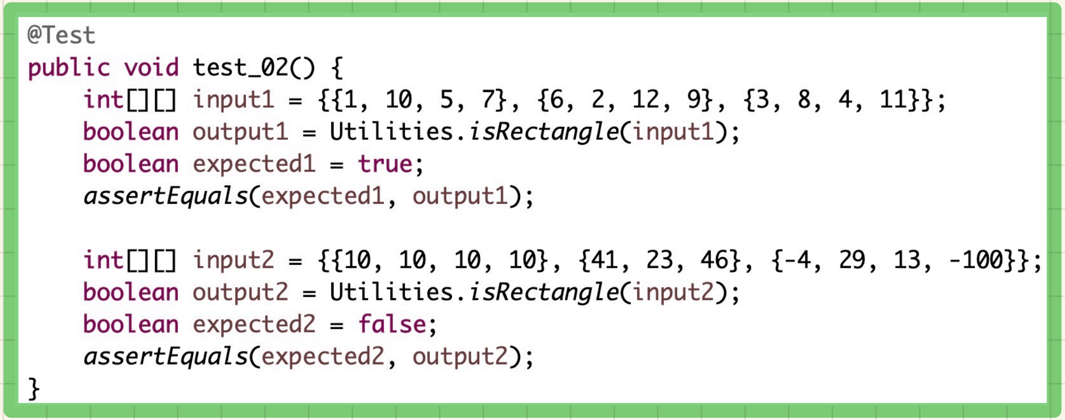
Last row : input1[input1.length - 1]

Cell at row 1, column 0 : input1[1][0]
index index
41

2D Array Algorithm: Is a 2D Array a Rectangle?

Problem: Given a 2D array a of integers, determine whether or not a is a rectangle.

Assume: a is not empty.



```
@Test
public void test_02() {
    int[][] input1 = {{1, 10, 5, 7}, {6, 2, 12, 9}, {3, 8, 4, 11}};
    boolean output1 = Utilities.isRectangle(input1);
    boolean expected1 = true;
    assertEquals(expected1, output1);

    int[][] input2 = {{10, 10, 10, 10}, {41, 23, 46}, {-4, 29, 13, -100}};
    boolean output2 = Utilities.isRectangle(input2);
    boolean expected2 = false;
    assertEquals(expected2, output2);
}
```

Proving f(n) is O(g(n))

If $f(n)$ is a polynomial of degree d, i.e.,

$$f(n) = a_0 \cdot n^0 + a_1 \cdot n^1 + \cdots + a_d \cdot n^d$$

and $a_0, a_1, \ldots, a_d$ are integers (i.e., negative, zero, or positive), then $f(n)$ *is* $O(n^d)$.

We prove by choosing

$$\begin{aligned} c &= |a_0| + |a_1| + \cdots + |a_d| \\ n_0 &= 1 \end{aligned}$$

Upper-bound effect: $n_0 = 1$?

$$[f(1) \leq (|a_0| + |a_1| + \cdots + |a_d|) \cdot 1^d]$$

Upper-bound effect holds?

$$[f(n) \leq (|a_0| + |a_1| + \cdots + |a_d|) \cdot n^d]$$

Exercise: **Prove** $f(n) = 5n^4 - 3n^3 + 2n^2 - 4n + 1$ is $O(\underbrace{n^4}_{g(n)})$

To prove, choose c and n_0.

$$c = |5| + |-3| + |2| + |-4| + 1 = 15$$

$$n_0 = 1$$

Verify

upper-bound effect starts at $n_0 = 1$

$$f \le c \cdot g$$

$$5 - 3 + 2 - 4 + 1 = 1 \leftarrow \underbrace{\boxed{f(1)}}_{1} \overset{\checkmark}{\le} \underbrace{15 \cdot 1^4}_{15}$$

u.b.e. verified at $n_0 = 1$

Asymptotic Upper Bounds: Example (1)

Given $f(n) = 5n^2 + 3n \cdot \log n + 2n + 5$:

(1) What is f(n)'s **most accurate** **asymptotic upper bound.**

(2) **Prove** your claim.

Asymptotic Upper Bounds: Example (2)

Given $f(n) = 20n^3 + 10n \cdot \log n + 5$:

(1) What is f(n)'s **most accurate** **asymptotic upper bound.**

(2) **Prove** your claim.

Asymptotic Upper Bounds: Example (3)

Given $f(n) = 3 \cdot \boxed{\log n} + 2$:

higher power (faster growth) than n^0

(1) What is f(n)'s **most accurate** **asymptotic upper bound**.

(2) **Prove** your claim.

(1) $O(\boxed{\log n})$

$g(n)$

Verify:

$\underline{f(1)} \le c \cdot g(1)$

$\underline{3 \cdot \log 1 + 2} \le 5 \cdot \log n$

$3 \cdot 0 + 2 \le 5 \cdot 0$

↳ $2 \le 0$ ✗

no u.b.e.

(2) Choose: $c = |3| + |2| = 5$

$n_0 \overset{?}{=} * 2$

↳ verify

↳ exercise

Asymptotic Upper Bounds: Example (4)

Given f(n) = 2^{n+2}:

(1) What is f(n)'s **most accurate** **asymptotic upper bound**.

(2) **Prove** your claim.

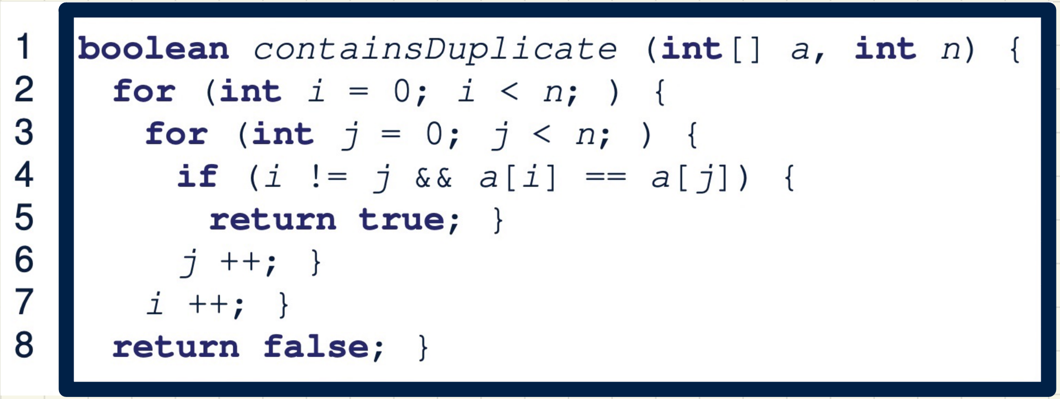
Asymptotic Upper Bounds: Example (5)

Given $f(n) = 2n + 100 \cdot \log n$:

(1) What is f(n)'s **most accurate** **asymptotic upper bound.**

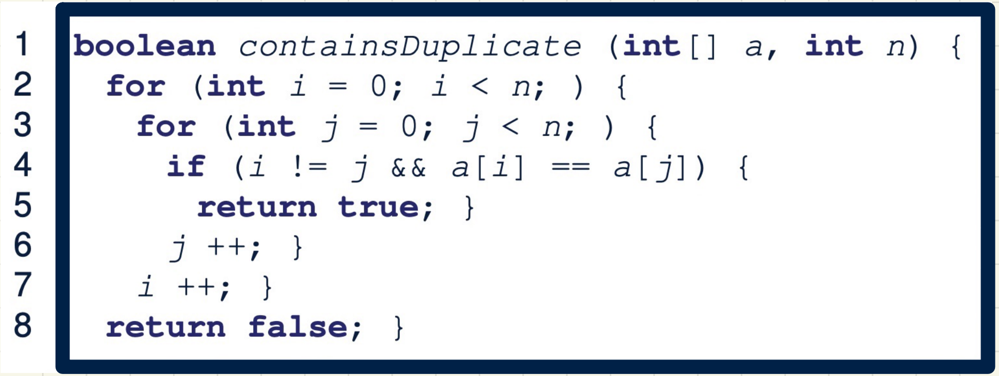
(2) **Prove** your claim.

Determining the Asymptotic Upper Bound (1.1)

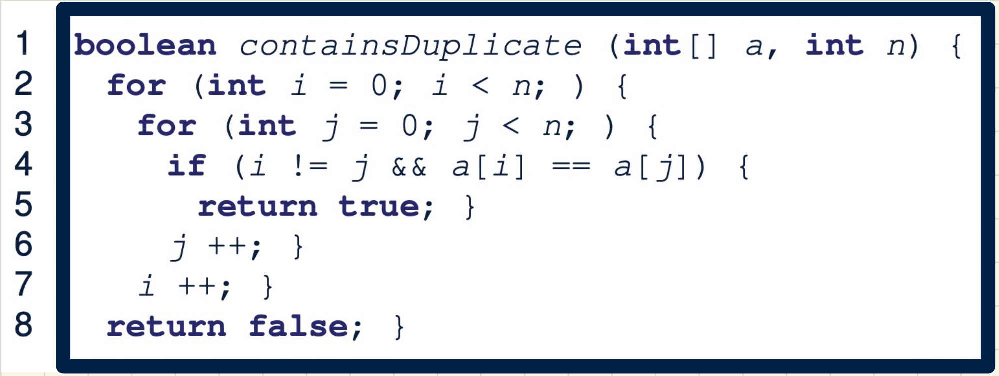


```
boolean containsDuplicate (int[] a, int n) {
  for (int i = 0; i < n; ) {
    for (int j = 0; j < n; ) {
      if (i != j && a[i] == a[j]) {
        return true; }
      j ++; }
    i ++; }
  return false; }
```

Determining the Asymptotic Upper Bound (1.2)

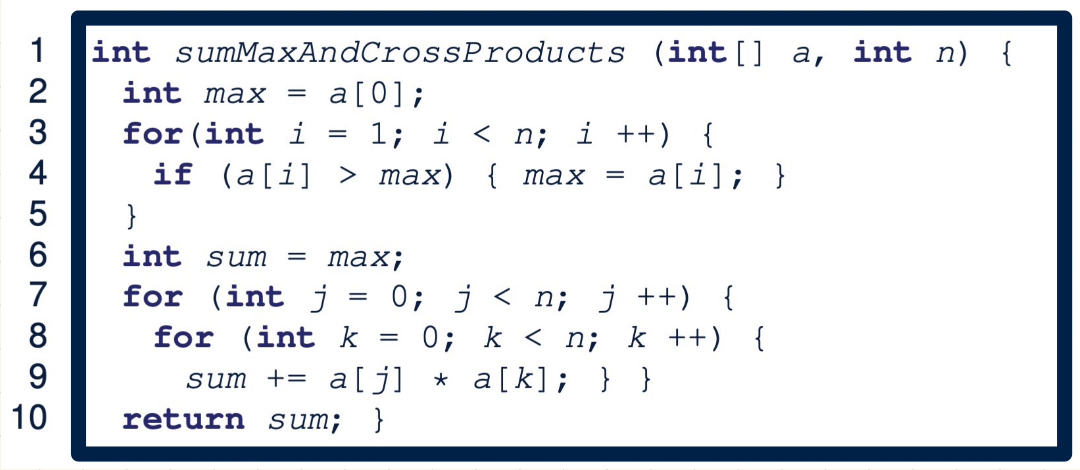


```
boolean containsDuplicate (int[] a, int n) {
  for (int i = 0; i < n; ) {
    for (int j = 0; j < n; ) {
      if (i != j && a[i] == a[j]) {
        return true; }
      j ++; }
    i ++; }
  return false; }
```



```
boolean containsDuplicate (int[] a, int n) {
  for (int i = 0; i < n; ) {
    for (int j = 0; j < n; ) {
      if (i != j && a[i] == a[j]) {
        return true; }
      j ++; }
    i ++; }
  return false; }
```

Determining the Asymptotic Upper Bound (2)



```
int sumMaxAndCrossProducts (int[] a, int n) {
  int max = a[0];
  for(int i = 1; i < n; i ++) {
    if (a[i] > max) { max = a[i]; }
  }
  int sum = max;
  for (int j = 0; j < n; j ++) {
    for (int k = 0; k < n; k ++) {
      sum += a[j] * a[k]; } }
  return sum; }
```

Lecture 4 - Sep 15

Asymptotic Analysis

Defining Big-O using Predicate Logic
Deriving Big-O: Triangular Sum
Dynamic Arrays: Constant Increments

Announcements/Reminders

- First Class (Syllabus) recording & notes posted
- Today's class: notes template posted
- Exercises:
  - + Tutorial Week 1 (2D arrays)
  - + Tutorial Week 2 (2D arrays, Proving Big-O)

$\mathbb{N} = \{0, 1, 2, ---\}$

Asymptotic Upper Bound (Big-O): Alternative Formulation

$\Rightarrow$: logical implication

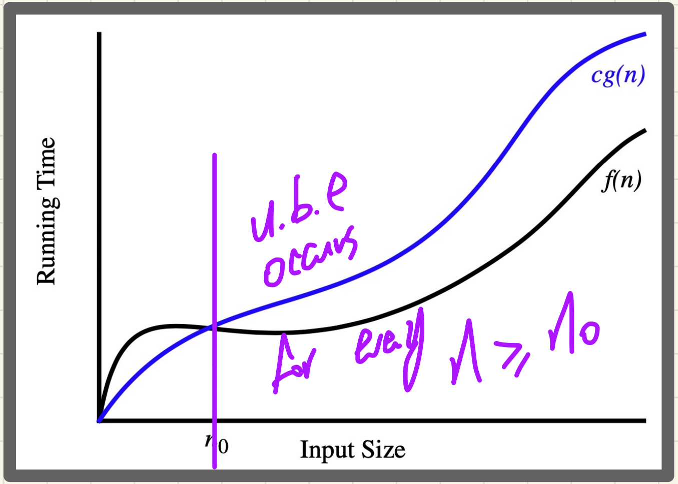
Known:

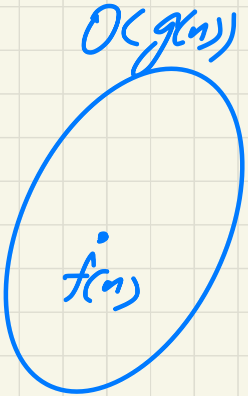
$f(n) \in O(g(n))$ if there are:

- ○ A real *constant* $c > 0$
- ○ An integer *constant* $n_0 \geq 1$

such that:

$$f(n) \leq c \cdot g(n) \quad \text{for } n \geq n_0$$





Q. Formulate the definition of "f(n) is order of O(g(n))" using **logical** operator(s): ¬, ∧, ∨, ⇒, ∀, ∃

input size

$c \in \mathbb{Z}$ $\quad n \in \mathbb{N}$

$$f(n) \in O(g(n)) \iff \exists c, n_0 \bullet c > 0 \wedge n_0 \geq 1 \wedge (\forall n \bullet n \geq n_0 \Rightarrow f(n) \leq c \cdot g(n))$$

$c \in \mathbb{N}$ $\quad$ u.b.e.

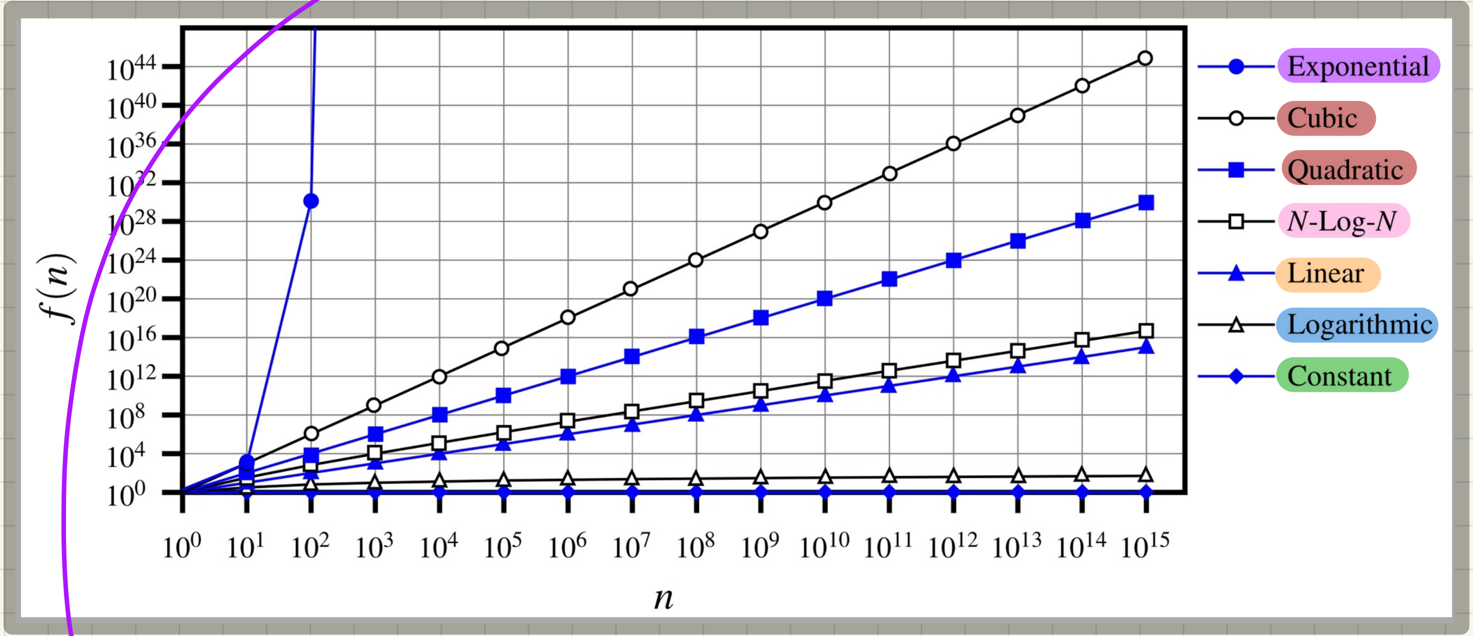
$\boxed{f(n) \leq c \cdot g(n)}$

Q. Why $\Rightarrow$ as opposed to $\wedge$

Hint Consider the truth tables

P	Q	$P \wedge Q$	$P \Rightarrow Q$
T	T	T	T
T	F	F	F
F	T	F	$\boxed{T}$
F	F	F	

RT Functions: Rates of Growth (w.r.t. Input Sizes)



the slower (flatter) relative to input size increase, the more efficient.

Size of integer interval

$[a, b] \rightsquigarrow \underline{a, a+1, a+2, \cdots, b}$

how many?

closed end

↳ end value included

$\parallel$

size $= \underline{b-a+1}$

e.g. $[34, 100] = 100 - 34 + 1 = \underline{\underline{67}}$

array

$[0, x] =$ array size

0 → min index

x → max index

$\parallel$

$(x-0)+1$

$\boxed{x+1}$

Asymptotic Upper Bound: Arithmetic Sequence/Progression

$$1 \xrightarrow{+1} 2 \xrightarrow{+1} 3 + 4 + \cdots \xrightarrow{+1} 100$$

100 terms

$$\frac{(1 + 100) * 100}{2}$$

n terms

common difference

$$\boxed{\underbrace{i}_{\substack{\text{start term}\\ \text{1st term}}} + \underbrace{(i + c)}_{\text{2nd term}} + \underbrace{(i + 2\cdot c)}_{\text{3rd term}} + \cdots + \underbrace{(i + (n-1)\cdot c)}_{n\text{th term}}}$$

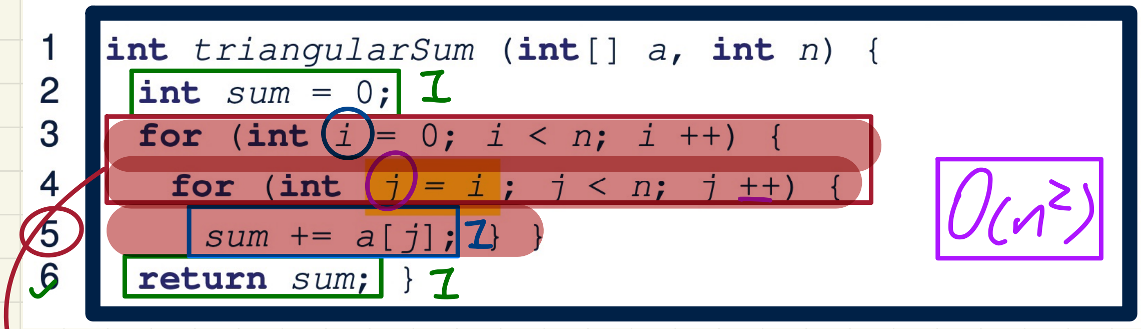
$$= \frac{(i + (i + (n-1)\cdot c)) * n}{2} = \boxed{\frac{c\cdot n^2 + (2i - c)n}{2}}$$

is $O(n^2)$

Remember

$$\frac{(\text{1st term} + \text{last term}) * \#\ \text{terms}}{2}$$

Determining the Asymptotic Upper Bound (3)



```
1  int triangularSum (int[] a, int n) {
2    int sum = 0;  1
3    for (int i = 0; i < n; i ++) {
4      for (int j = i; j < n; j ++) {
5        sum += a[j]; 1} }
6    return sum; } 1
```

$\boxed{O(n^2)}$

Each primitive op. takes O(1) time

Each combination of i and j corresponds to an exec. of L5:

Patterns of (i, j)

i: [0, n-1] = n

i	j				
0	0	1	2	...	n-1
1		1	2	...	n-1
2			2	...	n-1
⋮			⋮		
n-1					n-1

combination of (i, j) = (0, 2)

[0, n-1] = n

[1, n-1] = n-1

[2, n-1] = (n-1)-2+1 = n-2

⋮

[n-1, n-1] = 1

Executions of L5:

$$= n + (n-1) + (n-2) + \ldots + 1$$

$$= \frac{(n+1)\cdot n}{2}$$

is $O(n^2)$

$$O(\underbrace{1}_{L2} + \underbrace{n^2}_{L3\sim L5} + \underbrace{1}_{L6}) = O(n^2)$$

Implementing Stack / Queue

1. Using an array with some capacity MAX

s.push(..) ✓ s.push(..) ✓ - - - s.push ✓ | s.push

MAX pushes

(MAX+1)th push

↳ precondition violation (StackFullExcep.)

grow the size of the array when necessary

2. Using a dynamic array with "adapting" cap.

s.push(...)
s.push(...)
⋮

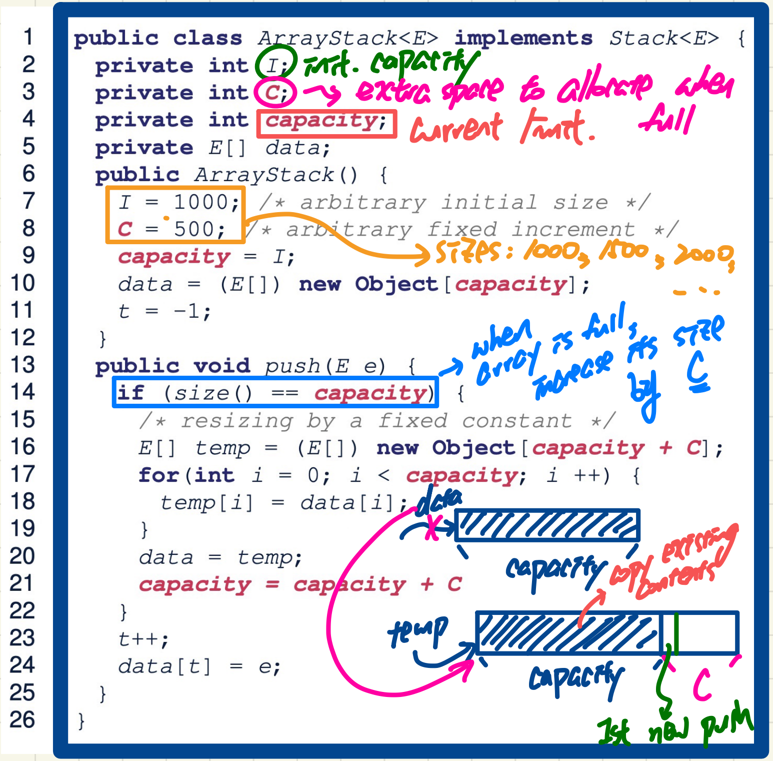
no worry about stack full.

2.1 constant increments

2.2 doubling

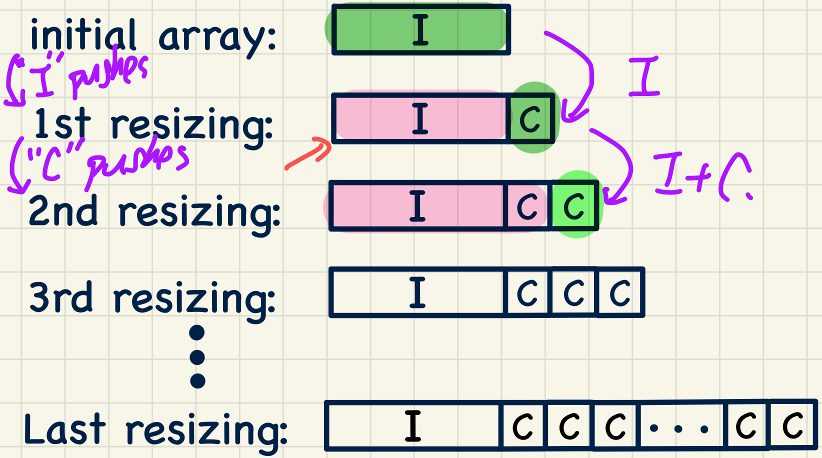
the one that demands less frequent resizing is asymptotically more efficient n pushes

Amortized Analysis: Dynamic Array with Const. Increments



```
public class ArrayStack<E> implements Stack<E> {
  private int I;
  private int C;
  private int capacity;
  private E[] data;
  public ArrayStack() {
    I = 1000; /* arbitrary initial size */
    C = 500; /* arbitrary fixed increment */
    capacity = I;
    data = (E[]) new Object[capacity];
    t = -1;
  }
  public void push(E e) {
    if (size() == capacity) {
      /* resizing by a fixed constant */
      E[] temp = (E[]) new Object[capacity + C];
      for(int i = 0; i < capacity; i ++) {
        temp[i] = data[i];
      }
      data = temp;
      capacity = capacity + C
    }
    t++;
    data[t] = e;
  }
}
```

init. capacity
extra space to allocate when full
current init.
sizes: 1000, 1500, 2000, ...
when array is full, increase its size by C
data
capacity
copy existing contents
temp
capacity
C
1st new push



W.L.O.G, assume: n pushes

and the last push triggers the last resizing routine.

Amortized/ Average RT:

Lecture 5 - Sep 17

Asymptotic Analysis,
Self-Balancing Binary Search Trees

Amortized RT: Constant Increments
Deriving Sum of Geometric Seq.
Height Balance Property

Announcements/Reminders

- First Class (Syllabus) recording & notes posted
- Today's class: notes template posted
- Exercises:
    - + Tutorial Week 1 (2D arrays)
    - + Tutorial Week 2 (2D arrays, Proving Big-O)
- Tutorial Week ~~2~~ 3 (this week)
    - + No in-person attendance.
    - + Exercises will be assigned.

average RT =* total RT / # ops. → e.g. n pushes

* over a seq of push operations

Amortized Analysis: Dynamic Array with Const. Increments

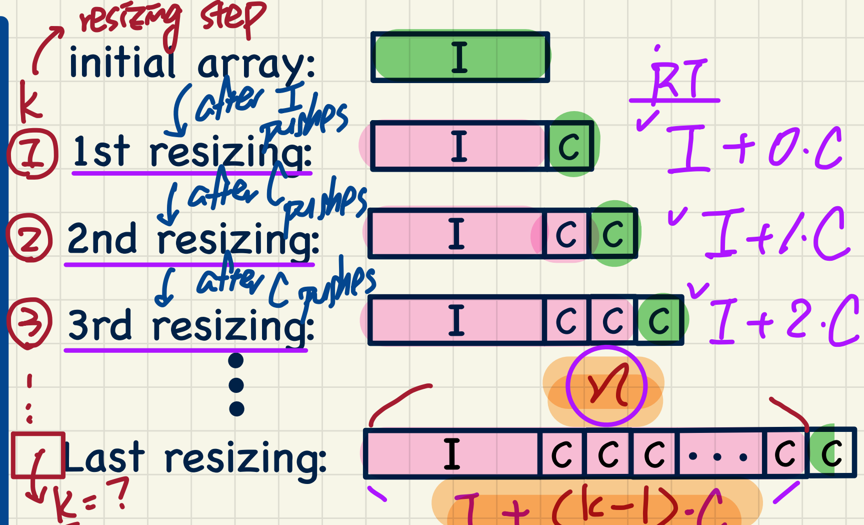
* without loss of generality

```
public class ArrayStack<E> implements Stack<E> {
  private int I;          init. capacity
  private int C;          → extra space to allocate when full
  private int capacity;   current init.
  private E[] data;
  public ArrayStack() {
    I = 1000; /* arbitrary initial size */
    C = 500; /* arbitrary fixed increment */     → sizes: 1000, 1500, 2000, ...
    capacity = I;
    data = (E[]) new Object[capacity];
    t = -1;
  }
  public void push(E e) {
    if (size() == capacity) {     → when array is full, increase its size by C
      /* resizing by a fixed constant */
      E[] temp = (E[]) new Object[capacity + C];
      for(int i = 0; i < capacity; i ++) {
        temp[i] = data[i];
      }
      data = temp;
      capacity = capacity + C
    }
    t++;
    data[t] = e;
  }
}
```

worst-case RT: O(n) ← # of elements in a full array

O(1) → RT when resizing not needed.

data, capacity, copy existing contents, temp, capacity, C, 1st new push



$n = I + (k-1)\cdot C \iff k = \frac{n-I}{C} + 1$

Total RT = Σ resizing steps

$= I + (I+C) + (I+2C) + \dots + n$

$\frac{(I+n)\cdot(\frac{n-I}{C}+1)}{2}$ is $O(n^2)$

Amortized/ Average RT: $O(\frac{n^2}{n}) = O(n)$

* W.L.O.G, assume: n pushes (consecutive) → first n elements stored.

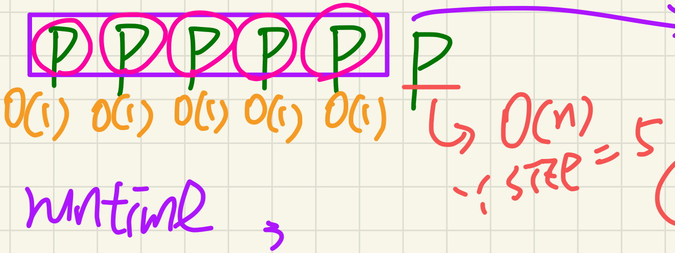
and the last push triggers the last resizing routine.

**

$$\frac{\overset{\text{highest power}}{\boxed{n^2}} + C \cdot n + (C \cdot I - I^2)}{2 \cdot C}$$

$$O(n^2)$$

assume:
I = 5



→ more accurate assessment of the strategy by consid ~~avg case~~

At runtime, the worst-case RT (of push operation) of a dynamic array occurs when that push op. triggers the resizing.

→ copying the existing contents to the new, bigger array.

Deriving the Sum of a Geometric Sequence

Initial Term: I

Common Factor: r

Number of Terms: k

$$\begin{tikzcd}
\boxed{3} \arrow[rr, bend left, "*2"] & + & 6 \arrow[rr, bend left, "*2"] & + & 12 \arrow[rr, bend left, "*2"] & + & 24
\end{tikzcd}$$

$\underbrace{I}_{3 \cdot 2^{0}}$ $\quad \underbrace{6}_{3 \cdot 2^{1}}$ $\quad \underbrace{12}_{3 \cdot 2^{2}}$ $\quad \underbrace{24}_{3 \cdot 2^{3}}$

$[0, 3] = 4$

terms : 4

$$S_{k} = \underbrace{\overset{I \cdot r^{0}}{I}}_{1st} + \underbrace{I \cdot r}_{2nd} + \underbrace{I \cdot r^{2}}_{3rd} + \underbrace{I \cdot r^{3}}_{4th} + \overset{I \cdot r^{k-2}}{\cdots} + \underbrace{I \cdot r^{k-1}}_{kth}$$

$$r \cdot S_{k} = I \cdot r + I \cdot r^{2} + I \cdot r^{3} + I \cdot r^{4} + \cdots + I \cdot r^{k-1} + I \cdot r^{k}$$

$$r \cdot S_{k} - S_{k} = (r-1) \cdot S_{k} = I \cdot r^{k} - I = I \cdot (r^{k} - 1) \Rightarrow S_{k} = \frac{I \cdot (r^{k} - 1)}{r - 1}$$

↓ useful for avg. RT of doubling.

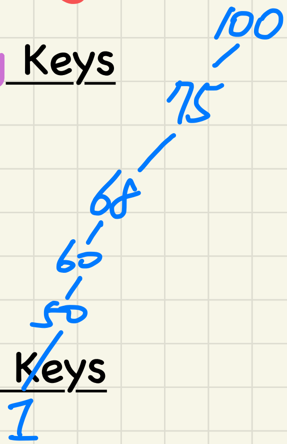
Worst-Case RT: BST with Linear Height

Example 1: Inserted Entries with Decreasing Keys

<100, 75, 68, 60, 50, 1>

key.

$n = 6$



$h = 5$

$(n-1)$

$\boxed{O(n)}$

linear height results in $O(n)$ search, insertion, deletion.

Example 2: Inserted Entries with Increasing Keys

<1, 50, 60, 68, 75, 100>

Exercises.

Example 3: Inserted Entries with In-Between Keys

<1, 100, 50, 75, 60, 68>

$\forall n .$ n is internal $\wedge$ difference of heights of n's children ≤ 1

BST + $\boxed{\text{height balance property}}$

||

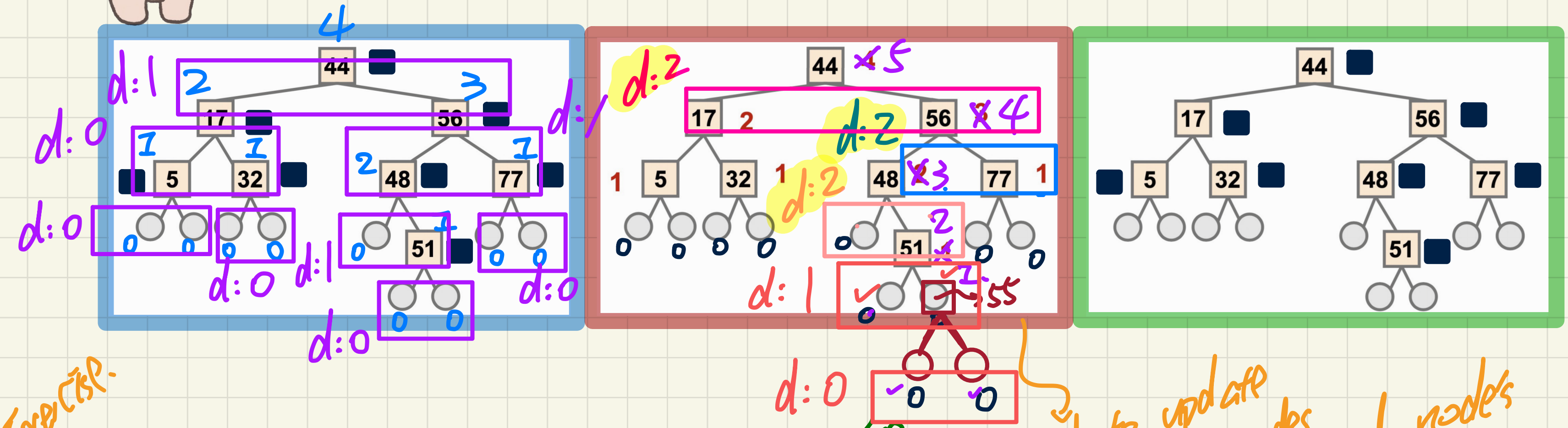
Balanced BST

Balanced BST: Definition

- internal node
- height
- height balance

Given a node p, the ***height*** of the subtree rooted at p is:

$$\textit{height}(p) = \begin{cases} 0 & \text{if } p \text{ is } \textit{external} \\ 1 + \textbf{MAX}(\{\, \textit{height}(c) \mid \textit{parent}(c) = p \,\}) & \text{if } p \text{ is } \textit{internal} \end{cases}$$



Exercise.

Q. Is the above tree a **balanced BST**? YES.

Q. Still a **balanced BST** after inserting **55**?

Q. Still a **balanced BST** after inserting **63**?

need to update heights of nodes along the inserted node's ancestor path

Lecture 6 - Sep 22

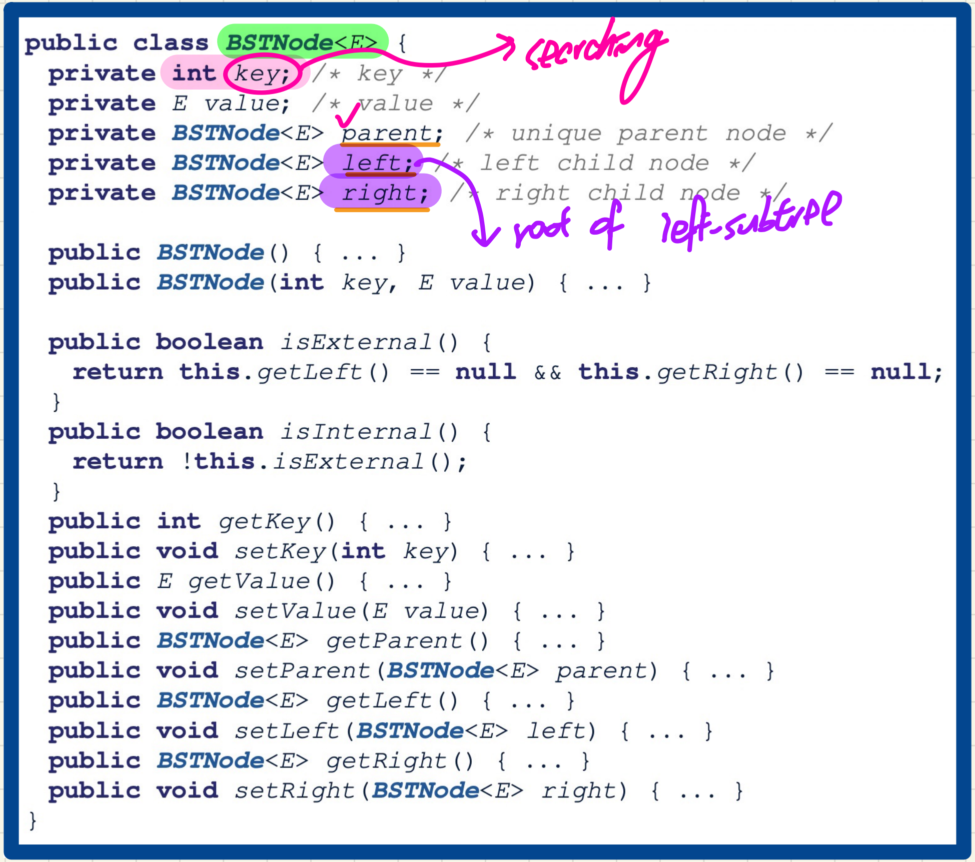
Self-Balancing Binary Search Trees

Implementing BST in Java
BST Operations: Search & Insert
Tree Rotation, In-Order Traversal

Announcements/Reminders

- First Class (Syllabus) recording & notes posted
- Today's class: notes template posted
- Exercises:
  - + Tutorial Week 1 (2D arrays)
  - + Tutorial Week 2 (2D arrays, Proving Big-O)
  - + Tutorial Week 3 (avg case analysis on doubling strategy)
- This Wednesday's class will have a **later** start: **4:10 PM**

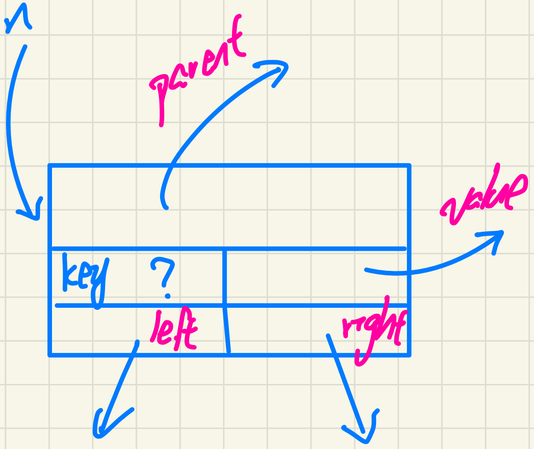
Generic, Binary Tree Nodes



```java
public class BSTNode<E> {
  private int key; /* key */
  private E value; /* value */
  private BSTNode<E> parent; /* unique parent node */
  private BSTNode<E> left; /* left child node */
  private BSTNode<E> right; /* right child node */

  public BSTNode() { ... }
  public BSTNode(int key, E value) { ... }

  public boolean isExternal() {
    return this.getLeft() == null && this.getRight() == null;
  }
  public boolean isInternal() {
    return !this.isExternal();
  }
  public int getKey() { ... }
  public void setKey(int key) { ... }
  public E getValue() { ... }
  public void setValue(E value) { ... }
  public BSTNode<E> getParent() { ... }
  public void setParent(BSTNode<E> parent) { ... }
  public BSTNode<E> getLeft() { ... }
  public void setLeft(BSTNode<E> left) { ... }
  public BSTNode<E> getRight() { ... }
  public void setRight(BSTNode<E> right) { ... }
}
```

searching

root of left-subtree



Compare:

+ prev ref.

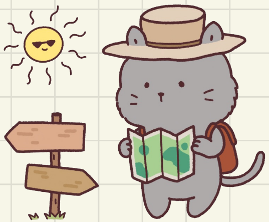
+ next ref.

in a DLN.

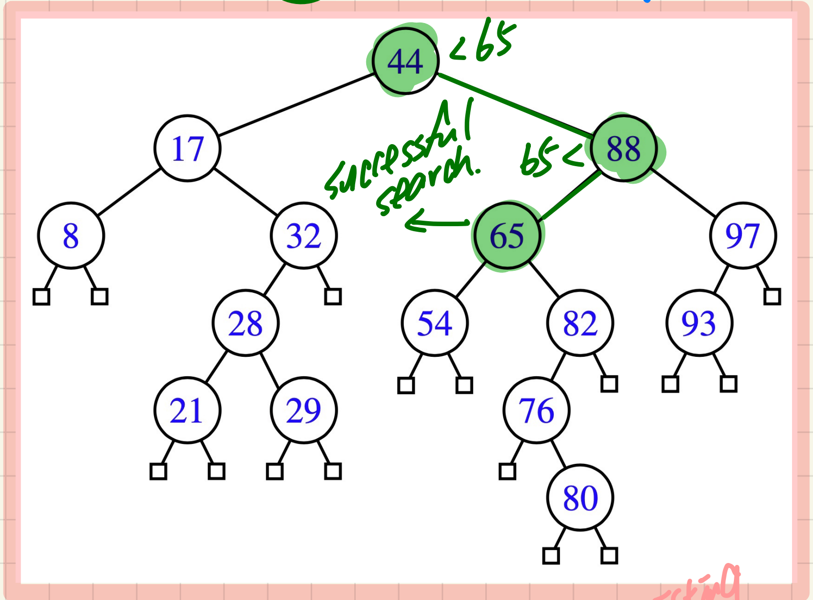
BST Operation: Searching a Key

BST ⇒ in-order traversal gives a sorted seq. of keys.

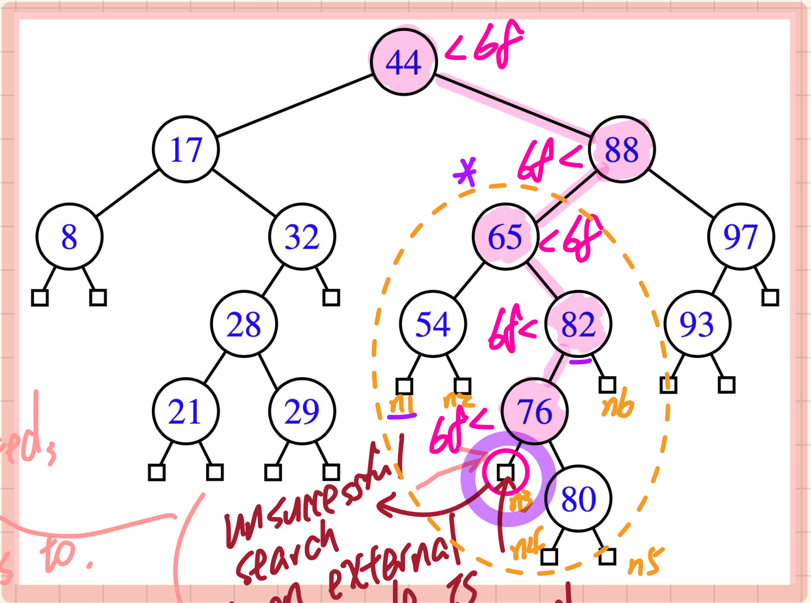
* T.o.t. n1 54 n2 65 ~~n3~~ 68 76 sorted! n4 80 n5 82 n6



Search key 65



Search key 68



If the non-existing key 68 is to be inserted, this ext. node is where it belongs to.

unsuccessful search → an external node is returned

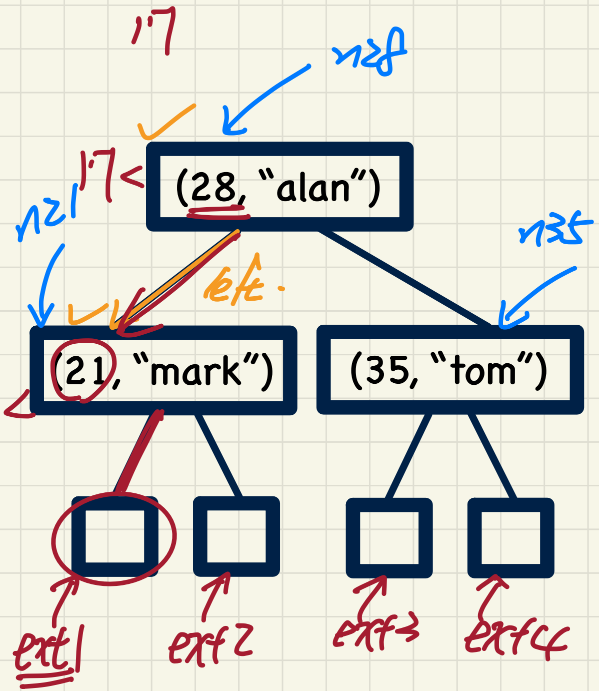
Tracing: Searching through a BST

```
@Test
public void test_binary_search_trees_search() {
  BSTNode<String> n28 = new BSTNode<>(28, "alan");
  BSTNode<String> n21 = new BSTNode<>(21, "mark");
  BSTNode<String> n35 = new BSTNode<>(35, "tom");
  BSTNode<String> extN1 = new BSTNode<>();
  BSTNode<String> extN2 = new BSTNode<>();
  BSTNode<String> extN3 = new BSTNode<>();
  BSTNode<String> extN4 = new BSTNode<>();
  n28.setLeft(n21); n21.setParent(n28);
  n28.setRight(n35); n35.setParent(n28);
  n21.setLeft(extN1); extN1.setParent(n21);
  n21.setRight(extN2); extN2.setParent(n21);
  n35.setLeft(extN3); extN3.setParent(n35);
  n35.setRight(extN4); extN4.setParent(n35);

  BSTUtilities<String> u = new BSTUtilities<>();
  /* search existing keys */
  assertTrue(n28 == u.search(n28, 28));
  assertTrue(n21 == u.search(n28, 21));
  assertTrue(n35 == u.search(n28, 35));
  /* search non-existing keys */
  assertTrue(extN1 == u.search(n28, 17)); /* *17* < 21 */
  assertTrue(extN2 == u.search(n28, 23)); /* 21 < *23* < 28 */
  assertTrue(extN3 == u.search(n28, 33)); /* 28 < *33* < 35 */
  assertTrue(extN4 == u.search(n28, 38)); /* 35 < *38* */
}
```

node creations

comparing nodes.

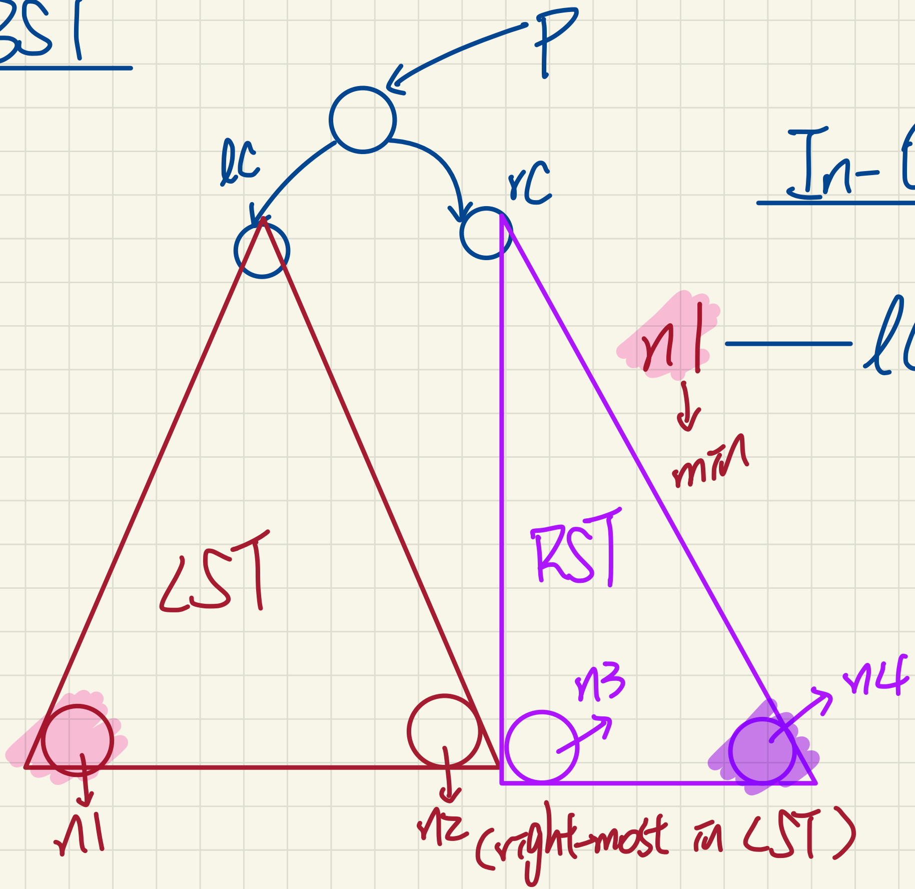
internal nodes (successful search).



exercise: why a particular ext. node is returned.

