Design by Contract
Modularity
Abstract Data Types (ADTs)

EECS3101 E:
Design and Analysis of Algorithms
Fall 2025

CHEN-WEI WANG

Learning Objectives

Upon completing this lecture, you are expected to understand:

1. Methodology of Design by Contract (DbC)
2. Criterion of *Modularity*, Modular Design
3. *Abstract Data Types* (*ADTs*)

Terminology: Contract, Client, Supplier

- A *supplier* implements/provides a service (e.g., microwave).
- A *client* uses a service provided by some supplier.
  - The client is required to follow certain instructions to obtain the service (e.g., supplier **assumes** that client powers on, closes door, and heats something that is not explosive).
  - If instructions are followed, the client would **expect** that the service does what is guaranteed (e.g., a lunch box is heated).
  - The client does not care how the supplier implements it.
- What then are the *benefits* and *obligations* os the two parties?

	benefits	*obligations*
CLIENT	obtain a service	follow instructions
SUPPLIER	assume instructions followed	provide a service

- There is a *contract* between two parties, violated if:
  - The instructions are not followed. [Client's fault]
  - Instructions followed, but service not satisfactory. [Supplier's fault]

Client, Supplier, Contract in OOP (1)

```
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. */
  } }
```

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

Method call *m.heat(obj)* indicates a client-supplier relation.

- **Client**: resident class of the method call [`MicrowaveUser`]
- **Supplier**: type of context object (or call target) `m` [`Microwave`]

Client, Supplier, Contract in OOP (2)

```java
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. */ } }
```

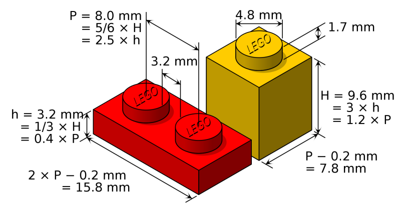
```java
class MicrowaveUser {
  public static void main(...) {
    Microwave m = new Microwave();
    Object obj = ???;
    m.power(); m.lock();
    m.heat(obj);
}}
```

- The *contract* is *honoured* if:
  - Right **before** the method call :
    - State of `m` is as assumed: `m.on==true` and `m.locked==ture`
    - The input argument `obj` is valid (i.e., not explosive).
  - Right **after** the method call : `obj` is properly heated.
- If any of these fails, there is a *contract violation*.
  - `m.on` or `m.locked` is `false` ⇒ `MicrowaveUser`'s fault.
  - `obj` is an explosive ⇒ `MicrowaveUser`'s fault.
    A fault from the client is identified ⇒ Method call will not start.
  - Method executed but `obj` not properly heated ⇒ `Microwave`'s fault

What is a Good Design?

- A "good" design should *explicitly* and *unambiguously* describe the *contract* between **clients** (e.g., users of Java classes) and **suppliers** (e.g., developers of Java classes).
  We call such a contractual relation a *specification*.
- When you conduct *software design*, you should be guided by the "appropriate" contracts between users and developers.
  - Instructions to **clients** should *not be unreasonable*.
    e.g., asking them to assemble internal parts of a microwave
  - Working conditions for **suppliers** should *not be unconditional*.
    e.g., expecting them to produce a microwave which can safely heat an explosive with its door open!
  - You as a designer should strike proper balance between **obligations** and **benefits** of clients and suppliers.
    e.g., What is the obligation of a binary-search user (also benefit of a binary-search implementer)? [The input array is sorted.]
  - Upon contract violation, there should be the fault of **only one side**.
  - This design process is called *Design by Contract (DbC)*.