Visual Summary: In-Order Traversal on BST

BST



In-Order Traversal

n1 — lc — n2 P n3 — rc — n4

n1 ↓ min

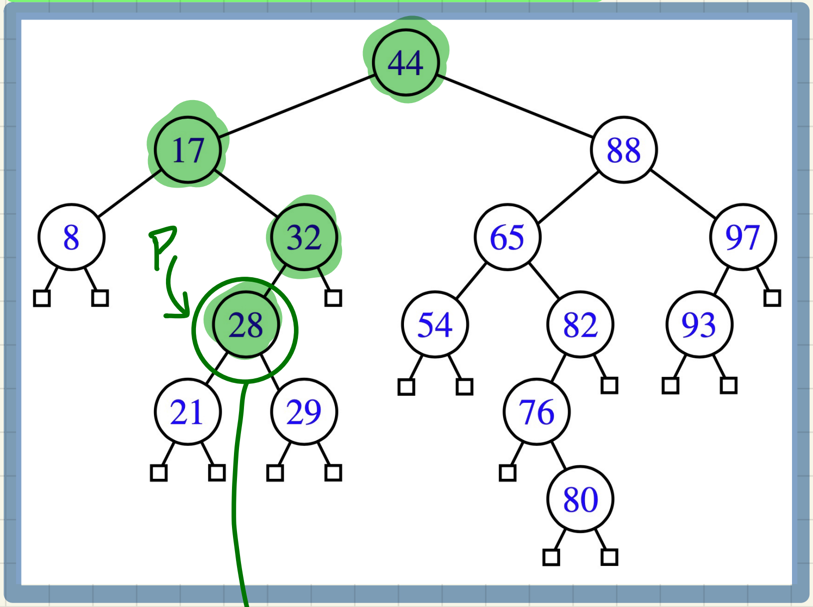
n2 ↳ largest key that is < P.

n3 ↳ smallest key that is > P.

n4 ↓ max

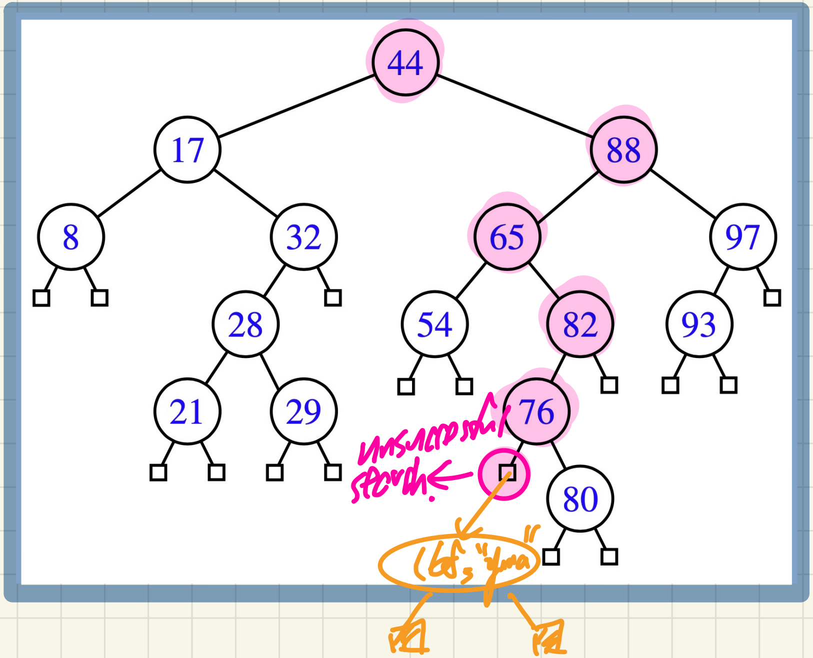
Visualizing BST Operation: Insertion

Insert Entry (28, "suyeon")

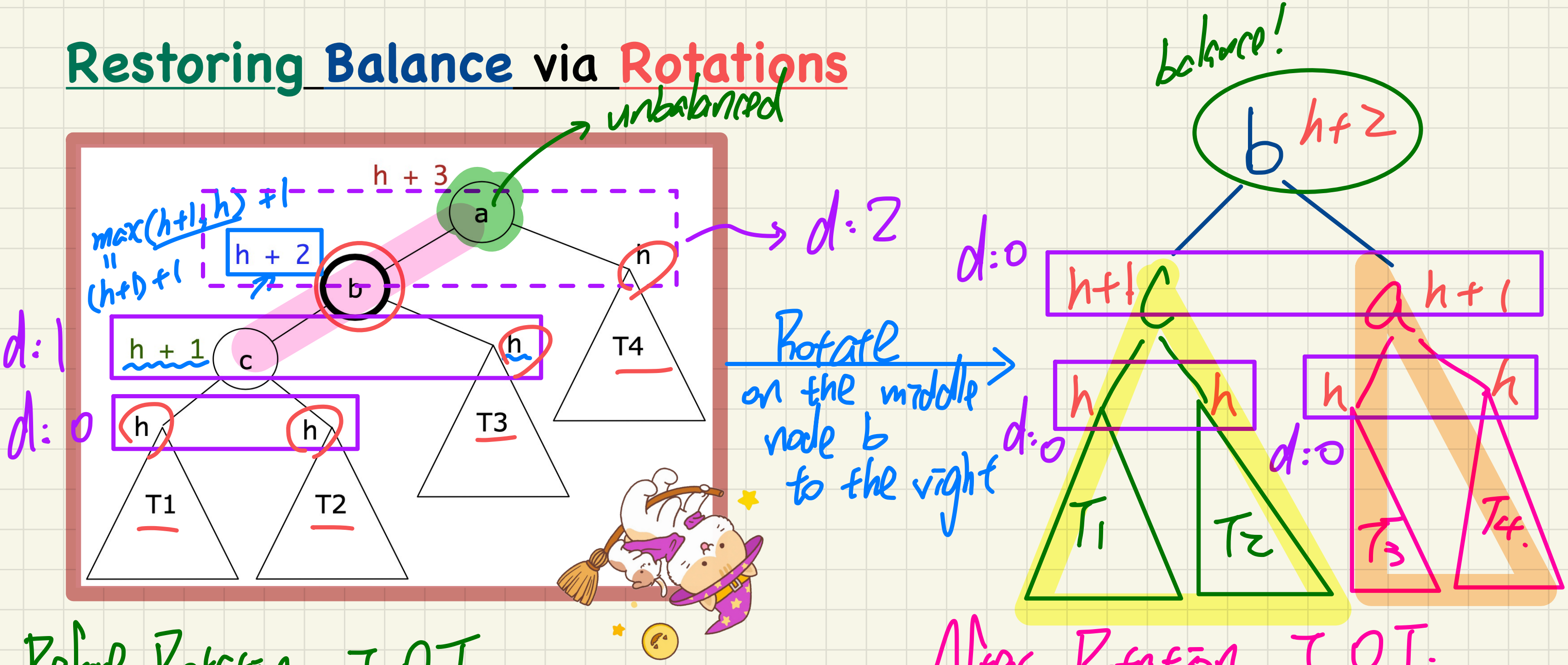


replace the value by the argument

Insert Entry (68, "yuna")



Restoring Balance via Rotations



Before Rotation, I.O.T.:

$\langle T_1, c, T_2, b, T_3, a, T_4 \rangle$

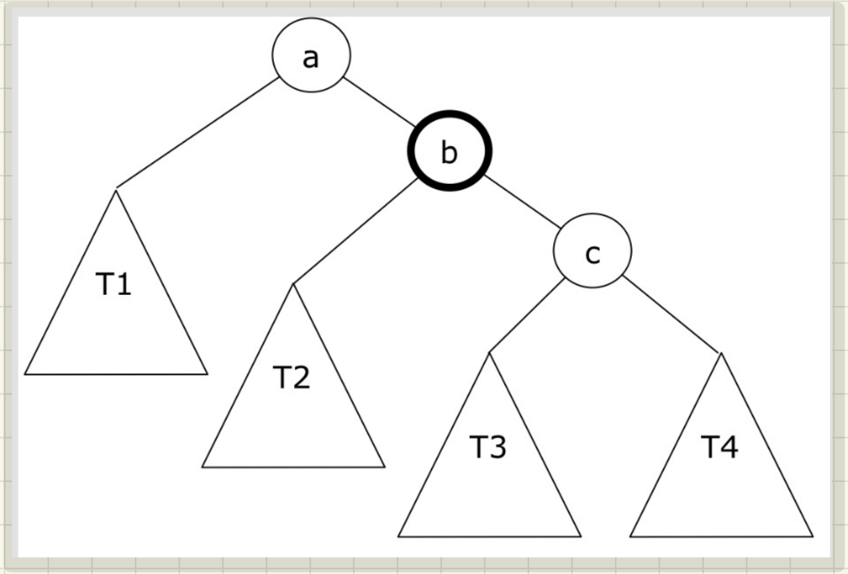
After Rotation, I.O.T.:

$\langle T_1, c, T_2, b, T_3, a, T_4 \rangle$

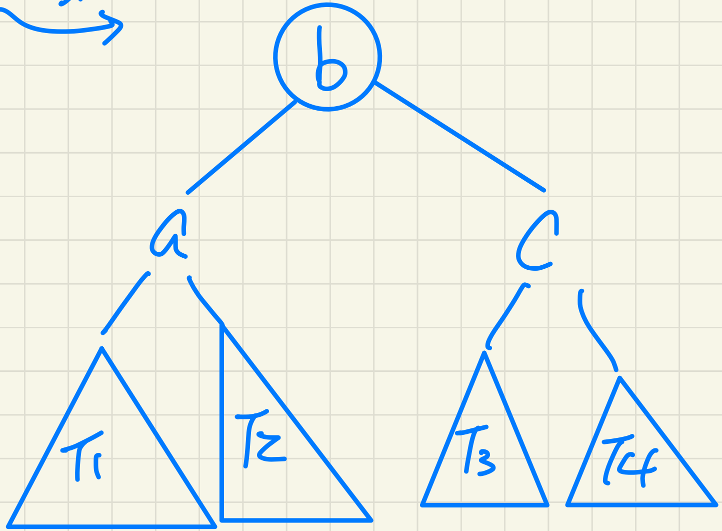
Q. Is the above tree **balanced**? YES

Q. After a **right-rotation** on node <u>b</u>, is the resulting tree still a **BST**? YES.

Trinode Restructuring: Single, Left Rotation



left rotation on b



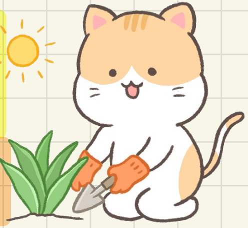
I.O.T.

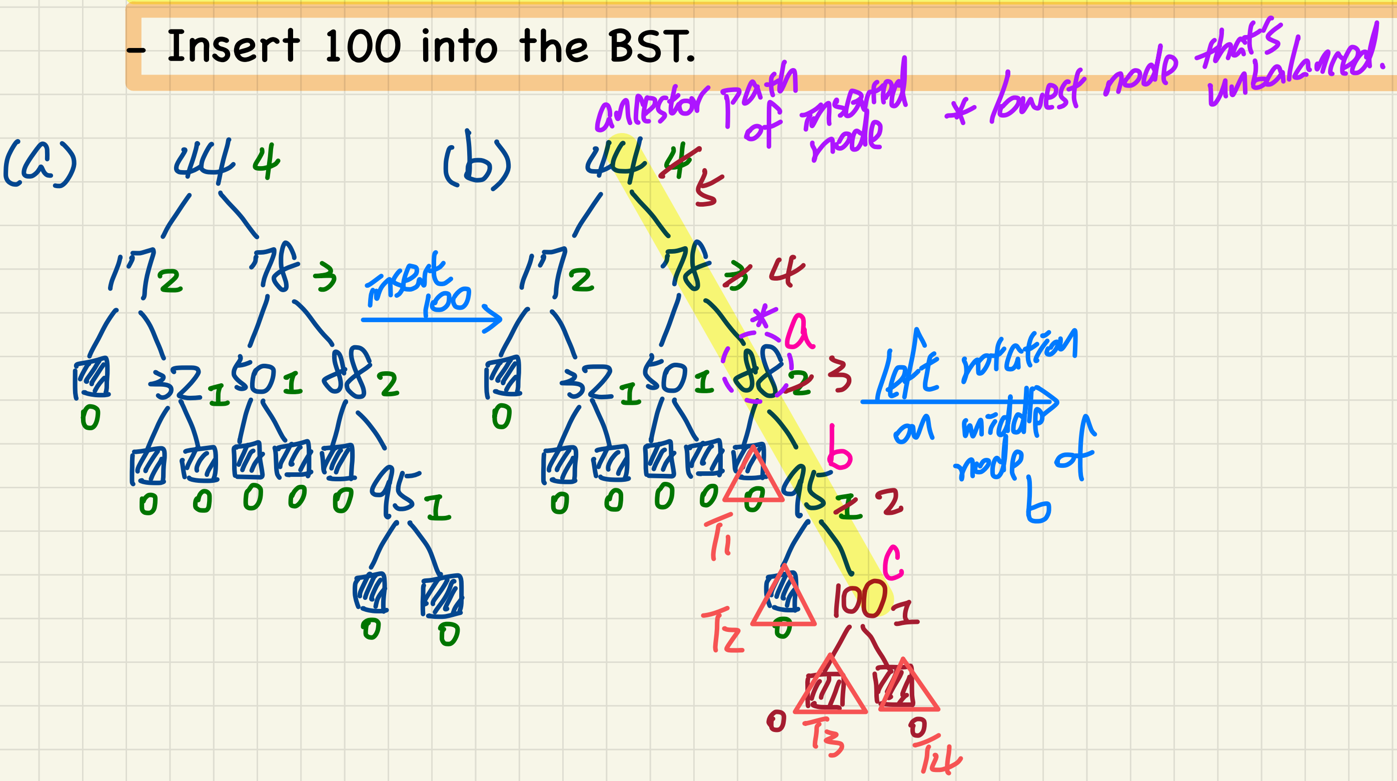
$\langle T_1, a, T_2, b, T_3, c, T_4 \rangle$

Trinode Restructuring after Insertion: Left Rotation

- Insert the following sequence of **keys** into an empty BST:
  <44, 17, 78, 32, 50, 88, 95>

- Insert 100 into the BST.





Lecture 7 - Sep 24

Self-Balancing Binary Search Trees

After Insertion: Left Rotation
After Insertion: Right-Left Rotations
BST Deletion: Cases 1 — 3

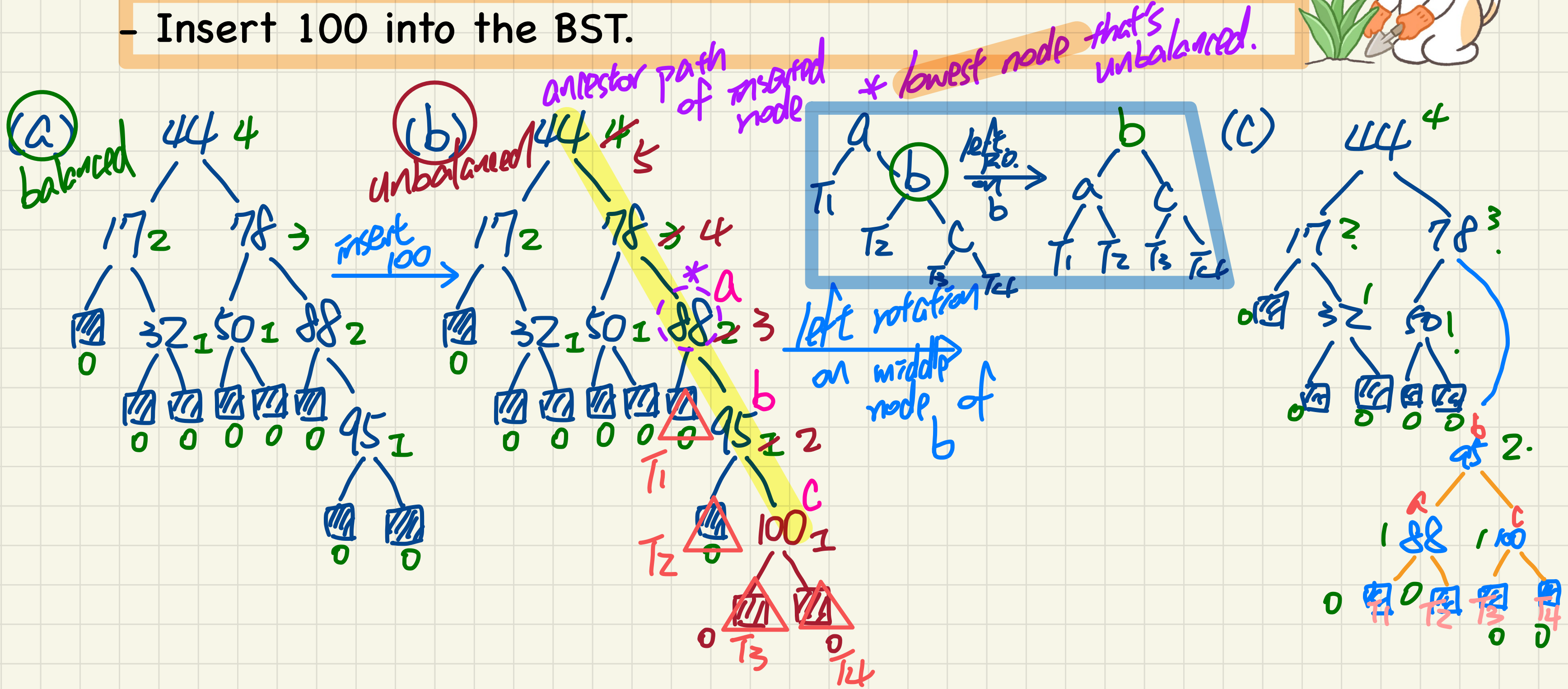
Announcements/Reminders

- First Class (Syllabus) recording & notes posted
- Today's class: notes template posted
- Exercises:
  - + Tutorial Week 1 (2D arrays)
  - + Tutorial Week 2 (2D arrays, Proving Big-O)
  - + Tutorial Week 3 (avg case analysis on doubling strategy)

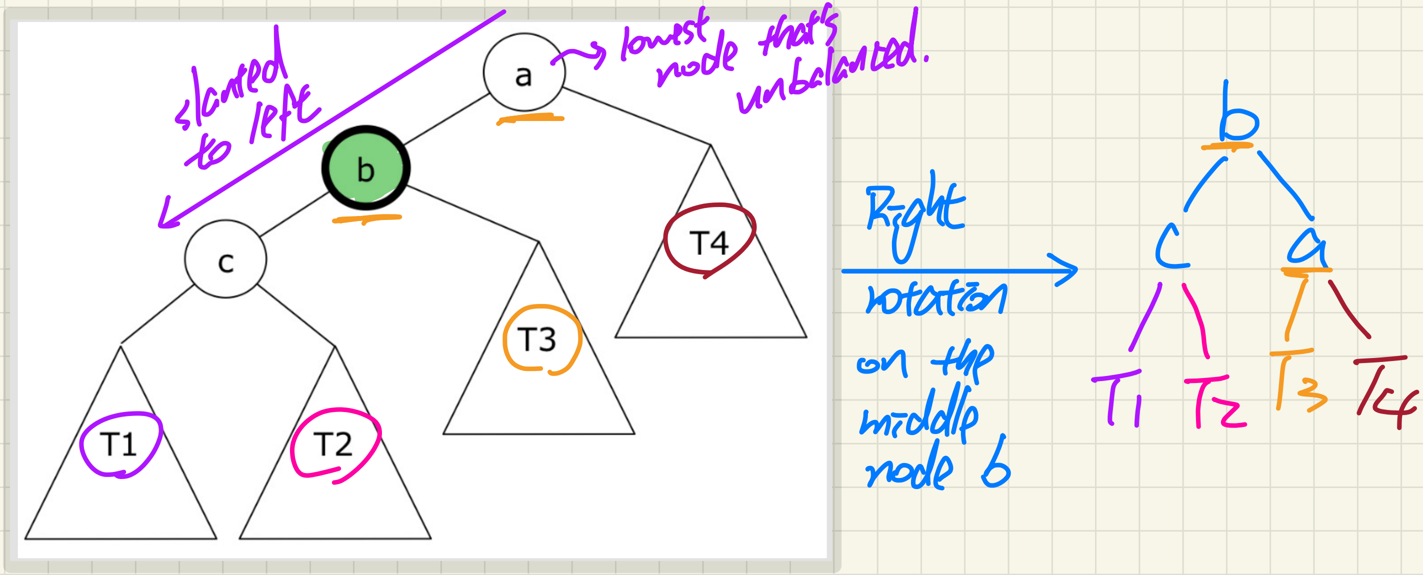
Trinode Restructuring after Insertion: Left Rotation

- Insert the following sequence of keys into an empty BST:
  <44, 17, 78, 32, 50, 88, 95>

- Insert 100 into the BST.



Trinode Restructuring: Single, Right Rotation



slanted to left

→ lowest node that's unbalanced.

Right rotation on the middle node b

I.O.T. : T_1, C, T_2, b, T_3, a, T_4

Trinode Restructuring after Insertion: Right Rotation

- Insert the following sequence of keys into an empty BST:
  <44, 17, 78, 32, 50, 88, 48>
- Insert 46 into the BST.

Trinode restructuring step   a , b , c

middle node

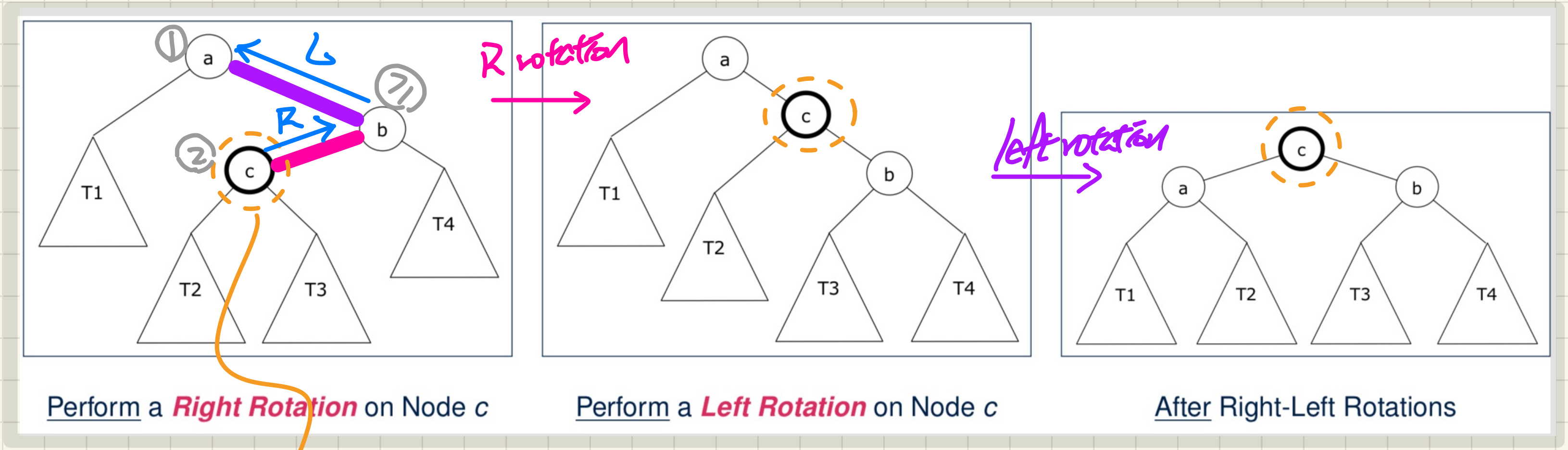
↳ one rotation (L , R)

↳ a, b, c slanted the same way.

↳ two rotations (R-L , L-R)

↳ a, b, c slanted differently.

Trinode Restructuring: Double, Right-Left Rotations



Perform a ***Right Rotation*** on Node *c*

Perform a ***Left Rotation*** on Node *c*

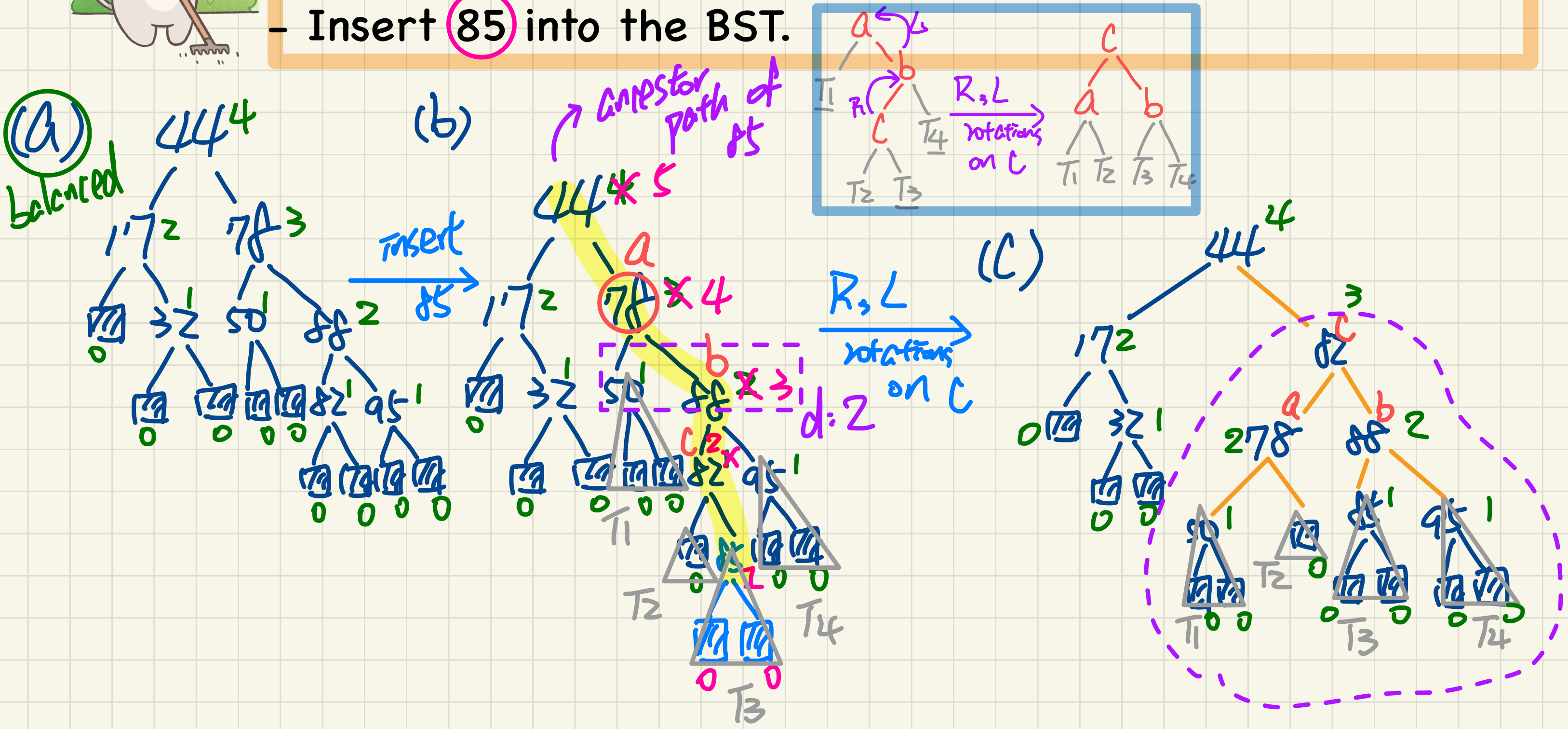
After Right-Left Rotations

to be promoted up 2 levels.

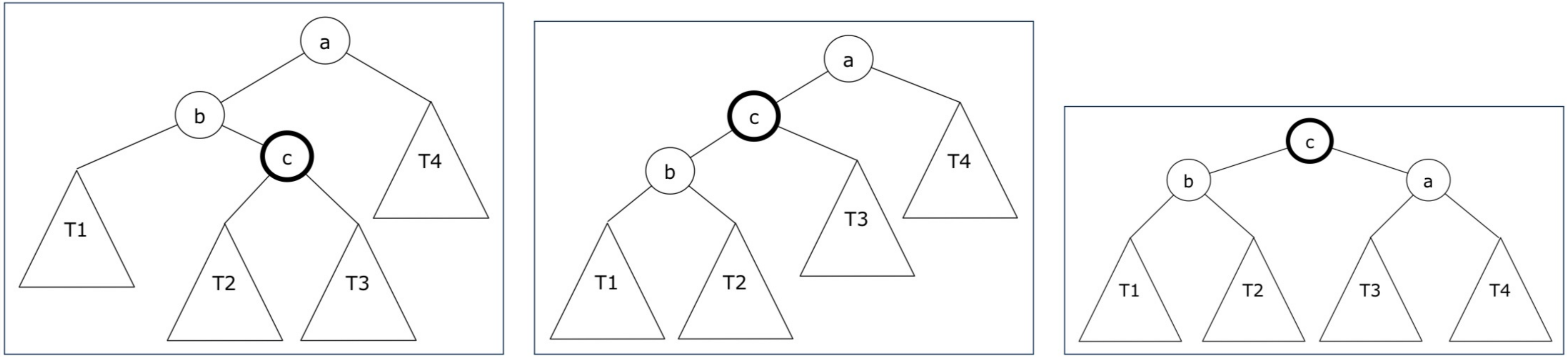
I.O.T. : T_1 $\overset{1}{a}$ T_2 $\overset{2}{c}$ T_3 $\overset{3}{b}$ T_4

Trinode Restructuring after Insertion: R-L Rotations

- Insert the following sequence of **keys** into an empty BST:
  <44, 17, 78, 32, 50, 88, 82, 95>
- Insert 85 into the BST.



Trinode Restructuring: Double, Left-Right Rotations



Perform a ***Left Rotation*** on Node *c*

Perform a ***Right Rotation*** on Node *c*

After Left-Right Rotations

Trinode Restructuring after Insertion: L-R Rotations

- Insert the following sequence of keys into an empty BST:

  <44, 17, 78, 32, 50, 88, 48, 62>

- Insert 54 into the BST.

BST Operation: Cases of Deletion

To *delete* an ***entry*** (with **key** *k*) from a BST rooted at ***node*** *n*:

root

Let node ***p*** be the return value from `search(n, k)`.

key of entry to be deleted.

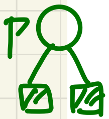
- ○ **Case 1**: Node ***p*** is ***external***.

  k is not an existing key ⇒ Nothing to remove

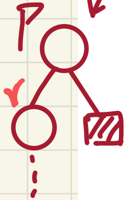
  internal

- ○ **Case 2**: Both of node ***p***'s child nodes are ***external***.

  No "orphan" subtrees to be handled ⇒ Remove ***p*** [Still BST?]



- ○ **Case 3**: One of the node ***p***'s children, say *r*, is ***internal***.
  - *r*'s sibling is ***external*** ⇒ Replace node ***p*** by node *r* [Still BST?]



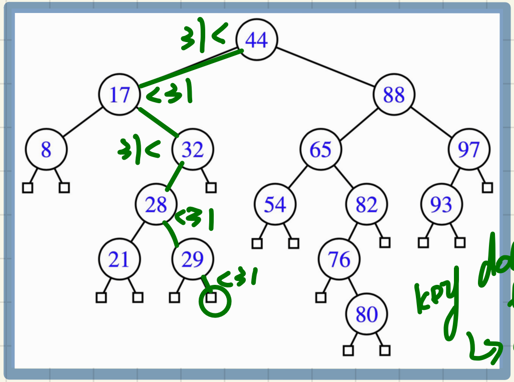
- ○ **Case 4**: Both of node ***p***'s children are ***internal***.
  - Let *r* be the ***right-most internal node p***'s ***LST***.

    ⇒ *r* contains the ***largest key s.t. key(r)*** < ***key(p)***.

    Exercise: Can *r* contain the ***smallest key s.t. key(r)*** > ***key(p)***?
  - Overwrite node ***p***'s entry by node *r*'s entry. [Still BST?]
  - *r* being the ***right-most internal node*** may have:
    - ⋄ Two ***external child nodes*** ⇒ Remove *r* as in **Case 2**.
    - ⋄ An ***external, RC*** & an ***internal LC*** ⇒ Remove *r* as in **Case 3**.

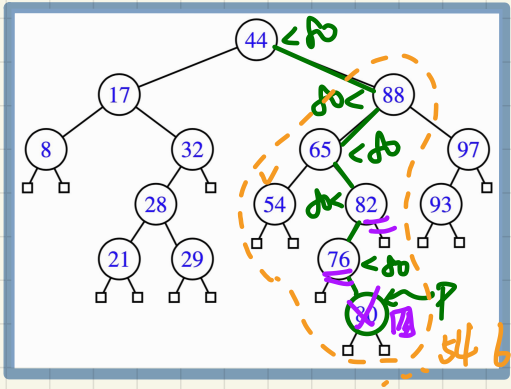
Visualizing BST Operation: Deletion

Case 1: Delete Entry with Key 31



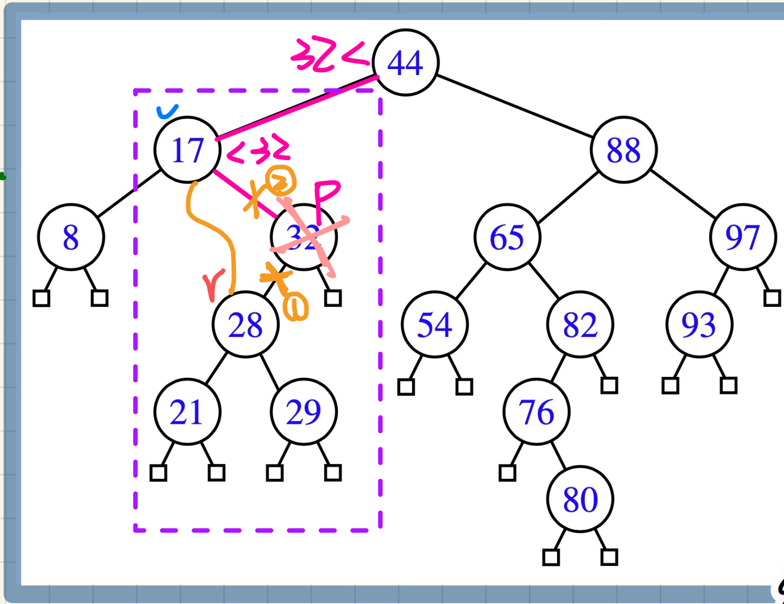
key doesn't exist
↳ do nothing.

Case 2: Delete Entry with Key 80



54 65 76 ~~80~~ 82 88 ...

Case 3: Delete Entry with Key 32



I.O.T. 17, 21, 28, 29, ~~32~~

Tutorials - Week 4 - Sep 26

Self-Balancing BST

Task Preview: Right Rotation in Java
BST Deletion: Case 4
Trinode Restructuring after Deletions

Exercise: BST Construction

Study: *constructExampleTree*

```
public class BSTNode<E> {
   private int key;
   private E value;

   private BSTNode<E> parent;
   private BSTNode<E> left;
   private BSTNode<E> right;

   . . .

   public void setLeft(BSTNode<E> left) {
      this.left = left;
      left.setParent(this);
   }

   public void setRight(BSTNode<E> right) {
      this.right = right;
      right.setParent(this);
   }

}
```

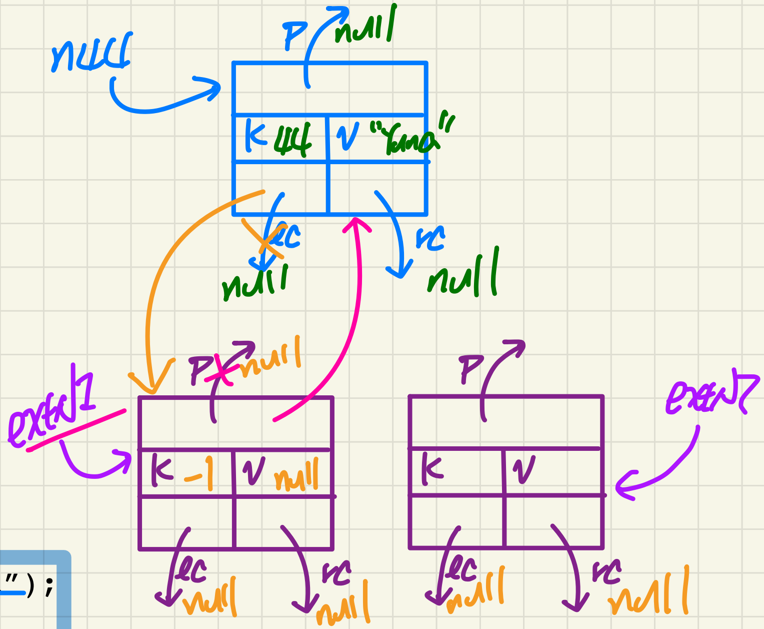
extN1

~~null~~ extN1

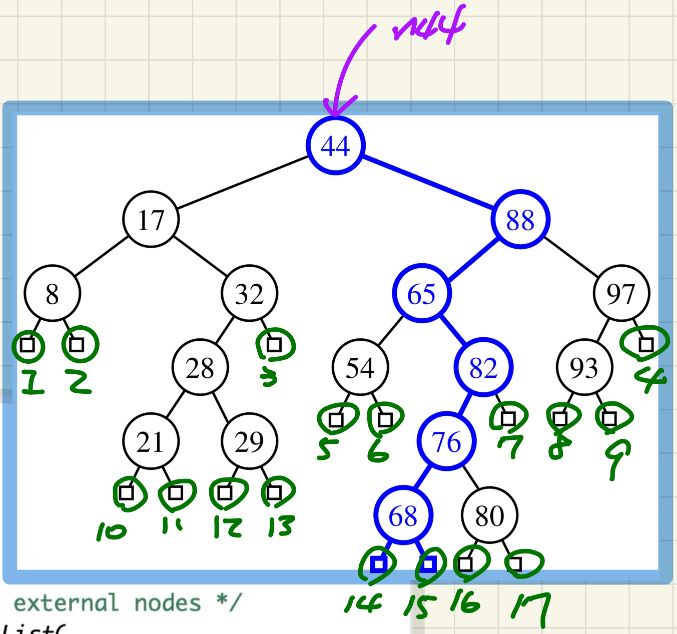
extN1

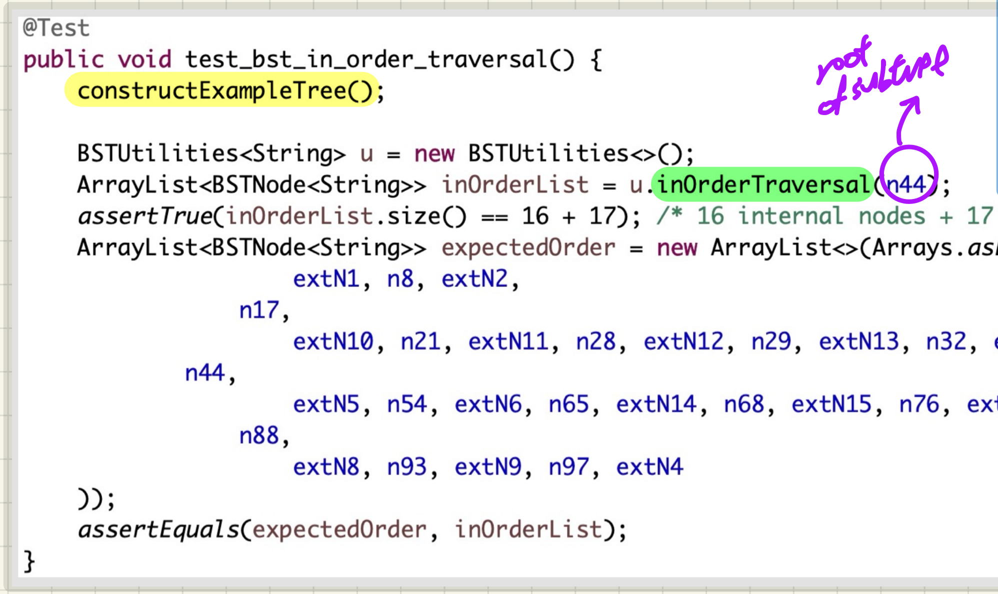
n44

```
BSTNode<String> n44 = new BSTNode<>(44, “Yuna”);
BSTNode<String> extN1 = new BSTNode<>();
BSTNode<String> extN2 = new BSTNode<>();
n44.setLeft(extN1);
n44.setRight(extN2);
```



Exercise: BST In-Order Traversal

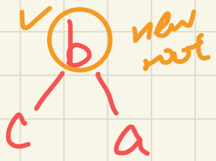


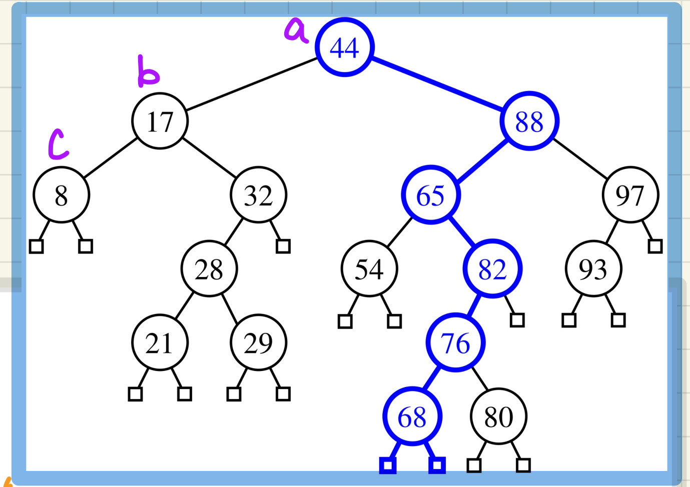


root of subtree

```
@Test
public void test_bst_in_order_traversal() {
    constructExampleTree();

    BSTUtilities<String> u = new BSTUtilities<>();
    ArrayList<BSTNode<String>> inOrderList = u.inOrderTraversal(n44);
    assertTrue(inOrderList.size() == 16 + 17); /* 16 internal nodes + 17 external nodes */
    ArrayList<BSTNode<String>> expectedOrder = new ArrayList<>(Arrays.asList(
                extN1, n8, extN2,
            n17,
                extN10, n21, extN11, n28, extN12, n29, extN13, n32, extN3,
        n44,
                extN5, n54, extN6, n65, extN14, n68, extN15, n76, extN16, n80, extN17, n82, extN7,
            n88,
                extN8, n93, extN9, n97, extN4
    ));
    assertEquals(expectedOrder, inOrderList);
}
```

Exercise: Trinode Restructuring via a Right Rotation





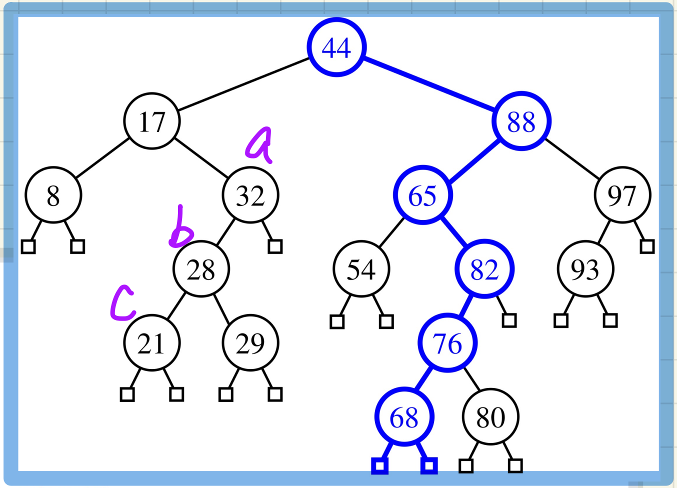
```
@Test
public void test_bst_right_rotation_1() {
    constructExampleTree();

    BSTUtilities<String> u = new BSTUtilities<>();
    u.rightRotate(n44, n17, n8);
                  a    b    c

    ArrayList<BSTNode<String>> inOrderList = u.inOrderTraversal(n17);
    assertTrue(inOrderList.size() == 16 + 17); /* 16 internal nodes + 17 external nodes */
    ArrayList<BSTNode<String>> expectedOrder = new ArrayList<>(Arrays.asList(
            extN1, // T1
            n8, // c
            extN2, // T2
            n17, // b
            extN10, n21, extN11, n28, extN12, n29, extN13, n32, extN3, // T3
            n44, // a
            extN5, n54, extN6, n65, extN14, n68, extN15, n76, extN16, n80, extN17, n82, extN7, n88, extN8, n93, extN9, n97, extN4 // T4
    ));
    assertEquals(expectedOrder, inOrderList);
}
```

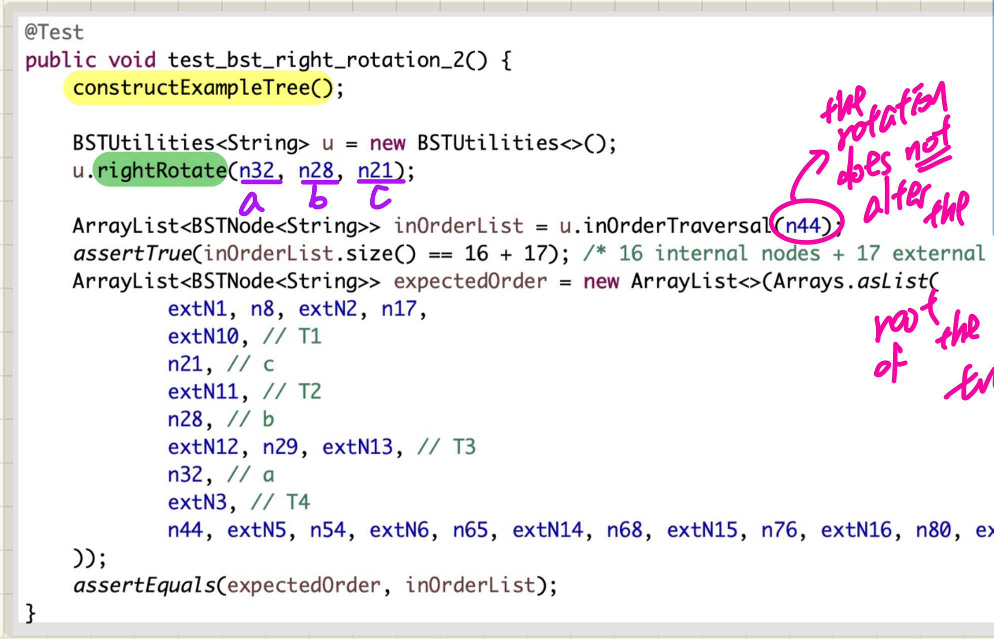
→ after R-rotation, the new root

→ new root

Identical to the T.O.A. seq. before rotation.

Exercise: Trinode Restructuring via a Right Rotation





```
@Test
public void test_bst_right_rotation_2() {
    constructExampleTree();

    BSTUtilities<String> u = new BSTUtilities<>();
    u.rightRotate(n32, n28, n21);
                  a    b    c

    ArrayList<BSTNode<String>> inOrderList = u.inOrderTraversal(n44);
    assertTrue(inOrderList.size() == 16 + 17); /* 16 internal nodes + 17 external nodes */
    ArrayList<BSTNode<String>> expectedOrder = new ArrayList<>(Arrays.asList(
            extN1, n8, extN2, n17,
            extN10, // T1
            n21, // c
            extN11, // T2
            n28, // b
            extN12, n29, extN13, // T3
            n32, // a
            extN3, // T4
            n44, extN5, n54, extN6, n65, extN14, n68, extN15, n76, extN16, n80, extN17, n82, extN7, n88, extN8, n93, extN9, n97, extN4
    ));
    assertEquals(expectedOrder, inOrderList);
}
```

the rotation does not alter the root of the entire tree.

BST Operation: Cases of Deletion

n1 ·· n2 ··· ~~n3~~ ~~p~~ n4 ··· n5 ·· n6

n3 ① replace p by n3

② remove n3

To *delete* an ***entry*** (with **key** *k*) from a BST rooted at ***node*** *n*:

Let node ***p*** be the return value from `search(n, k)`.

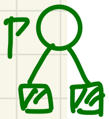
root

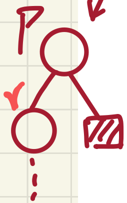
key of entry to be deleted.

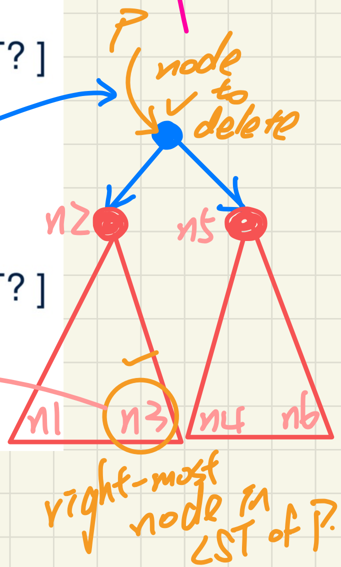
- **Case 1**: Node ***p*** is ***external***.
  - *k* is not an existing key ⇒ Nothing to remove
- **Case 2**: Both of node ***p***'s child nodes are ***external***.
  - internal
  - No "orphan" subtrees to be handled ⇒ Remove ***p*** [Still BST?]
- **Case 3**: One of the node ***p***'s children, say *r*, is ***internal***.
  - *r*'s sibling is ***external*** ⇒ Replace node ***p*** by node *r* [Still BST?]
- **Case 4**: Both of node ***p***'s children are ***internal***.
  - Let *r* be the ***right-most internal node p***'s ***LST***.

    ⇒ *r* contains the ***largest key s.t. key(r)*** < ***key(p)***.

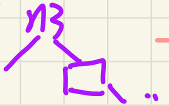
    Exercise: Can *r* contain the ***smallest key s.t. key(r)*** > ***key(p)***?
  - Overwrite node ***p***'s entry by node *r*'s entry. [Still BST?]
  - *r* being the ***right-most internal node*** may have:
    - ◇ Two ***external child nodes*** ⇒ Remove *r* as in **Case 2**.
    - ◇ An ***external, RC*** & an ***internal LC*** ⇒ Remove *r* as in **Case 3**.







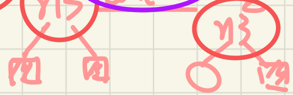
Q. Is it possible?



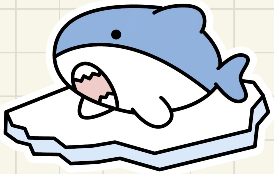
n3

Case 1

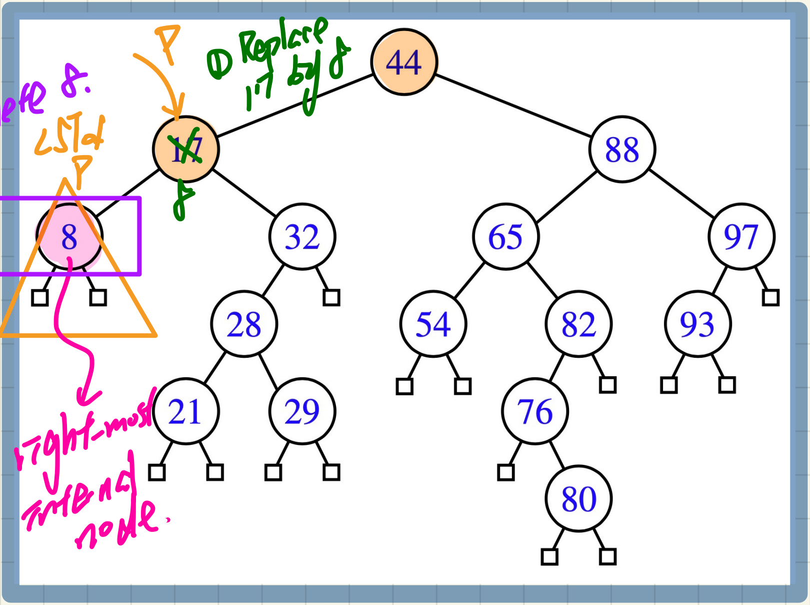
Case 2



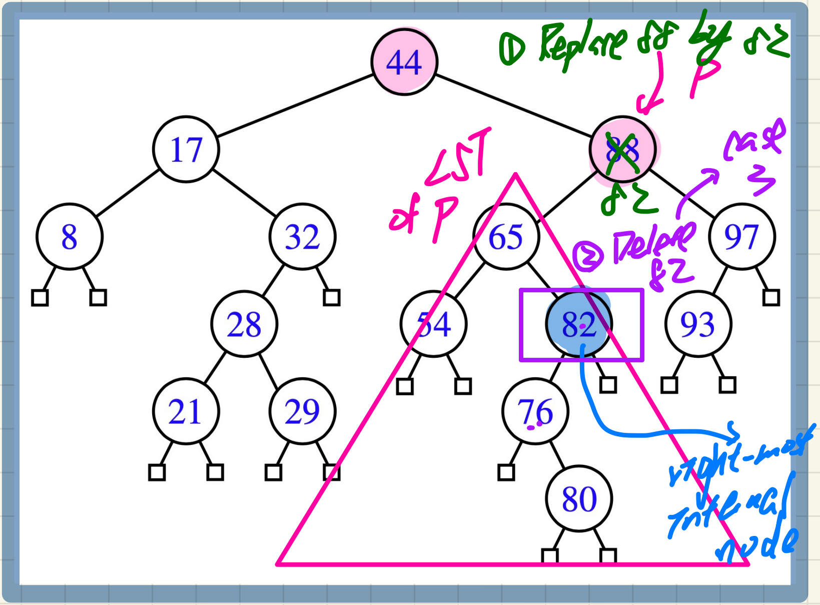
Visualizing BST Operation: Deletion



Case 4.1: Delete Entry with Key 17



Case 4.2: Delete Entry with Key 88



After Deletion: Continuous Trinode Restructuring

- Recall: *Deletion* from a BST results in removing a node with zero or one *internal* child node.
- After *deleting* an existing node, say its child is *n*: internal child

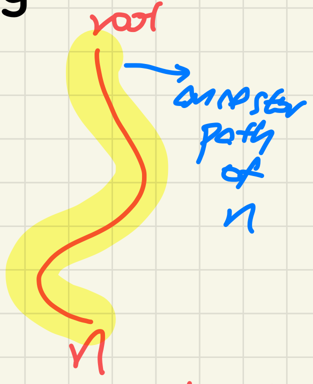
  Case 1: Nodes on *n*'s *ancestor path* remain *balanced*. ⇒ No rotations

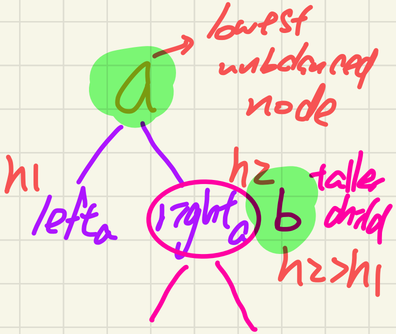
  Case 2: At least one of *n*'s *ancestors* becomes *unbalanced*.

  1. Get the **first**/**lowest** *unbalanced* node *a* on *n*'s *ancestor path*.
  2. Get *a*'s *taller* child node *b*. [$b \notin$ *n*'s *ancestor path*]
  3. Choose *b*'s child node *c* as follows:
     - *b*'s two child nodes have *different* heights ⇒ *c* is the *taller* child
     - *b*'s two child nodes have *same* height ⇒ *a*, *b*, *c* slant the *same* way
  4. Perform rotation(s) based on the *alignment* of *a*, *b*, and *c*:
     - Slanted the *same* way ⇒ ***single** rotation* on the **middle** node *b*
     - Slanted *different* ways ⇒ ***double** rotations* on the **lower** node *c*
- As *n*'s *unbalanced ancestors* are found, keep applying **Case 2**, until **Case 1** is satisfied. [*rotations*]

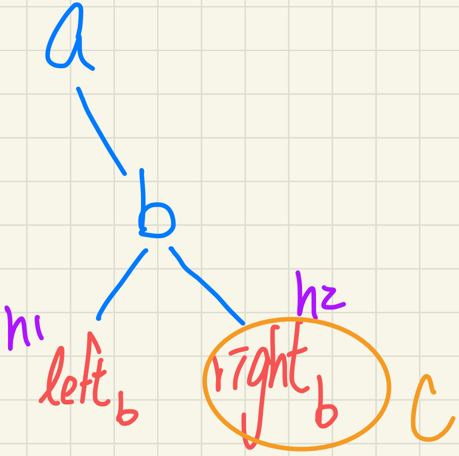
there might be multiple trinode restructuring steps to perform

~~$O(n)$~~ $O(h) = O(\log n)$.





(case 1)

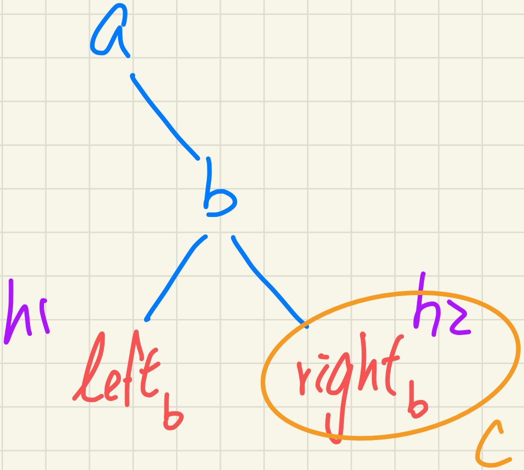


$h_1 \neq h_2$

⇒ choose the taller child.

say: $h_2 > h_1$

(case 2)

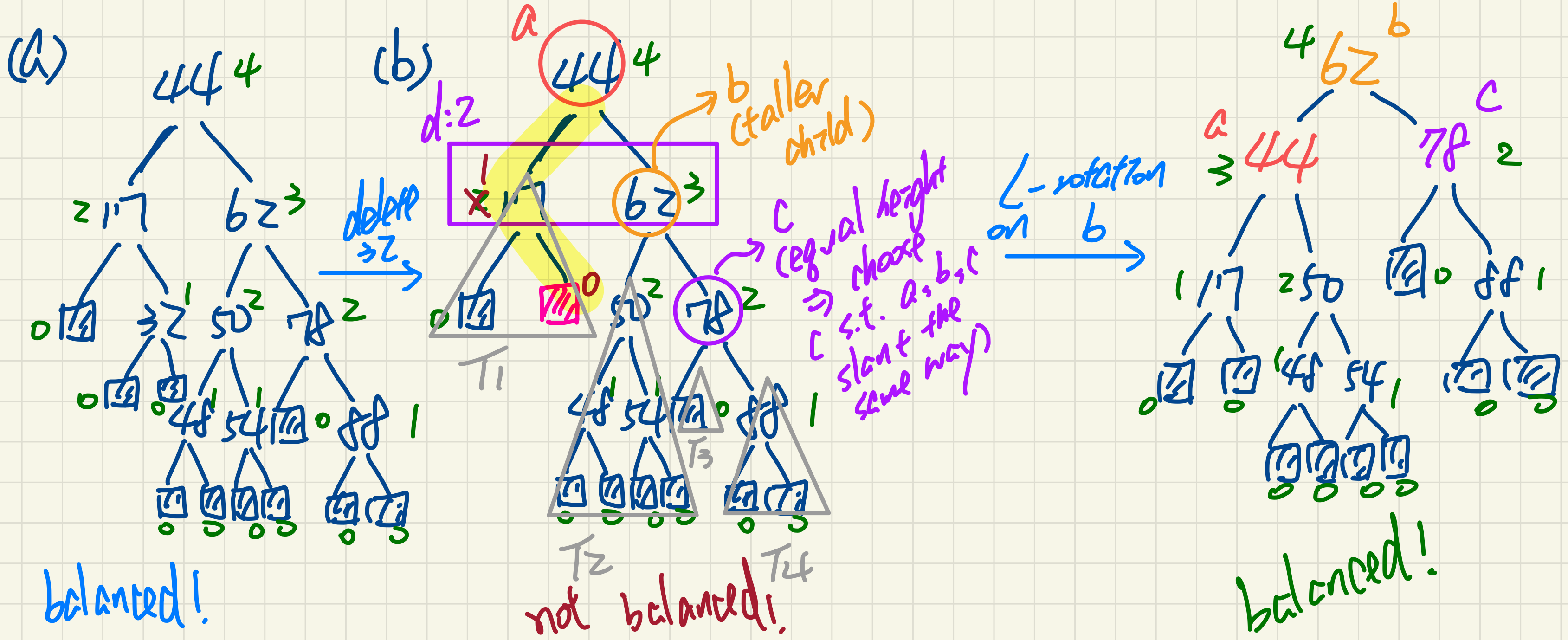


$h_1 = h_2$

⇒ choose the child that will make a, b, c slant the same way

Trinode Restructuring after Deletion: Single Rotation

- Insert the following sequence of **keys** into an empty BST:
  <44, 17, 62, 32, 50, 78, 48, 54, 88>
- Delete 32 from the BST.

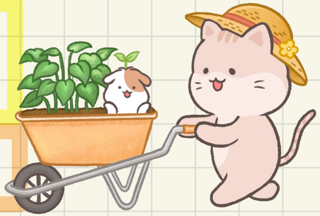


Trinode Restructuring after Deletion: Multiple Rotations

- Insert the following sequence of keys into an empty BST:

  <50, 25, 10, 30, 5, 15, 27, 1, 75, 60, 80, 55>

- Delete 80 from the BST.



Lecture 8 - Sep 29

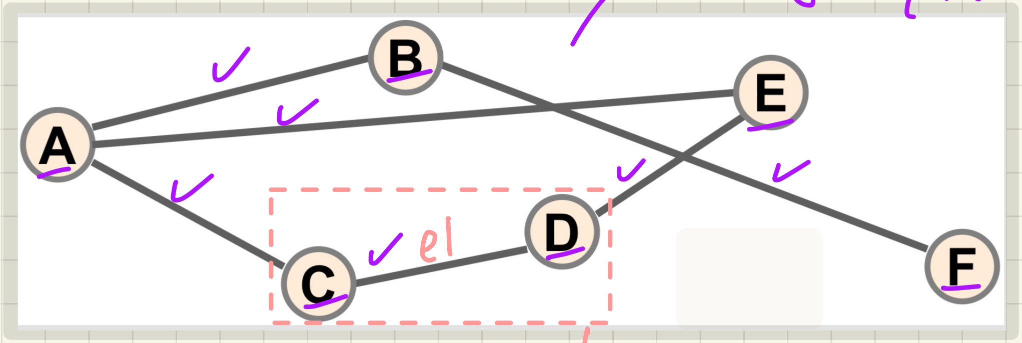
Graphs

Basic Definitions
Properties: Degrees, Number of Edges
Mathematical Induction on Vertices

Announcements/Reminders

- First Class (Syllabus) recording & notes posted
- Today's class: notes template posted
- Exercises:
  - + Tutorial Week 1 (2D arrays)
  - + Tutorial Week 2 (2D arrays, Proving Big-O)
  - + Tutorial Week 3 (avg case analysis on doubling strategy)
  - + Tutorial Week 4 (Trinode restructuring after deletions)

Graph: Definition



$V = \{A, B, C, D, E, F\}$

$E = \{(A,B), (A,C), (A,E), (C,D), (D,E), (B,F)\}$

$|V| = 6$

cardinality/
size

$|E| = 6$

$el = (C, D)$

$G = (V, E)$

vertices/
nodes

edges/
arcs

ordered pairs

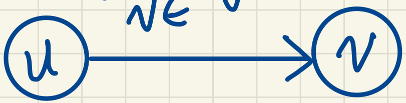
Edges: Directed vs. Undirected

Directed

dependency, flow, order among nodes

$(u, v) \neq (v, u)$

$u \in V$
$v \in V$

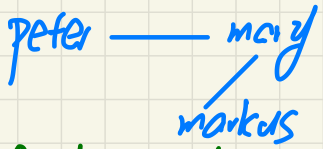


u: origin

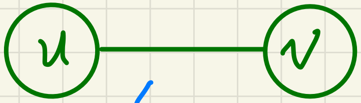
v: destination

(u, v)

(v, u) undefined.



Undirected

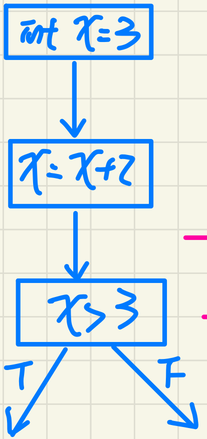


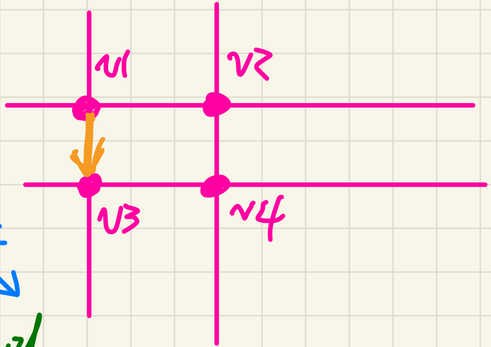
$\Updownarrow$

as if: (u, v) and (v, u) bi-directional.

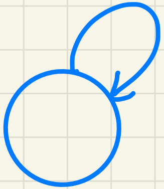
Examples:

- Control Flow/Data Flow Diagrams
- Social Network of Friendships (symmetry)
- Road Map of GPS → there's some one-way street
- Collaboration Network (Co-authorship)
- Degree Requirement pre-requisite. → topological sort.
- Web Pages (Hyperlinked)
- Protein-Protein Interaction Network → symmetry.



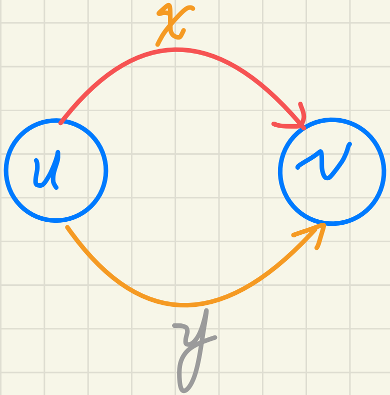


self edge/loop : (u, u)



multiple/parallel edges: (u, v)

(u, v)



Simple Graph : graph without self and parallel edges

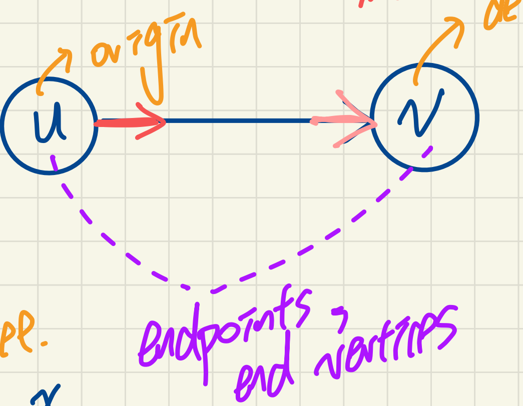
not simple graph : graph has self edges or parallel edges.

Vertices: Degree

→ # of edges incident on a vertex

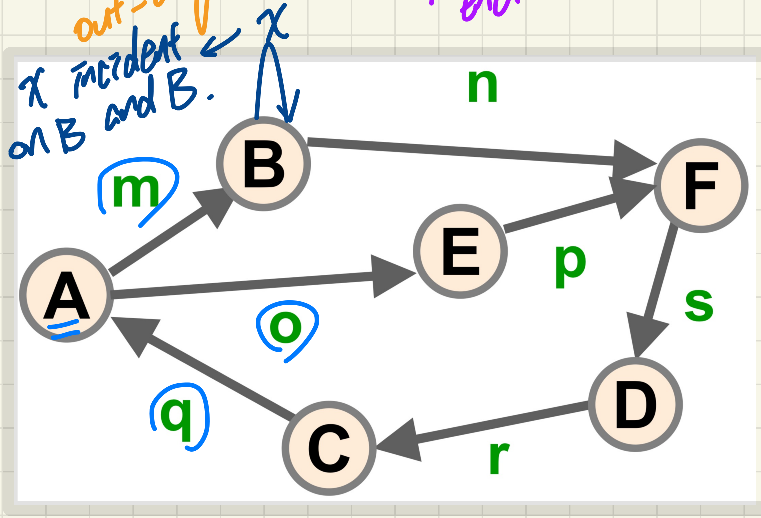
undirected: degree of vertex

directed: in-degree vs out-degree.



(u, v) is incident on vertices u and v.

- is outgoing edge of u
- is incoming edge of v



Exercises:

End vertices of m? A, B

Outgoing Edges of A? m, o

Incoming Edges of A? q

Edges incident on A? m, o, q.

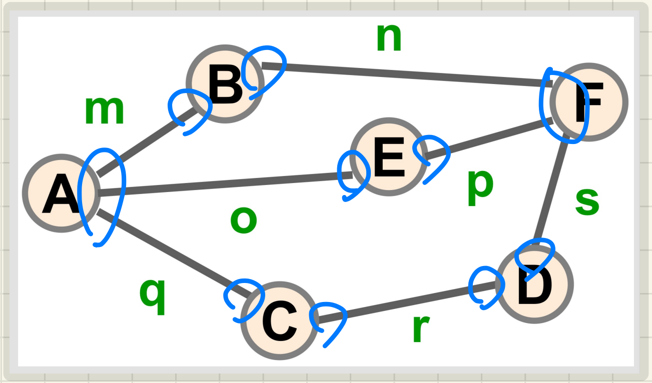
Degree of A? 3 (in-degree (A) = 1, out-degree (A) = 2)

Properties: Sum of Degrees for Undirected Graphs

Given a **simple**, **undirected** graph $G = (V, E)$ with $|E| = m$:

$$\sum_{v \in V} \mathtt{degree}(v) = 2 \cdot \underline{m}$$

↳ $|E|$



Vertex	Degree
A	3
B	2
C	2
D	2
E	2
F	3

$|E| = 7$

m

$\boxed{14} = 2 \cdot |E|$

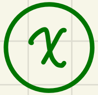
Properties: Sum of Degrees for Undirected Graphs

Given a non-empty **simple**, **undirected** graph $G = (V, E)$ with $|E| = m$:

$$\boxed{\sum_{v \in V} \text{degree}(v) = 2 \cdot m}$$ claim

Strategy of Proof : Perform a M.I. on |V|

(1) Base Case : $|V| = 1$



$|E| = 0.$

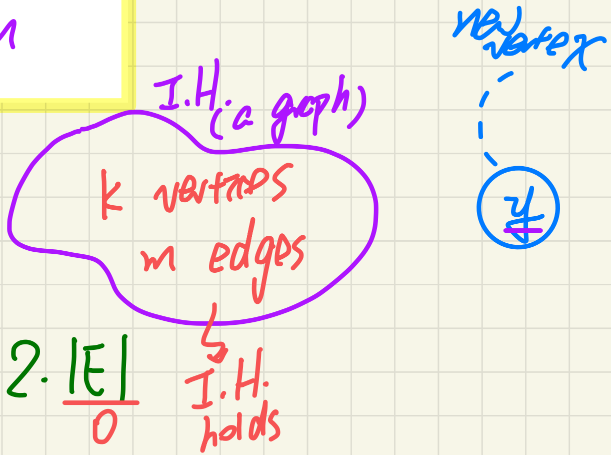
$\text{degree}(x) = 0$ $\quad \sum_{v \in \{x\}} \text{degree}(v) = \text{degree}(x) = 0 = 2 \cdot \underbrace{|E|}_{0}$

(2) Inductive Hypothesis (I.H.)

$$\sum_{v \in V} \text{degree}(v) = 2 \cdot m$$

where $|V| = k > 1$

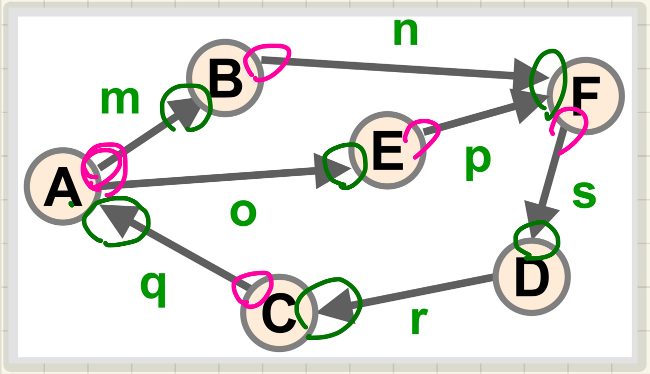
(3)* Make a strictly larger graph with $k+1$ vertices (by adding a new vertex y)



Properties: Sum of Degrees for Directed Graphs

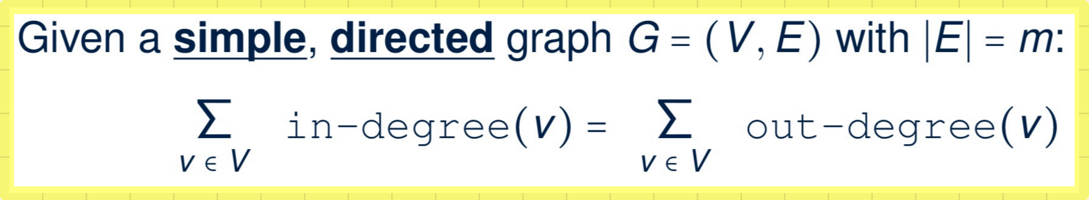
Given a **simple**, **directed** graph $G = (V, E)$ with $|E| = m$:

$$\sum_{v \in V} \texttt{in-degree}(v) = \sum_{v \in V} \texttt{out-degree}(v)$$



Vertex	in-degree	out-degree
A	1	2
B	1	1
C	1	1
D	1	1
E	1	1
F	2	1
	$\Sigma = 7$	$\Sigma = 7 = \lvert E \rvert = m$

Properties: Sum of Degrees for Directed Graphs



Given a **simple**, **directed** graph $G = (V, E)$ with $|E| = m$:

$$\sum_{v \in V} \texttt{in-degree}(v) = \sum_{v \in V} \texttt{out-degree}(v)$$

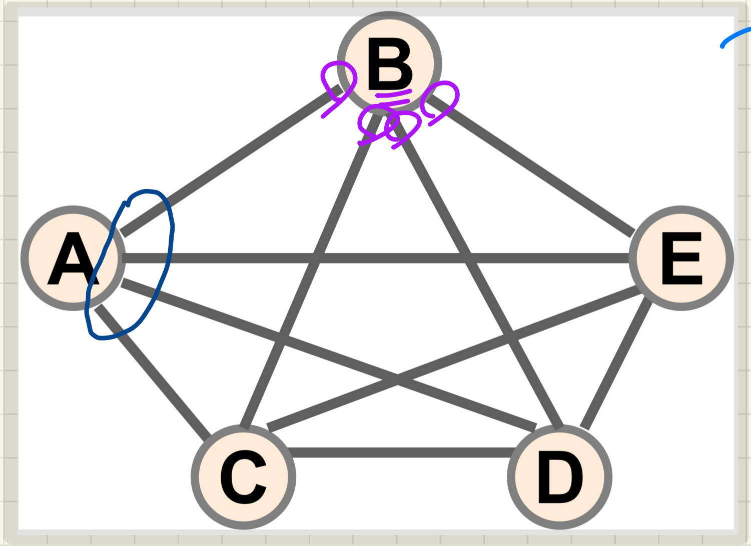
Properties: Sum of Degrees for Directed Graphs

Given a **simple**, **undirected** graph $G = (V, E)$, $|V| = n$, $|E| = m$:

$$m \le \frac{n \cdot (n-1)}{2}$$

→ # of edges is $O(|V|^2)$

|E|



$$\frac{|V| \times (|V|-1)}{2}$$

→ $|V| = 5$

$|E| = 10$

Vertex	edges
A	(A,B), (A,C), (A,D), (A,E)
B	(B,A), (B,C), (B,D), (B,E)
C	
D	
E	

|V|

max if a vertex is connected to all other $|V|-1$ vertices

$|V|-1$

2 edges should be counted as 1 edge towards m

Given a **simple**, **undirected** graph $G = (V, E)$, $|V| = n$, $|E| = m$:

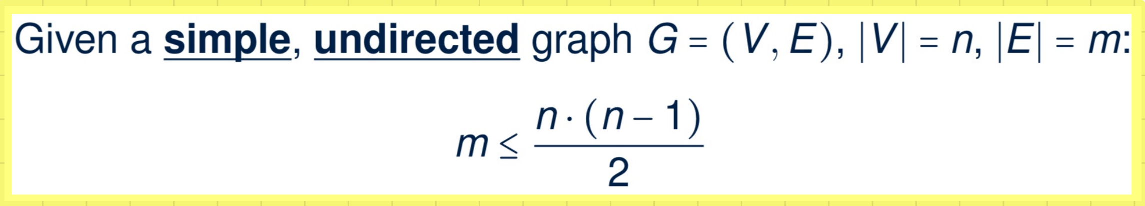
$$m \le \frac{n \cdot (n-1)}{2}$$

$=$

When $m = \frac{n.(n-1)}{2}$

$\Rightarrow$ G is Complete

Properties: Sum of Degrees for Directed Graphs



Given a **simple**, **undirected** graph $G = (V, E)$, $|V| = n$, $|E| = m$:

$$m \leq \frac{n \cdot (n-1)}{2}$$

Lecture 9 - Oct 1

Graphs

Mathematical Induction: Degree Sum
Paths, Cycles, Reachability
(Spanning vs. Connected) Subgraphs

Announcements/Reminders

- First Class (Syllabus) recording & notes posted
- Today's class: notes template posted
- Exercises:
  - + Tutorial Week 1 (2D arrays)
  - + Tutorial Week 2 (2D arrays, Proving Big-O)
  - + Tutorial Week 3 (avg case analysis on doubling strategy)
  - + Tutorial Week 4 (Trinode restructuring after deletions)

2. $(m+d)$ → # edges in the extended graph

Properties: Sum of Degrees for Undirected Graphs

Given a **simple** **undirected** graph $G = (V, E)$ with $|E| = m$:

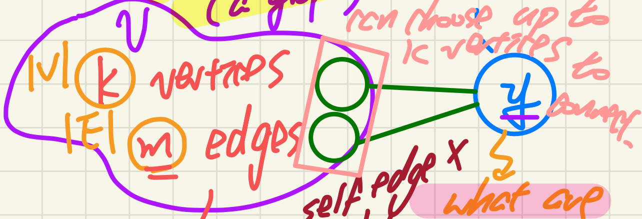
non-empty

$$\boxed{\sum_{v \in V} \text{degree}(v) = 2 \cdot m}$$

claim

$$\sum_{v \in V} \text{degree}(v) = 2 \cdot m$$

I.H. (a graph)



self edge ✗

parallel edge ✗

I.H. holds

what are the possible ways to connect y to the existing graph?

Strategy of Proof : Perform a M.I. on |V|

(1) Base Case : $|V| = 1$

$$\textcircled{x} \qquad |E| = 0. \qquad \text{degree}(x) = 0$$

$$\sum_{v \in \{x\}} \text{degree}(v) = \text{degree}(x) = 0 = 2 \cdot \underset{0}{\underline{|E|}}$$

(2) Inductive Hypothesis (I.H.)

$$\sum_{v \in V} \text{degree}(v) = 2 \cdot m$$

where $|V| = \boxed{k > 1}$

more substantial than base cases.

(3)* Make a strictly larger graph with $\underline{k+1}$ vertices (by adding a new vertex $\underline{y}$)

Choose $\textcircled{d}$ existing vertices and connect them to y. $d \le k$.

I.H. (excluding y)

$$\sum_{v \in \boxed{V \cup \{y\}}} \text{degree}(v) = \boxed{2 \cdot m} + \textcircled{d} + \textcircled{d}^{*}$$

k+1 vertices

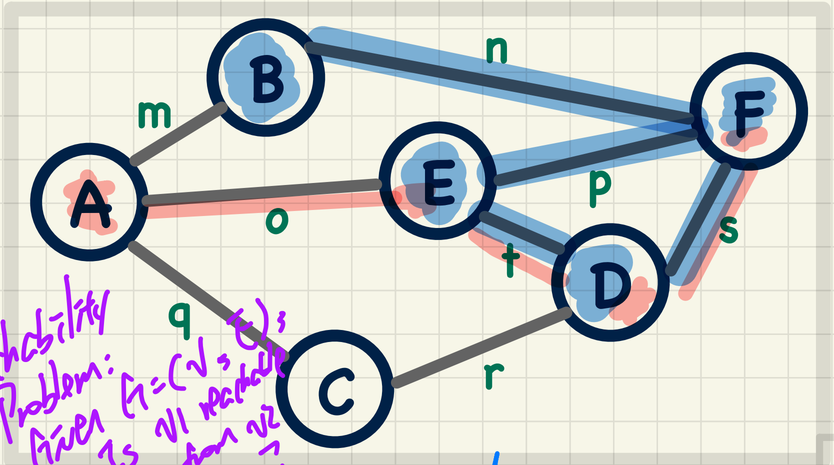
from y to existing nodes

from existing nodes to y

a graph with a cycle ⇒ cyclic

a graph without a cycle ⇒ acyclic

Graph: Paths and Cycles



Reachability Problem: Given G=(V,E); Is v1 reachable from v2?

- Path alternating v's and e's.
- Cycle
- Simple Path → a path without cycle
- Simple Cycle → 1. cycle 2. Except the two vertices forming the cycle, all remaining vertices are distinct
- Reachable
- Reachable Paths

Path: (F, s, D, t, E, p, F, n, B)

start vertex

end vertex

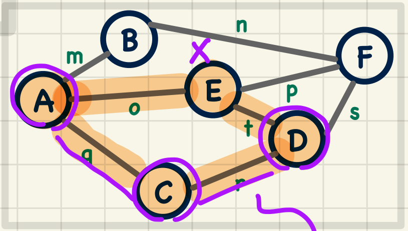
also a cycle.

Simple Path: (F, s, D, t, E, o, A)

cycle

Simple Cycle: (E, t, D, r, C, q, A, o, E)

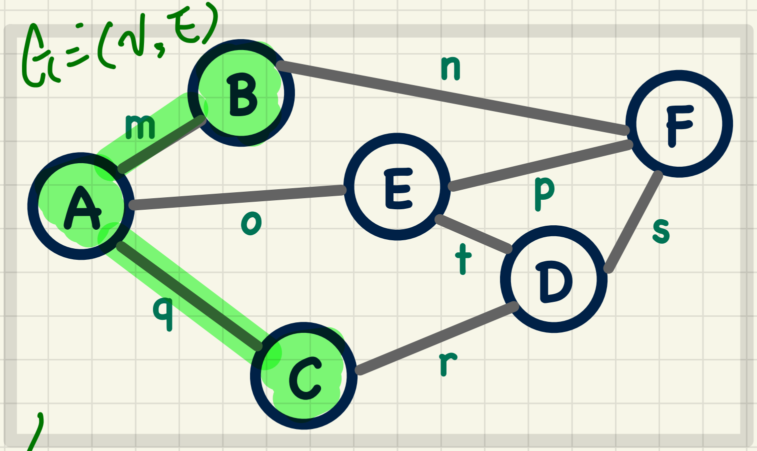
simple path (without any cycle)



a simple path

- subgraph
- spanning subgraph
- connected subgraph
- forest
- tree
- spanning tree

Graph: Subgraphs and Spanning Subgraphs



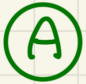
subgraph $G'=(V',E')$

$V' \subseteq V \wedge E' \subseteq E$

① G'_1 empty graph → min subgraph

$V'=\emptyset \wedge E'=\emptyset$,

② G'_2 one-vertex graph $V'=\{A\} \wedge E'=\emptyset$



③ a subgraph cannot contain just a single edge

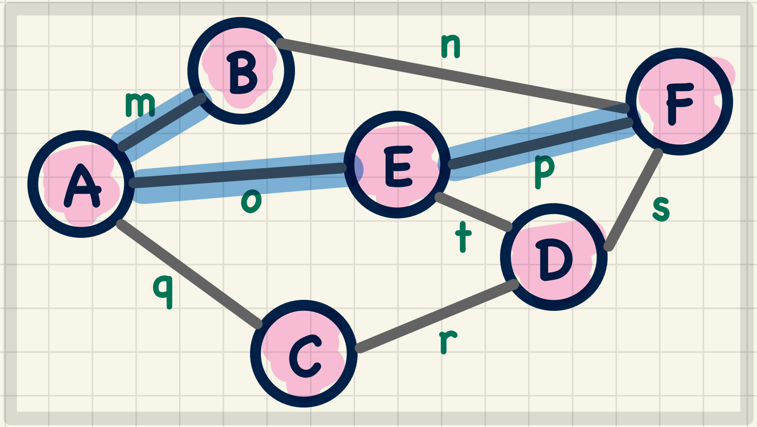
④ $G'_3 = (\{A,B,C\},\{m,q\})$ → a subgraph but not spanning.

⑤ G can be its own subgraph! → max subgraph

Spanning Subgraph

→ a subgraph that "spans" through all vertices.

↳ $G' = (V', E')$ is a spanning subgraph of G $\Leftrightarrow$ $V' = V \wedge E' \subseteq E$



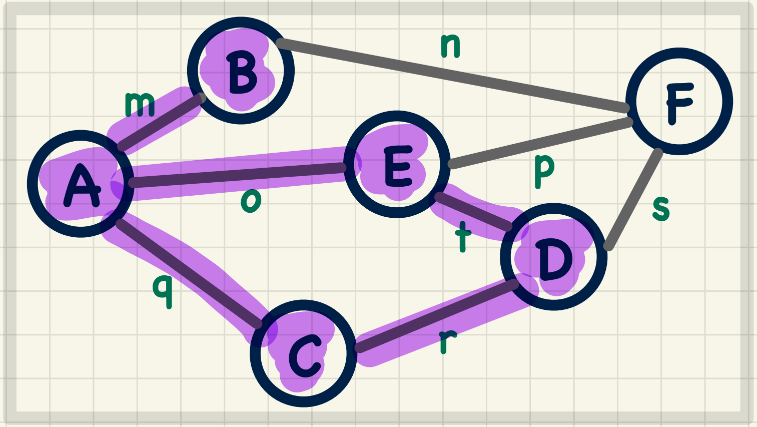
spanning ≠ connected

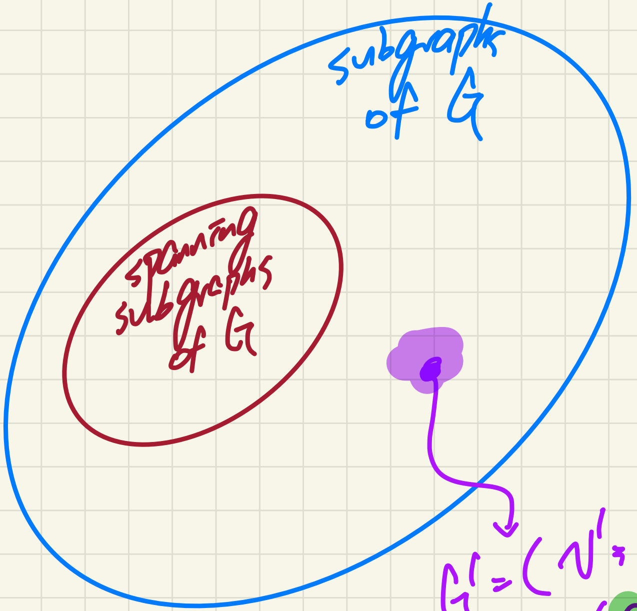
(1) $G_1' = (\underbrace{\{A, B, C, D, E, F\}}_{= V}, \underbrace{\emptyset}_{\subseteq E})$

(2) $G_2' = (\underbrace{\{A, B, C, D, E, F\}}_{= V}, \underbrace{\{m, o, p\}}_{\subseteq E})$

Graph: Subgraphs and Spanning Subgraphs

Formulate a condition of a graph G′ = ($\overset{\subseteq V}{V'}$, $\overset{\subseteq E}{E'}$) that is a subgraph, but **not** a spanning subgraph, of G = (V, E).





$G' = (V', E')$

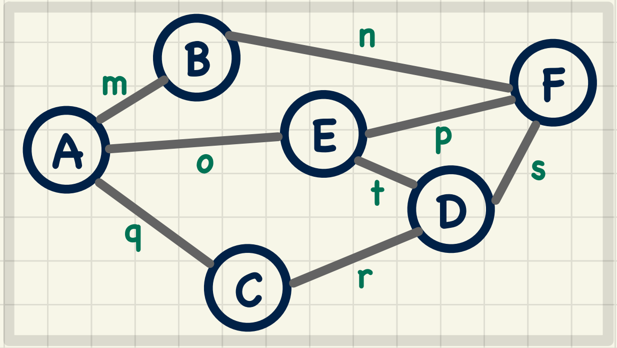
s.t. $V' \subset V \wedge E' \subseteq E$

Graph: Connected Graph

$G = (V, E)$

Connected$(G) \Leftrightarrow$

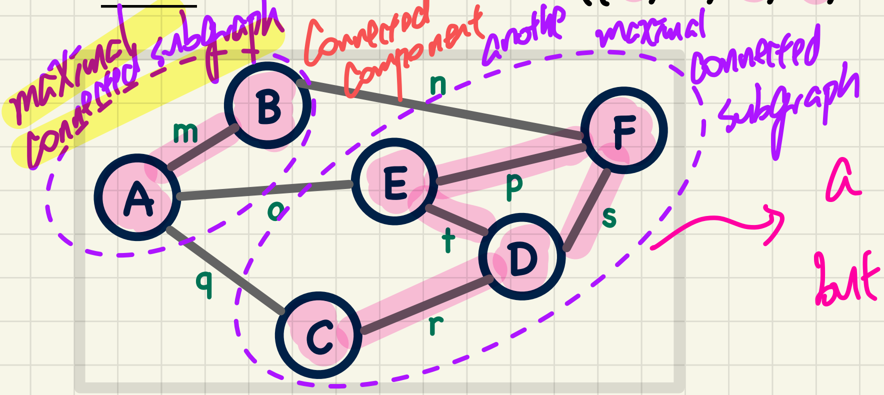
$\forall x, y \bullet x \in V \wedge y \in V \Rightarrow$ $\boxed{x \text{ is reachable}}$ from y.



only req. vertices to be covered, but not req. edges to build the necessary connections.

Is a spanning subgraph also connected subgraph?

Hint: Consider G2 = ({A, B, C, D, E, F}, {m, p, s, t, r})


maximal connected subgraph
Connected Component
another maximal connected subgraph
B
F
A
E
D
C
m
n
o
p
q
r
s
t

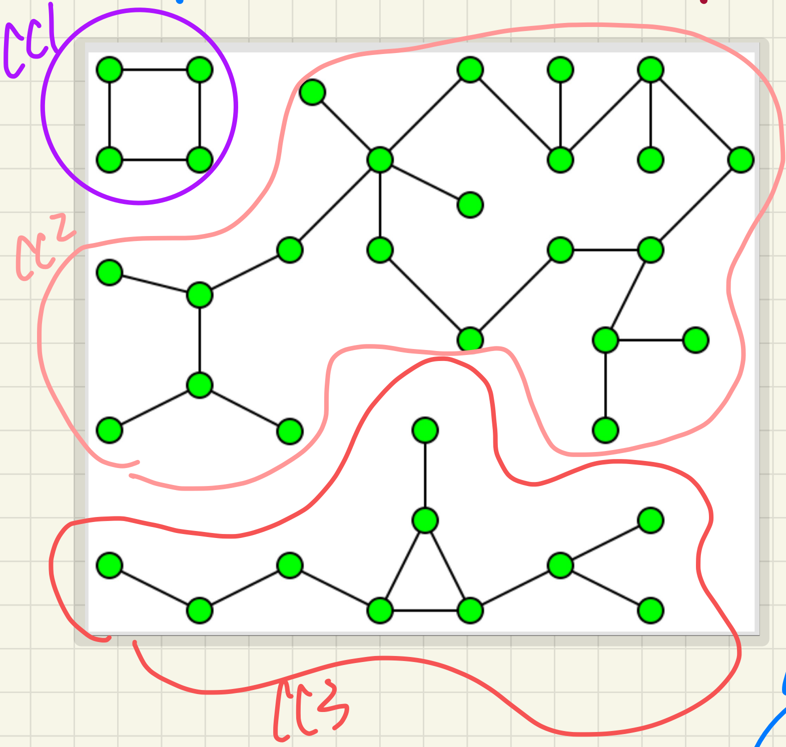
a spanning graph but not connected (i.e. no reachable path from B and F).

Connected Component of G

a maximal connected subgraph of G

no further extension is possible to make a larger connected subgraph

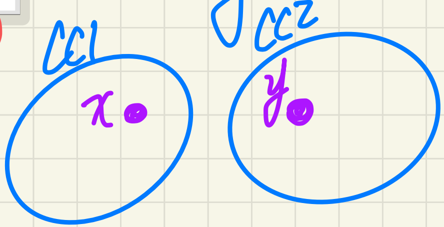
Graph: Connected Components



How many **connected components** does the graph have?

Between each pair of CCs, say CC1 and CC2,

$\forall x, y \cdot x$ is a vertex in CC1 $\wedge$ y is a vertex in CC2 $\Rightarrow$ x is not reachable from y



Lecture 10 - Oct 6

Graphs

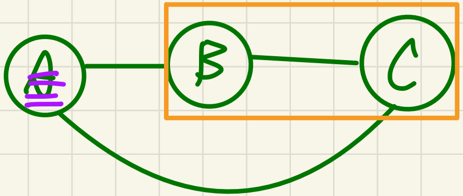
Forest vs. Tree vs. Spanning Tree
Graph Traversal: Depth-First Search (DFS)
DFS on a Tree vs. Pre-Order Traversal

Simple cycle:

A closed path that starts and ends at the same vertex and does not repeat any vertex or edge except for the starting/ending vertex.

simple path.

Example: A–B–C–A



Non-simple cycle:

A closed path that starts and ends at the same vertex but may repeat vertices or edges along the way.

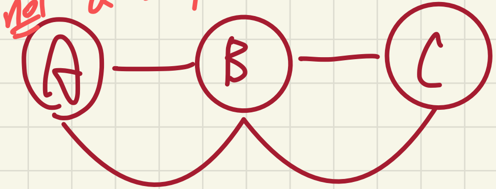
repeats.

→ not a simple path

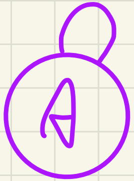
Example: A–B–C–B–A

cycle

⇒ not a simple cycle.



Self Loop



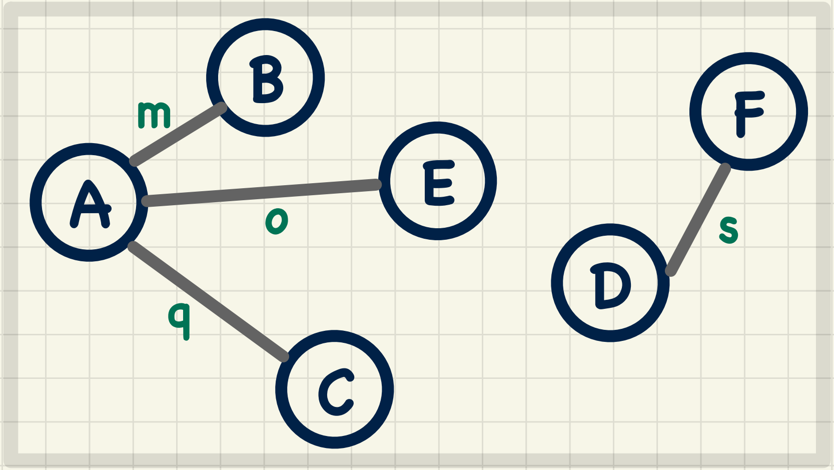
empty path

A–A

↳ simple path

cycle

Graph: Forests and Trees



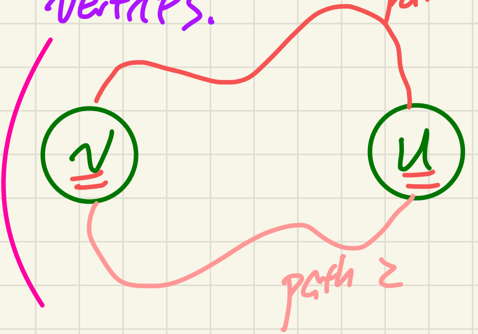
- undirected
- no cycle (acyclic)

Any two vertices are connected (v, u) via * at most one path.

≤ 1

What if $\boxed{\geq 2}$ edges connecting two vertices. >1

there's a cycle from v to v.



* Special case:

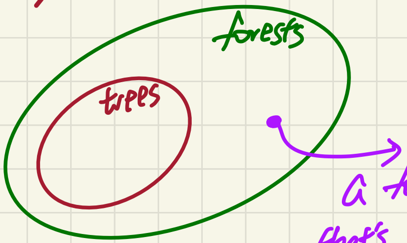
if u and v are connected by 0 edge ⇝ the graph is not connected.

special case of ≤ 1

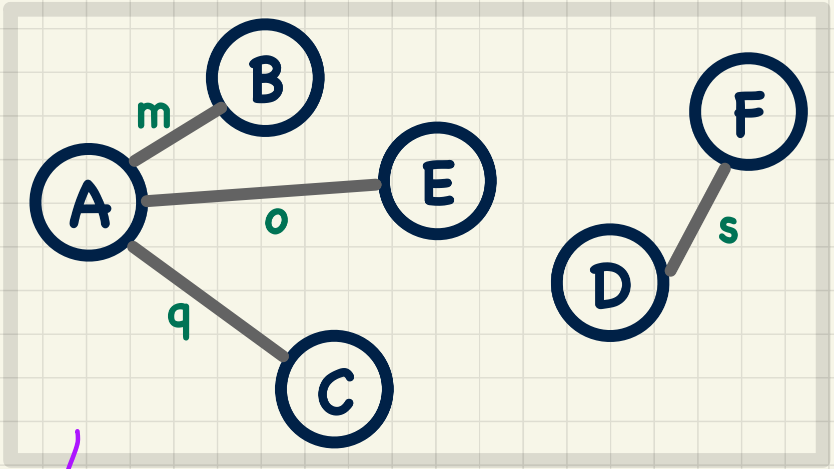
A forest may or may not be connected.

Graph: Forests and Trees

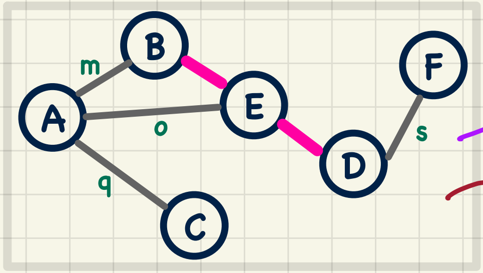
→ a connected forest.



a forest that's not connected.



↳ forest but not a tree.

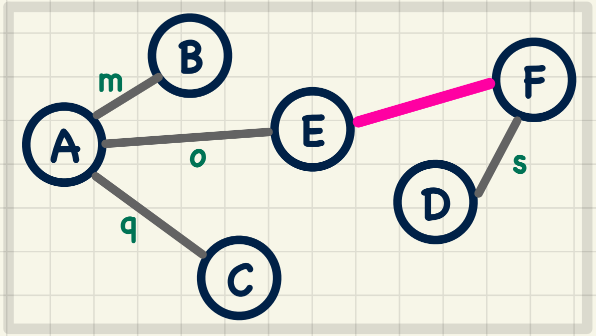


→ connected

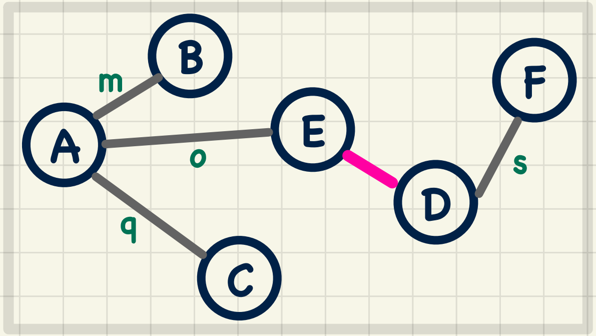
→ cyclic

↳ x forest

↳ x tree.



tree!



tree.