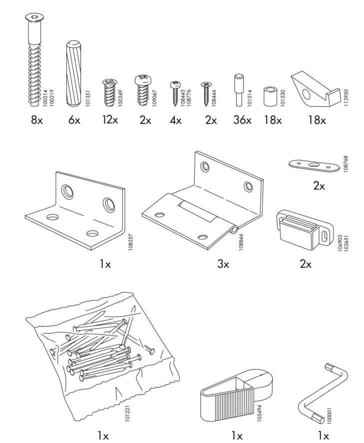
Modularity (1): Childhood Activity

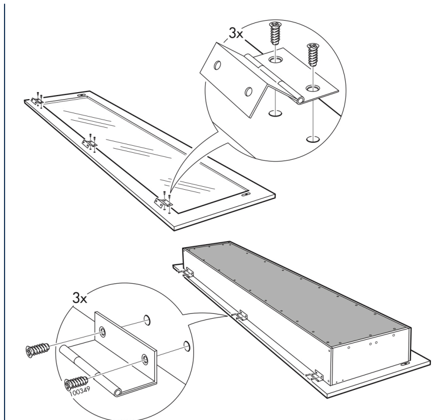


(INTERFACE) SPECIFICATION | (ASSEMBLY) ARCHITECTURE

Sources: https://commons.wikimedia.org and https://www.wish.com

Modularity (2): Daily Construction



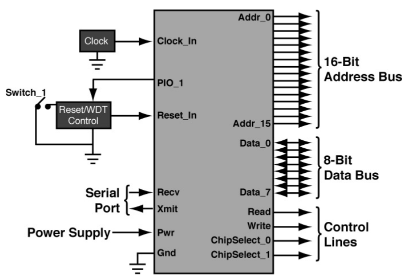


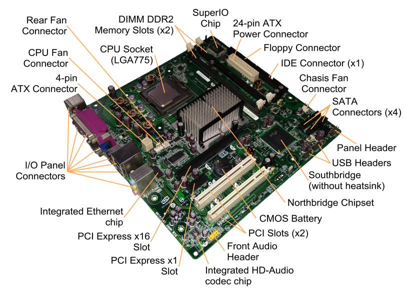
(INTERFACE) SPECIFICATION | (ASSEMBLY) ARCHITECTURE

Source: https://usermanual.wiki/

Modularity (3): Computer Architecture

Motherboards are built from functioning units (e.g., *CPUs*).





(INTERFACE) SPECIFICATION || (ASSEMBLY) ARCHITECTURE

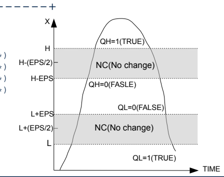
Sources: www.embeddedlinux.org.cn and https://en.wikipedia.org

Modularity (4): System Development

Safety-critical systems (e.g., *nuclear shutdown systems*) are built from *function blocks*.

```
(* 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
```



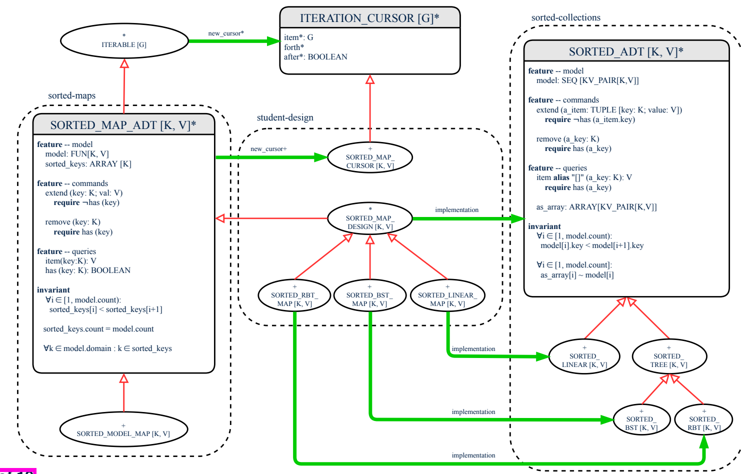

```
            (* Function block body in FBD language *)
                          HIGH_ALARM
                       +-------------+
                       | HYSTERESIS  |
X----------------------+--|XIN1        Q|--+----------QH
           +----+ w2|  |             |  |
H----------------| - |------|XIN2        |  |
        +---|    |   |  |             |  |
        |   +----+   |  |             |  |
        +--------------|EPS         |  |   +------+
   +---+w1|           |  +-------------+  +--| >=1  |
EPS --| / |--|           |                     |      |--Q
2.0 --|   |  |           |   LOW_ALARM      +--|      |
   +---+  |           |  +-------------+  |   +------+
        |   +----+ w3|  | HYSTERESIS  |  |
L --------------| + |------|XIN1        Q|--+----------QL
        |   |    |   |  |             |
        +---|    |   +--|XIN2        |
        |   +----+      |             |
        +--------------|EPS         |
                       +-------------+
```

(INTERFACE) SPECIFICATION || (ASSEMBLY) ARCHITECTURE

Sources: https://plcopen.org/iec-61131-3

Modularity (5): Software Design

Software systems are composed of well-specified *classes*.