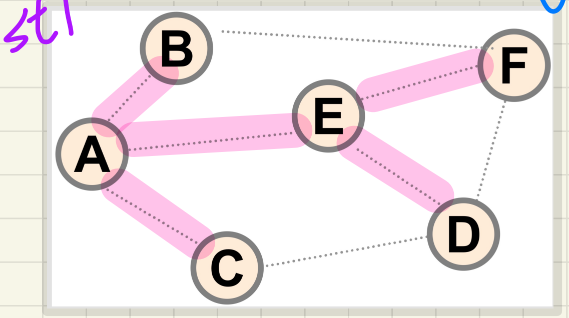
Graph: Spanning Trees

$\boxed{G \text{ is a spanning tree} \Rightarrow |E| = |V|-1}$

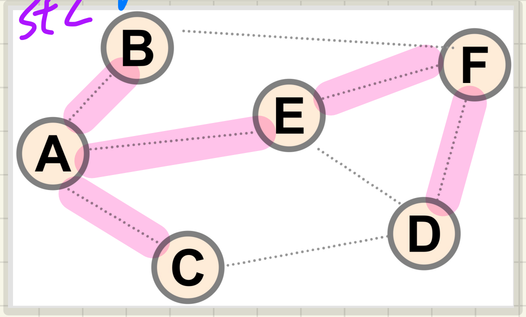
↳ a connected spanning subgraph that has $\boxed{\text{no}}$ cycle

↳ a spanning subgraph that is also a tree.

st1



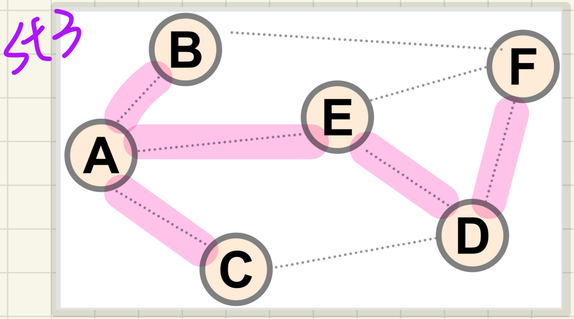
st2



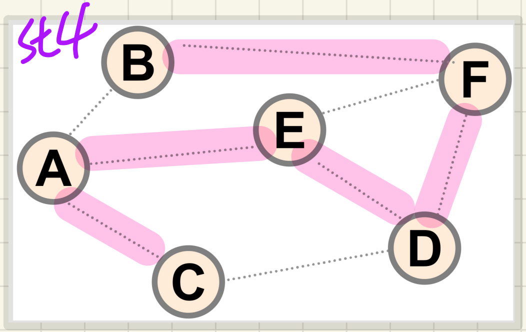
$|V| = 6$

$|E| = 5 \; (= |V| - 1)$

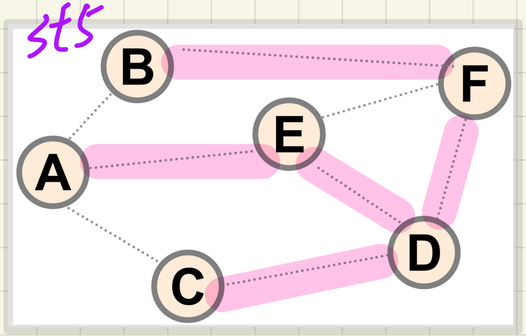
st3



st4



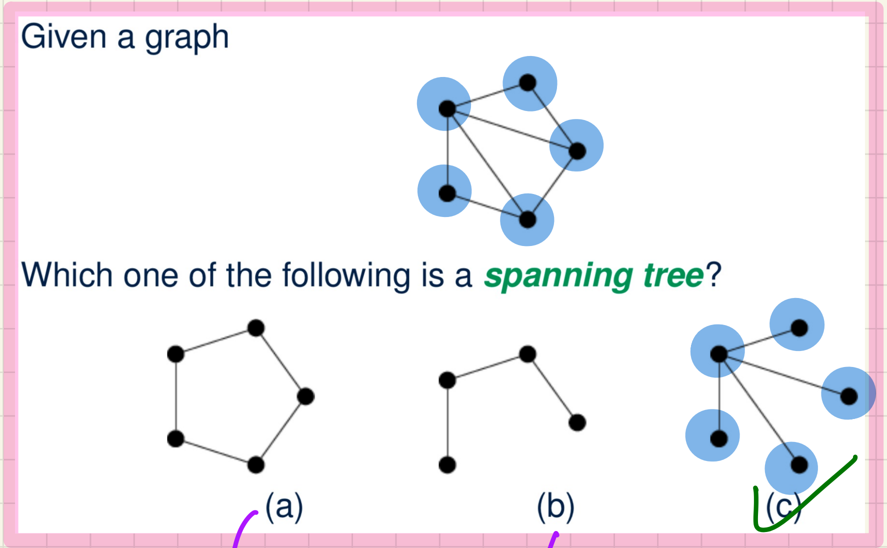
st5



Summary of Terms

	subgraph	spanning subgraph	forest	tree	spanning tree
undirected?	✓	✓	✓	✓	✓
acyclic?	? maybe maybe not	?	✓	✓	✓
connected?	?	?	?	✓	✓
spanning?	?	✓	?	?	✓

Graph: Exercises



Given a graph

Which one of the following is a ***spanning tree***?

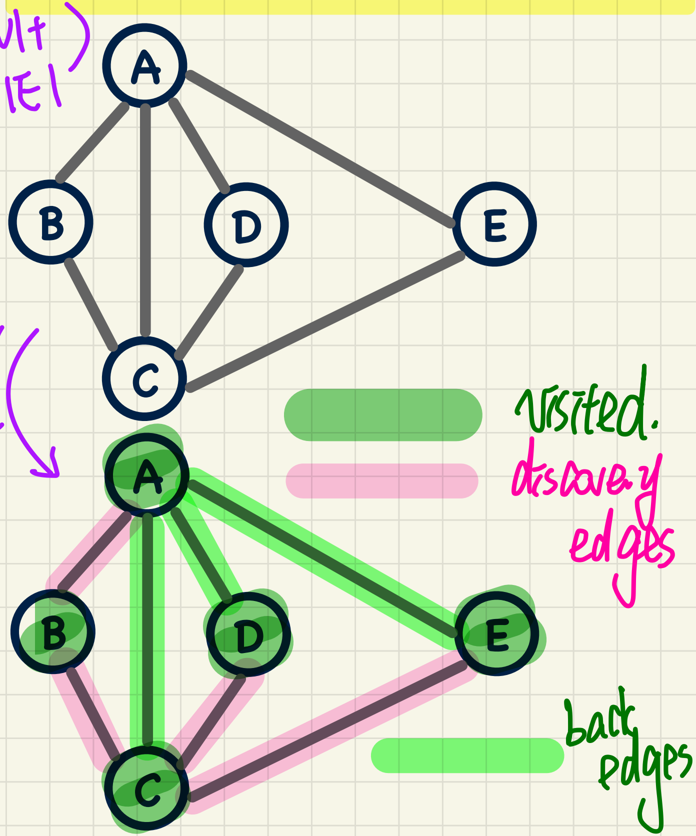
(a) spanning but cyclic.

(b) acyclic but not spanning.

Graph Traversals: Definition & Applications

Efficient Traversal of Graph G:

O(|V|+|E|)



Applications: (visit later).

- Reachable Vertices from v ∈ V
- A path between {u, v} ⊆ V
- The minimum path between {u, v} ⊆ V
- Is G connected?
- Compute a spanning tree of a connected G.
- Compute the connected components of G.
- If G is cyclic, return a cycle.

Graph Traversal: Depth-First Search (DFS)

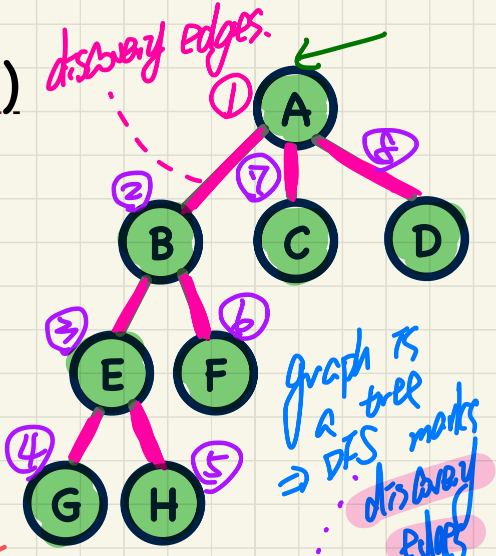
arbitrary.

e.g. A

A **Depth-First Search** (**DFS**) of graph G = (V, E), starting from some vertex *v* ∈ *V*, proceeds along a **path** from *v*.

- The **path** is constructed by following an *incident edge*.
- The **path** is extended **as far as possible**, until **all** *incident edges* lead to vertices that have already been *visited*.
- Once the **path** originated from *v* **cannot be extended further**, *backtrack* to the **latest** vertex whose *incident edges* lead to some *unvisited* vertices.

discovery edges.



graph is a tree ⇒ DFS marks discovery edges only.

Assumption: iterate through neighbours alphabetically.

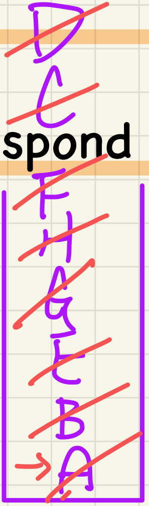
Q. When a graph is a tree,

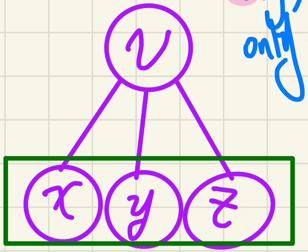
what kind of tree traversal does it correspond to?

pre-order (parent first, children next).

Q. What data structure should be used to keep track of the visited nodes?

↓ stack. (LIFO).

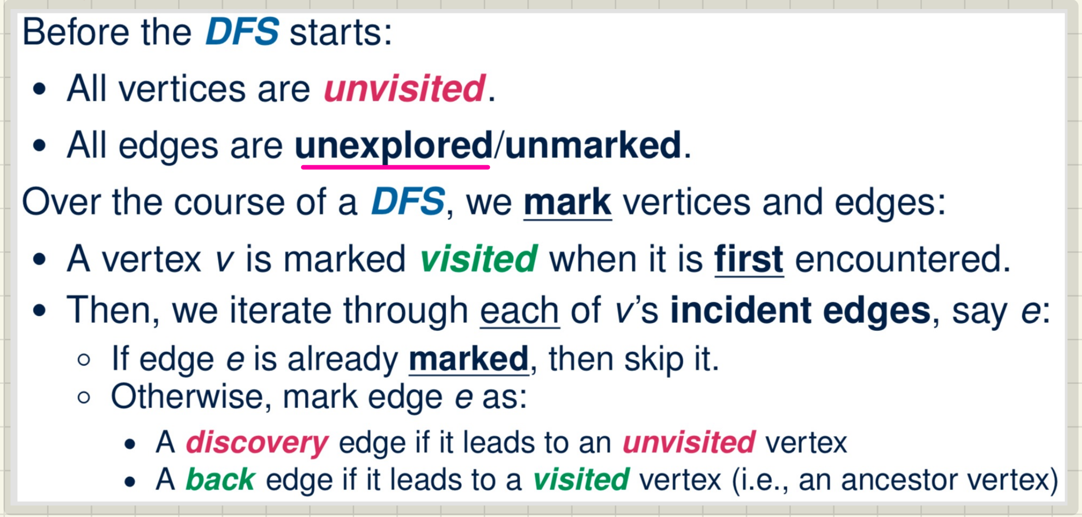




say all these neighbours have been visited.

↳ back track!

Depth-First Search (DFS): Marking Vertices & Edges



Before the *DFS* starts:

- All vertices are ***unvisited***.
- All edges are **unexplored**/**unmarked**.

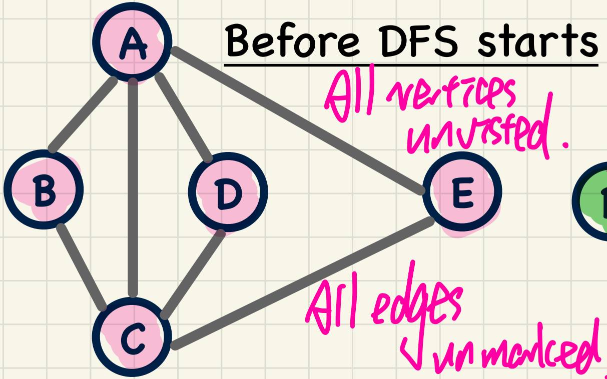
Over the course of a *DFS*, we **mark** vertices and edges:

- A vertex *v* is marked ***visited*** when it is **first** encountered.
- Then, we iterate through each of *v*'s **incident edges**, say *e*:
  - If edge *e* is already **marked**, then skip it.
  - Otherwise, mark edge *e* as:
    - A ***discovery*** edge if it leads to an ***unvisited*** vertex
    - A ***back*** edge if it leads to a ***visited*** vertex (i.e., an ancestor vertex)

the graph is cyclic

from C to A leading to vertex A that's already visited

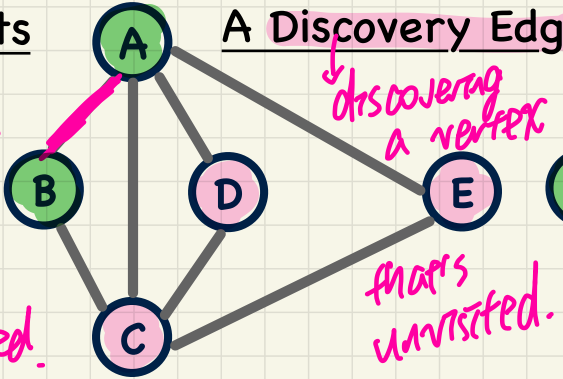
back edge



Before DFS starts

All vertices unvisited.

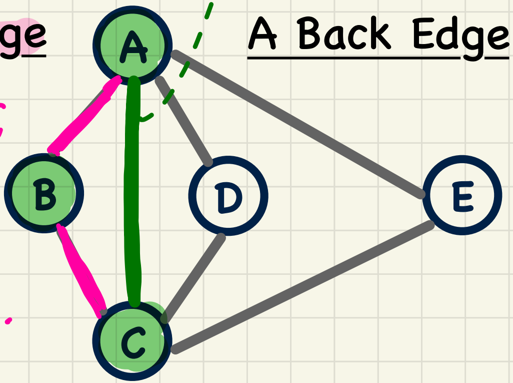
All edges unmarked.



A Discovery Edge

discovering a vertex

that's unvisited.



A Back Edge

Lecture 11 - Oct 8

Graphs

Proof: Spanning Tree and |V| vs. |E|
Tracing DFS using a Stack
Graphs in Java: Edge List

DFS

Vertex

- ↳ unvisited
- ↳ visited

Edge

- ↳ unmarked/unexplored
- ↳ discovery edge (leading to some unvisited vertex)
- ↳ back "edge" (leading "to some visited vertex)

Properties: Structure vs. |V| and |E|

Given $G = (V, G)$ an **undirected** graph with $|V| = n$, $|E| = m$:

$$\begin{cases} \boxed{m = n-1 \quad \text{if G is a } \textit{spanning tree}} \\ m \le n-1 \quad \text{if G is a } \textit{forest} \\ \underline{m \ge n-1} \quad \text{if G is } \textit{connected} \\ \boxed{m \ge n} \quad \text{if G contains a } \textit{cycle} \end{cases}$$

→ G is a spanning tree

⇐ ? ⇒

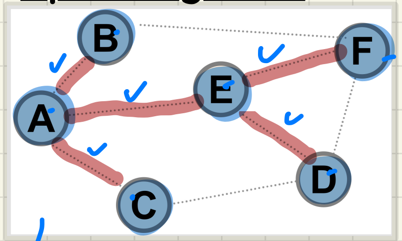
m = n − 1

? backs

|V| = 6
|E| = 4

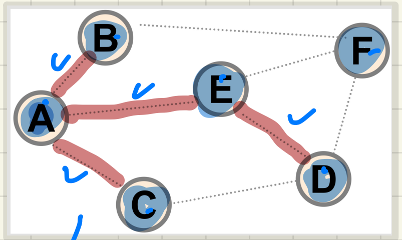
or may
may not have cycles.

Spanning Tree



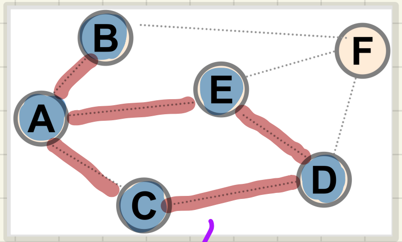
|V| = 6
|E| = 5 = |V| − 1

Forest



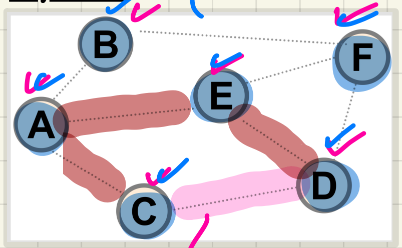
spanning but ¬ connected.
|V| = 6 |E| = 4 ≤ |V| − 1

Connected



Connected, ¬ spanning

Cyclic



more edges than just n−1 (≥ n).

Properties: Structure vs. |V| and |E|

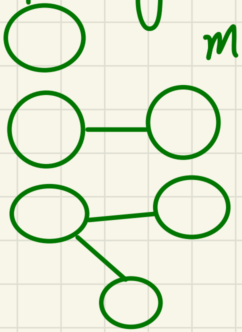
Given $G = (V, G)$ an **undirected** graph with $|V| = n$, $|E| = m$:

$$\begin{cases} m = n - 1 & \text{if G is a } \textit{spanning tree} \\ m \leq n - 1 & \text{if G is a } \textit{forest} \\ m \geq n - 1 & \text{if G is } \textit{connected} \\ m \geq n & \text{if G contains a } \textit{cycle} \end{cases}$$

I.H. G is a spanning tree ⇒ $|V| = n > 3$
$|E| = m$
s.t. $m = n - 1$

Mathematical Induction on $|V|$ $(|V| \geq 1)$

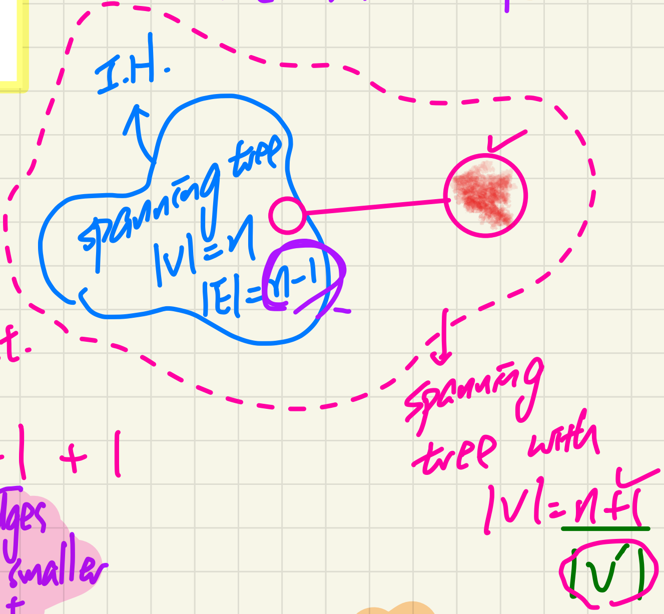
Base Cases: spanning trees with 1, 2, 3 vertices



$n = 1$ $m = 0$

$n = 2$ $m = 1$

$n = 3$ $m = 2$

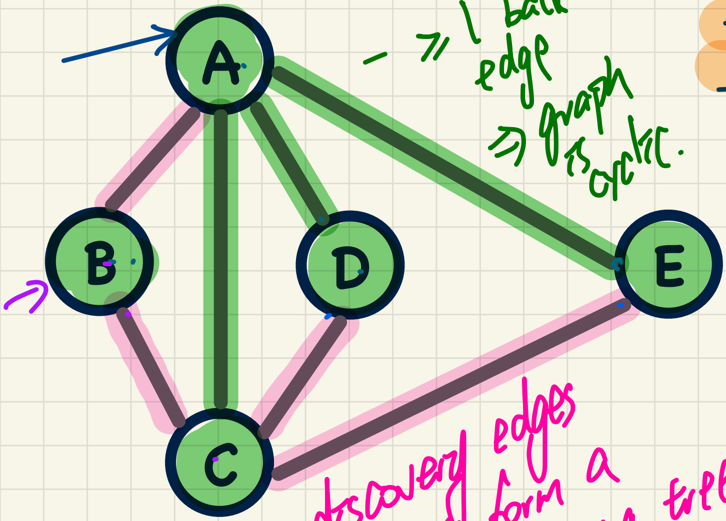


$$\overset{\text{\# edges in the larger s.t.}}{|E'|} = \underbrace{|E|}_{\text{\# edges in the smaller s.t.}} + 1 \overset{\text{I.H.}}{=} (n-1)+1 = n = |V'| - 1$$

Depth-First Search (DFS): Example 1 (a)

🟥 discovery edge
🟩 back edge.

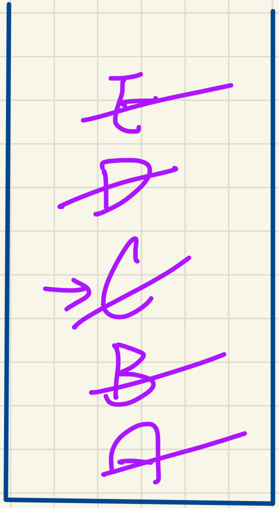
- start vertex of DFS : A
- order of visiting adjacent vertices.



- ⇒ 1 back edge ⇒ graph is cyclic.

- discovery edges happen to form a spanning tree.

	adjacent vertices			
A	~~B~~	C marked	~~D~~ marked	~~E~~ marked.
B	marked ~~A~~	~~C~~		
C	~~A.~~	marked ~~B~~	~~D~~	~~E.~~
D	~~A~~	~~C~~ marked.		
E	~~A~~	~~C~~ marked		



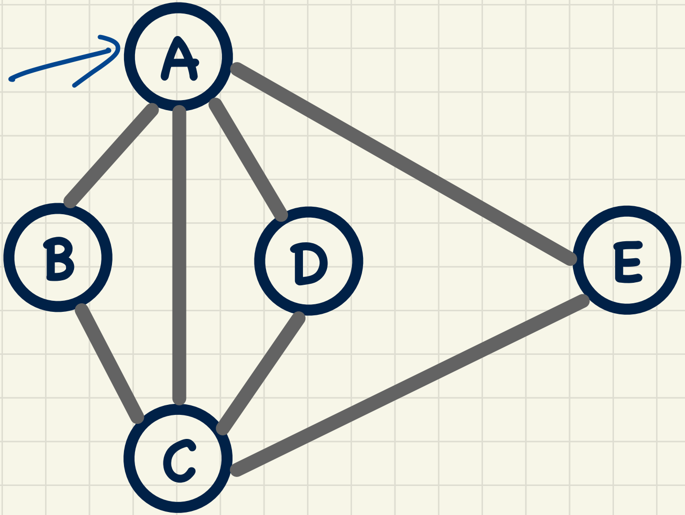
Assumptions:

- Adjacent vertices visited in alphabetic order

Questions

~ discovery edges ~ order of pushing vertices
~ back edges ~ order of popping off vertices: D E C B A

Depth-First Search (DFS): Example 1 (b)

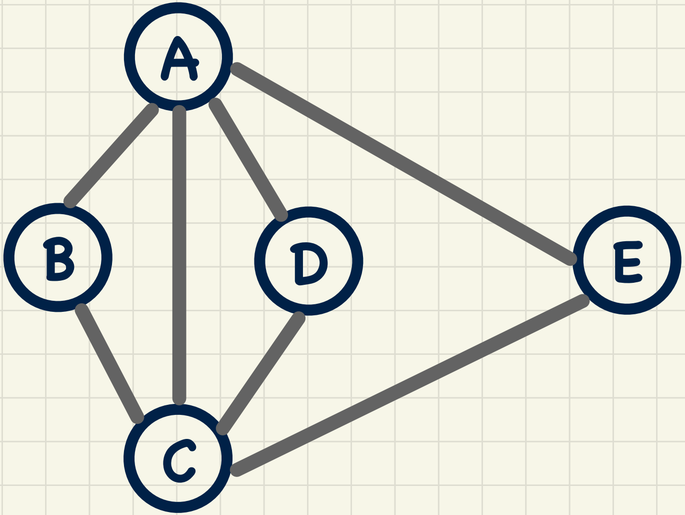


	adjacent vertices			
A	C	B	D	E
B	A	C		
C	A	B	D	E
D	A	C		
E	A	C		

Assumptions:

- ✓ Adjacent vertices visited in alphabetic order
- ✓ Exception: Edge AC visited first

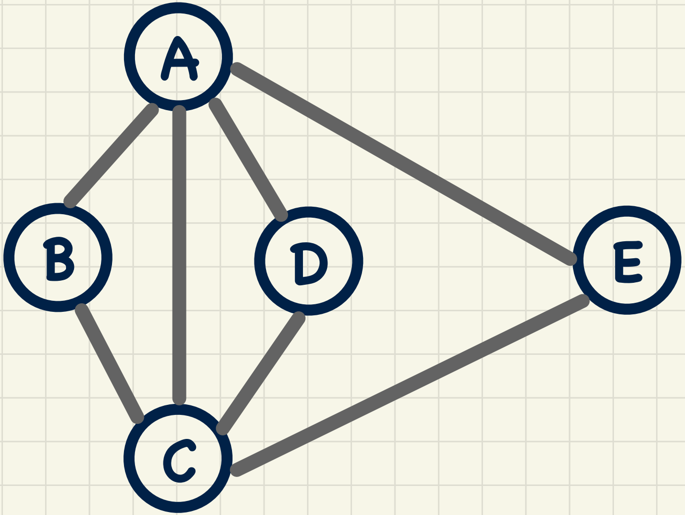
Depth-First Search (DFS): Example 1 (c)



Assumptions:

- Adjacent vertices visited in alphabetic order
- Exception: Edge AD visited first

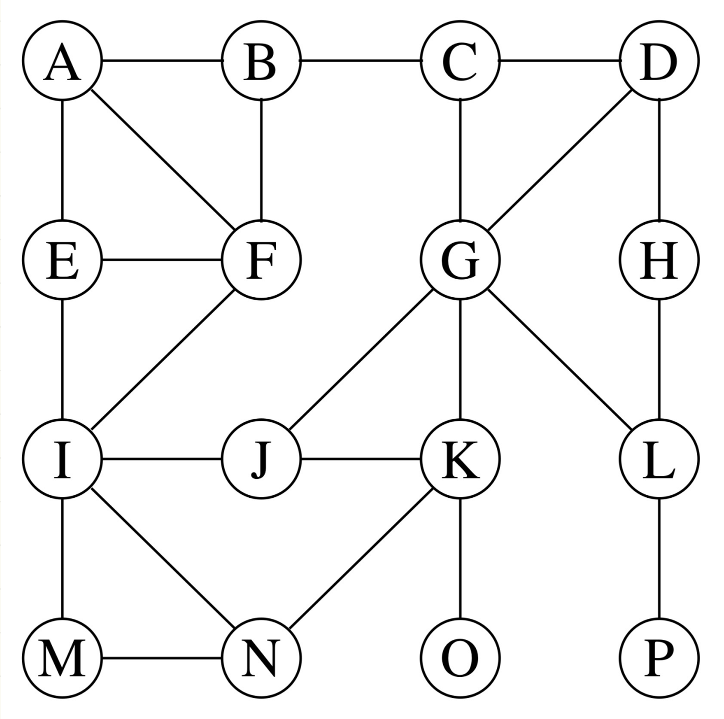
Depth-First Search (DFS): Example 1 (d)



Assumptions:

- Adjacent vertices visited in alphabetic order
- Exception: Edge AE visited first

Depth-First Search (DFS): Example 2



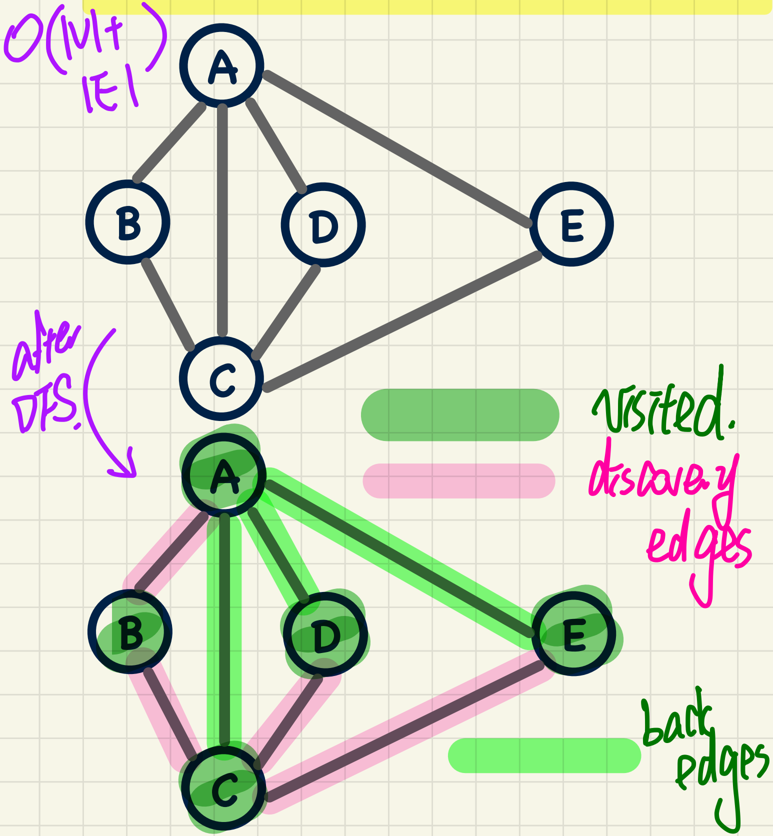
Assumptions:

- Adjacent vertices visited in alphabetic order

Graph Traversals: Definition & Applications

Efficient Traversal of Graph G:

O(|V| + |E|)

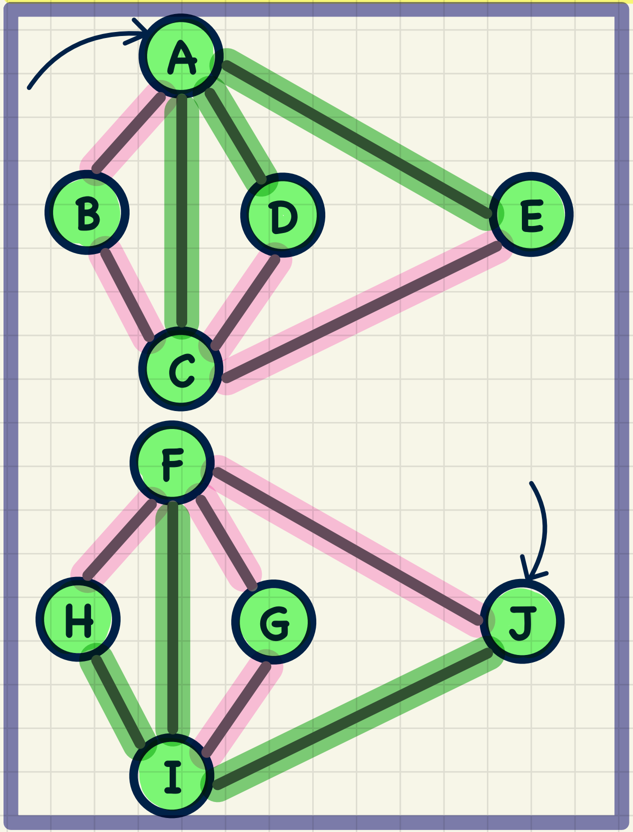


Applications: (visit later).

- Reachable Vertices from v ∈ V
- A path between {u, v} ⊆ V
- The minimum path between {u, v} ⊆ V
- Is G connected?
- Compute a spanning tree of a connected G.
- Compute the connected components of G.
- If G is cyclic, return a cycle.

Graph Traversals: Adapting DFS

Efficient Traversal of Graph G:



Graph Questions:

- Fina a **path** between {u, v} ⊆ V
- Is v **reachable** from v
- Find all **connected components** of G.
- Compute a **spanning tree** of a connected G.
- Is G **connected**?
- If G is cyclic, return a **cycle**.

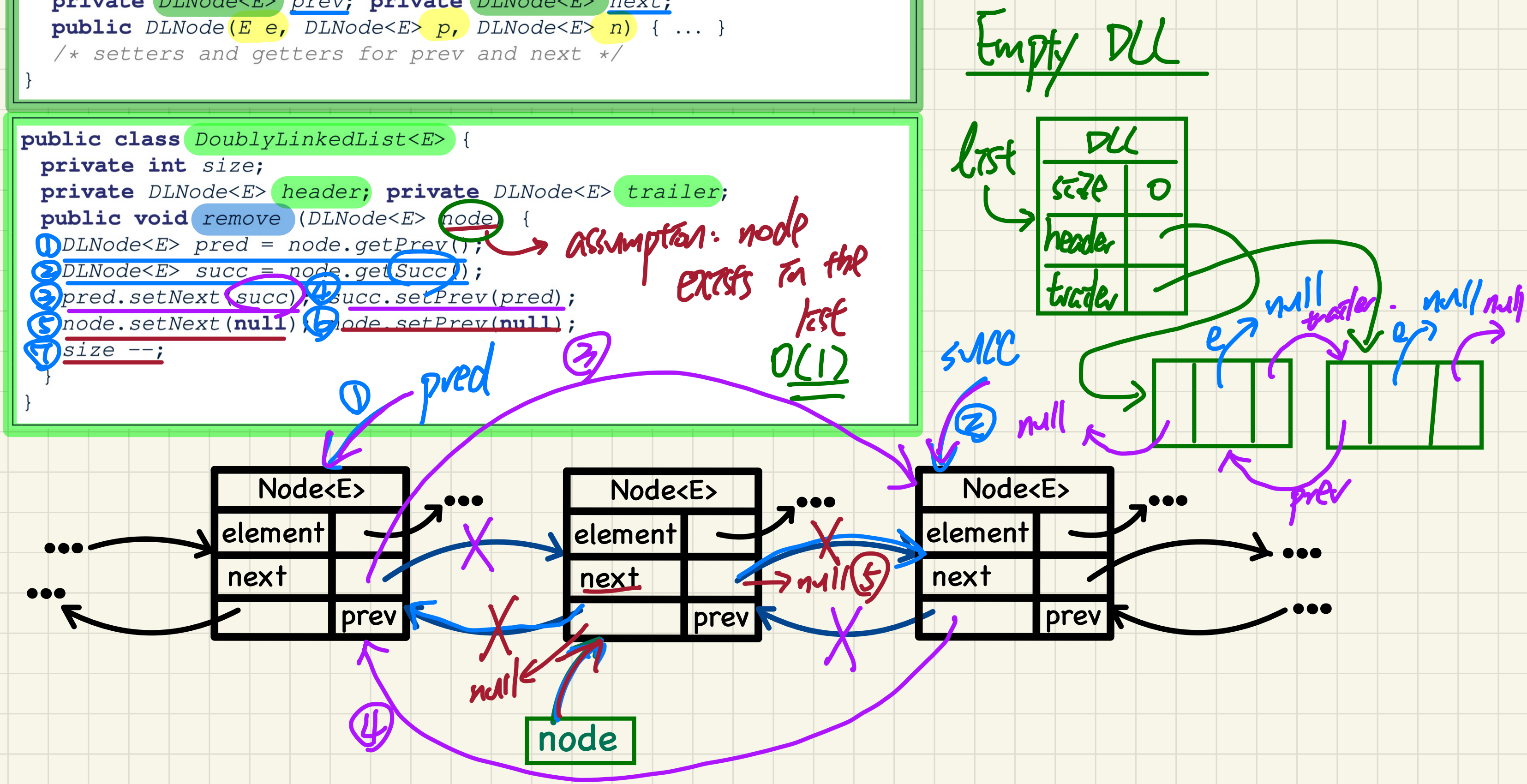
Graphs in Java: Doubly-Linked Nodes and Lists

```java
public class DLNode<E> { /* Doubly-Linked Node */
  private E element;
  private DLNode<E> prev; private DLNode<E> next;
  public DLNode(E e, DLNode<E> p, DLNode<E> n) { ... }
  /* setters and getters for prev and next */
}
```

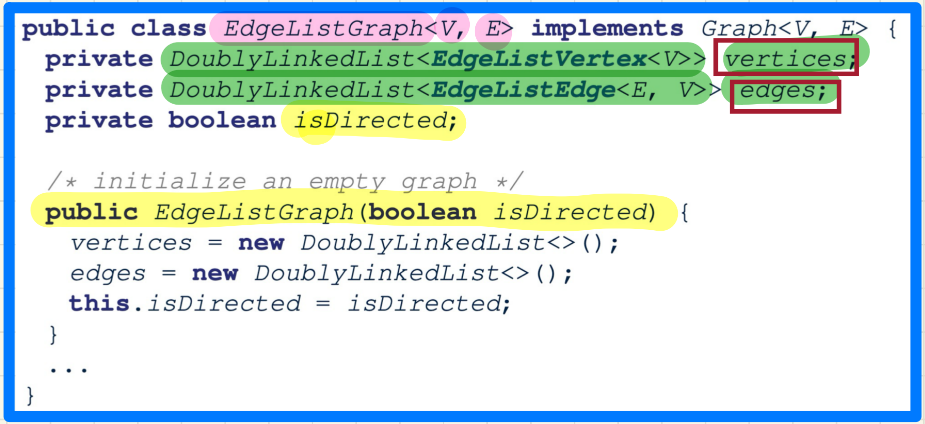
```java
public class DoublyLinkedList<E> {
  private int size;
  private DLNode<E> header; private DLNode<E> trailer;
  public void remove(DLNode<E> node) {
  ①DLNode<E> pred = node.getPrev();
  ②DLNode<E> succ = node.getSucc();
  ③pred.setNext(succ); ④succ.setPrev(pred);
  ⑤node.setNext(null); ⑥node.setPrev(null);
  ⑦size --;
  }
}
```

assumption: node exists in the list

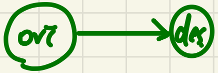
O(1)

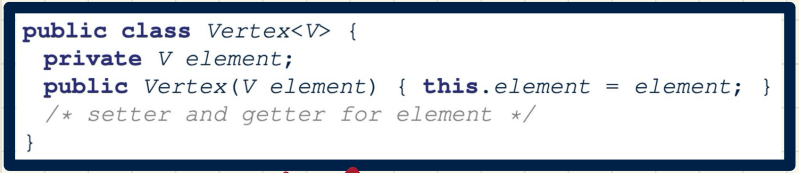


Graphs in Java: Edge List Strategy (1)

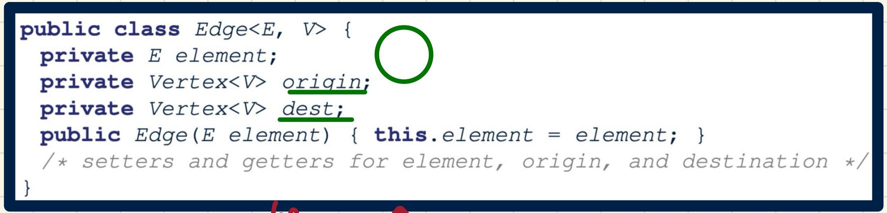


```java
public class EdgeListGraph<V, E> implements Graph<V, E> {
  private DoublyLinkedList<EdgeListVertex<V>> vertices;
  private DoublyLinkedList<EdgeListEdge<E, V>> edges;
  private boolean isDirected;

  /* initialize an empty graph */
  public EdgeListGraph(boolean isDirected) {
    vertices = new DoublyLinkedList<>();
    edges = new DoublyLinkedList<>();
    this.isDirected = isDirected;
  }
  ...
}
```





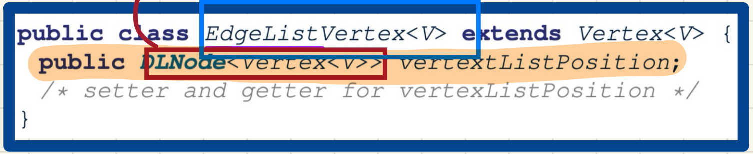
```java
public class Vertex<V> {
  private V element;
  public Vertex(V element) { this.element = element; }
  /* setter and getter for element */
}
```



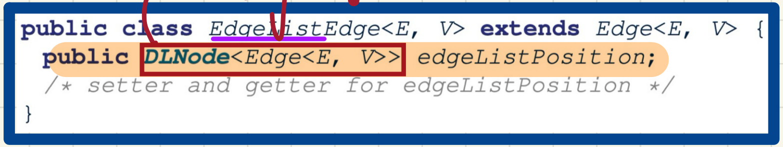
```java
public class Edge<E, V> {
  private E element;
  private Vertex<V> origin;
  private Vertex<V> dest;
  public Edge(E element) { this.element = element; }
  /* setters and getters for element, origin, and destination */
}
```

position in the vertex list.

position in the list of edges.

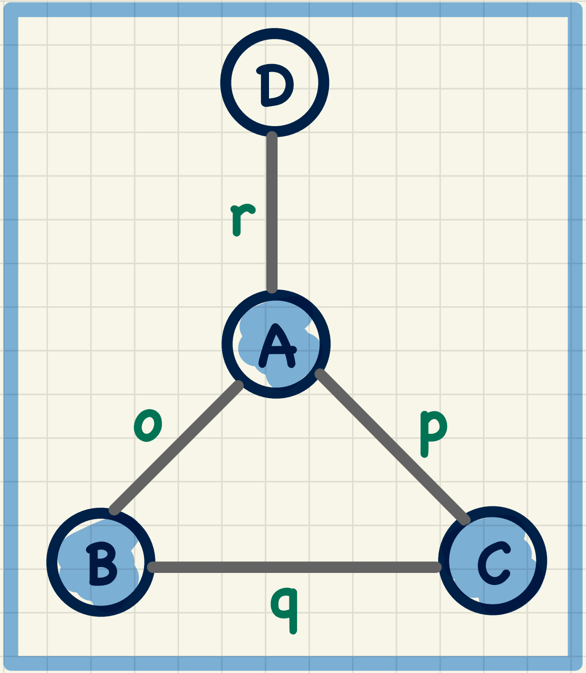


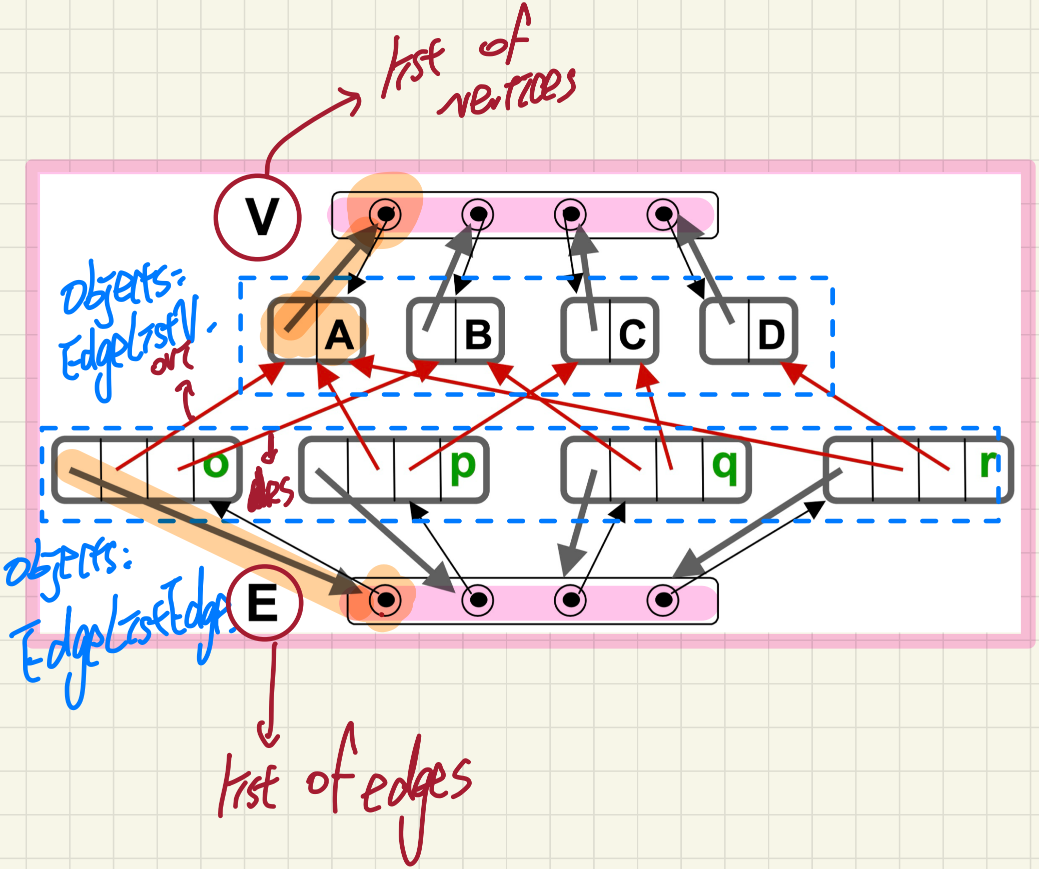
```java
public class EdgeListVertex<V> extends Vertex<V> {
  public DLNode<Vertex<V>> vertextListPosition;
  /* setter and getter for vertexListPosition */
}
```



```java
public class EdgeListEdge<E, V> extends Edge<E, V> {
  public DLNode<Edge<E, V>> edgeListPosition;
  /* setter and getter for edgeListPosition */
}
```

Graphs in Java: Edge List Strategy (2)





Lecture 12 - Oct 20

Graphs

Adapting DFS for Graph Questions
BFS: Marking Vertices and Edges
BFS: First Example on a Tree

Announcements/Reminders

- Today's class: notes template posted
- Assignment 1 due on Wednesday, October 22
- Test 1 next Monday, October 27:
  + Guide released not yet (Tuesday).
  + Review Session (slides, notes): Wednesday
  + Review Session (A1, more Q&A): Friday
- Tutorial Exercises so far:
  + Tutorial Week 1 (2D arrays)
  + Tutorial Week 2 (2D arrays, Proving Big-O)
  + Tutorial Week 3 (avg case analysis on doubling strategy)
  + Tutorial Week 4 (Trinode restructuring after deletions)

Test 1 (WSC, 4:30 PM to 5:20 PM)

Monday, Oct 27

Coverage

- Lecture materials (slides, notes, example code) up to and including Monday, October 20

slide 42 & 04-graph

- Tutorials 1 to 4
- Assignment 1

eClass: 100 marks

programming part: X cases

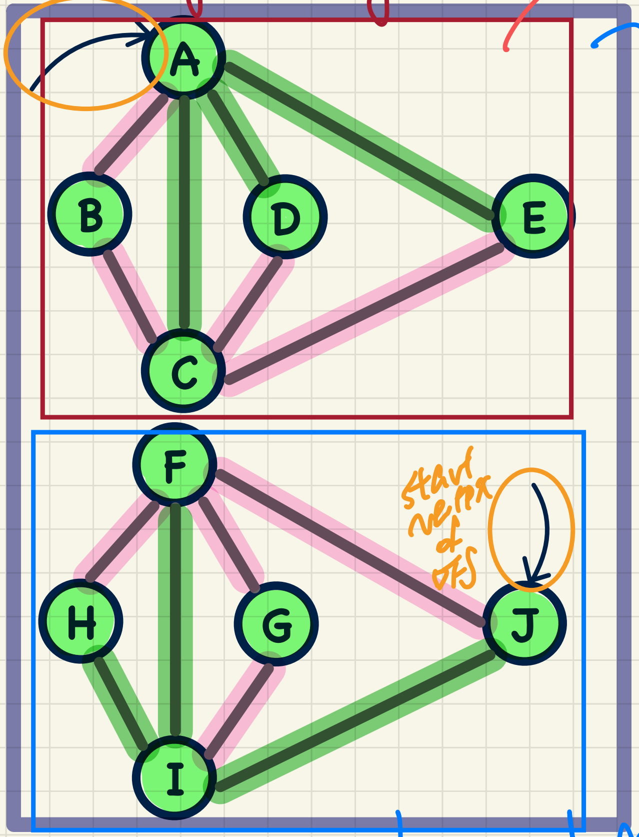
Format

- Programming Part (Eclipse):
  - Import a Java starter project (like A1)
  - Implement Java classes/methods to pass test cases
- Written Part (eClass):
  - Primarily MCQs
  - Written questions (e.g., short answers, justifications, proofs)

Duo Mobile

start vertex of DFS

subgraph traversed by 1st DFS



↓ connected component traversed by 2nd DFS.

- not connected
- >1 connected components

↳ multiple passes of traversal

↳ - DFS
  - BFS

a DFS starting at vertex A is guaranteed to visit all vertices in the C.C. that A belongs to

RT of DFS

$O(\underset{|V|}{m} + \underset{|E|}{n})$ ~> assumption getting a vertex's incident edges is $\boxed{O(1)}$

↳ not the case for edge list strategy

Graph Traversal

Assume G is connected

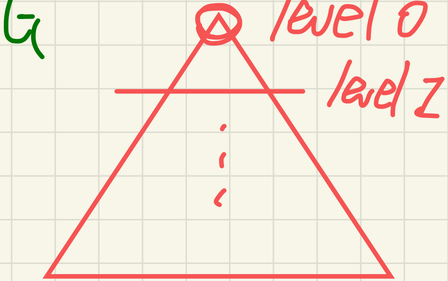
DFS tree

a DFS gives a spanning tree of G

a BFS gives a spanning tree of G

BFS tree,

level tree



If input graph G is <u>not</u> assumed to be connected.

```
boolean done = false;
while( !done. ){
    dfs(some unvisited vertex)
    if(# visited nodes so far = |V|){
        done = true
    }
}
```

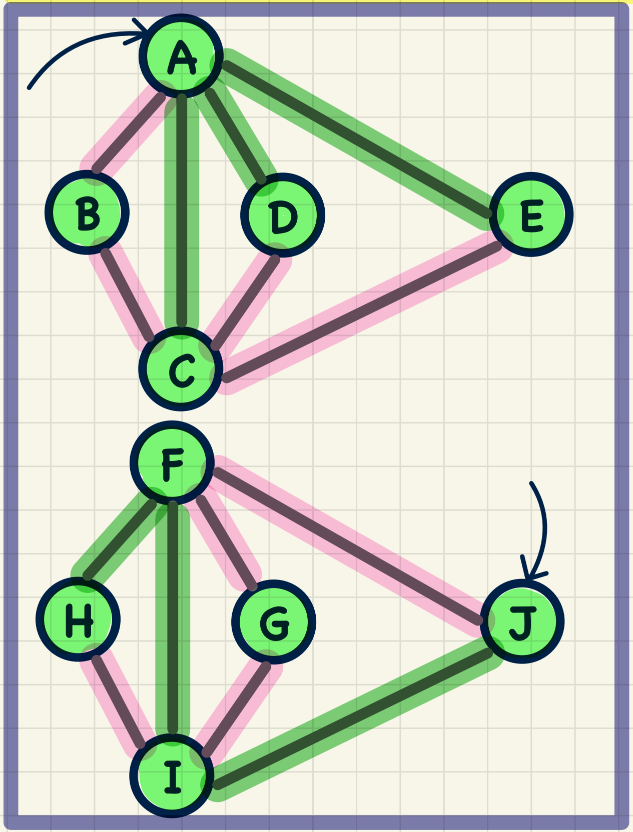
iterations
=
# connected components.

* a single DFS can return a single C.C.

Graph Traversals: Adapting DFS

4. if v is never encountered, then path does not exist!

Efficient Traversal of Graph G:



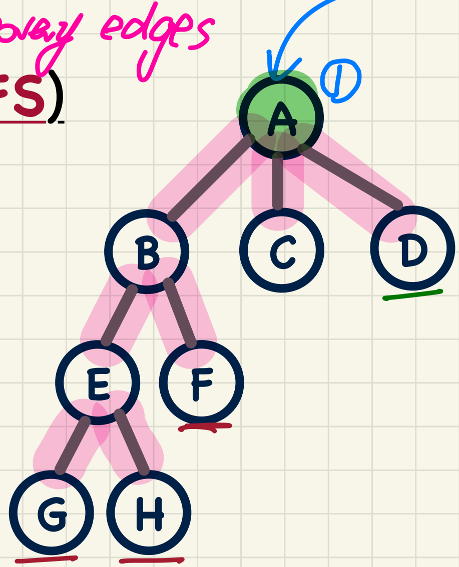
Graph Questions:

- Fina a path between {u, v} ⊆ V the list so far.
  1. start a DFS from u. 3. if v is visited, return
  2. Maintain a list to store the visited vertices
- Is u reachable from v
  similar to calculating a path
- * Find all connected components of G.
- Compute a spanning tree of a connected G.
  ↳ # discovery edges = # edges in G
  −1
- Is G connected?
  ↳ # visited nodes $\overset{?}{=}$ |V|
- If G is cyclic, return a cycle.
  ↳ return the path that leads to a back edge.

Graph Traversal: Breadth-First Search (BFS)

A ***breadth-first search (BFS)*** of graph G = (V, E), starting from some vertex *v* ∊ *V*:

- Visits every vertex ***adjacent*** to *v* before visiting any other (**more distant**) vertices
  - ***BFS*** attempts to stay as **close** as possible, whereas ***DFS*** attempts to move as **far** as possible
  - ***BFS*** proceeds in rounds and divides the vertices into ***levels***
- **No backtracking** in ***BFS***: it is completed **as soon as** the **most distant level** of vertices from the start vertex *v* are visited.



Q

before we dequeue A, enqueue all A's adjacent vertices

~~A B C D E F G H~~

adjacent vertices of A

adjacent vertices of B

adjacent vertices of E

Q. What data structure should be used to keep track of the visited nodes?

FIFO queue

Breadth-First Search (BFS): Marking Vertices & Edges

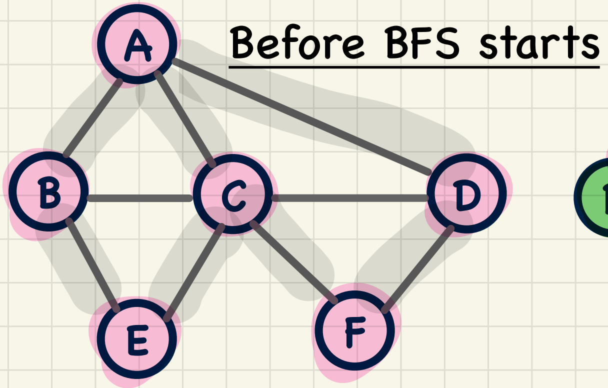
Before the ***BFS*** starts:

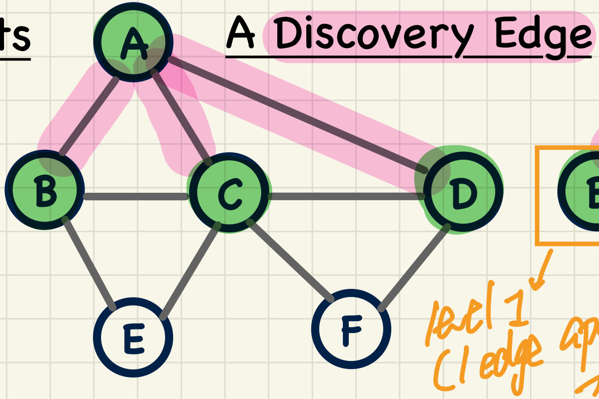
- All vertices are ***unvisited***.
- All edges are **unexplored/unmarked**.

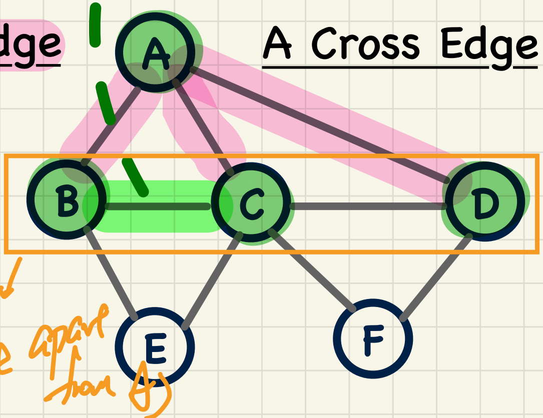
Over the course of a ***BFS***, we **mark** vertices and edges:

- A vertex is marked ***visited*** when it is **first** encountered.
- Then, we iterate through each of v's **incident edges**, say *e*:
  - If edge *e* is already **marked**, then skip it.
  - Otherwise, for an *undirected* graph, an edge is marked as:
    - A ***discovery*** edge if it leads to an ***unvisited*** vertex
    - A ***cross*** edge if it leads to a ***visited*** vertex (i.e., from a different ***branch*** at the same ***level***).

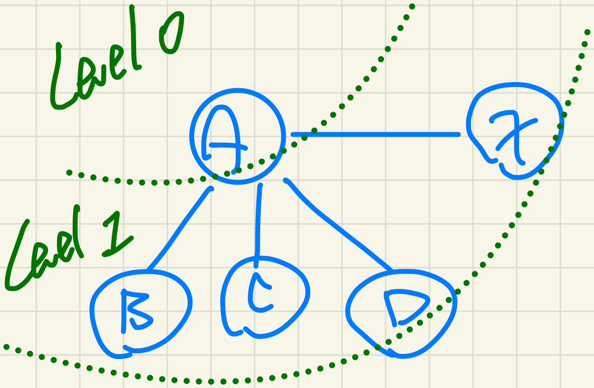
Cross edge: connecting nodes at the same level


Before BFS starts


A Discovery Edge


A Cross Edge

level 1 (1 edge apart from A)


Level 0
Level 1
A
X
B
C
D

Lecture 13 - Oct 22

Graphs

Test 1 Review

Announcements/Reminders

- Today's class: notes template posted
- Assignment 1 due on Wednesday, October 22
- Test 1 next Monday, October 27:
  - + Guide released not yet (Tuesday).
  - + Review Session (slides, notes): Wednesday
  - + Review Session (A1, more Q&A): Friday
- Tutorial Exercises so far:
  - + Tutorial Week 1 (2D arrays)
  - + Tutorial Week 2 (2D arrays, Proving Big-O)
  - + Tutorial Week 3 (avg case analysis on doubling strategy)
  - + Tutorial Week 4 (Trinode restructuring after deletions)

- A1 (Wed).

- 50 minutes

  Monday, Oct 27.

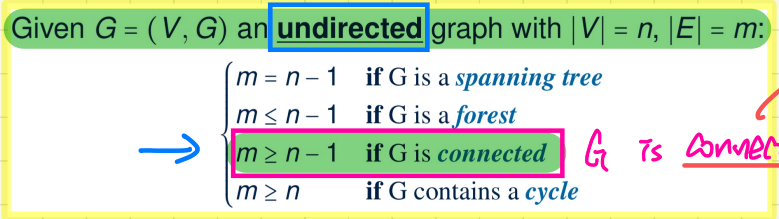
  4:30pm ~ 5:20pm (50 minutes)

  WSC

- 04-Graph.pdf

  ↳ up to and including slide 42.

Properties: Structure vs. |V| and |E|



Given $G = (V, G)$ an **undirected** graph with $|V| = n$, $|E| = m$:

$$\begin{cases} m = n - 1 & \text{if G is a } \textit{spanning tree} \\ m \leq n - 1 & \text{if G is a } \textit{forest} \\ \boxed{m \geq n - 1 \quad \text{if G is } \textit{connected}} \\ m \geq n & \text{if G contains a } \textit{cycle} \end{cases}$$

G is connected ⇒ $\underset{|E|}{m} \geq \underset{|V|}{n - 1}$

* new graph is connected and $|E'| \geq |V'| - 1$

every pair of vertices has some path connecting between them then $k+1$

↳ if there is no pair of vertices with no path between → it's connected.

In general, given a graph property,
to prove it: mathematical induction
to disprove it: give a witness graph G' that violates the property

Exercise: show that this is true (via I.H.).

Prove by induction on value of $|V| = n \geq 0$

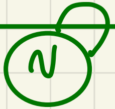
1. Base Cases

$\overset{n}{|V|} = 0$ empty graph.

$\overset{m}{|E|} = 0$ ↳ no violation pair of vertices can be found.

$\underset{0}{m} \geq \underset{0}{n} - 1$

$\overset{n}{|V|} = 1$ $\overset{m}{|E|} = 1$



$1 \geq (1-1)$

2. I.H.

For a undirected graph with k vertices ($|V| = k$, $k \geq 1$) if graph is connected, then $|E| \geq k - 1$

3. Inductive Case.

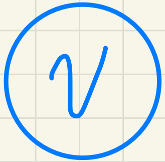
I.H.
k vertices
$|E| \geq k-1$
& $k \geq 1$
connected

x

Create a larger graph with $k+1$ vertices, s.t. *

one-vertex graph

connected.



say it's connected

↳ $|V| = 1$

$|E| = 0$

$0 \geq 1-1$ ✓.

```
try {

    g.m(v1);          throws SE.

}
catch ( SomeException  e ) {
  → fail();
}
```

fail the test if SomeException occurred

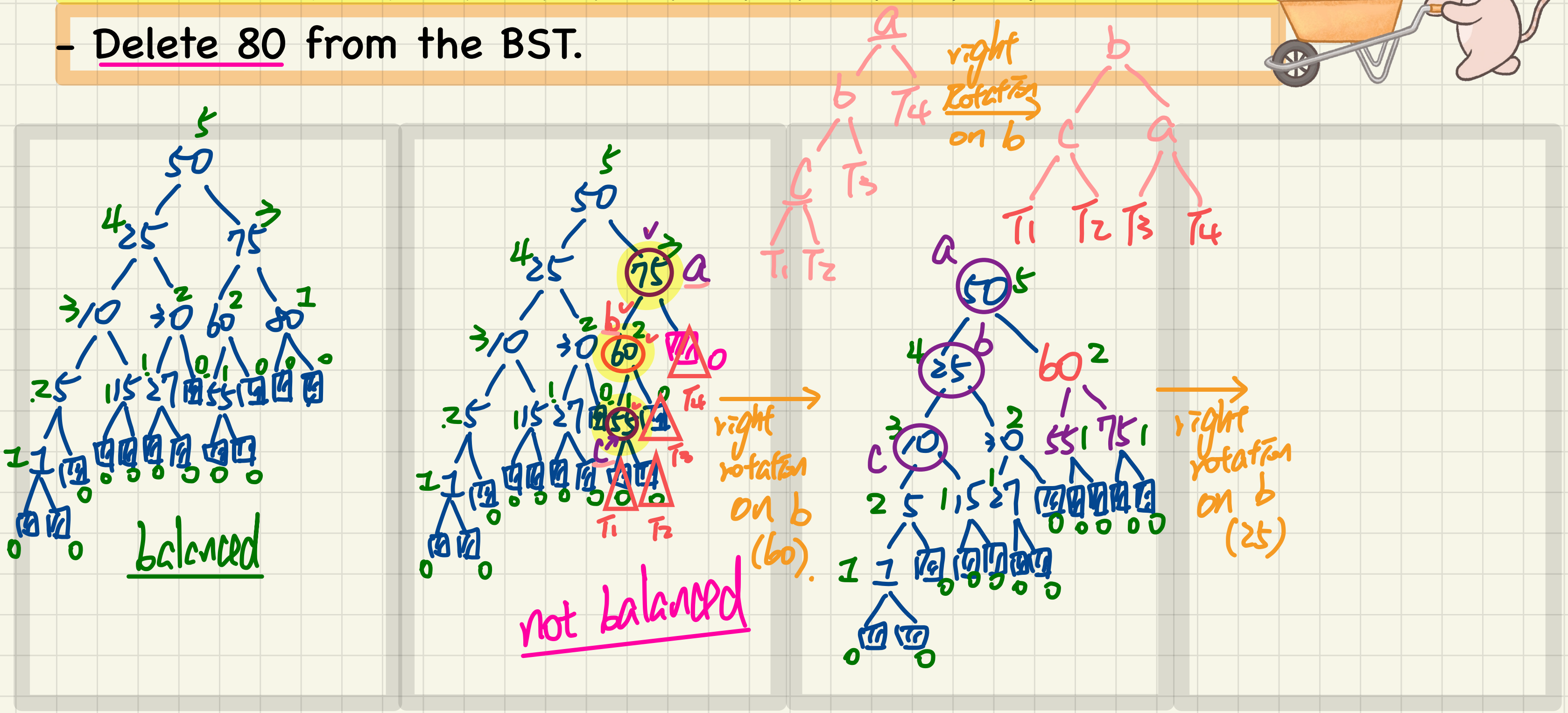
↳ not expecting S.E. when calling m on v1.

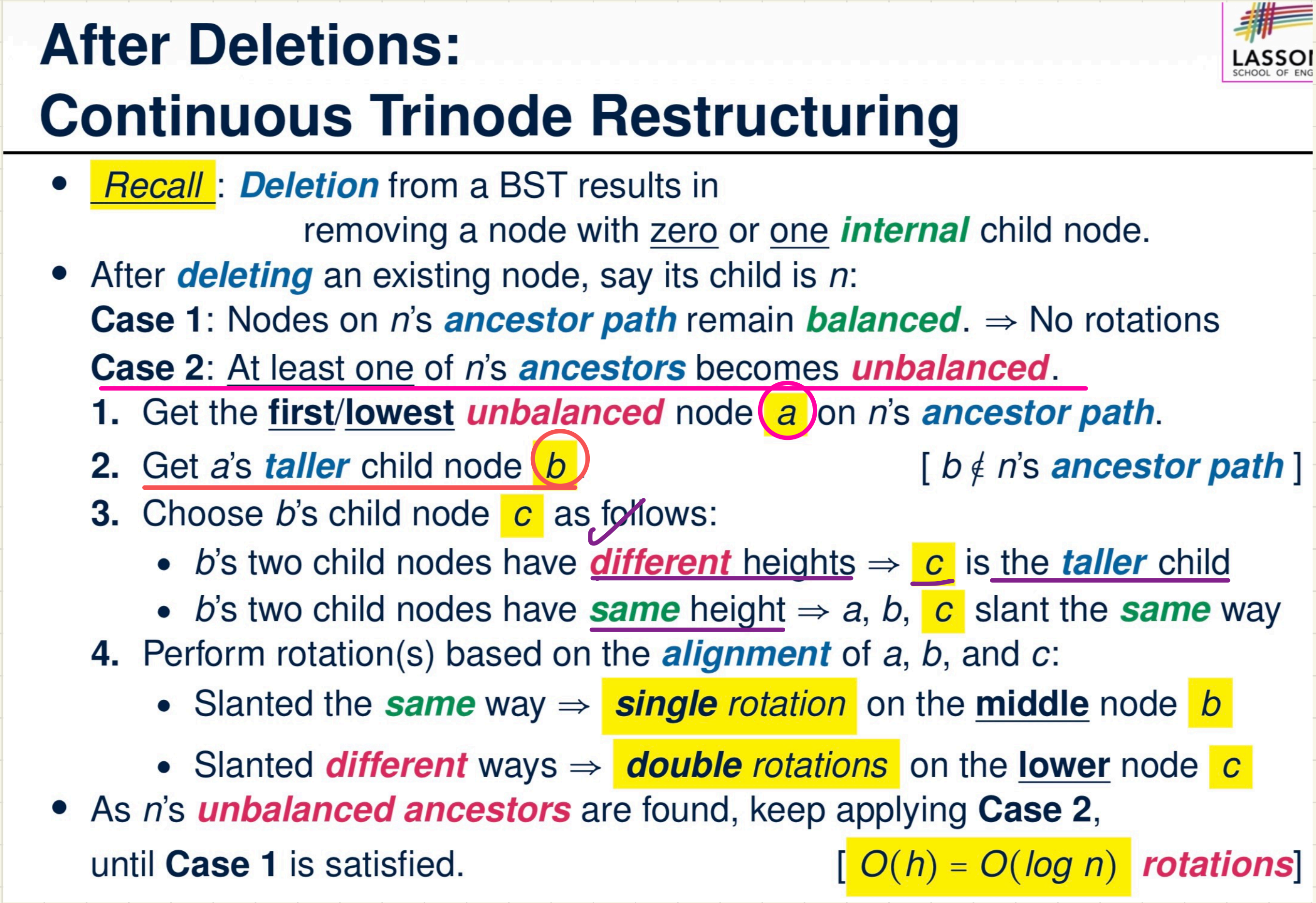
```
try {
                      ~> S.E. not thrown
    g.m(v2);
    fail();           ~> fail the test if S.E. did not occur.
}
catch ( SomeException  e ) {

}
```

expecting S.E. when calling m on v2.

Trinode Restructuring after Deletion: Multiple Rotations

- Insert the following sequence of **keys** into an empty BST:
  <50, 25, 10, 30, 5, 15, 27, 1, 75, 60, 80, 55>
- Delete 80 from the BST.





After Deletions: Continuous Trinode Restructuring

- *Recall*: ***Deletion*** from a BST results in removing a node with zero or one ***internal*** child node.
- After ***deleting*** an existing node, say its child is *n*:

  Case 1: Nodes on *n*'s ***ancestor path*** remain ***balanced***. ⇒ No rotations

  Case 2: At least one of *n*'s ***ancestors*** becomes ***unbalanced***.

  1. Get the **first**/**lowest** ***unbalanced*** node *a* on *n*'s ***ancestor path***.
  2. Get *a*'s ***taller*** child node *b* [$b \notin$ *n*'s ***ancestor path***]
  3. Choose *b*'s child node *c* as follows:
     - *b*'s two child nodes have ***different*** heights ⇒ *c* is the ***taller*** child
     - *b*'s two child nodes have ***same*** height ⇒ *a*, *b*, *c* slant the ***same*** way
  4. Perform rotation(s) based on the ***alignment*** of *a*, *b*, and *c*:
     - Slanted the ***same*** way ⇒ ***single rotation*** on the **middle** node *b*
     - Slanted ***different*** ways ⇒ ***double rotations*** on the **lower** node *c*
- As *n*'s ***unbalanced ancestors*** are found, keep applying **Case 2**, until **Case 1** is satisfied. [$O(h) = O(\log n)$ ***rotations***]

Tutorials - Week 7 - Oct 24

Test 1 Review

Assignment 1 Solution Walkthrough

Strategy (50 minutes)

- ≤ 25 minutes

4:30 pm

4:55 pm -

Programming Part

↓ partially completed projects with no compilation error

↳ export & submit.

- 25 minutes → eClass

→ start with 1st minute navigating through the questions.

- leave 5~10 minutes

↳ check answers

↳ go back to Eclipse.

1. Programming Part

↳ No extra class (not required by the JUnit tests)

↳ No modifications on "base" classes

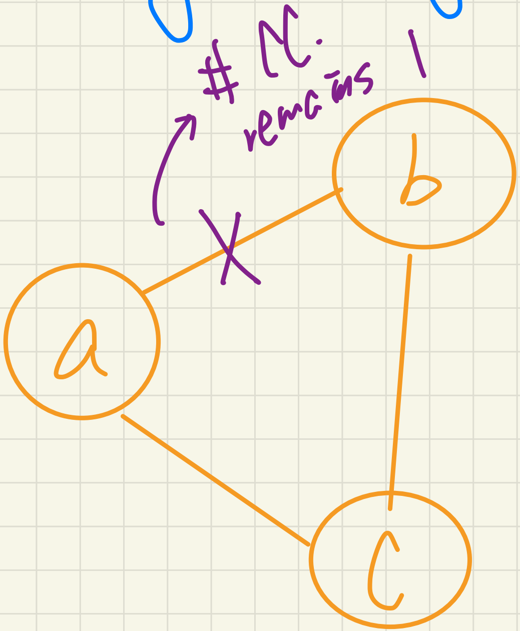
↳ OK to add extra helper methods and attributes.

Starter tests may not cover all edge cases.

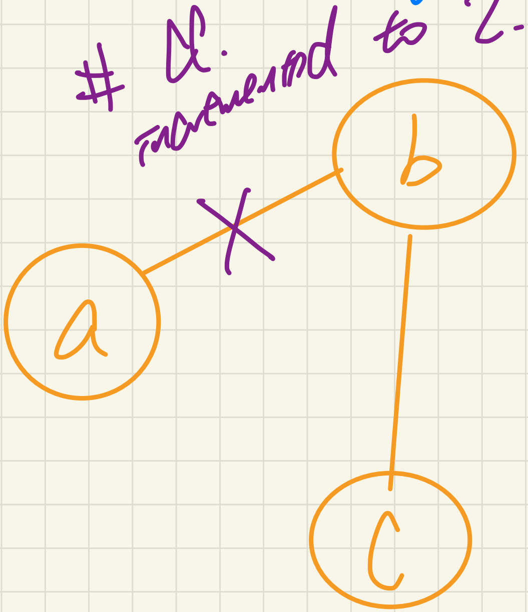
↳ make sure to program your methods generally.

Assume: graph for programming part is undirected (like A1)

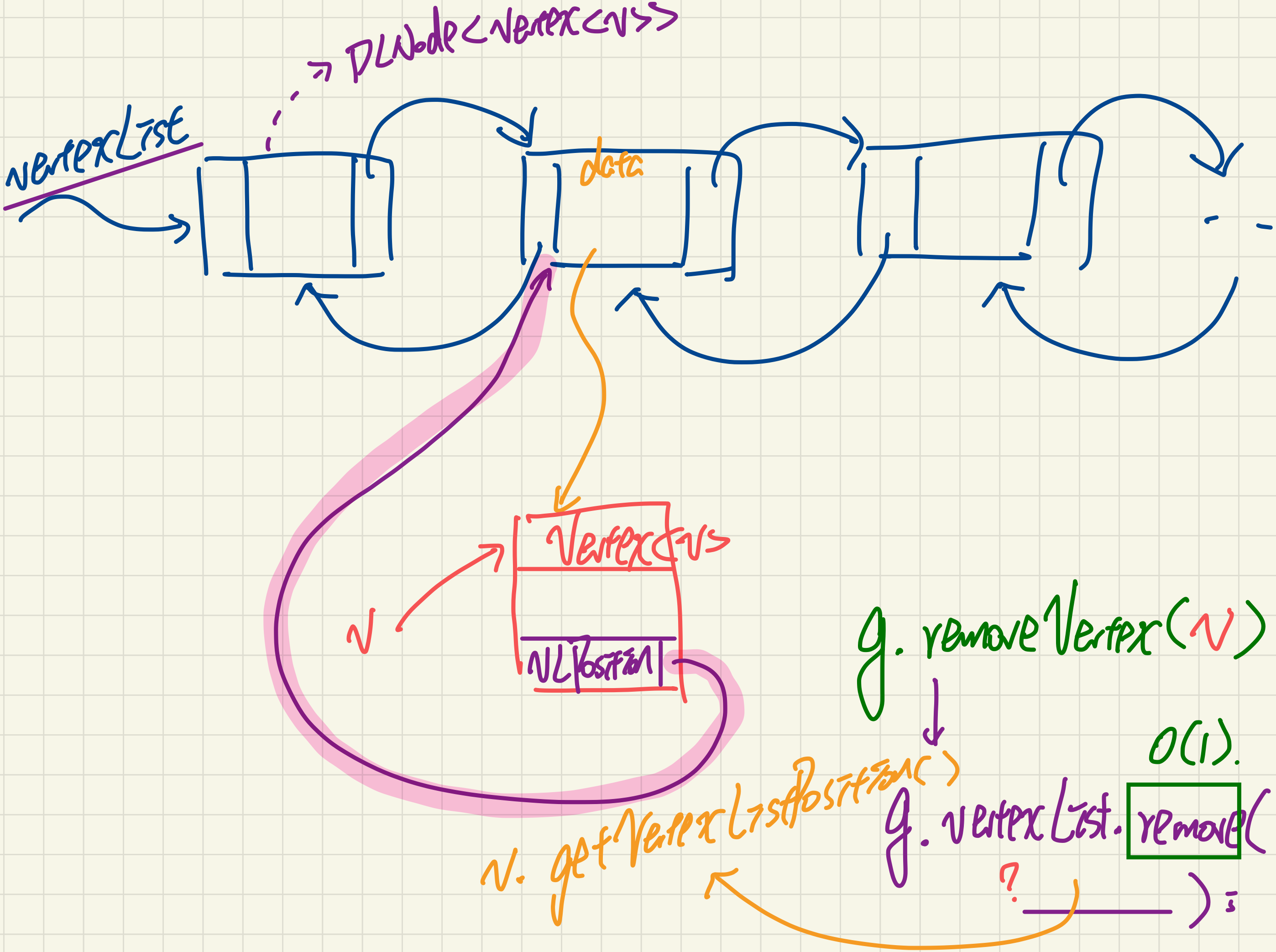
Removing 1 edge from a connected graph

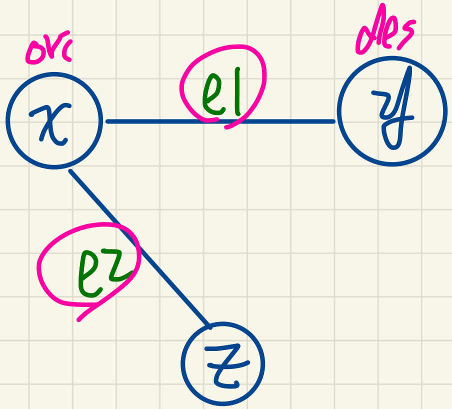


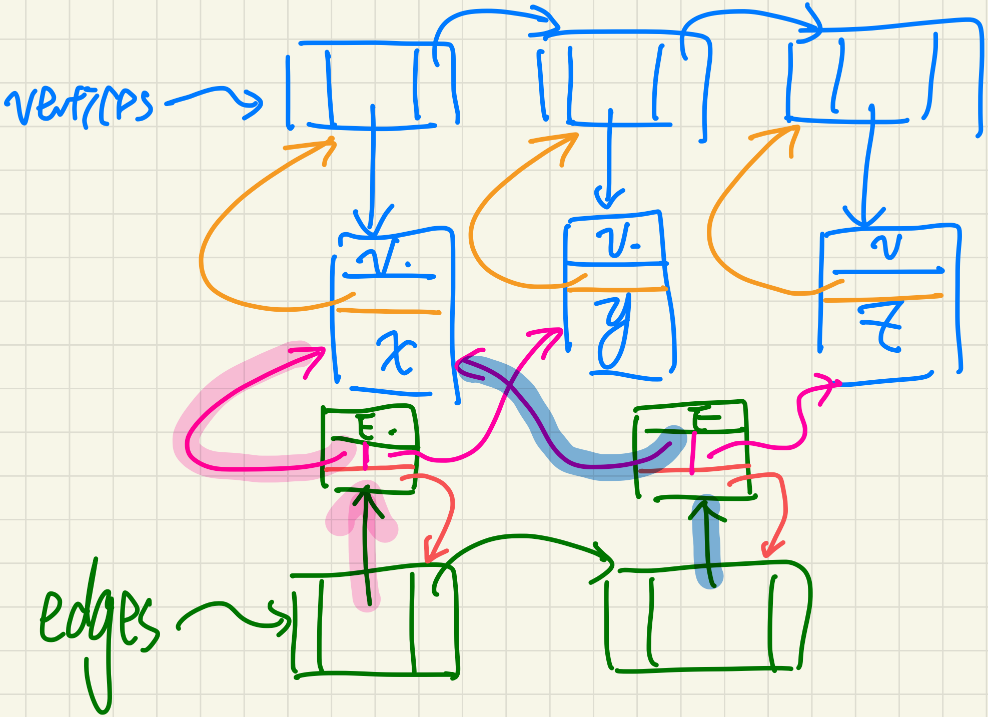
Connected cyclic



Connected acyclic


DLNode<Vertex<V>>
vertexList
data
Vertex<V>
v
VLPosition
g.removeVertex(v)
O(1).
g.vertexList.remove(? ____);
v.getVertexListPosition()





deg(x)

Lecture 14 - Nov 3

Graphs

Tracing BFS using a FIFO Queue
Back Edge (DFS) vs. Cross Edge (BFS)
Implementing Graphs: Adjacency Lists

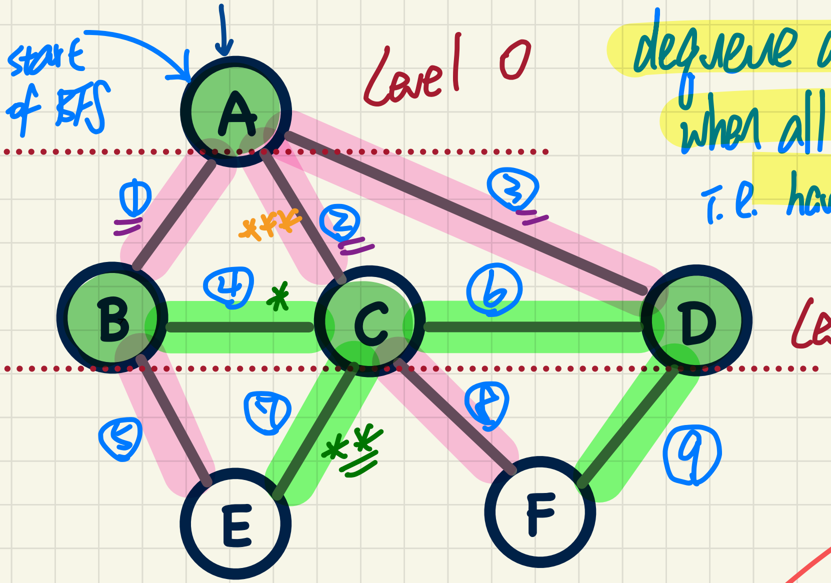
Announcements/Reminders

- Today's class: notes template posted
- Test 1 results to be released on Tuesday (Nov 4)
- Change of Dates:
    - + Assignment 2 to be released on Wed, Nov 12
    - + Assignment 2 to be due on Wed, Nov 19
    - + Test 2 to be take place on Mon, Nov 24

*** not possible to have a cross edge linking back to a previous level

Breadth-First Search (BFS): Example 1 (a)

* Cross edge connecting vertices at the same level.



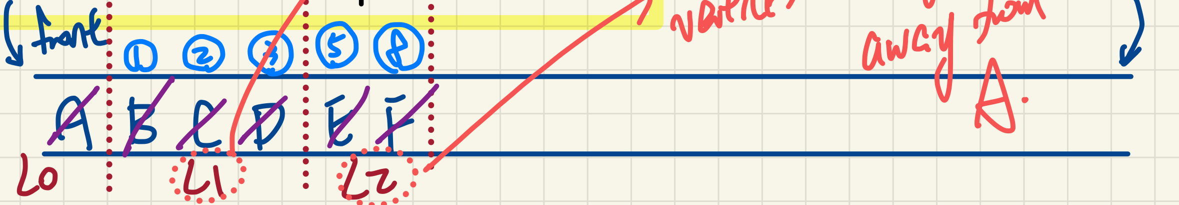
dequeue a vertex when all its i.e. have been marked.

enqueued	dequeued
A	A
B	B
C	C
D	D
E	E
F	F

** cross edge connecting vertices at the next level.

Assumptions:

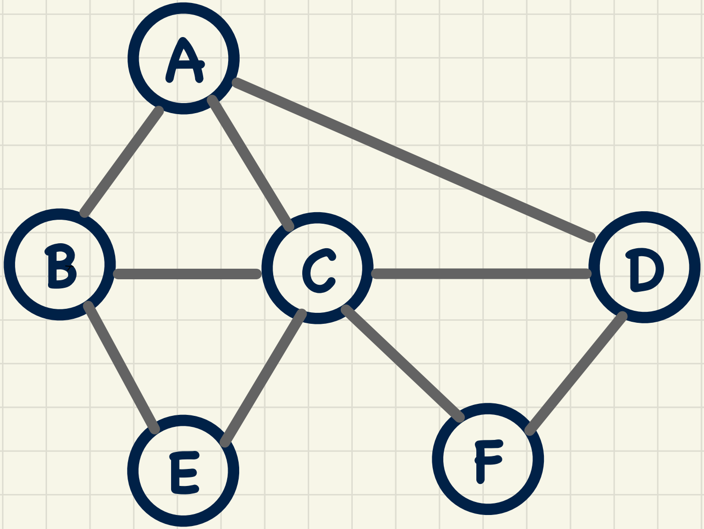
- Adjacent vertices visited in alphabetic order



vertices 1 edge away from A

vertices 2 edges away from A.

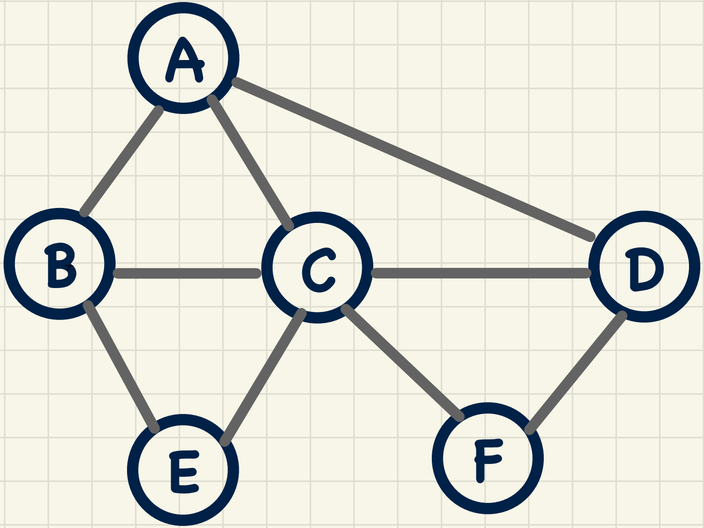
Breadth-First Search (BFS): Example 1 (b)



Assumptions:

- Adjacent vertices visited in alphabetic order
- Exception: Edge AC visited first

Breadth-First Search (BFS): Example 1 (c)

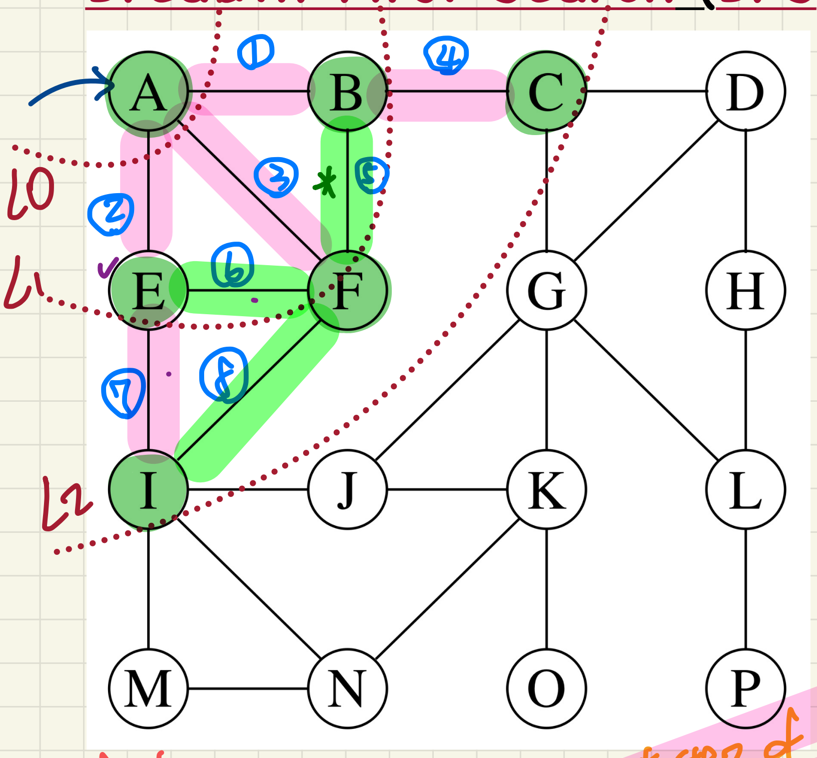


Assumptions:

- Adjacent vertices visited in alphabetic order
- Exception: Edge AD visited first

Breadth-First Search (BFS): Example 2

* Though organized vertically, vertices B and F are at the same level.



enqueued	dequeued
A	A
B ①	B
E ②	E
F ③	F
C ④	

front

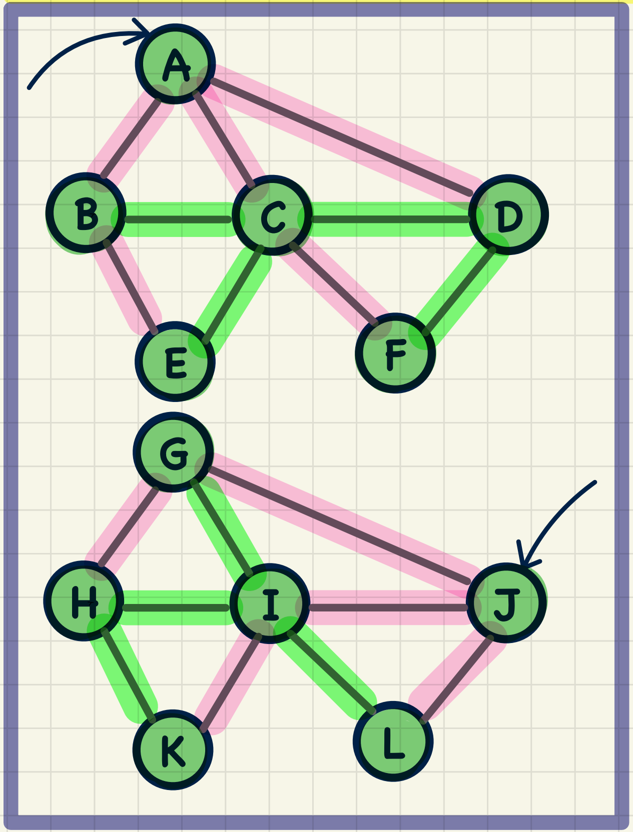
① ② ③ ④ ⑦

~~A~~ ~~B~~ ~~E~~ ~~F~~ C I

next step of BFS: go over each vertex at L2 and reach out to L3

Graph Traversals: Adapting BFS

Efficient Traversal of Graph G:



Graph Questions:

- Fina a **path** between {u, v} ⊆ V

  Start BFS from u, maintain the path until v is encountered.

- Is v **reachable** from v

  start BFS from u, is v ever encountered?

- Find <u>all</u> **connected components** of G.

  Keep doing BFSs until all vertices are visited.

- Compute a **spanning tree** of a connected G.

  Each BFS will identify the spanning tree containing the start vertex

- Is G **connected**?

  How many BFSes are needed to cover all vertices? 1

- If G is cyclic, return a **cycle**.

back edge/ cross edge input graph	(DFS) back edge found	(BFS). cross edge found
undirected	YES	YES
directed	YES	not necessarily .

Does there exist a cycle?

```
int numVertexVisited = 0;
while (numVertexVisited < |V|){
    Pick some vertex x (unvisited),
        start BFS on x.
}
```

exit loop as soon as numV.Visited = |V|

numVertexVisited properly updated (for every discovery edge)

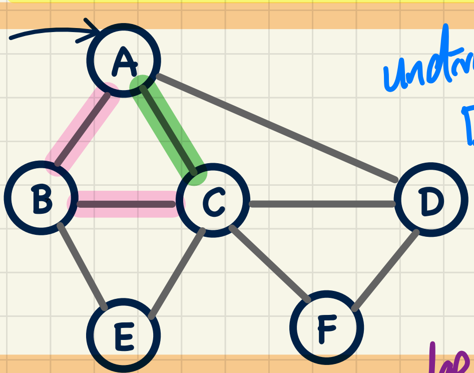
iterations
||
CCs.

Back Edge (DFS) vs. Cross Edge (BFS): Cyclic?

Does a back edge always imply the existence of a cycle? YES.

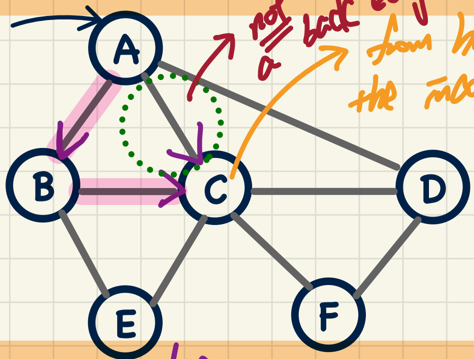
Does a cross edge always imply the existence of a cycle?

focus on finding a path leading to a back edge.



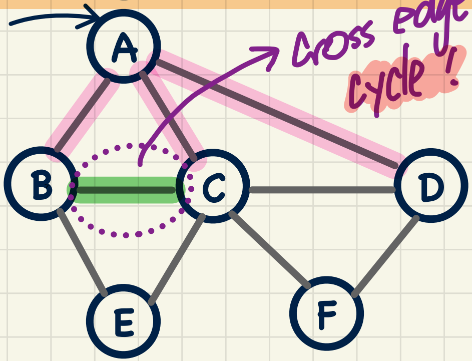
undirected DFS

cycle!

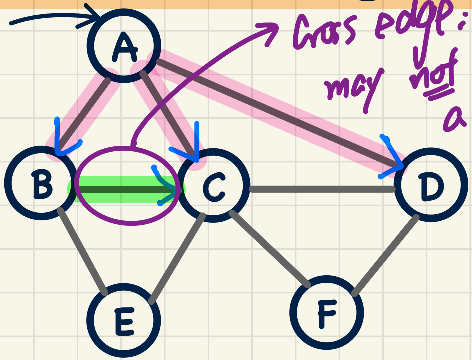


not a back edge.

from here the incoming edge will not be considered as a back edge.



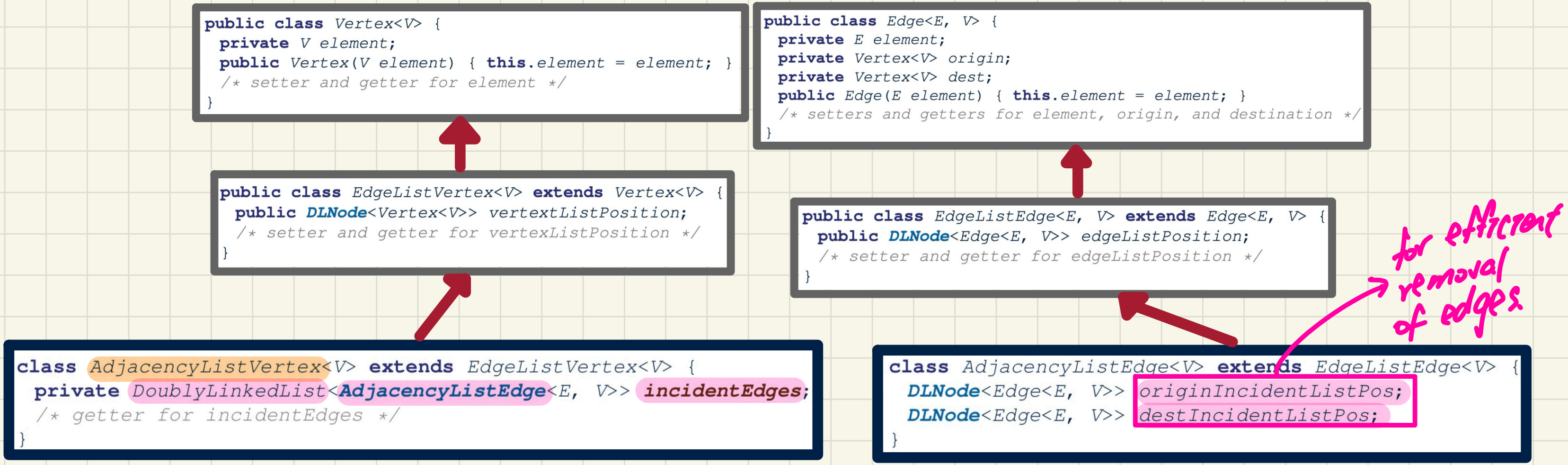
cross edge: cycle!



cross edge: may not be a cycle

Graphs in Java: Adjacency List Strategy (1)

```java
class AdjacencyListGraph<V, E> implements Graph<V, E> {
  private DoublyLinkedList<AdjacencyListVertex<V>> vertices;
  private DoublyLinkedList<AdjacencyListEdge<E, V>> edges;
  private boolean isDirected;

  /* initialize an empty graph */
  AdjacencyListGraph(boolean isDirected) {
    vertices = new DoublyLinkedList<>();
    edges = new DoublyLinkedList<>();
    this.isDirected = isDirected;
  }
```



```java
public class Vertex<V> {
  private V element;
  public Vertex(V element) { this.element = element; }
  /* setter and getter for element */
}
```

```java
public class EdgeListVertex<V> extends Vertex<V> {
  public DLNode<Vertex<V>> vertextListPosition;
  /* setter and getter for vertexListPosition */
}
```

```java
class AdjacencyListVertex<V> extends EdgeListVertex<V> {
  private DoublyLinkedList<AdjacencyListEdge<E, V>> incidentEdges;
  /* getter for incidentEdges */
}
```

```java
public class Edge<E, V> {
  private E element;
  private Vertex<V> origin;
  private Vertex<V> dest;
  public Edge(E element) { this.element = element; }
  /* setters and getters for element, origin, and destination */
}
```

```java
public class EdgeListEdge<E, V> extends Edge<E, V> {
  public DLNode<Edge<E, V>> edgeListPosition;
  /* setter and getter for edgeListPosition */
}
```

```java
class AdjacencyListEdge<V> extends EdgeListEdge<V> {
  DLNode<Edge<E, V>> originIncidentListPos;
  DLNode<Edge<E, V>> destIncidentListPos;
}
```

for efficient removal of edges

Lecture 15 - Nov 5

Graphs

Visualizing Adjacency Lists Strategy
Shortest Paths in Weighted Graphs
Dijkstra's Algorithm: Intro, Example 1

Announcements/Reminders

- Today's class: notes template posted
- Test 1 results released on Tuesday (Nov 4)
- Change of Dates:
  + Assignment 2 to be released on Wed, Nov 12
  + Assignment 2 to be due on Wed, Nov 19
  + Test 2 to be take place on Mon, Nov 24

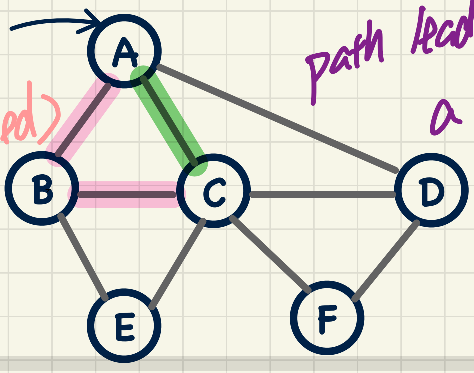
Back Edge (DFS) vs. Cross Edge (BFS): Cyclic?

Does a **back edge** always imply the existence of a **cycle**?

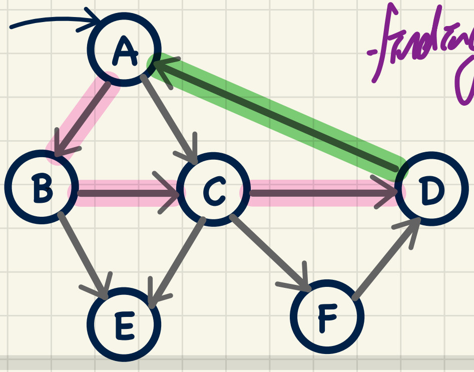
Does a **cross edge** always imply the existence of a **cycle**?

DFS

(undirected)



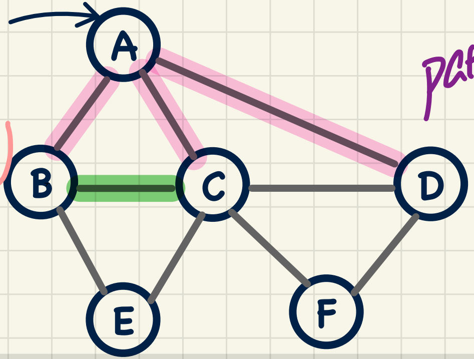
path leading to a back edge denotes a cycle



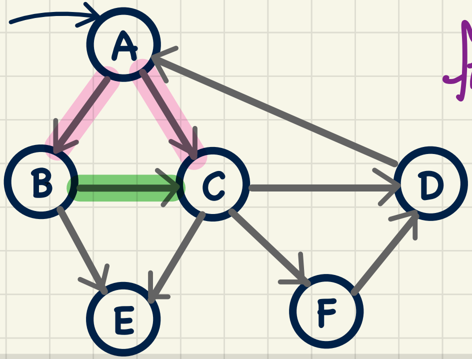
finding a back edge ⇒ ∃ a cycle.

BFS

(undirected)

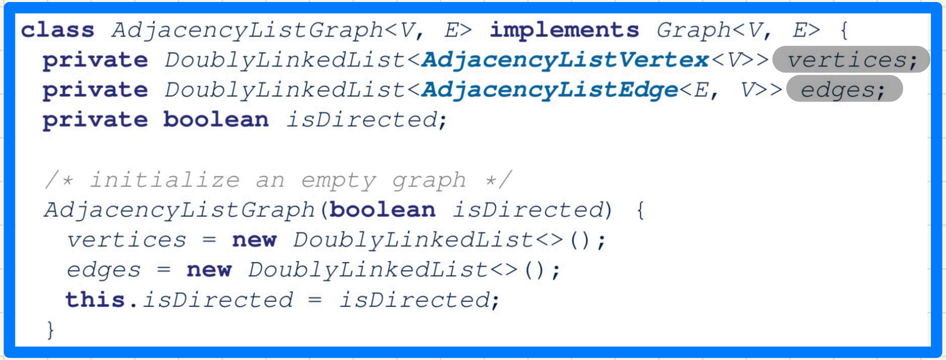


path leading to a cross edge imply a cycle.