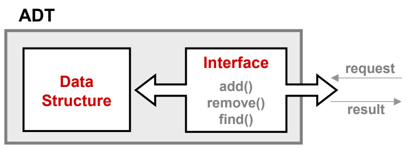


Design Principle: Modularity

- **Modularity** refers to a sound quality of your design:
  1. **Divide** a given complex ***problem*** into inter-related ***sub-problems*** via a logical/justifiable functional decomposition.
     e.g., In designing a game, solve sub-problems of: 1) rules of the game; 2) actor characterizations; and 3) presentation.
  2. **Specify** each ***sub-solution*** as a *module* with a clear **interface**: inputs, outputs, and **input-output relations**.
     - The UNIX principle: Each command does one thing and does it well.
     - In objected-oriented design (OOD), each class serves as a module.
  3. **Conquer** original ***problem*** by assembling ***sub-solutions***.
     - In OOD, classes are assembled via client-supplier relations (aggregations or compositions) or inheritance relations.
- A *modular design* satisfies the criterion of modularity and is:
  - ***Maintainable***: fix issues by changing the relevant modules only.
  - ***Extensible***: introduce new functionalities by adding new modules.
  - ***Reusable***: a module may be used in different compositions
- Opposite of modularity: A *superman module* doing everything.

Abstract Data Types (ADTs)

- Given a problem, <u>decompose</u> its solution into *modules*.
- Each *module* implements an *abstract data type (ADT)*:
  - filters out *irrelevant* details
  - contains a list of declared data and *well-specified* operations



- <u>Supplier's Obligations</u>:
  - Implement all operations
  - Choose the "right" data structure (DS)
- <u>Client's Benefits</u>:
  - <u>Correct</u> output
  - Efficient performance
- The internal details of an *implemented ADT* should be **hidden**.

Building ADTs for Reusability

- ADTs are *reusable software components*
  e.g., Stacks, Queues, Lists, Dictionaries, Trees, Graphs
- An ADT, once thoroughly tested, can be reused by:
  - Suppliers of other ADTs
  - Clients of Applications
- As a supplier, you are obliged to:
  - *Implement* given ADTs using other ADTs (e.g., arrays, linked lists, hash tables, etc.)
  - *Design* algorithms that make use of standard ADTs
- For each ADT that you build, you ought to be clear about:
  - The list of supported operations (i.e., *interface*)
    - The interface of an ADT should be *more than* method signatures and natural language descriptions:
    - How are clients supposed to use these methods? [*preconditions*]
    - What are the services provided by suppliers? [*postconditions*]
  - Time (and sometimes space) *complexity* of each operation

Why Java Interfaces ≈ ADTs (1)

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.

It is useful to have:

- A *generic collection class* where the ***homogeneous type*** of elements are parameterized as `E`.
- A reasonably ***intuitive overview*** of the ADT.

Java 8 List API

Why Java Interfaces ≈ ADTs (2)

Methods described in a ***natural language*** can be ***ambiguous***:

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

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())`

Beyond this lecture...

1. **Q.** Can you think of more real-life examples of leveraging the power of ***modularity***?
2. Visit the Java API page:

   `https://docs.oracle.com/javase/8/docs/api`

   Visit collection classes which you used in EECS2030 (e.g., `ArrayList`, `HashMap`) and EECS2011.

   Q. Can you identify/justify some example methods which illustrate that these Java collection classes are **not** true ***ADTs*** (i.e., ones with well-specified interfaces)?
3. Constrast with the corresponding library classes and features in EiffelStudio (e.g., `ARRAYED_LIST`, `HASH_TABLE`).

   Q. Are these Eiffel features ***better specified*** w.r.t. obligations/benefits of clients/suppliers?

Index (1)

Index (2)

Asymptotic Analysis of Algorithms

EECS3101 E:
Design and Analysis of Algorithms
Fall 2025

CHEN-WEI WANG

What You're Assumed to Know

- You will be required to ***implement*** Java classes and methods, and to ***test*** their correctness using JUnit.