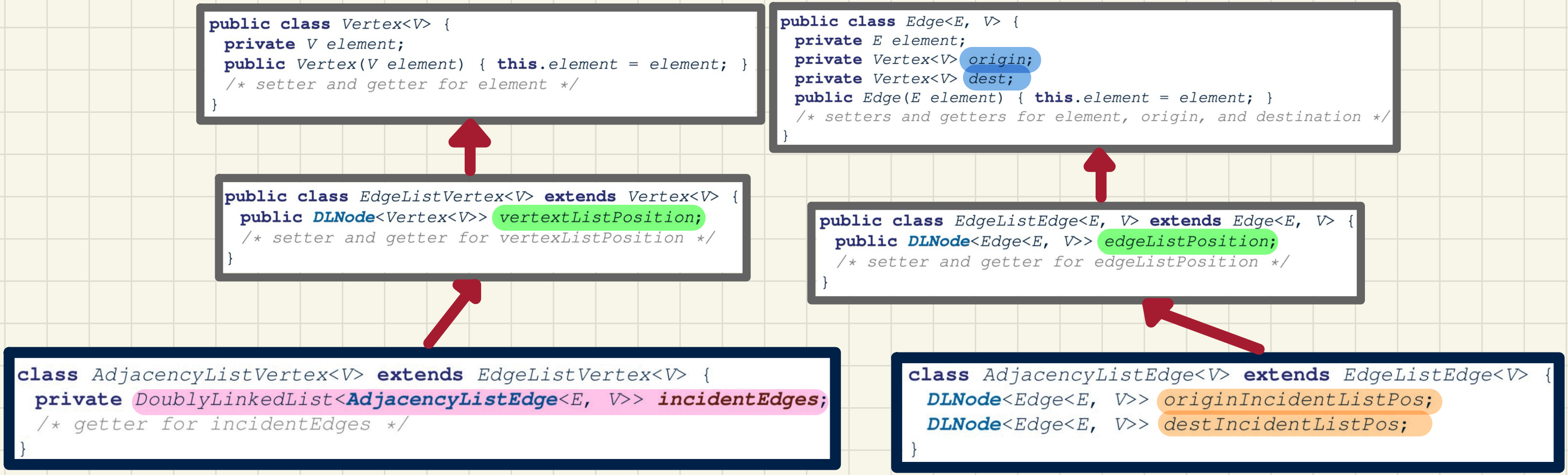


finding a cross edge ⇏ ∃ a cycle

Graphs in Java: Adjacency List Strategy (1)



```java
class AdjacencyListGraph<V, E> implements Graph<V, E> {
  private DoublyLinkedList<AdjacencyListVertex<V>> vertices;
  private DoublyLinkedList<AdjacencyListEdge<E, V>> edges;
  private boolean isDirected;

  /* initialize an empty graph */
  AdjacencyListGraph(boolean isDirected) {
    vertices = new DoublyLinkedList<>();
    edges = new DoublyLinkedList<>();
    this.isDirected = isDirected;
  }
```



```java
public class Vertex<V> {
  private V element;
  public Vertex(V element) { this.element = element; }
  /* setter and getter for element */
}
```

```java
public class Edge<E, V> {
  private E element;
  private Vertex<V> origin;
  private Vertex<V> dest;
  public Edge(E element) { this.element = element; }
  /* setters and getters for element, origin, and destination */
}
```

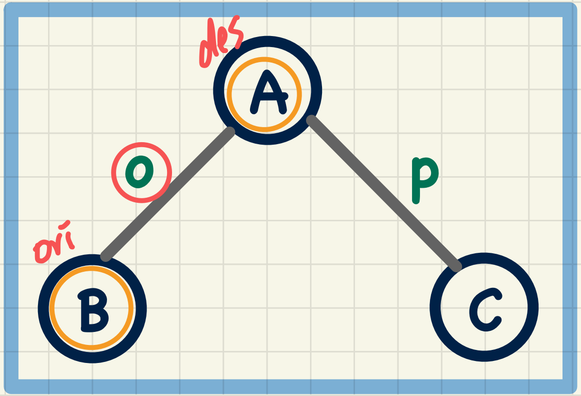
```java
public class EdgeListVertex<V> extends Vertex<V> {
  public DLNode<Vertex<V>> vertextListPosition;
  /* setter and getter for vertexListPosition */
}
```

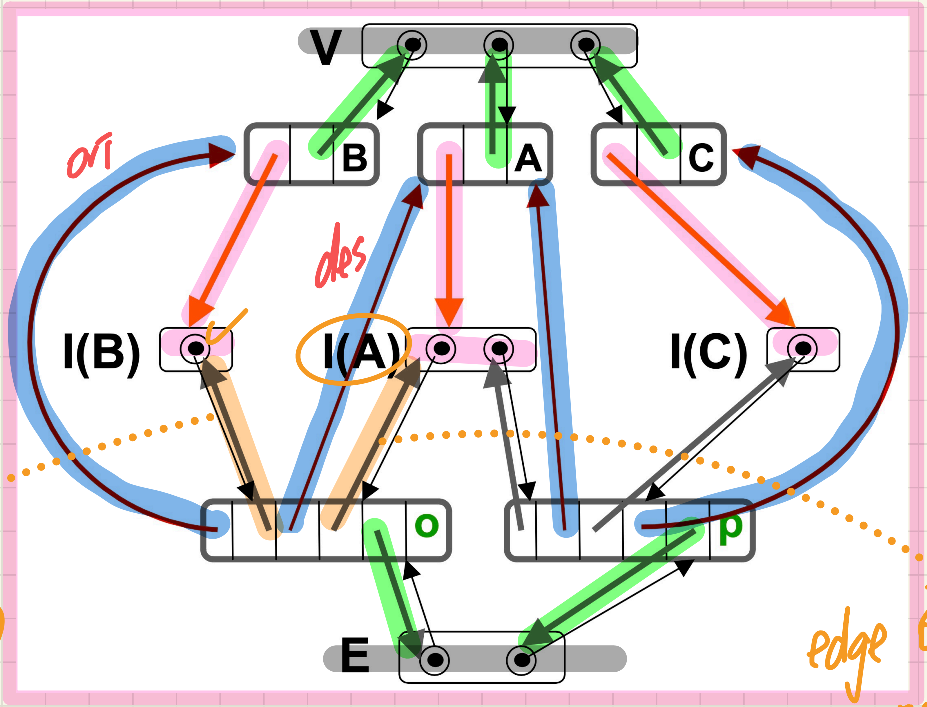
```java
public class EdgeListEdge<E, V> extends Edge<E, V> {
  public DLNode<Edge<E, V>> edgeListPosition;
  /* setter and getter for edgeListPosition */
}
```

```java
class AdjacencyListVertex<V> extends EdgeListVertex<V> {
  private DoublyLinkedList<AdjacencyListEdge<E, V>> incidentEdges;
  /* getter for incidentEdges */
}
```

```java
class AdjacencyListEdge<V> extends EdgeListEdge<V> {
  DLNode<Edge<E, V>> originIncidentListPos;
  DLNode<Edge<E, V>> destIncidentListPos;
}
```

Graphs in Java: Adjacency List Strategy (2)

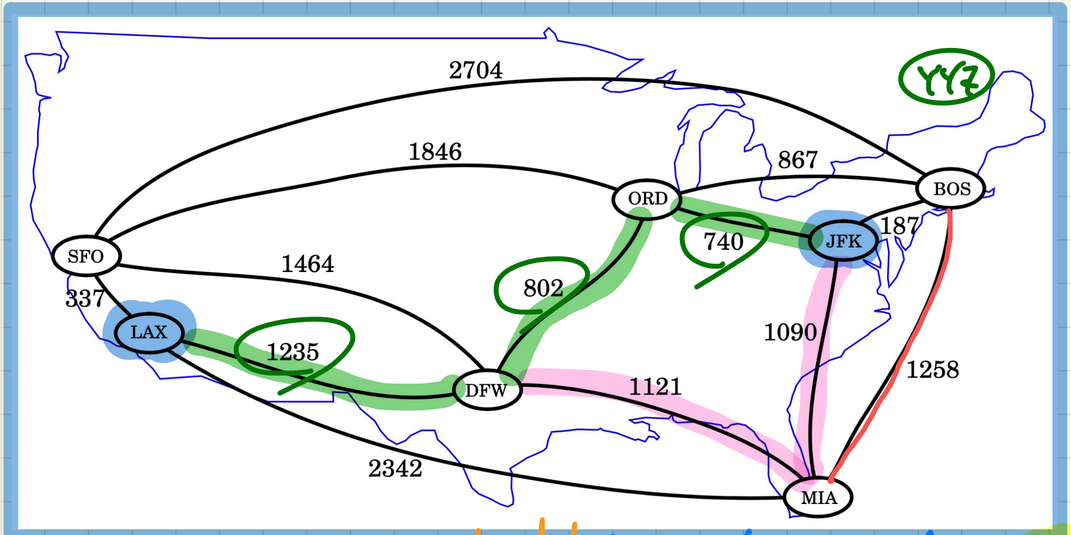




edge O's position in its origin's (B) incident edges

edge O's position in its des.'s (A) i.e. list.

Shortest Paths in Weighted Graphs



e.g. $d(YYZ, BOS) = \infty$

$d(JFK, LAX) = ??$

distance/shortest path between two vertices

out of all paths connecting u and v, $d(u, v)$ denotes the minimum length/weight among them.

$d(u, v) = \infty$ if not connected.

"cost"

Weights on edges: $w(u, v)$

overloaded length/weight of a path

$w((v_0, v_1, \ldots, v_k))$ ↳ path with $k+1$ vertices

weight of edge

$= w(v_0, v_1) + w(v_1, v_2) + \cdots + w(v_{k-1}, v_k)$

$$= \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

e.g. $w(BOS, MIA) = 1258$

Assumption: $w \geq 0$ (non-negative)

Dijkstra's Shortest Path Algorithm

→ shortest path of some vertex so far (eventually $\boxed{D(v)} = \boxed{d(v)}$)

intermediate result in loop

↓ true shortest length to get to.

Starting from a **source vertex s**, perform a ***BFS***-like procedure:

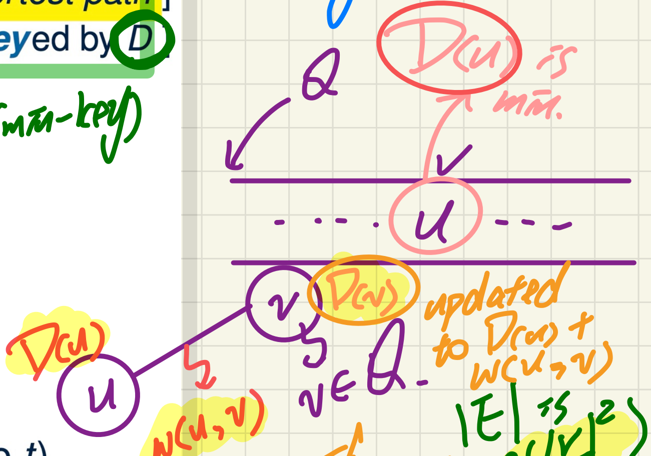
1. Initially:
   - 1.1 Set $D(s) = 0$, and every other vertex $t \neq s$, $D(t) = \infty$. [***distance***]
   - 1.2 Set $a(v) = nil$ for every vertex v. [***ancestor*** in *shortest path*]
   - 1.3 Insert all vertices into a ***priority queue*** Q [***key***ed by D]

→ heap. (min-key)

2. While Q is not empty, repeat the following:
   - 2.1 Find vertex u in Q s.t. $D(u)$ is the **minimum**.
   - 2.2 For every vertex v ***adjacent to*** u **if**: $v \in Q \wedge \boxed{D(u) + w(u, v) < D(v)}$, then:
     - Set $D(v) = D(u) + w(u, v)$
     - Set $a(v) = u$
   - 2.3 Remove vertex u from Q. → "visited"

Upon completion, for every vertex t ($t \neq s$):

- $\boxed{D(t) = d(s, t)}$ (i.e., weight of *shortest path* from s to t).
- **Reversing** t's ***ancestor path*** → *shortest path*: $\langle s, \ldots, a(t), t\rangle$



$|E|$ is $O(|V|^2)$

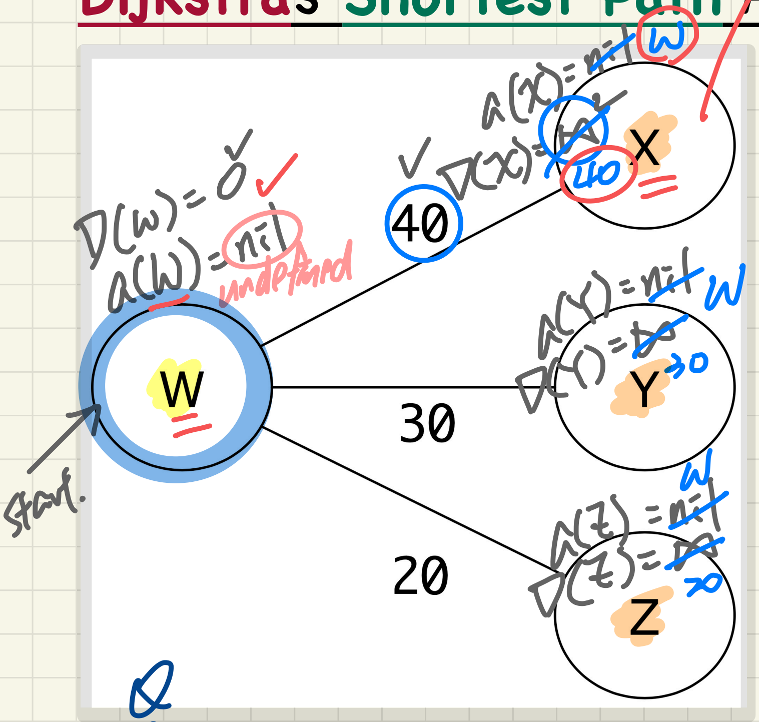
iterations ≤ degree(u)

often not necessary to go over all incident edges (time consuming if the graph is complete)

Dijkstra's Shortest Path Algorithm: Example 1

shortest path from W to X has weight: 40.

<X, W, ~~nil~~>



Q

① ②

~~W~~ X Y ~~Z~~

① ②

Iteration	vertex with min D	changes?
Init.		
1	W Reverse: <W, X>	$\underbrace{D(W)}_{0} + \underbrace{w(W,X)}_{40} < \underbrace{D(X)}_{\infty}$ D(X) = 40 a(X) = W D(Y) = 30 a(Y) = W D(Z) = 20 a(Z) = W
2	Z D(Z) = 20	n.C. ∵ W ∉ Q
3	Y D(Y) = 30	n.C. ∵ W ∉ Q
4	X D(X) = 20	n.C ∵ W ∉ Q.

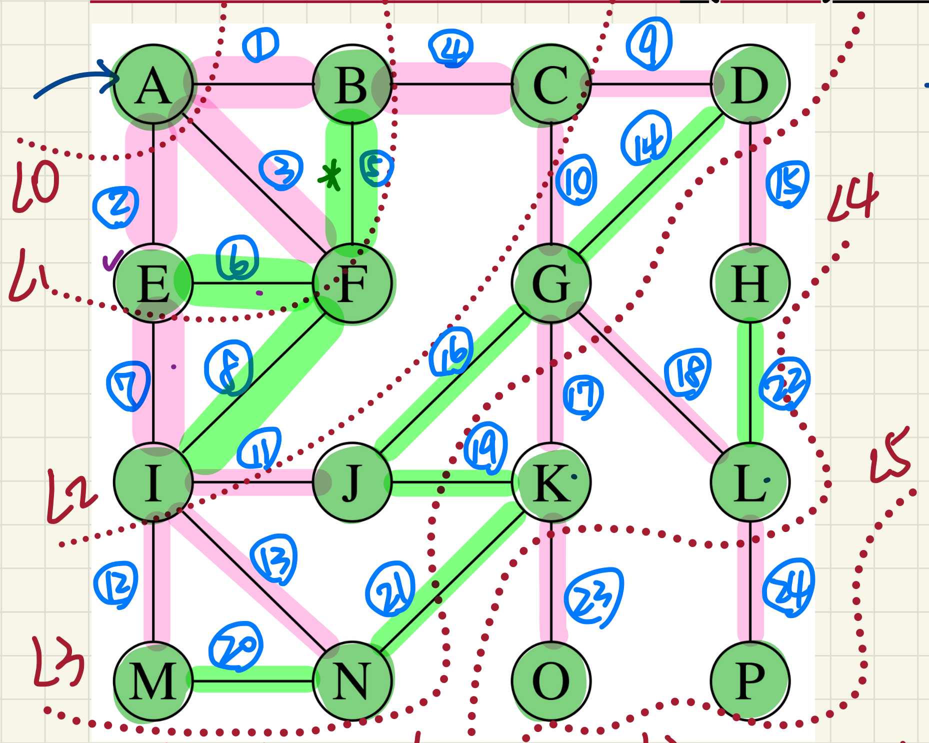
Tutorials - Week 9 - Nov 7

Graphs

Breadth-First Search (BFS)

Breadth-First Search (BFS): Example 2

* Though organized vertically, vertices B and F are at the same level.



enqueued	dequeued
A	A
B ①	B
E ②	E
F ③	F
C ④	C
D ⑨	I
G ⑩	D
J ⑪	G
M ⑫	J
N ⑬	N
H ⑮	H
	K
	L O P

K	⑰
L	⑱
O	㉓
P	㉔

front

L0	L1	L2	L3	L4	L5
	① ② ③	④ ⑦	⑨ ⑩ ⑪ ⑫ ⑬	⑮ ⑰ ⑱	㉓ ㉔
~~A~~	~~B~~ ~~E~~ ~~F~~	~~C~~ ~~I~~	~~D~~ ~~G~~ ~~J~~ ~~M~~ ~~N~~	~~H~~ ~~K~~ ~~L~~	~~O~~ ~~P~~

Lecture 16 - Nov 10

Graphs

Dijkstra’s Algorithm: Tracing
Dijkstra’s Algorithm: Pre- and Post-cond.

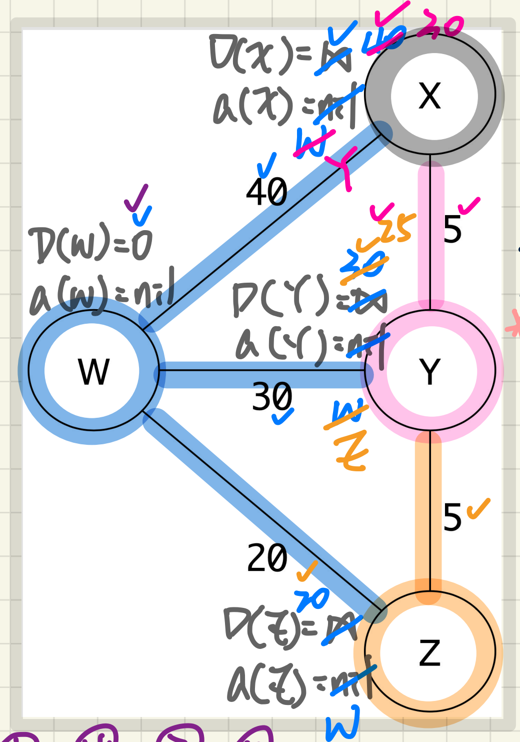
Announcements/Reminders

- Today's class: notes template posted
- Test 1 results released on Tuesday (Nov 4)
- Change of Dates:
  - Assignment 2 to be released on Wed, Nov 12
  - Assignment 2 to be due on Wed, Nov 19
  - Test 2 to be take place on Mon, Nov 24

single source. * # Iterations = |V|

Dijkstra's Shortest Path Algorithm: Example 2

** remove X from Q



$d(W,W) = 0$ ✓
$d(W,X) = 30$
$d(W,Y) = 25$
$d(W,Z) = 20$

Init.	W (src)	X	Y	Z
D	0	∞	∞	∞
a	nil	nil	nil	nil

D(X) = d(W,X)

# Iteration	min D	changes
1	W D(W)=0	X, Y, Z in Q
	a(X)=W ←	0 + 40 < ∞ → D(X) = 40 ≠ d(W,X)
	a(Y)=W ←	0 + 30 < ∞ → D(Y) = 30 ≠ d(W,Y)
	a(Z)=W ←	0 + 20 < ∞ → D(Z) = 20 = d(W,Z) (not final).
		remove W from Q. ↳ D(W) = d(W,W) final
2	Z D(Z)=20	W ∉ Q → not considered.
	a(Y)=Z ←	20 + 5 < 30 → D(Y) = 25 = d(W,Y) not final
		remove Z from Q D(Z) = d(W,Z) final
3	Y D(Y)=25	W ∉ Q → not considered.
	a(X)=Y ←	25 + 5 < 40 → D(X) = 30 = d(W,X) not final
		remove Y from Q D(Y) = d(W,Y) final
4	X D(X)=30	W ∉ Q ∧ Y ∉ Q → not considered. **

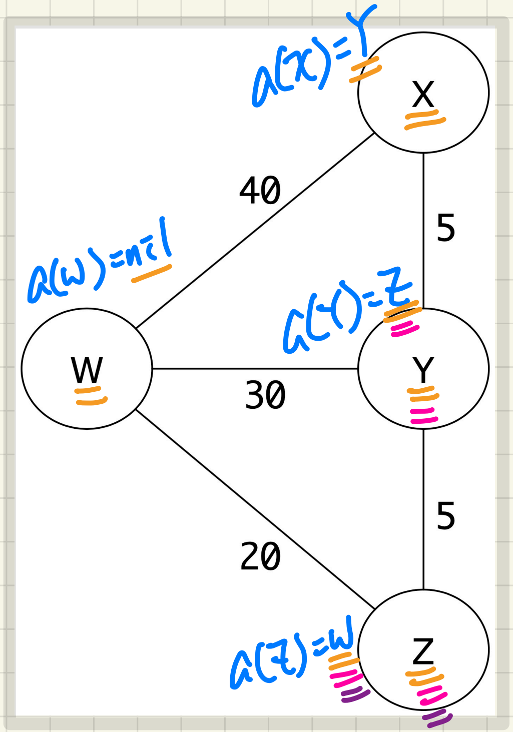
Q*

① ④ ③ ②

~~W~~ ~~X~~ ~~Y~~ ~~Z~~

* min key (D value) priority queue

Upon termination of Dijkstra's algorithm



dest.	ancestor path.
x	x Y Z W
Y	Y Z W
Z	Z W

dest.	shortest path (from source W)
x	W Z Y x
Y	W Z Y
Z	W Z

programming statements.

Correctness of Loops: Syntax

precondition

```
{Q}
Sinit *
while B ** {
  Sbody
}
{R}
```

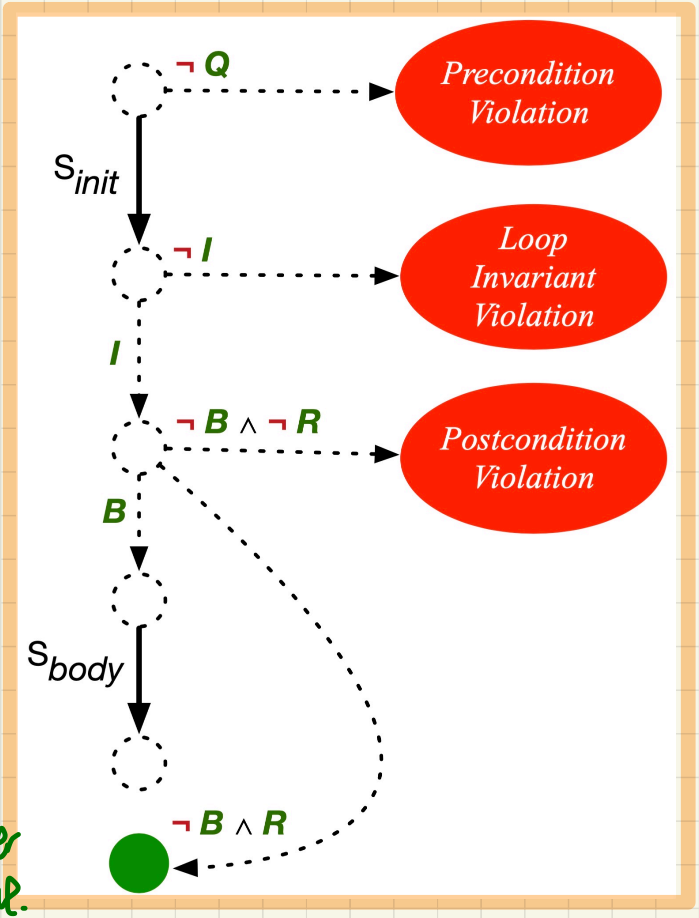
postcondition.

```
void myAlgorithm() {
  assert Q; /* Precondition */
  Sinit
  assert I; /* Is LI established? */
  while( B ) {
    Sbody
    assert I; /* Is LI preserved? */
  }
  assert R; /* Postcondition */
}
```

Iterative Algorithm

B : stay condition

¬B : exit condition.



* Initialization/Preparation for iterations.

** As long as B is true, execute S_{body} another time.
As soon as B is false, exit from the loop.

$\{Q\}$

precondition:

$w(u, v) \geq 0 \quad u \in V \quad v \in V$

non-negative weight

Dijkstra's algorithm

$\{R\}$

1. $D(v) = \boxed{d(x, v)}$ where x is the start vertex

implicitly; shortest path so far from start vertex to v

true shortest-path length between x and v

shortest-path length known so far

Q. when is this $D(v)$ finalized?

A. when v is removed from Q

2. Reverse of each ancestor path gives the shortest path.

Lecture 17 - Nov 12

Graphs

Loop Invariant (LI): Execution Flow
Relating Exit Condition, LI, Postcondition
Dijkstra's Algorithm: LI, Assumption

Announcements/Reminders

- Today's class: notes template posted
- Assignment 2 released

  Edge List

  ① given. working version of BFS

  ② answer graph question
- Change of Dates:
  - + Assignment 2 to be due on Wed, Nov 19
  - + Test 2 to be take place on Mon, Nov 24

Correctness of Loops: Syntax

→ programming statements.

precondition

```
{Q}
Sinit *
while (B) **  {
  Sbody
}
{R}
```

postcondition.

```
void myAlgorithm() {
  assert Q; /* Precondition */
  Sinit
  assert I; /* Is LI established? */
  while ( B ) {
    Sbody
    assert I; /* Is LI preserved? */
  }
  assert R; /* Postcondition */
}
```

establishing LI

Iterative Algorithm

B : stay condition

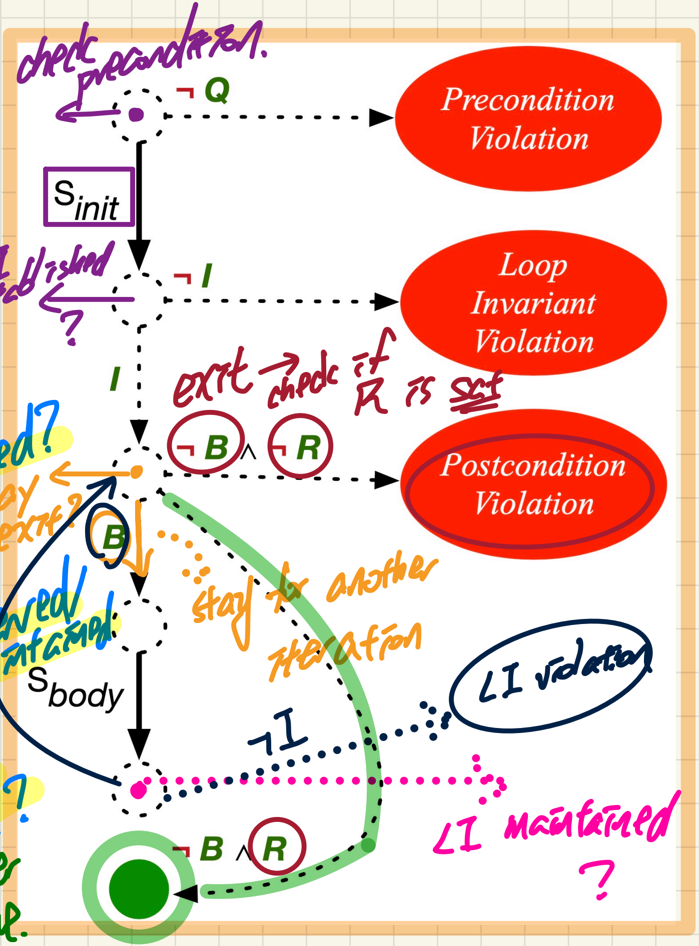
¬B : exit condition.

Is LI established?

Is LI preserved/maintained at the end of current it?

* Initialization/Preparation for iterations.

** As long as B is true, execute Sbody another time.
As soon as B is false, exit from the loop.



alt

```
S_init
while (B) {
    assert(I);
*
}
```

if B is false right after S_{init}, we will not even get a chance to check * for establishments

Correctness of Loops: Example

* $\underset{i>5}{\underline{\neg(i \leq 5)}} \wedge (1 \leq i \leq 6) \Rightarrow i = 6$

proved!

LI

```
void testLI() { /* Assume: integer attribute i */
  assert i == 1; /* Precondition */
  assert (1 <= i) && (i <= 6); /* Is LI established? */
  while (i <= 5) {
    i = i + 1;
    assert (1 <= i) && (i <= 6); /* Is LI maintained? */
  }
  assert i == 6; /* Postcondition */
}
```

Q

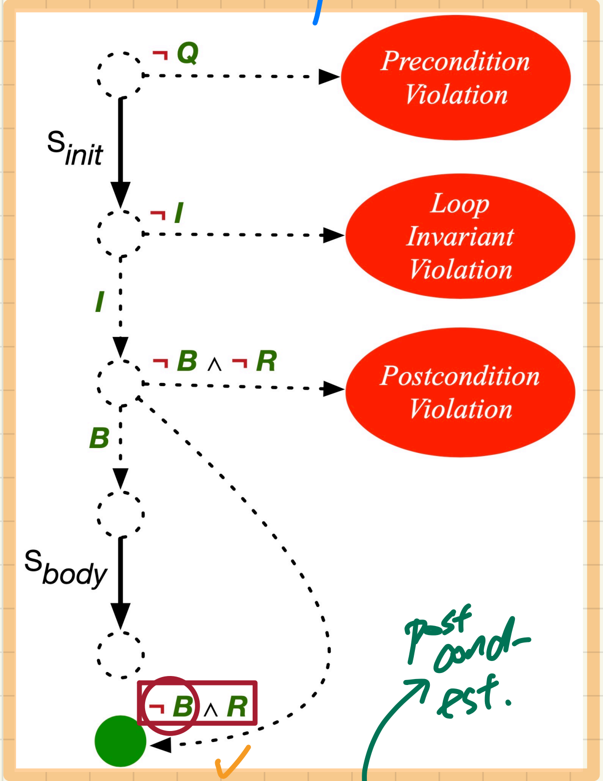
B

R

LI : $1 \leq i \leq 6$

stay condition

It #	i	LI	B	R
init	1	✓(est.)	✓	
1	changed to: 2	✓(maintained)	✓	
2	3	✓	✓	
3	4	✓	✓	
4	5	✓	✓	
5	6	✓	X exit	✓



post cond- est.

General $\neg B \wedge LI \Rightarrow R$

exit

maintained in last iteration

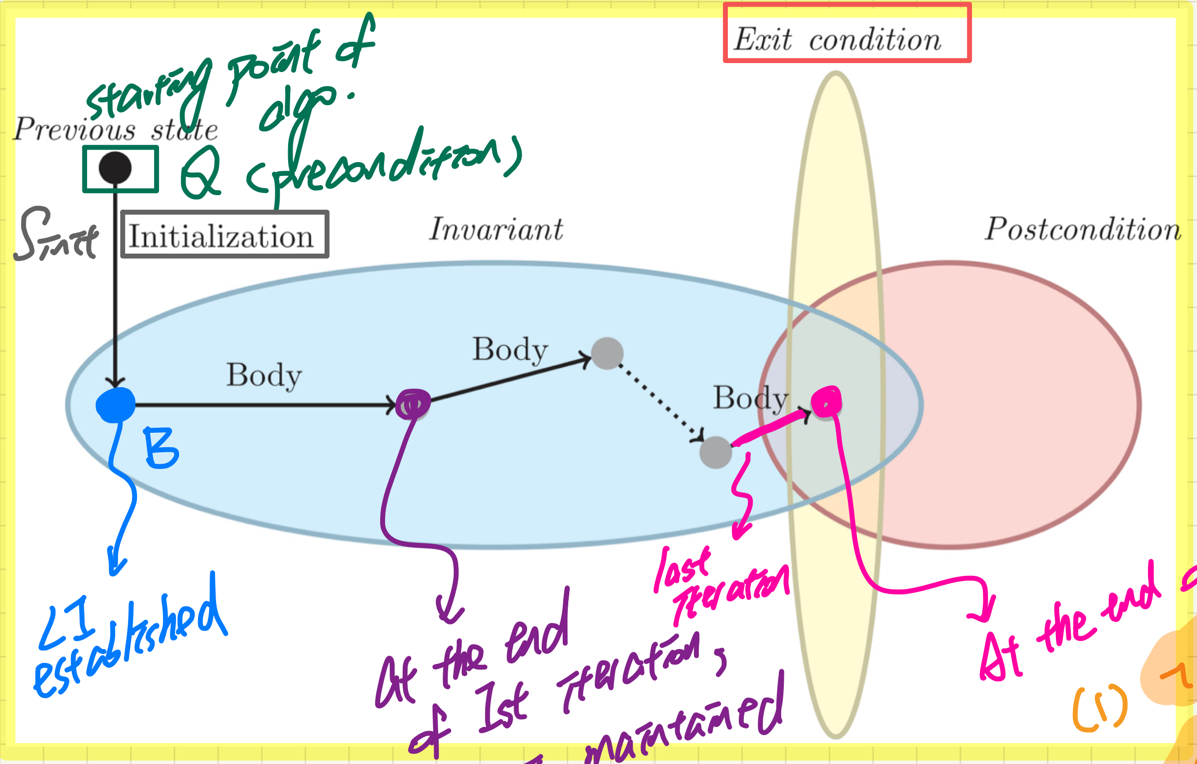
*

Exercises

(1) $\angle I'$: $1 \leq i \leq 5$

(2) $\angle I''$: $1 < i \leq 6$

Contracts of Loops: Visualization



¬B.

starting point of algo.

Q (precondition)

Start

B

LI established

At the end of 1st iteration, LI maintained

last iteration

At the end of the last iteration:

(1) ¬B (exit)

(2) LI (maintained)

⇒ R (postcondition)

$* \ \forall u \bullet u \in V \wedge u \in S \Rightarrow D(u) = d(\underset{w}{s}, u)$

Correctness of Loops: Dijkstra's Shortest-Path Algorithm

Recall: A *loop invariant* (*LI*) is a Boolean condition.

- *LI* is establisehd before the 1st iteration.
- *LI* is preserved **at the end of** each subsequent iteration.

The (iterative) Dijkstra's algorithm has *LI*:

> For every vertext *u* that has already been removed from the priority queue *Q* (i.e., *u* is considered visited), *D*(*u*) equals the *true* shortest-path distance from source *s* to *u*.

min length so far

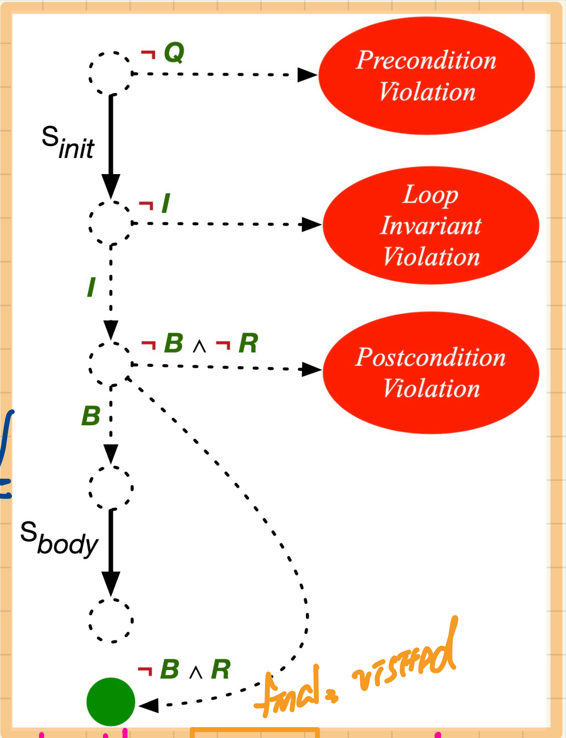
$d(s, u)$

$G = (V, E)$ Apply Dijkstra from $s \in V$

S: set of vertices removed so far.

at the end of It.

It	Remove	LI	S
1	w	LI_1: $D(w) = d(w, w)$	$\{w\}$
2	z	LI_2: $LI_1 \wedge D(z) = d(w, z)$	$\{w, z\}$
3	y	LI_3: $LI_2 \wedge D(y) = d(w, y)$	$\{w, z, y\}$
4	x	LI_4: $LI_3 \wedge D(x) = d(w, x)$	$\{w, z, y, x\}$



final, visited

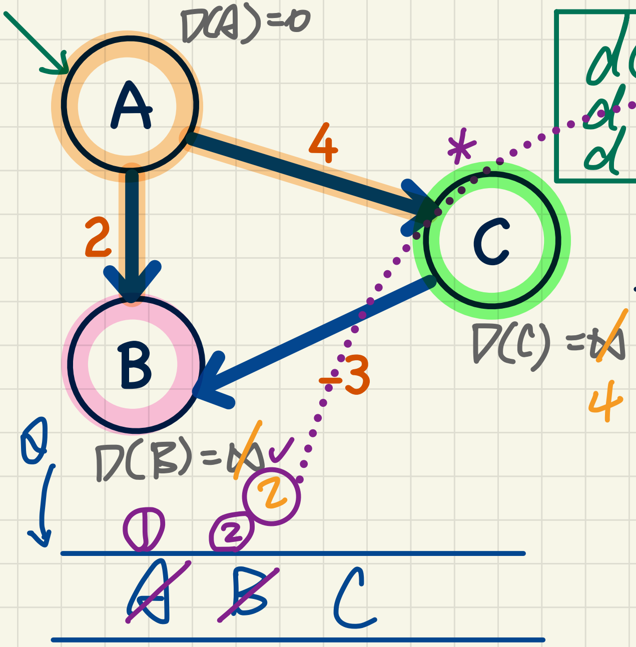
$LI \ v1: \ \forall u \bullet u \notin Q \wedge u \in V \Rightarrow D(u) = d(\underset{w}{s}, u)$

$LI \ v2: \ *$

Dijkstra's Shortest Path Algorithm: Negative Weights

The (iterative) Dijkstra's algorithm has ***LI***:

> For every vertext *u* that has already been removed from the priority queue *Q* (i.e., *u* is considered visited), *D*(*u*) equals the ***true*** shortest-path distance from source *s* to *u*.



$d(A, A) = 0$
$d(A, B) = \boxed{1}$
$d(A, C) = 4$

* At the end of 2nd iteration, the removed vertex B's final value D=2, not equal to the true d(A,B)=1
↳ violation of LI

It.	min D	changes
1	A D(A)=0	B∈Q C∈Q D(B)=2 D(C)=4 remove A from Q → S={A}
2	B D(B)=2	n.c. remove B from Q → S={A, B} ✓
3	C D(C)=4	B∉Q no change.

Tutorials - Week 10 - Nov 14

Graphs

Precondition and Postcondition
Deriving and Tracing Loop Invariant
Correctness of Loops

Iterative Algorithm: Precondition and Postcondition

input

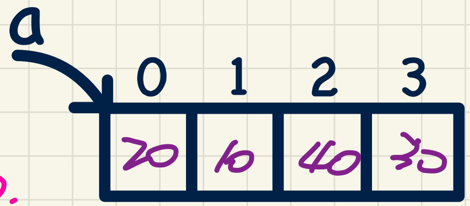
```
int findMax (int[] a) {
  assert Q; /* precondition satisfied? */
  int i = 0;
  int result = a[i];
  assert LI; /* invariant established? */
  while(i != a.length) {
    if(a[i] > result) {
      result = a[i];
    }
    i = i + 1;
    assert LI; /* invariant preserved? */
  }
  assert R; /* postcondition satisfied? */
  return result;
}
```

Sinit

Sbody

Postcondition: relation between input and output.

output



result = 40.

Q. a != null ∧ a.length > 0 or !=

R. result ≥ each value in a

result ≥ a[0]
∧ result ≥ a[1]
⋮
∧ result ≥ a[3]

S in Dijkstra's algo.

result is the temporary computed max result so far ; index (i)
(≈ D value)

0..i specify the range of array already considered.

a

0	1	2	3
20	10	40	30

result ≥ each value in a

$$\forall j \cdot \boxed{0 \le j \le \boxed{a.length - 1}} \Rightarrow \boxed{\underline{a[j] \le result}}$$

< a.length

✓ result ≥ a[j]

largest index (right-most)

↳ to derive the [I], result may only be applicable as the max of some part/prefix of the array a.

if the j value is within this range

↳ then this condition should be true.

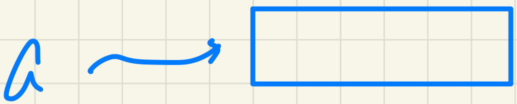
Q (precondition) ~ input

R (postcondition)

- ~ input
- ~ output
- ~ no need to mention local variables

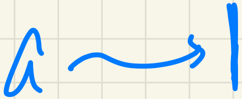
LI (Loop invariant)

- ~ input
- ~ output
- ~ loop counter
- ~ other local variable(s)



```
a != null
```

```
int[] a = new int[0];
```



```
a != null && a.length == 0
```

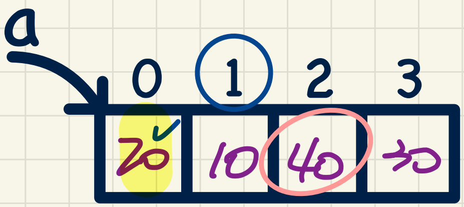
Iterative Algorithm: Loop Invariant (1)

```
int findMax (int[] a) {
  assert Q; /* precondition satisfied? */
  int i = 0;
  int result = a[i];
  assert LI; /* invariant established? */
  while(i != a.length) {
    if(a[i] > result) {
      result = a[i];
    }
    i = i + 1;
    assert LI; /* invariant preserved? */
  }
  assert R; /* postcondition satisfied? */
  return result;
}
```

Start

B

4



Proposed LI:

$\forall j \cdot 0 \leq j \leq i \Rightarrow result \geq a[j]$

inappropriate!

after it.	i	result	LI	B
Start	0	20	✓ est.	✓
1	1	20	* $i=1$ ✓	✓
2	2	20	✗ ** $i=2$	

* $\forall j \cdot 0 \leq \overset{0,1}{j} \leq 1 \Rightarrow result \geq a[j]$

** $\forall j \cdot 0 \leq \overset{0,1,2}{j} \leq 2 \Rightarrow \overset{20}{result} \geq a[j]$

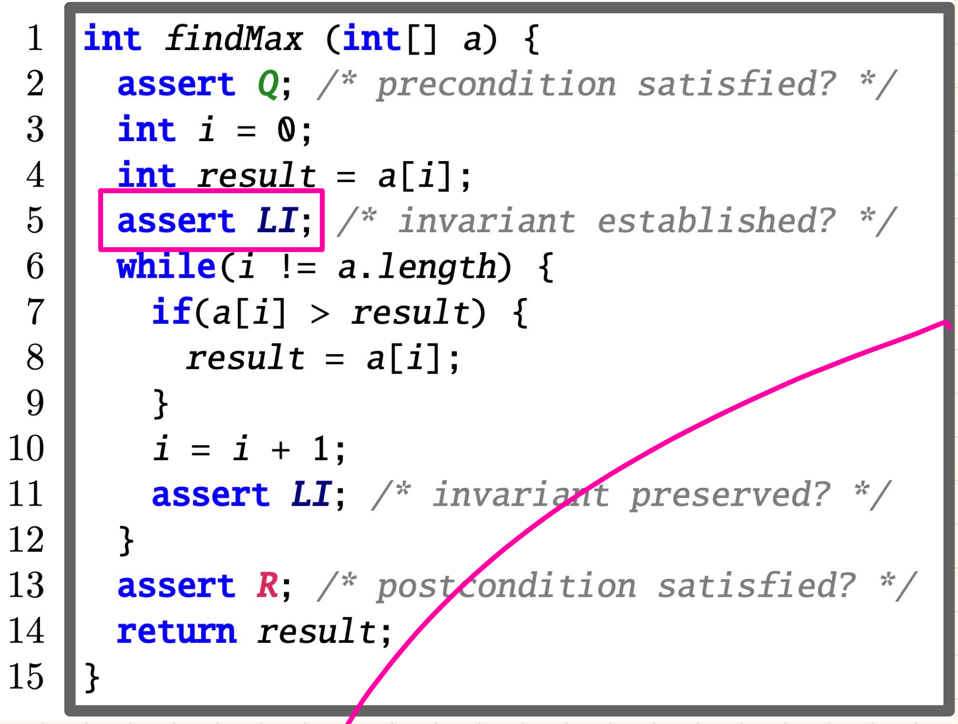
LI violation

"$\overset{20}{result} \geq \overset{40}{a[2]}$"

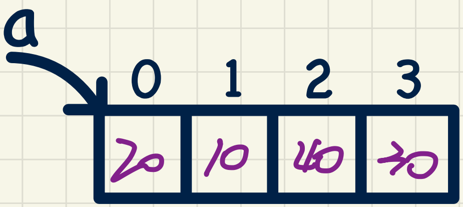
"false"

Iterative Algorithm: Loop Invariant (2)

$false \Rightarrow P \equiv true$



```
int findMax (int[] a) {
  assert Q; /* precondition satisfied? */
  int i = 0;
  int result = a[i];
  assert LI; /* invariant established? */
  while(i != a.length) {
    if(a[i] > result) {
      result = a[i];
    }
    i = i + 1;
    assert LI; /* invariant preserved? */
  }
  assert R; /* postcondition satisfied? */
  return result;
}
```



Fixed LI:

$$\forall j \bullet 0 \le j \overset{\le i-1}{\boxed{< i}} \Rightarrow result \ge a[j]$$

Exercise

show this LI is appropriate:

(1) established

(2) maintained.

$$\forall j \bullet \underbrace{\boxed{0 \le j < 0}}_{false} \Rightarrow result \ge a[j]$$

Iterative Algorithm: Correctness

```
int findMax (int[] a) {
  assert Q; /* precondition satisfied? */
  int i = 0;
  int result = a[i];
  assert LI; /* invariant established? */
  while(i != a.length) {
    if(a[i] > result) {
      result = a[i];
    }
    i = i + 1;
    assert LI; /* invariant preserved? */
  }
  assert R; /* postcondition satisfied? */
  return result;
}
```

B

$\neg B \wedge LI \Rightarrow R$

exit from loop

maintained in last iteration.

Post-condition

A1

$\boxed{\neg(i \mathrel{!=} a.length)}$

$\wedge$ A2

$\boxed{\forall j \cdot 0 \le j < \boxed{i} \Rightarrow result \ge a[j]}$

$\Rightarrow \forall j \cdot 0 \le j < a.length \Rightarrow result \ge a[j]$

Assuming the antecedent

① Assume A1, A2.

② From A1: $i = a.length$

③ Substitute i by $a.length$ in A2: this gives us the consequent to prove.

Lecture 18 - Nov 17

Graphs

Priority Queues ADT: Introduction
Heap: Structural Property
Heap: Relational Property

Announcements/Reminders

- Today's class: notes template posted
- Assignment 2 released
- Change of Dates:
  - Assignment 2 to be due on Wed, Nov 19
  - Test 2 to be take place on Mon, Nov 24
- Online Course Evaluation

Wednesday class: $\approx$ 40-50 min lecture.

$\approx$ $\boxed{\leq 20 \text{ min. Q \& A}}$ ~ or Zoom office hour.

Test 2 (WSC 106, 4:30 PM to 5:20 PM, Monday Nov 24)

Coverage

lecture on PQ & heap

+ Graphs lecture (slides 33 — 72, notes, example code)
+ Tutorials Weeks 9 and 10
+ Assignment 2

Format

+ Programming Part (Eclipse):
  * Import a Java starter project (like A2)
  * Implement Java classes/methods to pass test cases
  * e.g., Implement graph op from scratch.
  * e.g., Implement graph op based on given DFS (A1) or BFS (A2).

incident edges

ref. solutions

Hashmap
Hashset
ArrayList
(API page accessible)

+ Written Part (eClass):
  * MCQs ✓
  * Written questions ✓ (e.g., short answers, justifications, proofs)

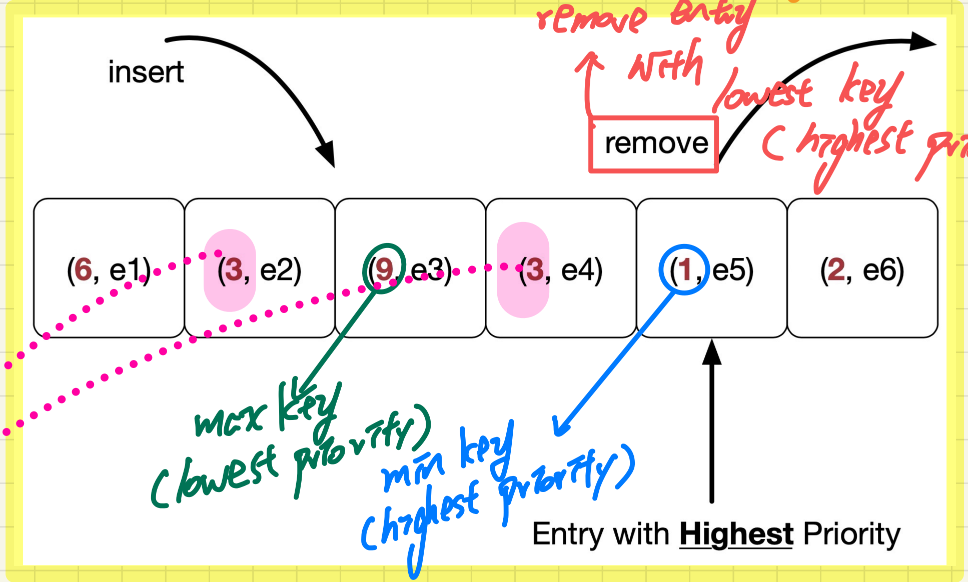
What is a Priority Queue (PQ)

min-key PQ.

(k) ↓ priority ↑

e.g. D(v)

↳ min D(v) chosen first.



insert

remove

remove entry with lowest key (highest priority).

(6, e1)	(3, e2)	(9, e3)	(3, e4)	(1, e5)	(2, e6)

max key (lowest priority)

min key (highest priority)

Entry with **Highest** Priority

entries with the same priority (does not matter which one to choose)

Compare PQ with FIFO Queue

1. entries in PQ removed according to priority values

2. entries in FIFO queue removed according to chronological order of insertions.

role of key D.S.	
BST	uniquely identifying an entry
PQ/heap	keys may be duplicated ↓ not for searching purpose

$*(2^0+2^1+\cdots+2^{h-1}) = 2^h - 1$

BT Terminology: Complete vs. Full BTs

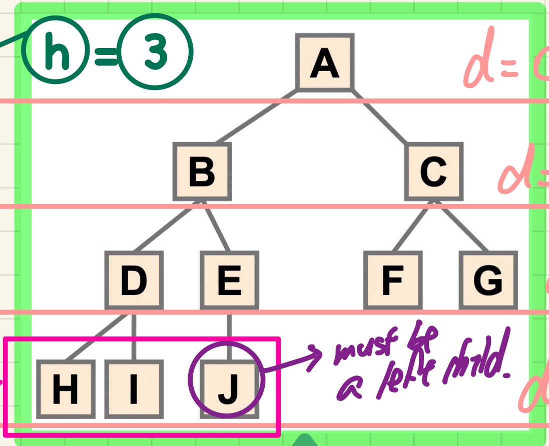
C1: level 0 ~ level h-2
↳ each node has 2 child nodes

Complete BT

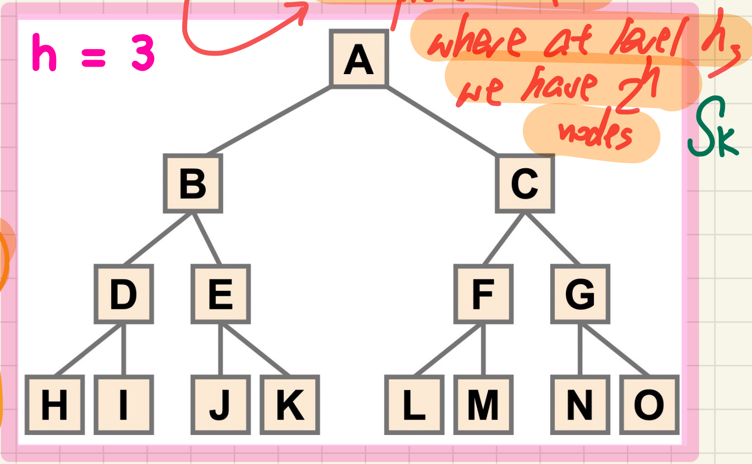
where at level h, we have 2^h nodes

terms (k)

$S_k = 2^k - 1$

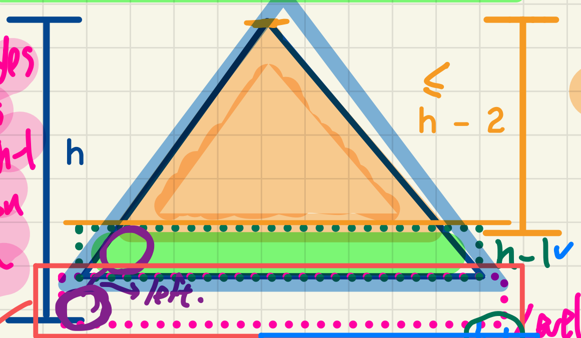


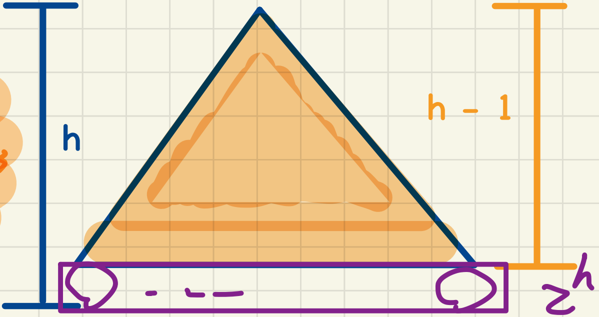
level
h-3
d=0
h-2
d=1
h-1
d=2
h
d=3



C3: child nodes of nodes at level h-1 filled from L to R

C2: each node has 0, 1, 2 child nodes





max: 2^h

Min # nodes? $*(2^0+2^1+\cdots+2^{h-1})+1 = 2^h$

Max # nodes? $(2^0+2^1+\cdots+2^{h-1})+2^h = 2^{h+1}-1$

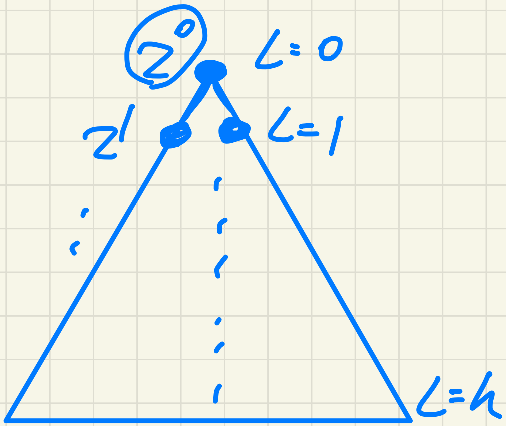
$2^h - 1$

Min # nodes?

Max # nodes? $(2^0+2^1+\cdots+2^h) = 2^{h+1}-1$

Geometric Sequence

$$S_{\underset{\text{\# terms}}{k}} = \frac{\overset{\text{first term}}{1} \cdot (\overset{\text{common factor}}{r}^{k} - 1)}{r - 1}$$



In case of BT with height = h

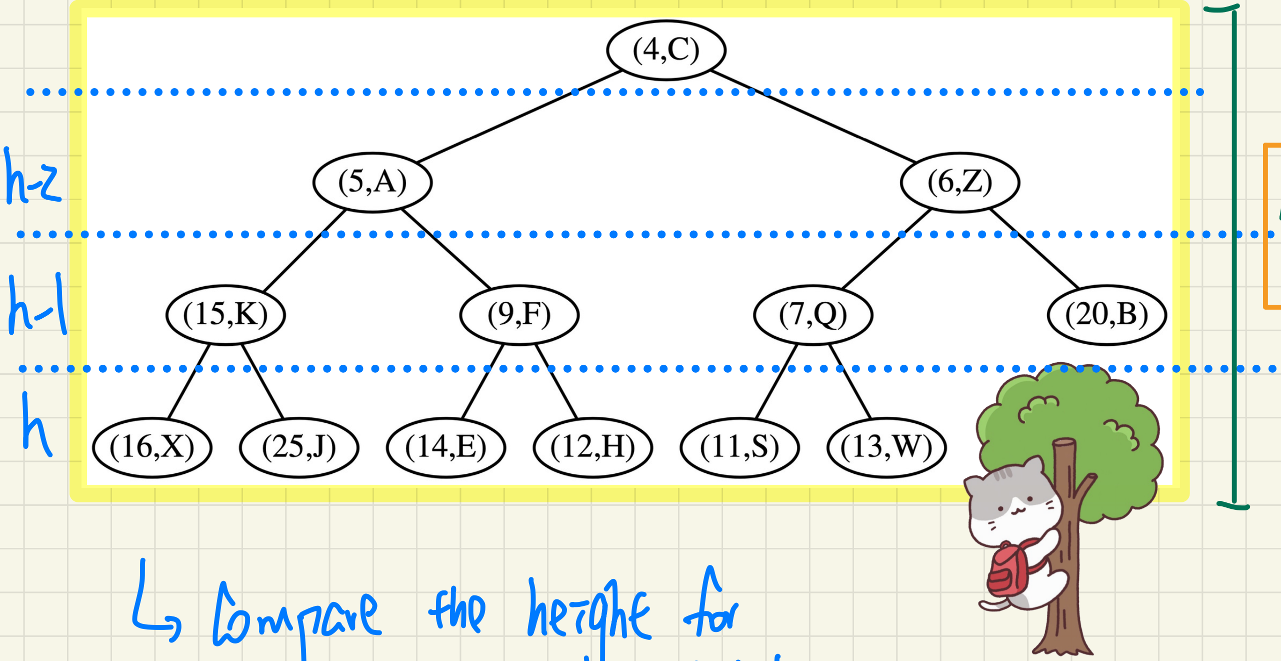
$$\frac{1 \cdot (2^{h+1} - 1)}{2 - 1} = 2^{h+1} - 1$$

Heaps: Structural Properties of Nodes

$2^{h+1} - 1 = n$ max # nodes

h is $O(\log n)$

Property: The tree is a complete Binary Tree



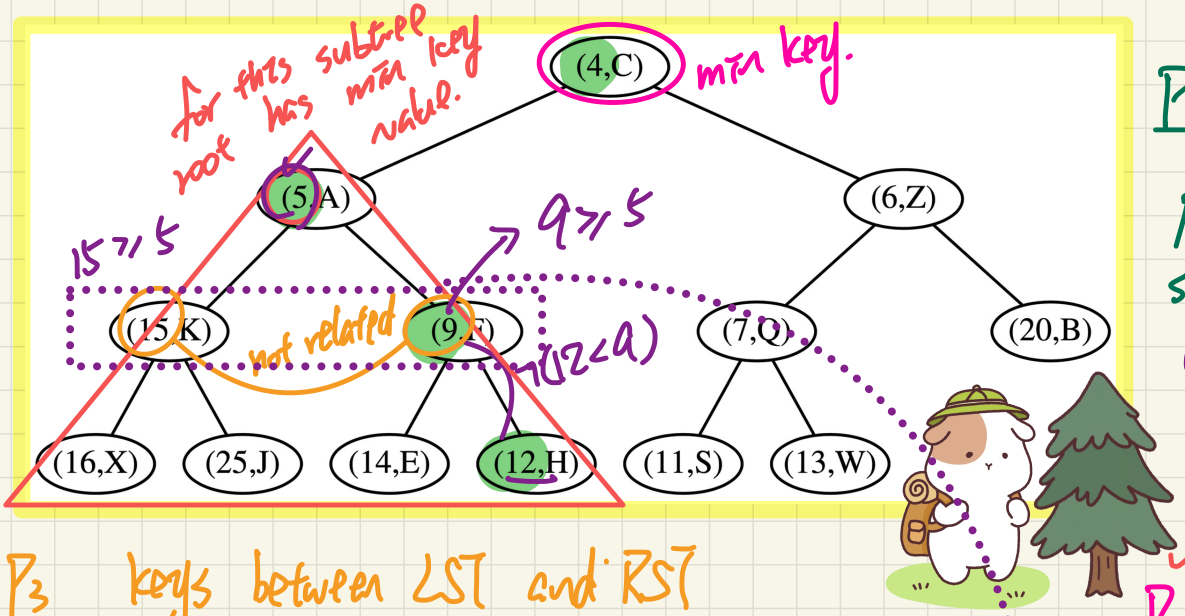
h is $O(\log n)$

↳ Compare the height for self-balancing BSTs.

Heaps: Relational Properties of Keys

Property: Each non-root node n is s.t. key(n) ≥ key(parent(n))

child



P1 Any leaf-to-root path generates a sorted seq. of keys. (non ascending order)

↳ child key = parent key
child key > parent key

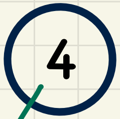
P2 min key stored in the root.

P3 . keys between LST and RST are not related

↳ the only property is w.r.t. the parent

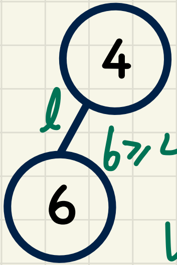
Example Heaps

Example 1



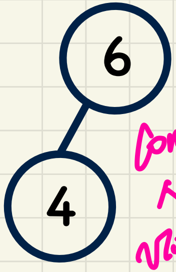
one-noded heap.

Example 2



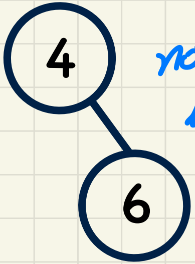
complete ^ sat. HOP. ✓

Example 3



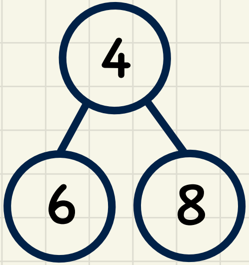
complete ^ violates HOP

Example 4

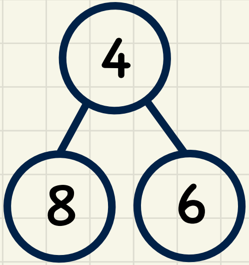


not complete but sat HOP.

Example 5



Example 6



heaps!

Lecture 19 - Nov 19

Graphs

Heap Operations: Insertion vs. Deletion
Dijkstra's Algorithm: Time Complexity
Implementing Graphs: Adjacency Matrix

Announcements/Reminders

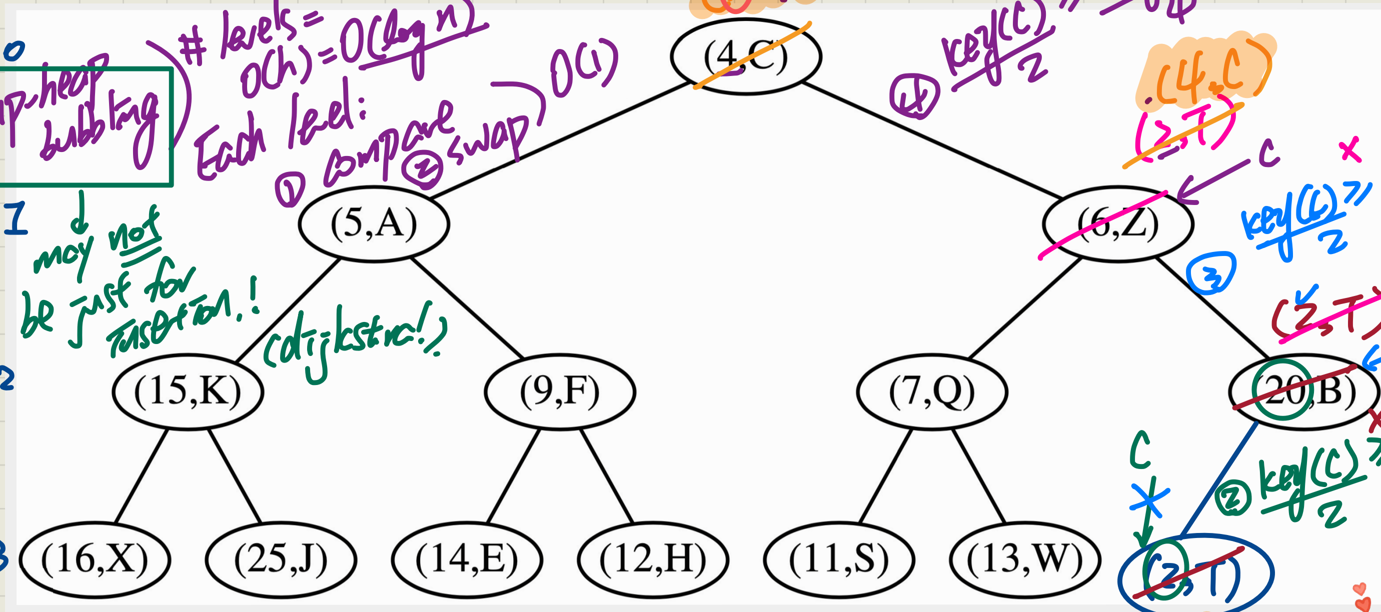
- Today's class: notes template posted
- Assignment 2 released
- Change of Dates:
    - + Assignment 2 to be due on Wed, Nov 19
    - + Test 2 to be take place on Mon, Nov 24
- Online Course Evaluation

Heap Operations: Insertion

new min → RT: O(log n · 1) = O(log n)

key

Insert a new entry (2, T)



up-heap bubbling

levels = O(h) = O(log n)

Each level: ① compare ② swap } O(1)

↓ may not be just for insertion! (dijkstra!)

① Store new entry as right-most node at level h.

← keep a ref. to the node for which new node should be inserted as a child.

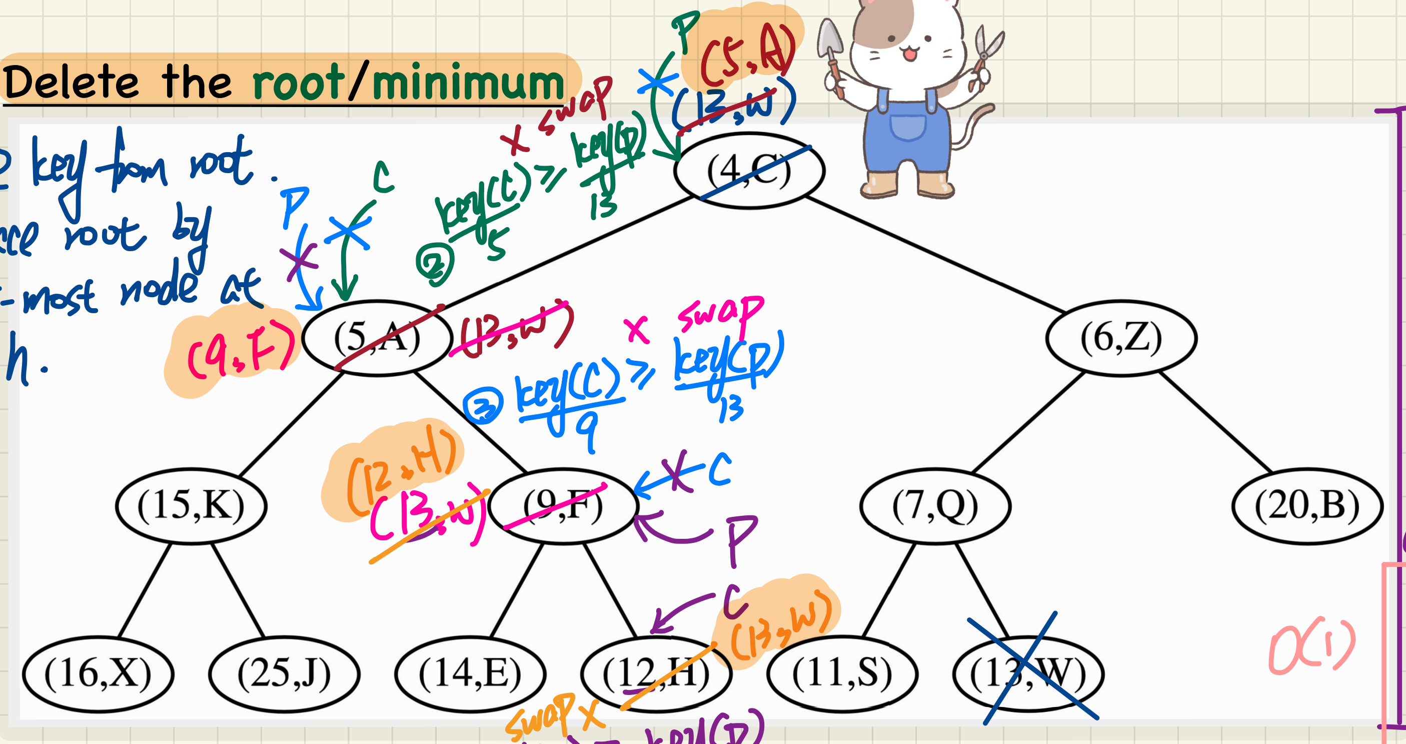
trading space for time

Heap Operations: Deletion

K = 4

Delete the root/minimum

① store key from root. Replace root by right-most node at level h.



④ key(C) ≥ key(P)
12 13

⑤ return k: 4.

down-heap bubbling:

levels: O(h) " O(log n)

Each level:
① Choose C
② Compare
③ swap

O(1)

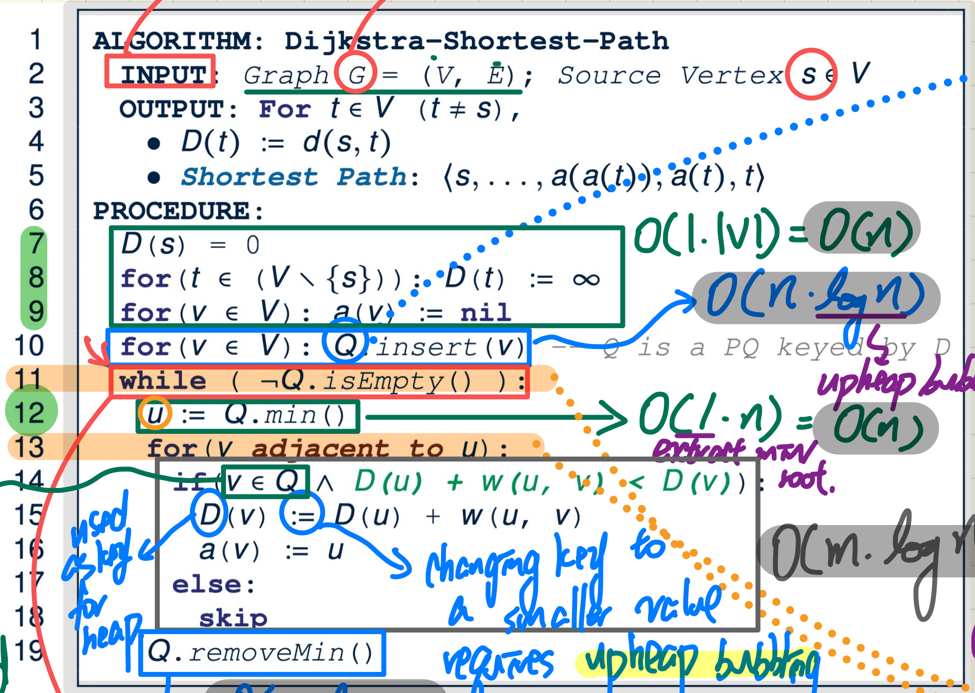
RT: O(1 · log n) = O(log n)

$|V| = n \quad |E| = m$

Dijkstra's Shortest Path Algorithm: Time Complexity

$w \geq 0$

directed or undirected



```
ALGORITHM: Dijkstra-Shortest-Path
INPUT: Graph G = (V, E); Source Vertex s ∈ V
  OUTPUT: For t ∈ V (t ≠ s),
    • D(t) := d(s,t)
    • Shortest Path: ⟨s, ..., a(a(t)), a(t), t⟩
PROCEDURE:
  D(s) = 0
  for(t ∈ (V \ {s})): D(t) := ∞
  for(v ∈ V): a(v) := nil
  for(v ∈ V): Q.insert(v)    -- Q is a PQ keyed by D
  while ( ¬Q.isEmpty() ):
    u := Q.min()
    for(v adjacent to u):
      if(v ∈ Q ∧ D(u) + w(u, v) < D(v)):
        D(v) := D(u) + w(u, v)
        a(v) := u
      else:
        skip
    Q.removeMin()
```

$O(1 \cdot |V|) = O(n)$

$O(n \cdot \log n)$

↳ upheap bubbling

$O(1 \cdot n) = O(n)$

extract min root.

$O(m \cdot \log n)$

used as key for heap

changing key to a smaller value requires upheap bubbling to restore the rel. property.

this special op. should not be more expensive than $O(\log n)$

↳ e.g. balanced BST.

↳ $O(n \cdot \log n)$

iterations = $|V| = n$

Priority Queue

~ Implemented by heap

(h is $O(\log n)$) $|V|$

~ extract min: $O(1)$

~ insertion & deletion:

$O(h) = O(\log n)$

RT: $O((m+n) \cdot \log n)$

graph is complete

$O(n^2 \cdot \log n)$

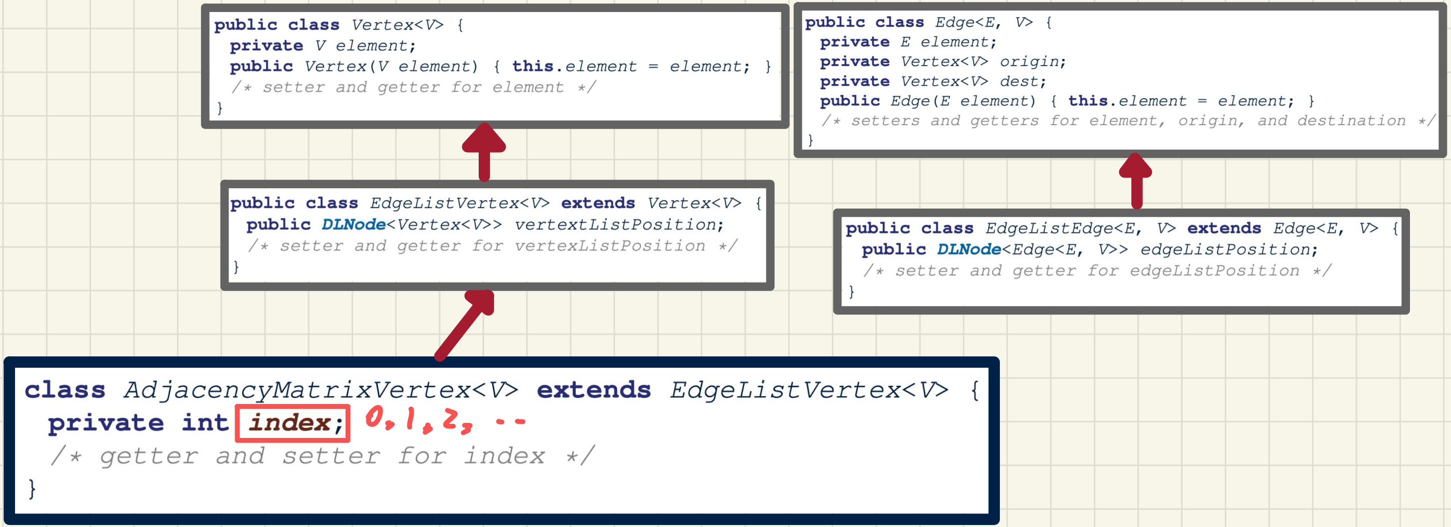
$\sum_{u \in V}$ # adjacent vertices of u

$= \sum_{u \in V} deg(u) = |E| = m$

directed graph

Graphs in Java: Adjacency Matrix Strategy (1)

```
class AdjacencyMatrixGraph<V, E> implements Graph<V, E> {
  private DoublyLinkedList<AdjacencyMatrixVertex<V>> vertices;
  private DoublyLinkedList<EdgeListEdge<E, V>> edges;
  private boolean isDirected;

  private EdgeListEdge<E, V>[][] matrix;

  /* initialize an empty graph */
  AdjacencyMatrixGraph(boolean isDirected) {
    this.vertices = new DoublyLinkedList<>();
    this.edges = new DoublyLinkedList<>();
    this.isDirected = isDirected;
  }
```



```
public class Vertex<V> {
  private V element;
  public Vertex(V element) { this.element = element; }
  /* setter and getter for element */
}
```

```
public class Edge<E, V> {
  private E element;
  private Vertex<V> origin;
  private Vertex<V> dest;
  public Edge(E element) { this.element = element; }
  /* setters and getters for element, origin, and destination */
}
```

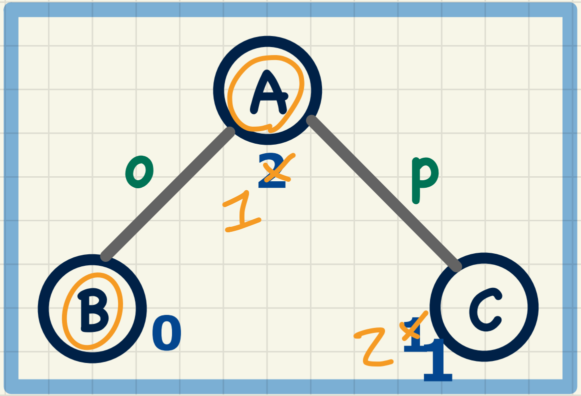
```
public class EdgeListVertex<V> extends Vertex<V> {
  public DLNode<Vertex<V>> vertextListPosition;
  /* setter and getter for vertexListPosition */
}
```

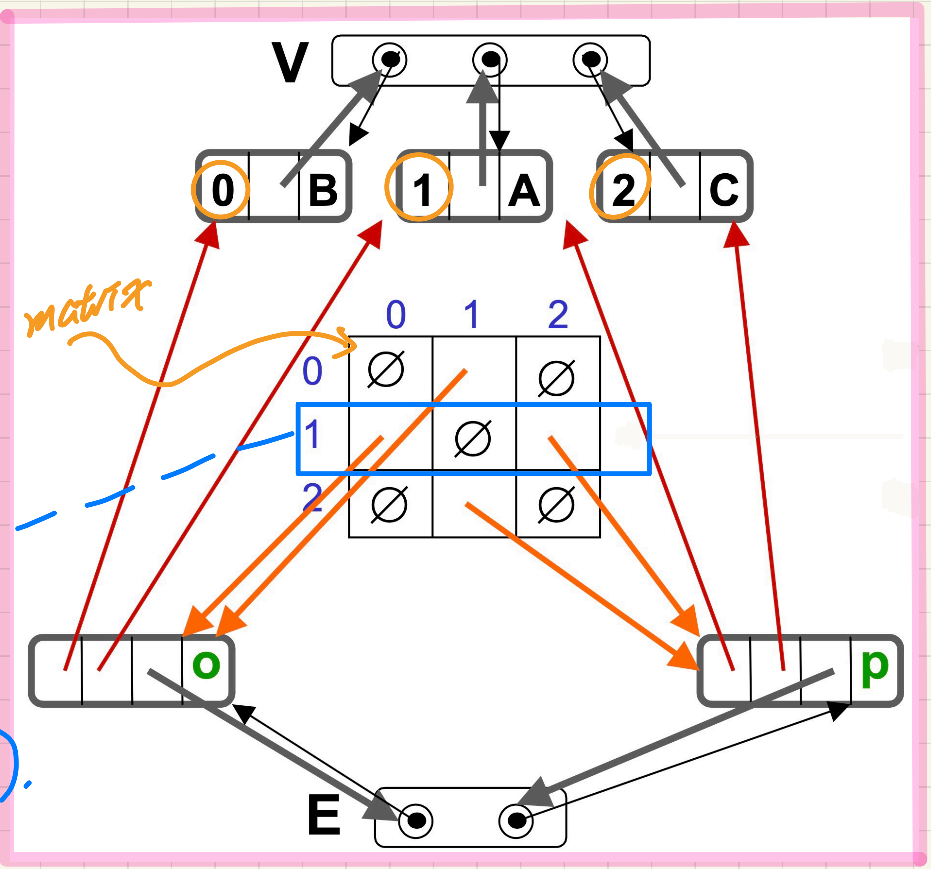
```
public class EdgeListEdge<E, V> extends Edge<E, V> {
  public DLNode<Edge<E, V>> edgeListPosition;
  /* setter and getter for edgeListPosition */
}
```

```
class AdjacencyMatrixVertex<V> extends EdgeListVertex<V> {
  private int index;
  /* getter and setter for index */
}
```

0, 1, 2, --

Graphs in Java: Adjacency Matrix Strategy (2)





	0	1	2
0	Ø		Ø
1		Ø	
2	Ø		Ø

matrix[1]
(incident edges list
for vertex A
with index 1).

Tutorials - Week 11 - Nov 21

Graphs

Assignment 2 Solution

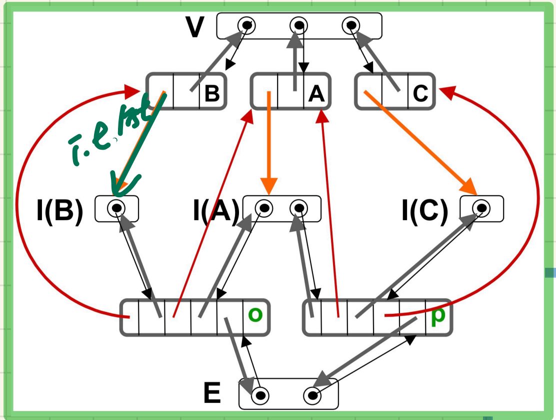
Graph Implementation Strategies

Graphs in Java: Strategies

matrix[1][0] == matrix[0,1]

adjacency matrix

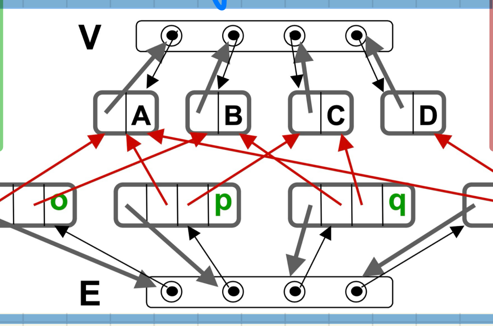
<table>
<tr><th></th><th>VERTEX</th><th>EDGE</th><th>GRAPH</th></tr>
<tr><td rowspan="3">ADJACENCY LIST</td><td rowspan="2">incidentEdges</td><td>originIncidentListPos</td><td rowspan="5">isDirected
vertices
edges</td></tr>
<tr><td>destIncidentListPos</td></tr>
<tr><td rowspan="3">vertexListPosition</td><td rowspan="4">edgeListPosition</td></tr>
<tr><td>EDGE LIST</td></tr>
<tr><td rowspan="2">ADJACENCY MATRIX</td></tr>
<tr><td>index</td><td>matrix</td></tr>
</table>

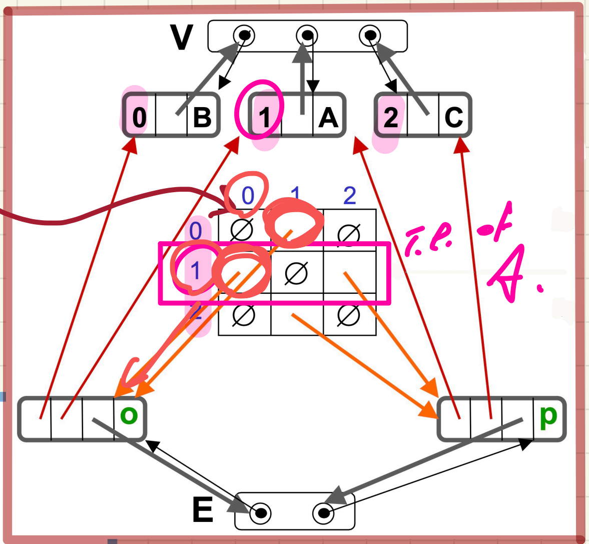


Adjacency List.

matrix (edges).

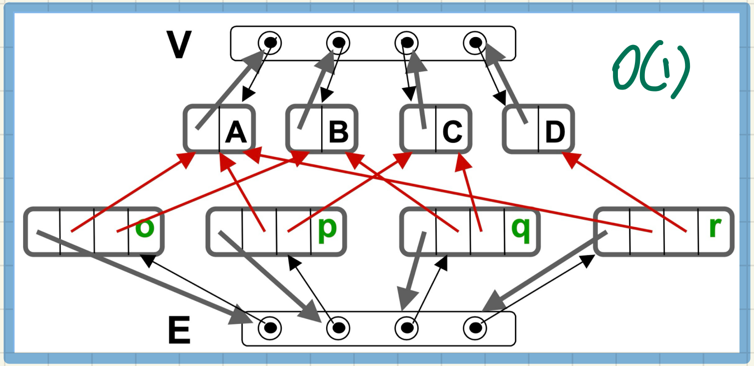
Edge List





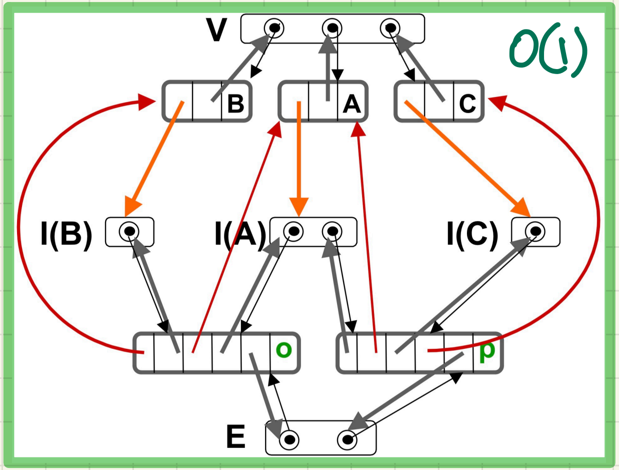
	0	1	2
0	∅		∅
1	∅	∅	
2	∅		∅

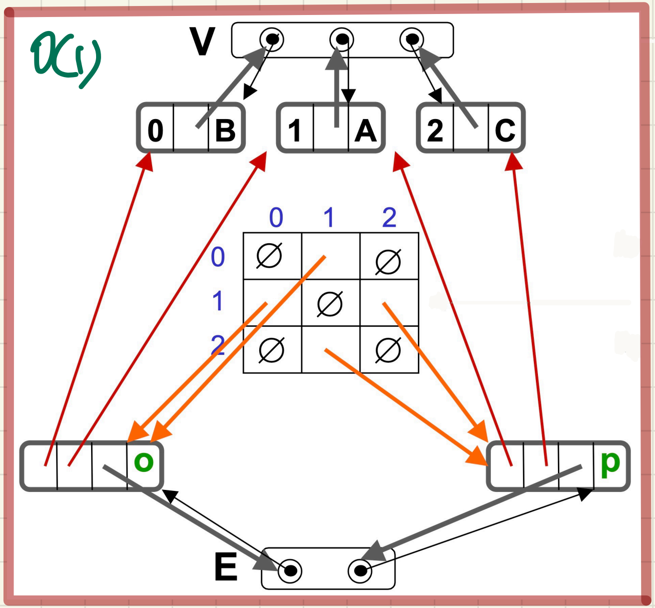
Graphs in Java: Time Complexities (1)



numVertices(), numEdges()

attributes of DLL. (size).





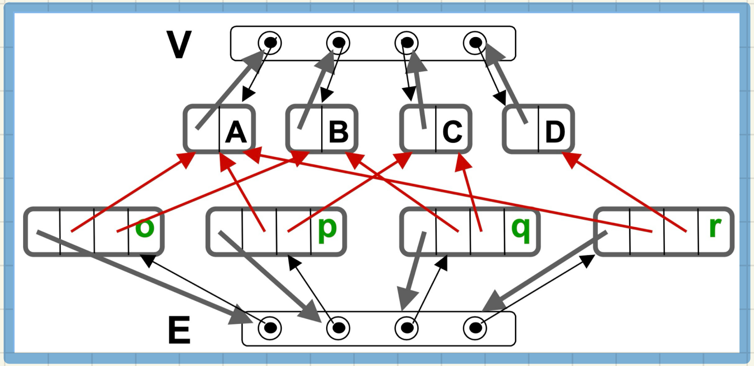
Graphs in Java: Time Complexities (2)

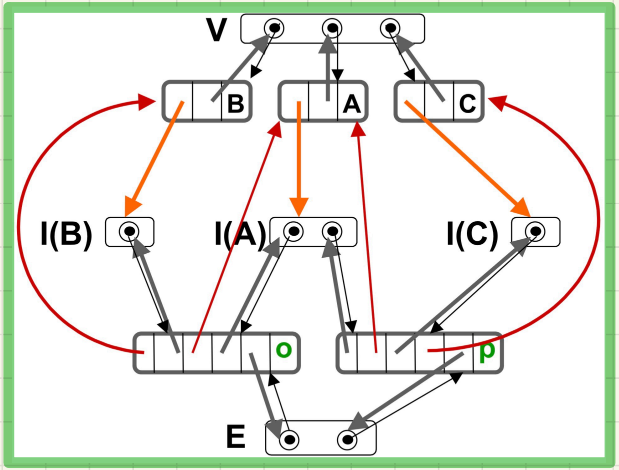
|V| = n
|E| = m

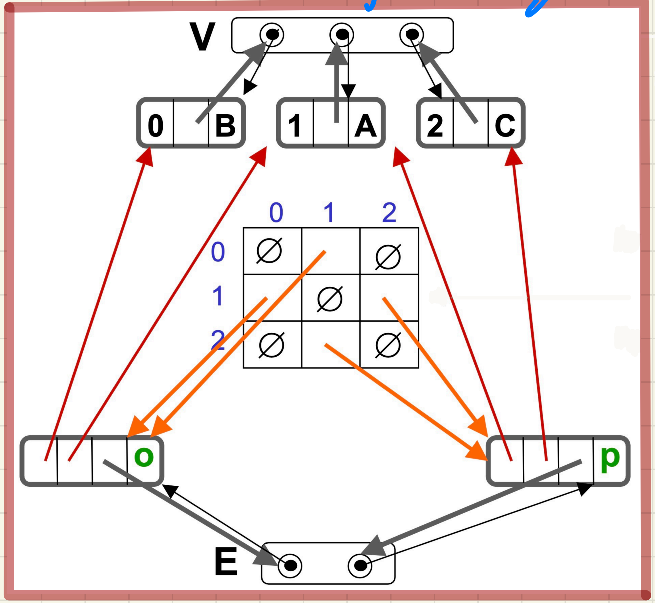
vertices(), edges()

O(n) O(m)

go over DLLs to create the iterable objects (e.g., ArrayList).







	0	1	2
0	∅		∅
1		∅	
2	∅		∅

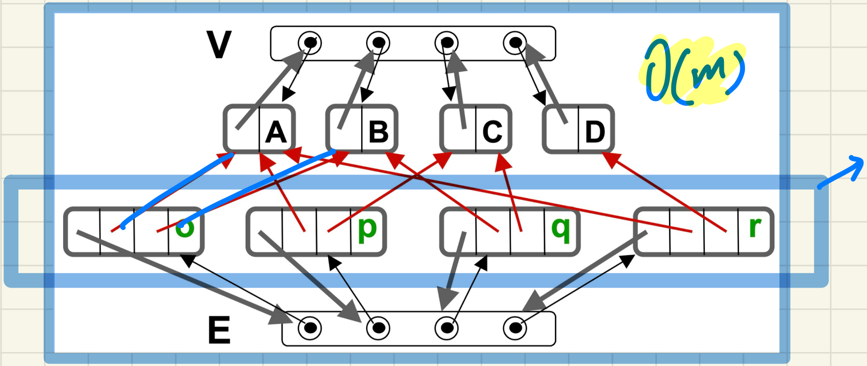
Graphs in Java: Time Complexities (3)

* A.getIndex()

** C.getIndex()

✓

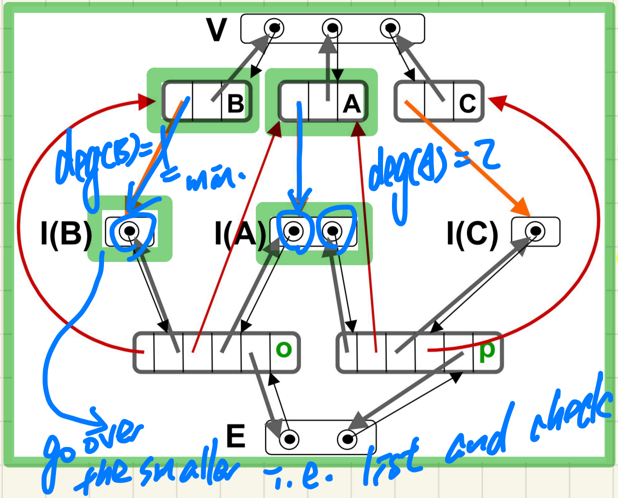
getEdge(u, v)



O(m)

go over m edges and check ori. & des. against u & v.

getEdge(A, C)
↳ matrix[*][**]



deg(B)=1 = min.

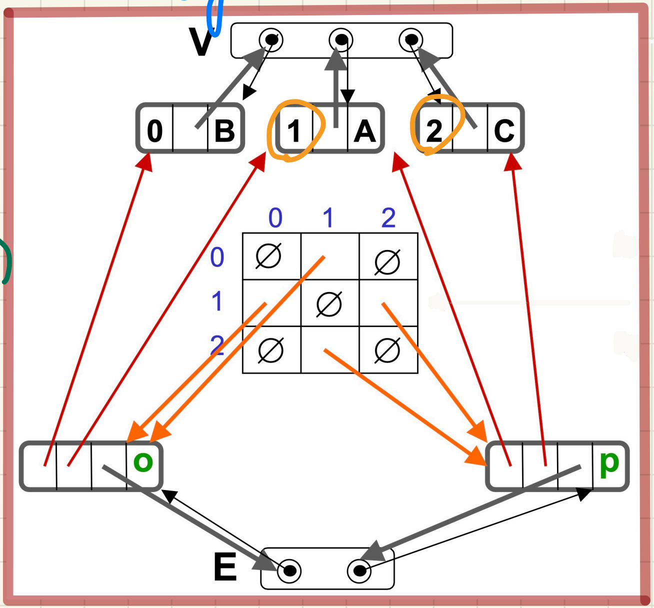
deg(A)=2

go over the smaller i.e. list and check ori. & des. against A & B.

getEdge(A, B)

$O(\min(d_u, d_v))$

↳ possible n-1



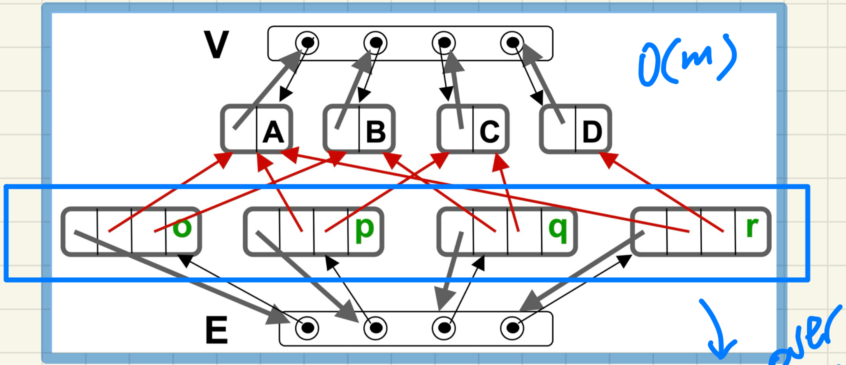
	0	1	2
0	∅		∅
1		∅	
2	∅		∅

Graphs in Java: Time Complexities (4)

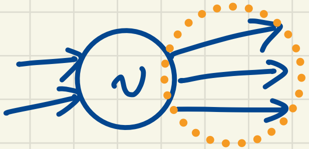
go over columns at the row indexed by v, count those whose ori. is v.

outDegree(u), inDegree(u)

inEdges(v), outEdges(v)



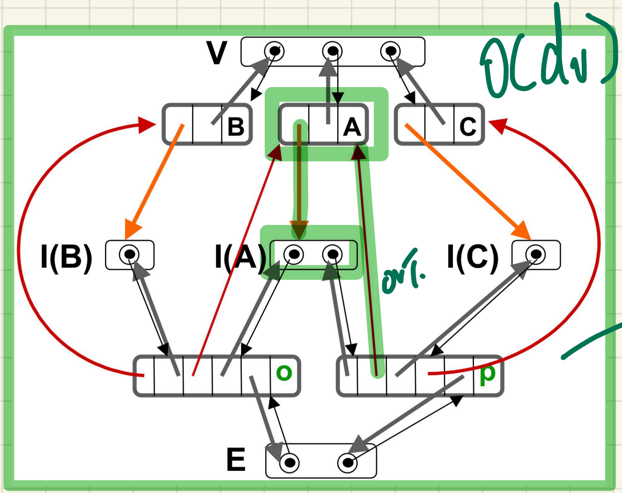
O(m)



v is the ori. vertex.

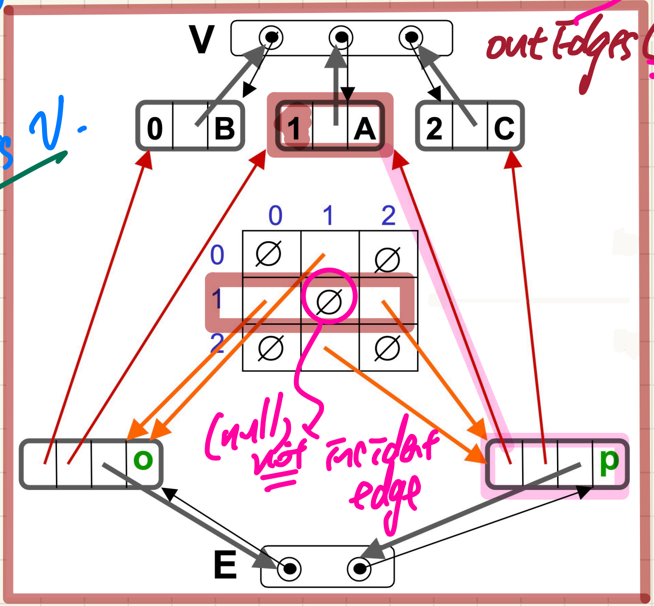
go over m edges

count those whose origins is v.



O(dv)

go over v's i.e. list count those whose origin is v.

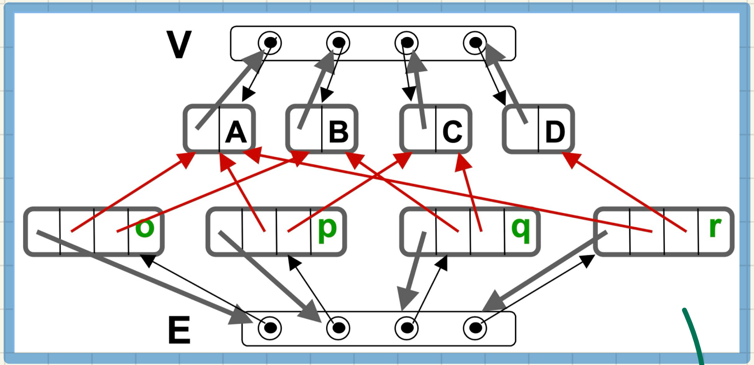


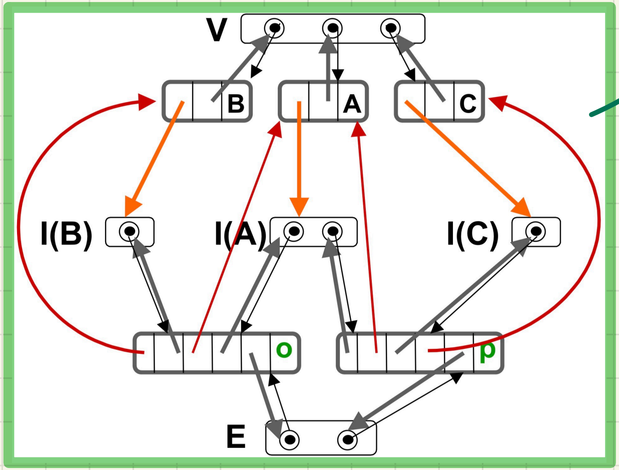
	0	1	2
0	∅		∅
1		∅	
2	∅		∅

outEdges(A)

(null) ~~not~~ incident edge

Graphs in Java: Time Complexities (5)





O(1)

insert last to DLL.