  Review them if necessary:

  https://www.eecs.yorku.ca/~jackie/teaching/lectures/index.html#EECS2030_F21

  - Implementing classes and methods in Java [Weeks 1 – 2]
  - Testing methods in Java [Week 4]
- Also, make sure you know how to trace programs using a ***debugger***:

  https://www.eecs.yorku.ca/~jackie/teaching/tutorials/index.html#java_from_scratch_w21

  - Debugging actions (Step Over/Into/Return) [Parts C – E, Week 2]

Learning Outcomes

This module is designed to help you learn about:

- Notions of ***Algorithms*** and ***Data Structures***
- Measurement of the "goodness" of an algorithm
- Measurement of the ***efficiency*** of an algorithm
- Experimental measurement vs. ***Theoretical*** measurement
- Understand the purpose of *asymptotic* analysis.
- Understand what it means to say two algorithms are:
  - equally efficient, **asymptotically**
  - one is more efficient than the other, **asymptotically**
- Given an algorithm, determine its ***asymptotic upper bound***.

Algorithm and Data Structure

- A *data structure* is:
  - A systematic way to store and organize data in order to facilitate ***access*** and ***modifications***
  - Never suitable for all purposes: it is important to know its ***strengths*** and ***limitations***
- A **well-specified** *computational problem* precisely describes the desired ***input/output relationship***.
  - **Input:** A sequence of n numbers $\langle a_1, a_2, \dots, a_n \rangle$
  - **Output:** A permutation (reordering) $\langle a'_1, a'_2, \dots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$
  - An ***instance*** of the problem: $\langle 3, 1, 2, 5, 4 \rangle$
- An *algorithm* is:
  - A solution to a **well-specified** computational problem
  - A **sequence of computational steps** that takes value(s) as ***input*** and produces value(s) as ***output***
- An ***algorithm*** manipulates some chosen ***data structure(s)***.

Measuring "Goodness" of an Algorithm

1. ***Correctness*** :
   - Does the ***algorithm*** produce the **expected** output?
   - Use ***unit & regression testing*** (e.g., JUnit) to ensure this.
2. Efficiency:
   - ***Time Complexity***: processor time required to complete
   - ***Space Complexity***: memory space required to store data

   Correctness is always the priority.

   How about efficiency? Is time or space more of a concern?

Measuring Efficiency of an Algorithm

- ***Time*** is more of a concern than is ***storage***.
- Solutions (run on computers) should be ***as fast as possible***.
- Particularly, we are interested in how ***running time*** depends on two ***input factors***:
  1. ***size***
     e.g., sorting an array of 10 elements vs. 1m elements
  2. ***structure***
     e.g., sorting an already-sorted array vs. a hardly-sorted array

Q. How does one determine the ***running time*** of an algorithm?

1. Measure time via ***experiments***
2. Characterize time as a ***mathematical function*** of the input size

Experimental Analysis: Challenges

1. An algorithm must be ***fully implemented*** (e.g., in Java) in order study its runtime behaviour **experimentally**.
   - What if our purpose is to ***choose among alternative*** data structures or algorithms to implement?
   - Can there be a *higher-level analysis* to determine that one algorithm or data structure is more "**superior**" than others?
2. Comparison of multiple algorithms is only ***meaningful*** when experiments are conducted under the **same** working environment of:
   - ***Hardware***: CPU, running processes
   - ***Software***: OS, JVM version, Version of Compiler
3. Experiments can be done only on ***a limited set of test inputs***.
   - What if ***worst-case*** inputs were **not** included in the experiments?
   - What if "***important***" inputs were **not** included in the experiments?