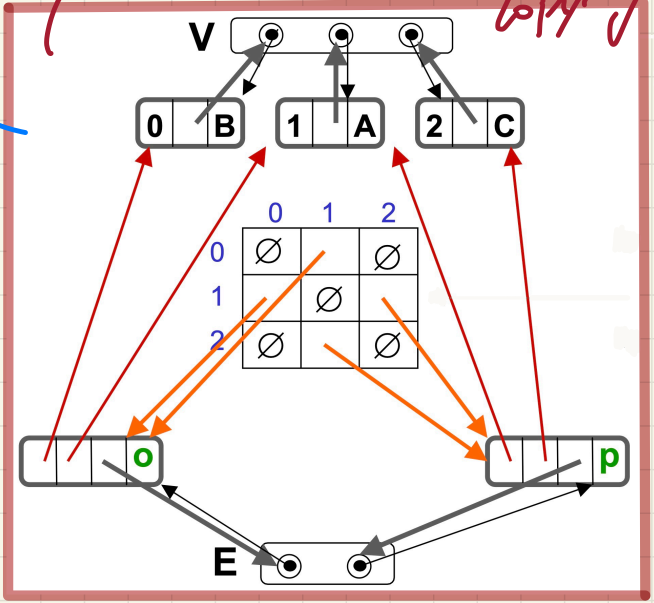
insertVertex(x)

what if AL<AL<E>> is used?

n x n

↳ (n+1) x (n+1) matrix

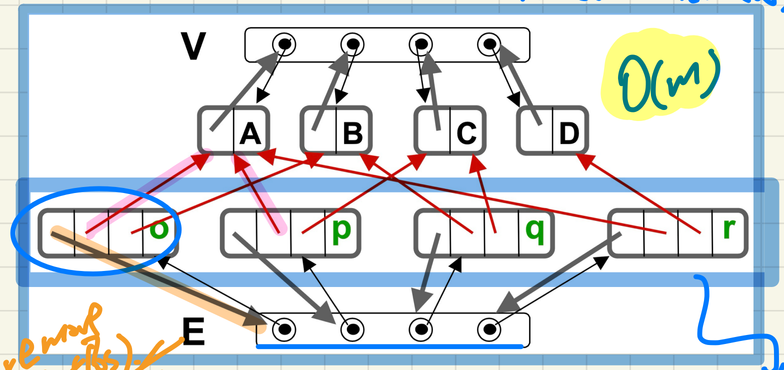
O(n²) copying.



	0	1	2
0	Ø		Ø
1		Ø	
2	Ø		Ø

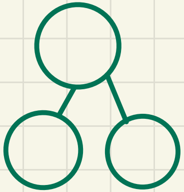
Graphs in Java: Time Complexities (6)

removeVertex(A).



E.remove(o.getPos());

removeVertex(v)



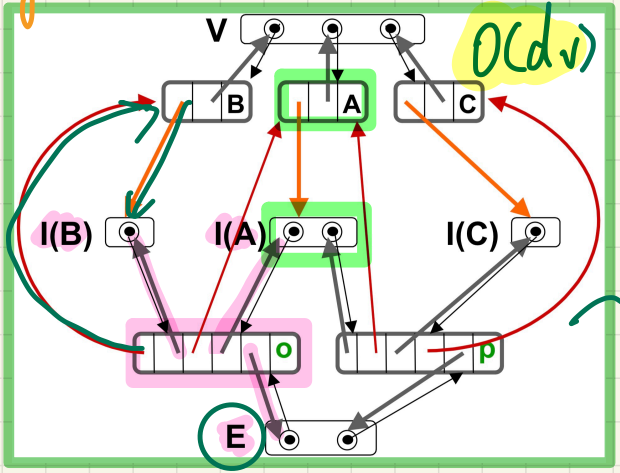
* each edge should be removed from:

(1) list of edges

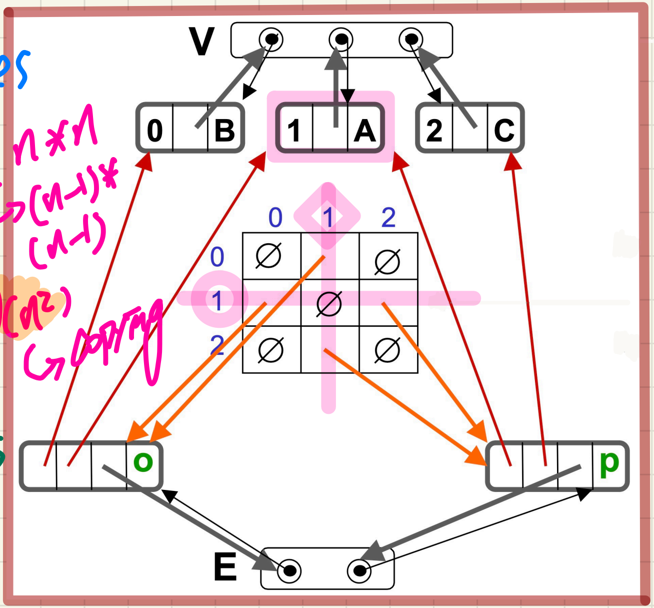
(2) ori's i.e. list

(3) des's i.e. list

go over m edges and remove those whose ori. or des. is v.



go over v's i.e. lists and remove all edges from there *



	0	1	2
0	∅		∅
1		∅	
2	∅		∅

$n*n$

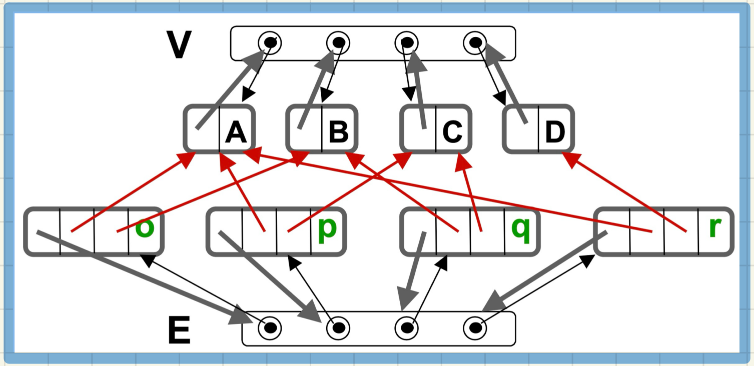
↳ $(n-1)*(n-1)$

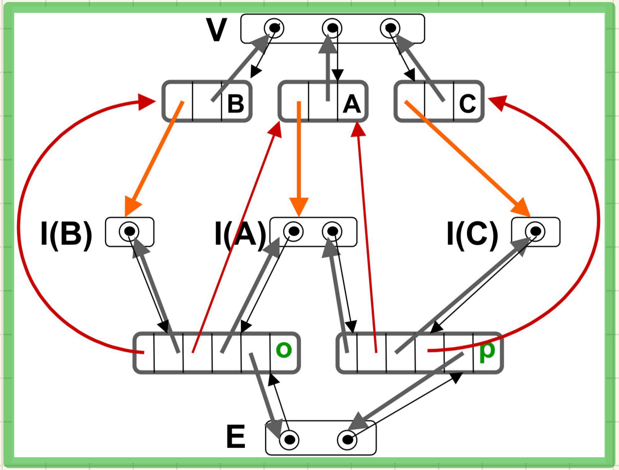
$O(n^2)$

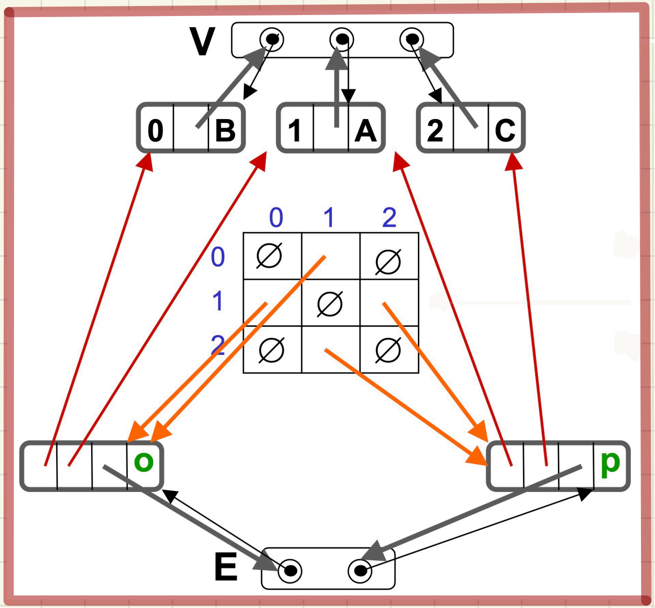
↳ copying

Graphs in Java: Time Complexities (7)

insertEdge(u, v, x),
removeEdge(e)







	0	1	2
0	∅		∅
1		∅	
2	∅		∅

Lecture 20 - Nov 26

Graphs

Directed Acyclic Graphs (DAGs)
Topological Ordering
Topo. Sort: Time Complexity, Tracing
Topo. Sort: Sequentializing Updates

Announcements/Reminders

- Today's class: notes template posted
- One in-person make-up lecture
- One or two exam review sessions

M	T	W	T	F
1	2	3 study day	4	5
8	9	10	11	12
15	16	17 Exam		

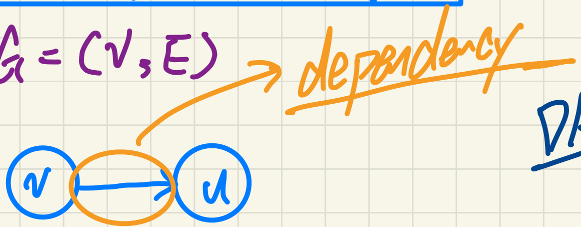
12 → 11 am

12 noon.

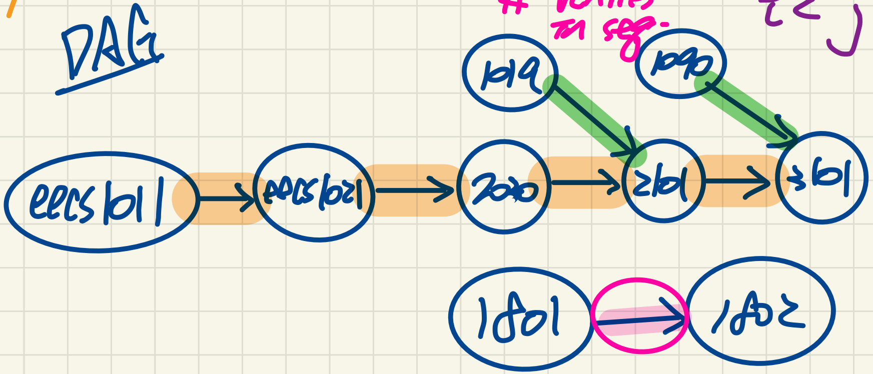
Directed Acyclic Graphs (DAGs)

$$\forall i, j \bullet 1 \le i, j \le \underbrace{n}_{\text{\# vertices in seq.}} \land (v_i, v_j) \in E \Rightarrow i < j$$

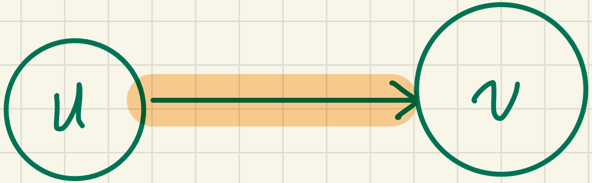
DAG $G = (V, E)$

↳ 

↳ no cycles.



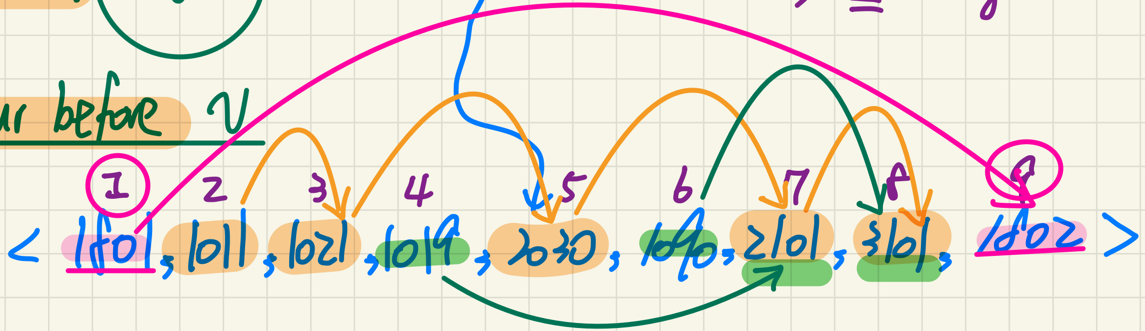
Exercise : DAG vs. Tree



u must occur before v

Topological Order (sequentialized DAG)

↳ not unique.



< 1801, 1011, 1021, 1019, 2030, 1090, 2101, 3101, 1802 >

input DAG → a topological order

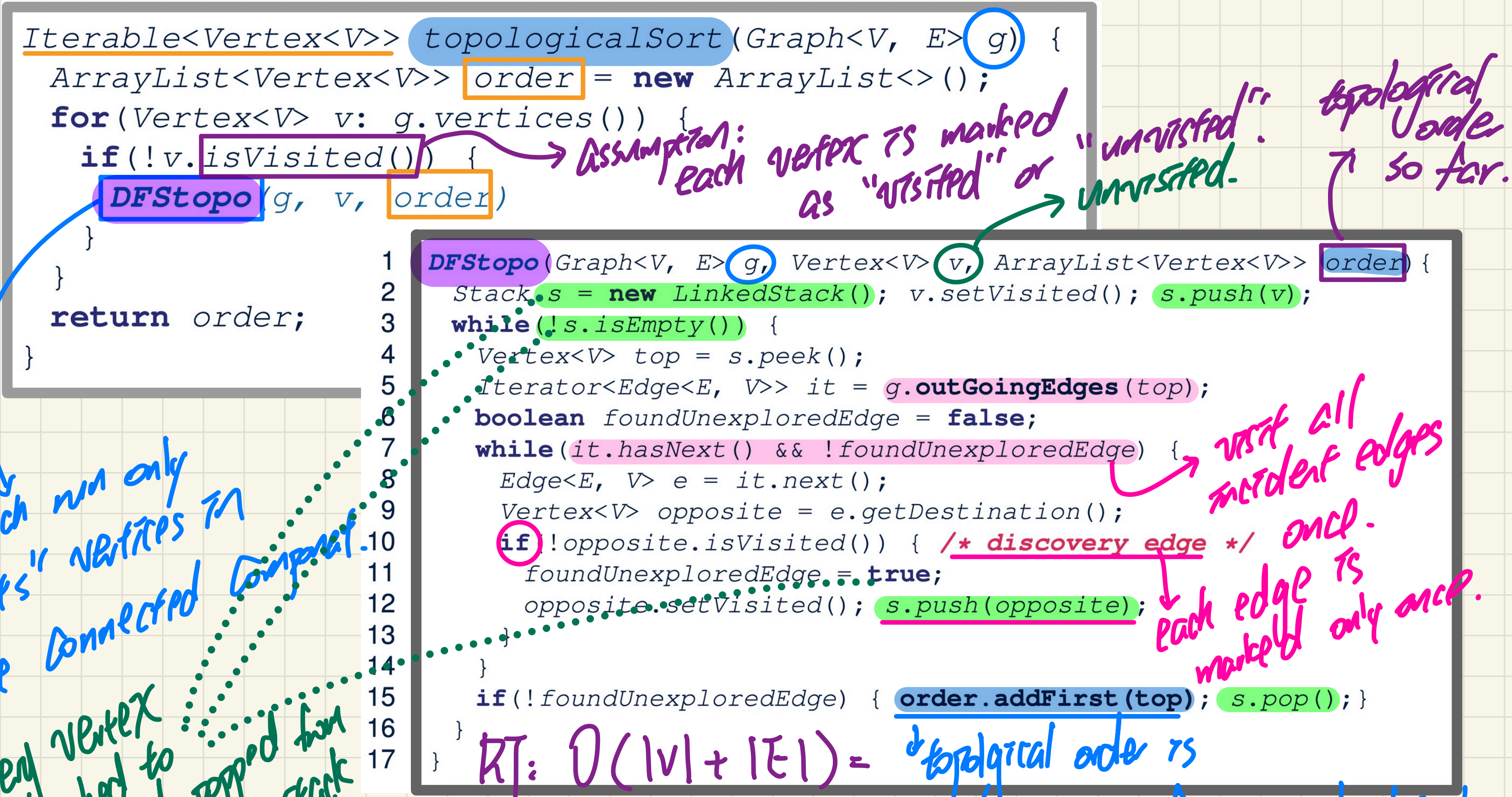
topological sort.

slight extension to DFS.

(a standard BFS cannot achive this).

based on dependency links in the DAG.

Topological Sort on a DAG: Time Complexity



```
Iterable<Vertex<V>> topologicalSort(Graph<V, E> g) {
  ArrayList<Vertex<V>> order = new ArrayList<>();
  for(Vertex<V> v: g.vertices()) {
    if(!v.isVisited()) {
      DFStopo(g, v, order)
    }
  }
  return order;
}
```

```
DFStopo(Graph<V, E> g, Vertex<V> v, ArrayList<Vertex<V>> order) {
  Stack s = new LinkedStack(); v.setVisited(); s.push(v);
  while(!s.isEmpty()) {
    Vertex<V> top = s.peek();
    Iterator<Edge<E, V>> it = g.outGoingEdges(top);
    boolean foundUnexploredEdge = false;
    while(it.hasNext() && !foundUnexploredEdge) {
      Edge<E, V> e = it.next();
      Vertex<V> opposite = e.getDestination();
      if(!opposite.isVisited()) { /* discovery edge */
        foundUnexploredEdge = true;
        opposite.setVisited(); s.push(opposite);
      }
    }
    if(!foundUnexploredEdge) { order.addFirst(top); s.pop();}
  }
}
```

Assumption: each vertex is marked as "visited" or "unvisited".

unvisited.

topological order so far.

visit all incident edges once.

each edge is marked only once.

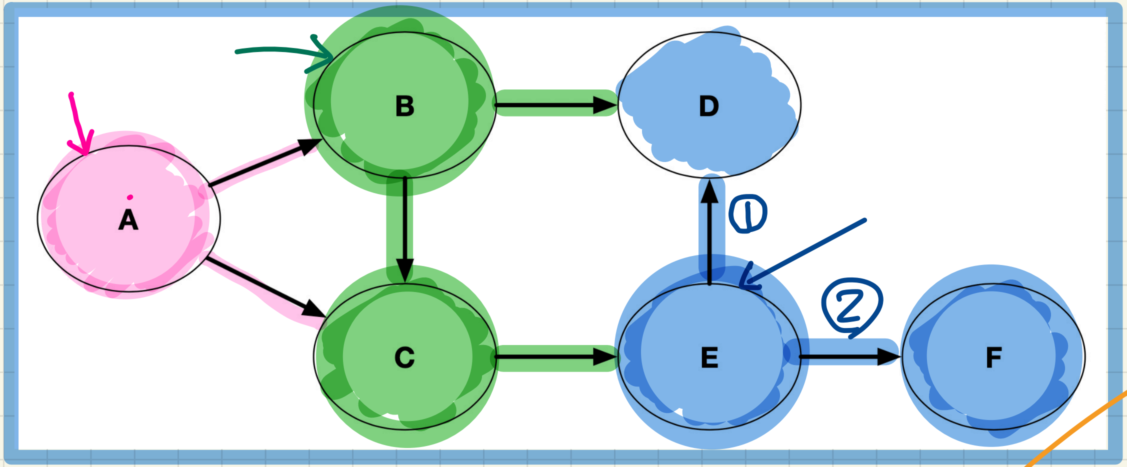
each run only "sorts" vertices in some connected component.

every vertex is pushed to and popped from the stack once.

RT: O(|V| + |E|) = O(n + m)

topological order is reverse of vertices backtracking.

Topological Sort on a DAG: Example (1)



Assume T.E. visited in alphabetic order.

ultimate topological order to return

A B C E F D

Iteration 1 DFS$_{topo}$(E, < >)

push	pop
E	D.
D	F
F	E

~~F~~
~~D~~
~~E~~

order :< E F D >

Iteration 2 DFS$_{topo}$(B, <E, F, D>)

push	pop
B	C.
C	B

~~C~~
~~B~~

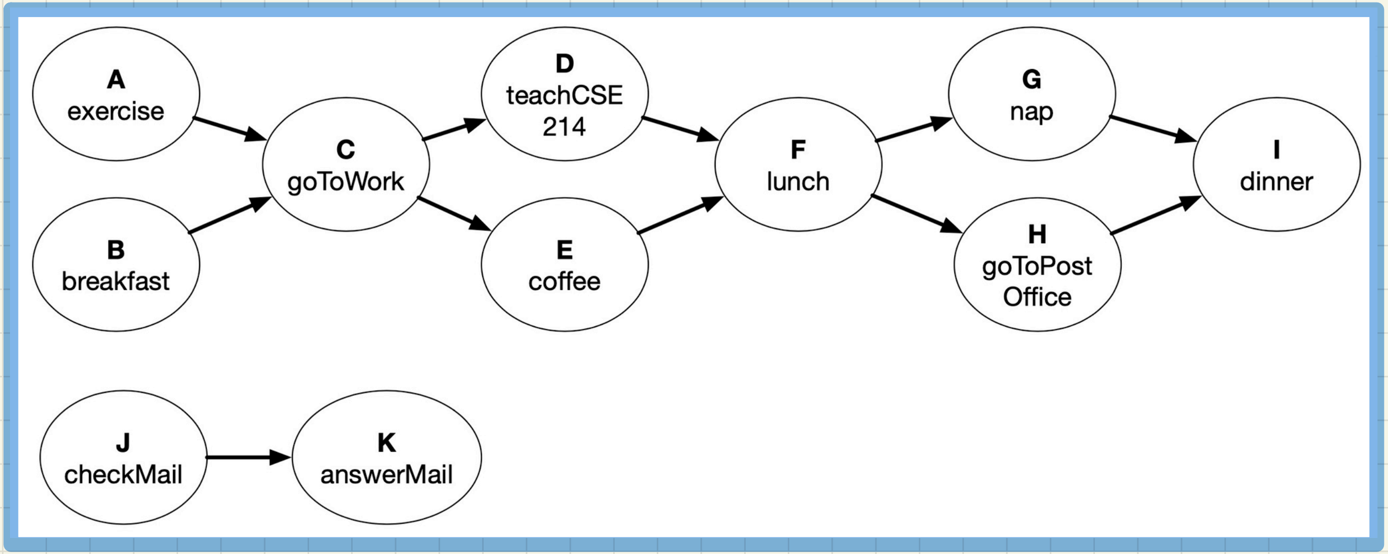
B C E F D

Iteration 3. DFS$_{topo}$ (A, <B, C, E, F, D>)

~~A~~

push	pop
A	A

Topological Sort on a DAG: Example (2)



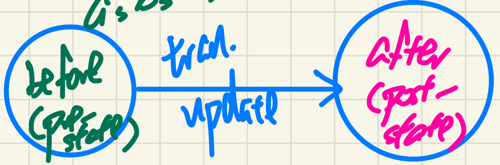
Topological Sort on a DAG: Example (3)

$$
\begin{array}{rl}
 & c' = b' - a' + d \\
\wedge & b' = a' * d' \\
\wedge & d' = a + c + 5 \\
\wedge & a' = a + d'
\end{array}
$$

post-state values of vars.

→ specification of transactional updates (as constraints, not var. assignments).

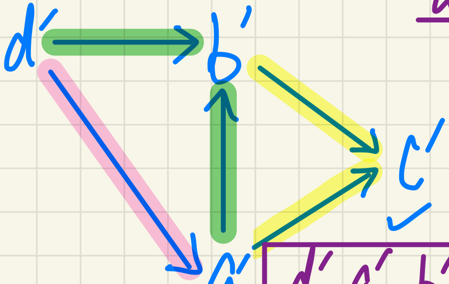
* c' = b'
↳ b' should be computed before c'



Task: Sequentialize the specified transactional updates.

```
old_a := a
old_b := b
old_c := c
old_d := d
d := old_a + old_c + 5
a := old_a + d
b := a * d
c := b - a + old_d
```

DAG



topological sort

DFS_topo(d', <>)

~~c'~~
~~b'~~
~~a'~~
~~d'~~

push	pop
d'	c'
a'	b'
b'	a'
c'	d'

*

d'	a'	b'	c'

topological order.

Lecture 21 - Dec 1

Graphs

Minimum Spanning Tree (MST) Problem
Greedy Method
Kruskal’s Greey Algorithm

Announcements/Reminders

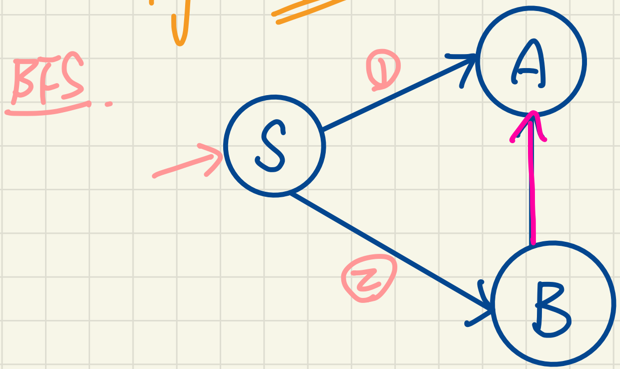
- Today's class: notes template posted
- Survey on in-person make-up lecture & exam review **active**!

M	T	W	T	F
1	2	study day 3	4	5
8	9	10	11	12
15	16	17 Exam		

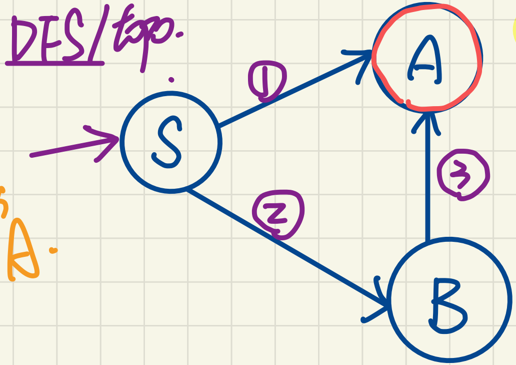
Topological Sort on a DAG: BFS?

Consider: g = ({S, A, B}, {(S, A), (S, B), (B, A)})

legal DAG.



B happens before A.



* when A is popped out, all outgoing edges of A have been handled. (anything that A depends on have been handled)

enqueue	dequeue
S	S
A	A
B	B

this order does not consider the fact that B depends on A

↳ not suitable for gen. topo. order.

push	pop
S	A *
A	B
B	S

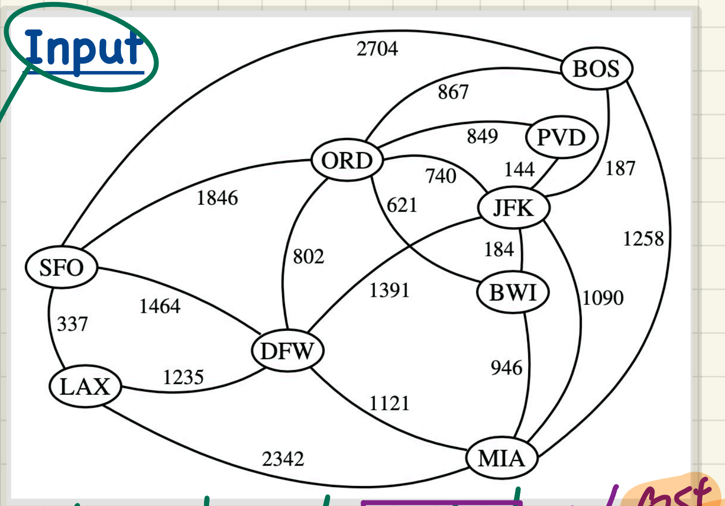
~~B~~
~~A~~
~~S~~.

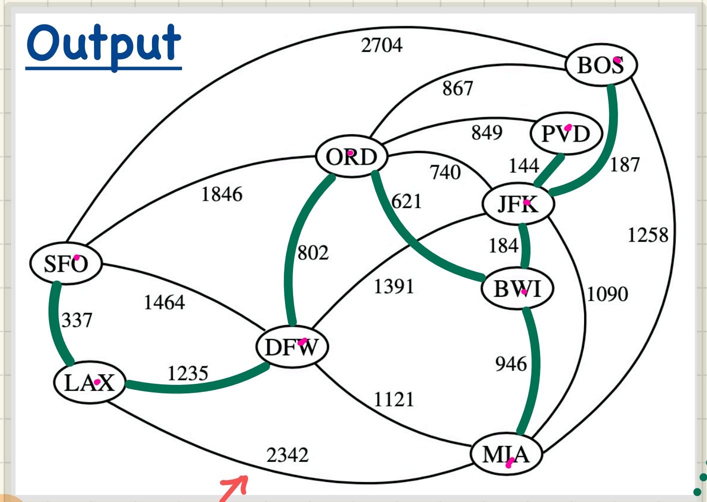
S, B, A

topological order.

T: a set of edges.

Minimum Spanning Tree (MST) Problem





simple, undirected, weighted → 1. cost (min) 2. benefit/score (max)

subgraph. forest.

- Tree : undirected, acyclic, connected
- Spanning Tree : tree ∧ spanning subgraph
- Minimum Spanning Tree : $W(T) = \sum_{(u,v) \in T} w(u,v)$

e.g. MST T = { (BOS, JFK), (PVD, JFK), (JFK, BWI), (BWI, ORD), (BWI, MIA), (ORD, DFW), (DFW, LAX), (LAX, SFO) }

when building a MST iteratively, as soon as |T| = |V| - 1 terminate.

|V| = 9

graph is connected

↳ $\underbrace{|T|}_{\text{\# edges in spanning tree}} = \boxed{|V| - 1} = 8$

MST Problem: Greedy Method

- a design method for algorithms

① - step-by-step (iterative) construction of some complete solution → globally-optimal.

- from each step/iteration:

② • multiple feasible choices exist to "extend" the partial solution so far.

③ • pick the choice that's the best "right now" w.r.t. a * cost/score func.

↳ locally-optimal, not necessarily making the solution so far the globally-optimal

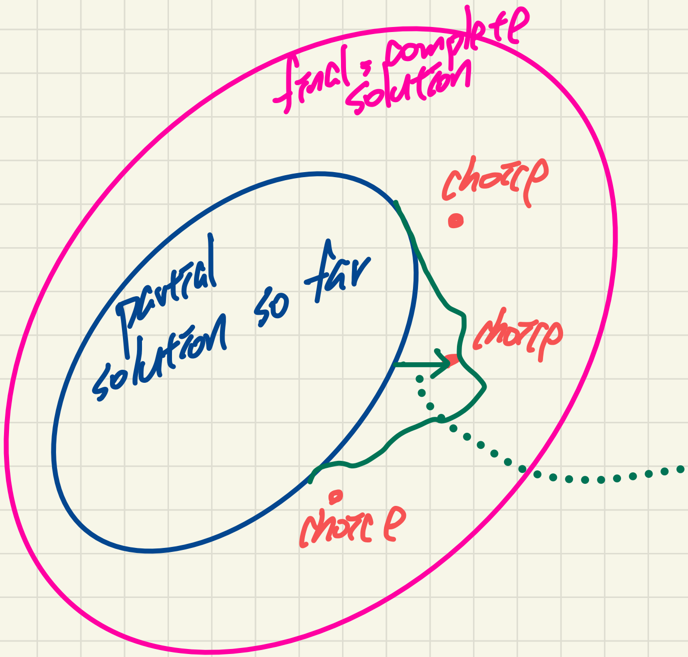
④ • once a choice is made, never undo it.

backtrack

* cost/score function

- rank choices (e.g. D in Dijkstra)
- measure partial solution so far.

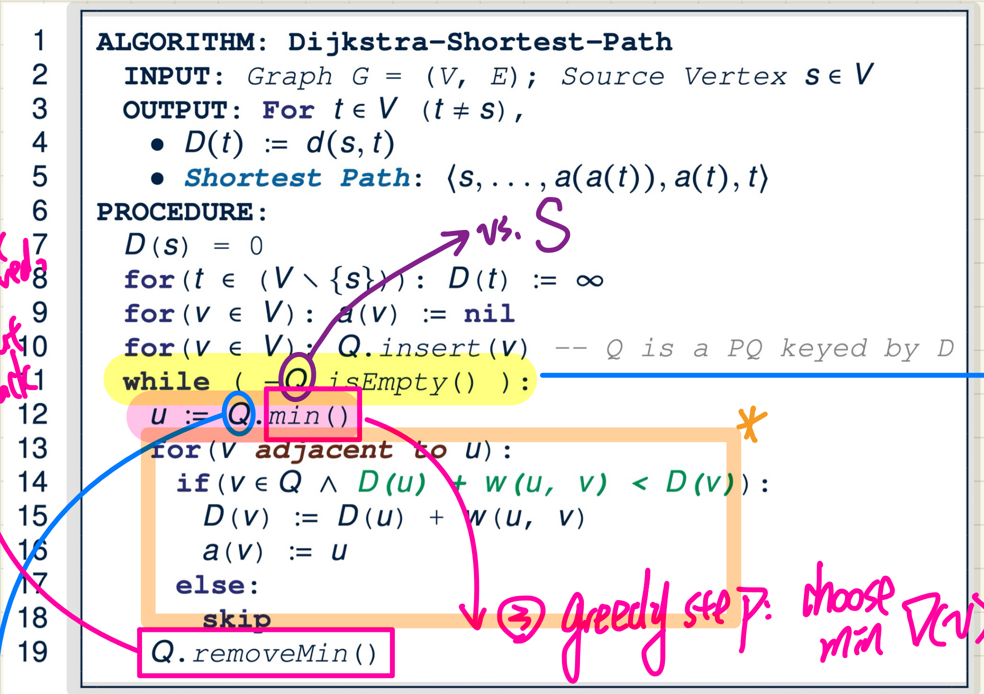
Greedy Method



greedy step: opt for the choice with min cost

each g.s. guarantees that we're one step closer to const. the complete solution.

Greedy Method Example: Dijkstra's Algorithm



```
ALGORITHM: Dijkstra-Shortest-Path
  INPUT: Graph G = (V, E); Source Vertex s ∈ V
  OUTPUT: For t ∈ V (t ≠ s),
    • D(t) := d(s, t)
    • Shortest Path: ⟨s, . . . , a(a(t)), a(t), t⟩
PROCEDURE:
  D(s) = 0
  for(t ∈ (V \ {s})): D(t) := ∞
  for(v ∈ V): a(v) := nil
  for(v ∈ V): Q.insert(v) -- Q is a PQ keyed by D
  while ( ¬Q.isEmpty() ):
    u := Q.min()
    for(v adjacent to u):
      if(v ∈ Q ∧ D(u) + w(u, v) < D(v)):
        D(v) := D(u) + w(u, v)
        a(v) := u
      else:
        skip
    Q.removeMin()
```

vs. S

* After each greedy step, the solution is only partial, S is still subject to extension.

① step-by-step Const. of solution (S)

all finalized vertices

*

③ greedy step: choose min D(v)

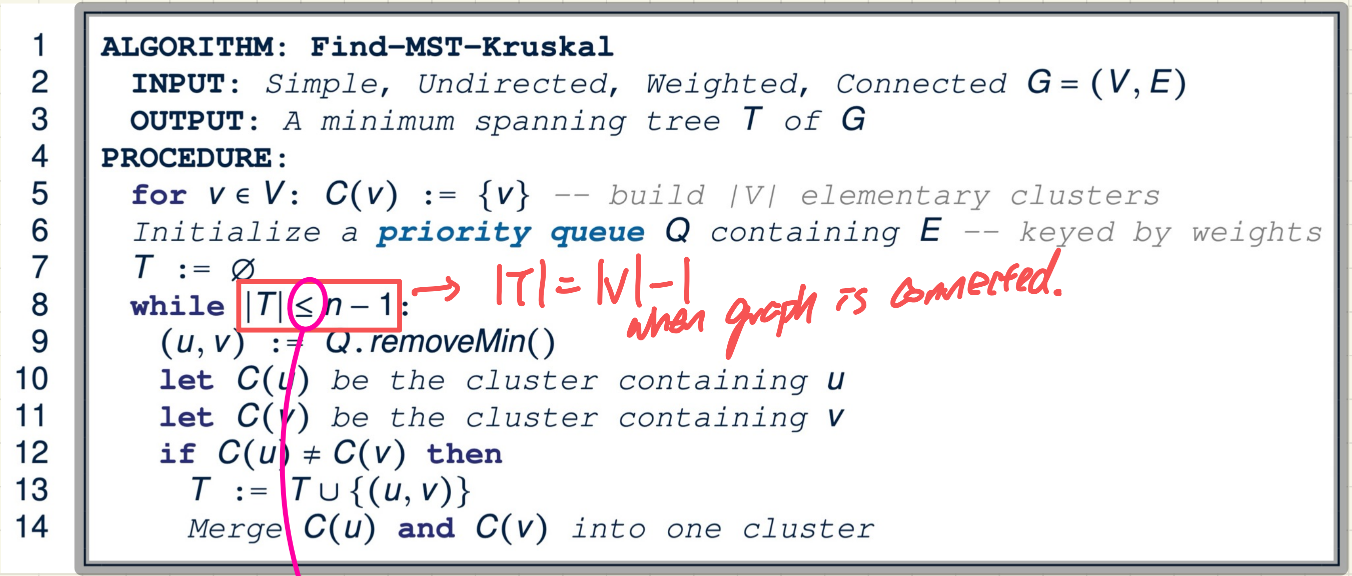
④ once vertex removed never put back

Cost function: D(v)

② multiple feasible choices exist to extend S

vertices that remain in Q.

MST Problem: Kruskal's Algorithm

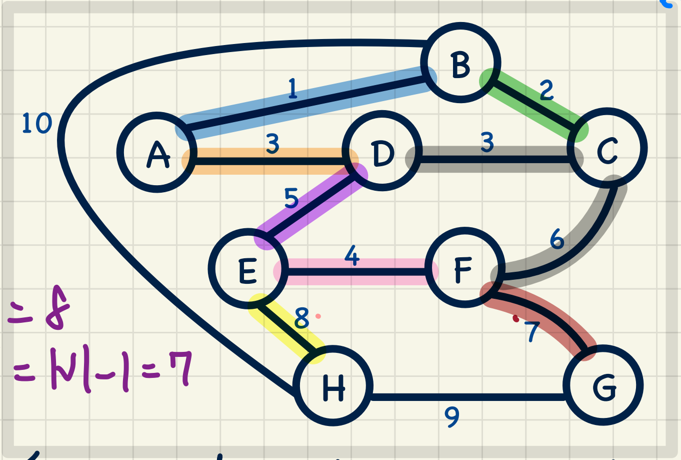


```
ALGORITHM: Find-MST-Kruskal
  INPUT: Simple, Undirected, Weighted, Connected $G = (V, E)$
  OUTPUT: A minimum spanning tree $T$ of $G$
PROCEDURE:
  for $v \in V$: $C(v)$ := $\{v\}$ -- build |V| elementary clusters
  Initialize a priority queue $Q$ containing $E$ -- keyed by weights
  $T$ := $\emptyset$
  while $|T| \leq n - 1$:
    $(u, v)$ := $Q$.removeMin()
    let $C(u)$ be the cluster containing $u$
    let $C(v)$ be the cluster containing $v$
    if $C(u) \neq C(v)$ then
      $T$ := $T \cup \{(u, v)\}$
      Merge $C(u)$ and $C(v)$ into one cluster
```

→ $|T| = |V| - 1$ when graph is connected.

should be:
$|T| < n-1$

MST Problem: Example



$|V| = 8$

$|T| = |V| - 1 = 7$

put all edges in Q.

sets of vertices

It.	min edge	clusters	T

Init.		{A} {B} {C} {D}
		{E} {F} {G} {H}
			{ }

1	w(A,B)=1
	e1
		{A, B} {C} {D}
		{E} {F} {G} {H}
			{e1}

2	w(B,C)=2
	e2
		{A, B, C} {D}
		{E} {F} {G} {H}
			{e1, e2}

3	w(A,D)=3
	e3
		{A, B, C, D}
		{E} {F} {G} {H}
			{e1, e2, e3}

4	w(C,D)=3	n.C.	n.C

5	w(E,F)=4
	e4
		{A, B, C, D}
		{E, F} {G} {H}
			{e1, e2, e3, e4}

6	w(D,E)=5
	e5
		{A, B, C, D, E, F}
		{G} {H}
			{e1, e2, e3, e4, e5}

7	w(C,F)=6	n.C.	n.C
8	w(F,G)=7 e6	{A, B, C, D, E, F, G} {H}	{e1, e2, e3, e4, e5, e6}
9	w(E,H)=8 e7	V	{e1, e2, e3, e4, e5, e6, e7}

$|T| = |V| - 1$ → terminated

Lecture 22 - Dec 10

Graphs

Partition, Cluster, Cut
Kruskal’s Algorithm: Cut Property
Kruskal’s Algorithm: Time Complexity

MST Problem: Partition, Cluster, Cut*

a set, where each member is a set of vertices

a set of vertices is a piece/member of some partition.

of members ~~& CCs~~

Iteration	Min Edge	Processing	Resulting Partition	T: MST Under Construnction
Init.	—	—	$\{A\},\{B\},\{C\},\{D\},\{E\},\{F\},\{G\},\{H\}$	$\varnothing$
1	$w(A,B)=1$	$\because C(A) \neq C(B) \therefore$ Tree Edge	$\{A,B\},\{C\},\{D\},\{E\},\{F\},\{G\},\{H\}$	$\{\ (A,B)\ \}$
2	$w(B,C)=2$	$\because C(B) \neq C(C) \therefore$ Tree Edge	$\{A,B,C\},\{D\},\{E\},\{F\},\{G\},\{H\}$	$\{\ (A,B),(B,C)\ \}$
3	$w(A,D)=3$	$\because C(A) \neq C(D) \therefore$ Tree Edge	$\{A,B,C,D\},\{E\},\{F\},\{G\},\{H\}$	$\{\ (A,B),(B,C),(A,D)\ \}$
4	$w(C,D)=3$	$\because C(C) = C(D) \therefore$ Internal Edge	No Change	
5	$w(E,F)=4$	$\because C(E) \neq C(F) \therefore$ Tree Edge	$\{A,B,C,D\},\{E,F\},\{G\},\{H\}$	$\{\ (A,B),(B,C),(A,D),(E,F)\ \}$
6	$w(D,E)=5$	$\because C(D) \neq C(E) \therefore$ Tree Edge	$\{A,B,C,D,E,F\},\{G\},\{H\}$	$\{\ (A,B),(B,C),(A,D),(E,F),(D,E)\ \}$
7	$w(C,F)=6$	$\because C(C) = C(F) \therefore$ Internal Edge	No Change	
8	$w(F,G)=7$	$\because C(F) \neq C(G) \therefore$ Tree Edge	$\{A,B,C,D,E,F,G\},\{H\}$	$\{\ (A,B),(B,C),(A,D),(E,F),(D,E),(F,G)\ \}$
9	$w(E,H)=8$	$\because C(E) \neq C(H) \therefore$ Tree Edge	$\{A,B,C,D,E,F,G,H\}$	$\{\ (A,B),(B,C),(A,D),(E,F),(D,E),(F,G),(E,H)\ \}$

initial partition: each vertex in its own cluster

C(A)

$\bigcup \{\{A\},\{B\},\{C\},\{D\},\{E\},\{F\},\{G\},\{H\}\} = V$

C(A)

C(A)

C(B)

not a cut

$V \setminus C(B)$

C(B) = C(D) → same C.C.

C(F) ≠ C(G) → diff. C.C.s

partition & cut.

$\bigcup \{\{A,B,C,D,E,F,G,H\}\} = \{\ldots\}$

1 member

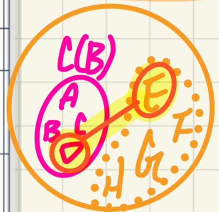
1 CC

final partition: all vertices in a single cluster

* a special case of partition: two member sets of vertices

1. 2 clusters

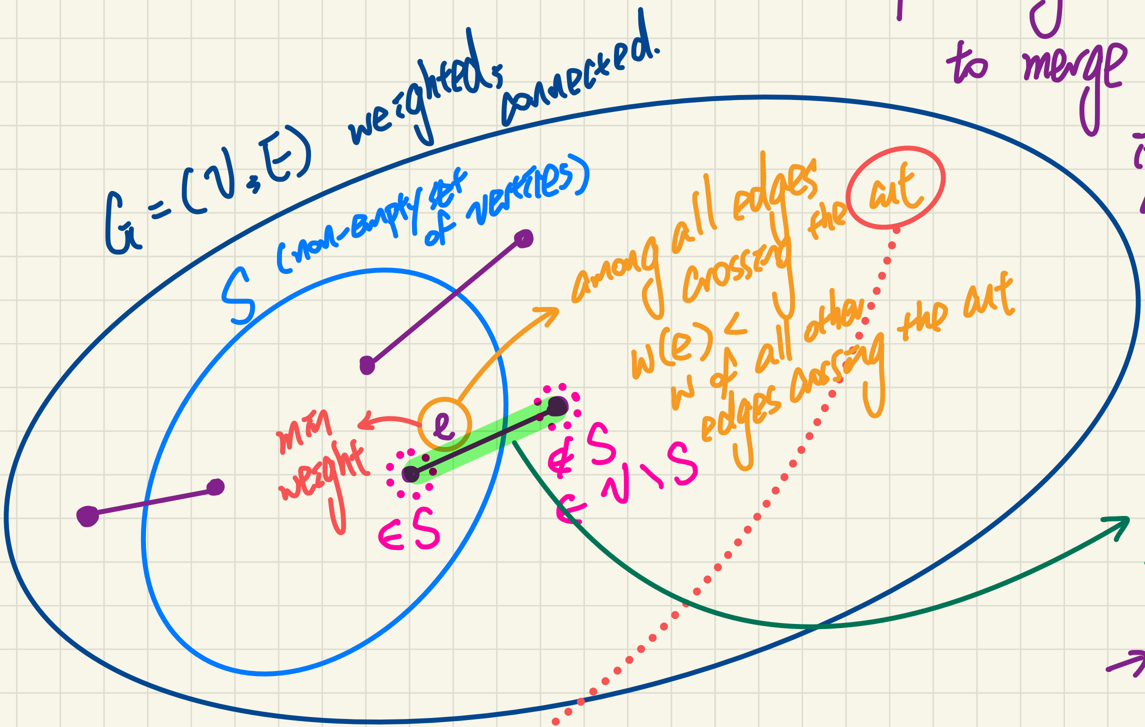
2. 1 cluster vs. rest of vertices.



$V \setminus C(B)$

Edge (D,E) crosses the cut.

MST Problem: Cut Property


G = (V, E) weighted, connected.
S (non-empty set of vertices)
among all edges crossing the cut
w(e) < w of all other edges crossing the cut
min weight
e
∈ S
∉ S
∈ V \ S

cut: {S, V \ S}

member sets of partition

Kruskal's algo.

keep choosing the min edge to merge clusters, it's safe to assume that eventually a MST will be obtained.

there's an MST T s.t. e ∈ T

→ e is a safe edge

MST Problem: Cut Property in Kruskal's Algorithm

```
ALGORITHM: Find-MST-Kruskal
  INPUT: Simple, Undirected, Weighted, Connected $G = (V, E)$
  OUTPUT: A minimum spanning tree $T$ of $G$
PROCEDURE:
  for $v \in V$: $C(v)$ := $\{v\}$ -- build |V| elementary clusters
  Initialize a priority queue $Q$ containing $E$ -- keyed by weights
  $T$ := $\varnothing$
  while $|T| \neq n-1$:
    $(u, v)$ := $Q.removeMin()$
    let $C(u)$ be the cluster containing $u$
    let $C(v)$ be the cluster containing $v$
    if $C(u) \neq C(v)$ then
      $T$ := $T \cup \{(u, v)\}$
      Merge $C(u)$ and $C(v)$ into one cluster
```

(u, v): min-weight edge crossing the cut

non-decreasing order

In the beginning of It. 6

$C(D) = \{A, B, C, D\}$

↳ $C(D) \cap V \setminus C(D) = \varnothing$

$C(E) \subseteq V \setminus C(D)$

all edges with an end point $\in C(D)$

$(A, B), (B, C), (A, D), (C, D)$

Iteration	Min Edge	Processing	Resulting Partition	T: MST Under Construnction
Init.	—	—	$\{\{A\}, \{B\}, \{C\}, \{D\}, \{E\}, \{F\}, \{G\}, \{H\}\}$	$\varnothing$
1	$w(A, B) = 1$	$\because C(A) \neq C(B) \therefore$ Tree Edge	$\{\{A, B\}, \{C\}, \{D\}, \{E\}, \{F\}, \{G\}, \{H\}\}$	$\{(A, B)\}$
2	$w(B, C) = 2$	$\because C(B) \neq C(C) \therefore$ Tree Edge	$\{\{A, B, C\}, \{D\}, \{E\}, \{F\}, \{G\}, \{H\}\}$	$\{(A, B), (B, C)\}$
3	$w(A, D) = 3$	$\because C(A) \neq C(D) \therefore$ Tree Edge	$\{\{A, B, C, D\}, \{E\}, \{F\}, \{G\}, \{H\}\}$	$\{(A, B), (B, C), (A, D)\}$
4	$w(C, D) = 3$	$\because C(C) = C(D) \therefore$ Internal Edge	No Change	
5	$w(E, F) = 4$	$\because C(E) \neq C(F) \therefore$ Tree Edge	$\{\{A, B, C, D\}, \{E, F\}, \{G\}, \{H\}\}$	$\{(A, B), (B, C), (A, D), (E, F)\}$
6	$w(D, E) = 5$	$\because C(D) \neq C(E) \therefore$ Tree Edge	$\{\{A, B, C, D, E, F\}, \{G\}, \{H\}\}$	$\{(A, B), (B, C), (A, D), (E, F), (D, E)\}$
7	$w(C, F) = 6$	$\because C(C) = C(F) \therefore$ Internal Edge	No Change	
8	$w(F, G) = 7$	$\because C(F) \neq C(G) \therefore$ Tree Edge	$\{\{A, B, C, D, E, F, G\}, \{H\}\}$	$\{(A, B), (B, C), (A, D), (E, F), (D, E), (F, G)\}$
9	$w(E, H) = 8$	$\because C(E) \neq C(H) \therefore$ Tree Edge	$\{\{A, B, C, D, E, F, G, H\}\}$	$\{(A, B), (B, C), (A, D), (E, F), (D, E), (F, G), (E, H)\}$

$\neq$ means cross → merge

merge

no cut: $\because V \setminus V = \varnothing$

* Merging $C(A)$ & $C(B)$ $\because$ (A, B) crosses cut $\{\underbrace{\{A\}}_{C(A)}, \underbrace{V \setminus \{A\}}_{C(B) \subseteq}\}$

** Merging $C(B)$ & $C(C)$ $\because$ (B, C) crosses cut $\{\underbrace{\{A, B\}}_{C(B)}, \underbrace{V \setminus \{A, B\}}_{C(C) \subseteq}\}$

safe to include (D, E) in T $\because$ cut property.

I hope you enjoyed learning with me

All the best to you!