Measure Running Time via Experiments

- Once the algorithm is implemented (e.g., in Java):
  - Execute program on ***test inputs*** of various ***sizes*** & ***structures***.
  - For each test, record the ***elapsed time*** of the execution.

```
long startTime = System.currentTimeMillis();
/* run the algorithm */
long endTime = System.currenctTimeMillis();
long elapsed = endTime - startTime;
```

  - ***Visualize*** the result of each test.
- To make ***sound statistical claims*** about the algorithm's ***running time***, the set of ***test inputs*** should be "***complete***".
  e.g., To experiment with the ***RT*** of a sorting algorithm:
  - **Unreasonable**: **only** consider small-sized and/or almost-sorted arrays
  - **Reasonable**: **also** consider large-sized, randomly-organized arrays

Moving Beyond Experimental Analysis

- A better approach to analyzing the ***efficiency*** (e.g., ***running time***) of algorithms should be one that:
  - Can be applied using a *high-level description* of the algorithm (**without** fully implementing it).
    [e.g., Pseudo Code, Java Code (with "tolerances")]
  - Allows us to calculate the *relative efficiency* (rather than absolute elapsed time) of algorithms in a way that is ***independent of*** the hardware and software environment.
  - Considers *all* possible inputs (esp. the ***worst-case scenario***).
- We will learn a better approach that contains 3 ingredients:
  1. Counting ***primitive operations***
  2. Approximating running time as ***a function of input size***
  3. Focusing on the ***worst-case*** input (requiring most running time)

Counting Primitive Operations

- A ***primitive operation*** (***POs***) corresponds to a low-level instruction with a ***constant** execution time*.
  - (Variable) Assignment [e.g., `x = 5;`]
  - Indexing into an array [e.g., `a[i]`]
  - Arithmetic, relational, logical op. [e.g., `a + b`, `z > w`, `b1 && b2`]
  - Accessing an attribute of an object [e.g., `acc.balance`]
  - Returning from a method [e.g., `return result;`]

  Q: Is a ***method call*** a primitive operation?

  A: **Not** in general. It may be a call to:
  - a "***cheap***" method (e.g., printing `Hello World`), or
  - an "***expensive***" method (e.g., sorting an array of integers)
- ***RT*** of an ***algorithm*** is approximated as the number of ***POs*** involved (**despite** the execution environment).

From Absolute RT to Relative RT

- Each ***primitive operation*** (***PO***) takes approximately the same, constant amount of time to execute. [say **t**]

  The absolute value of **t** depends on the ***execution environment***.

Q. How do you relate the ***number of POs*** required by an algorithm and its ***actual RT*** on a specific working environment?

A. ***Number of POs*** should be *proportional* to the actual ***RT***.

$$\boxed{RT = \mathbf{t} \cdot \textit{number of POs}}$$

- e.g., `findMax (int[] a, int n)` has ***7n - 2*** POs

  RT = (***7n - 2***) · **t**
- e.g., Say two algorithms with ***RT*** (***7n - 2***) · **t** and ***RT*** (***10n + 3***) · **t**:
  It suffices to compare their *relative* running time:

  7n - 2 vs. ***10n + 3***.

∴ To determine the ***time efficiency*** of an algorithm, we only focus on their ***number of POs***.

Example: Approx. # of Primitive Operations

- Given # of primitive operations counted **precisely** as $7n - 2$, we view it as

$$7 \cdot n^1 - 2 \cdot n^0$$

- We say
  - n is the ***highest power***
  - 7 and 2 are the ***multiplicative constants***
  - 2 is the ***lower term***
- When **approximating** a ***function*** [e.g., RT ≈ f(n)]
  (considering that ***input size*** may be very large):
  - **Only** the ***highest power*** matters.
  - ***multiplicative constants*** and ***lower terms*** can be dropped.

  ⇒ $7n - 2$ is approximately n

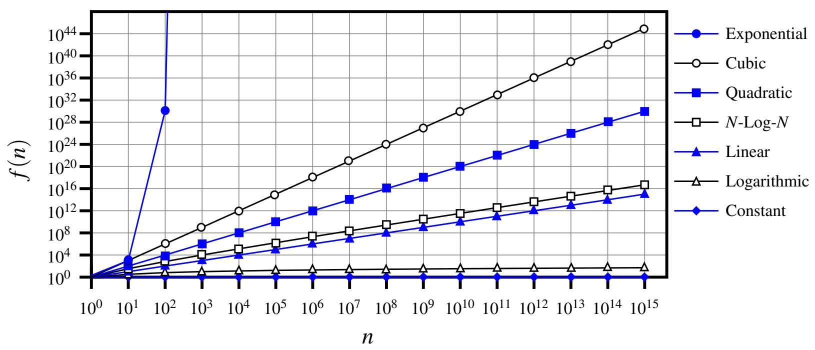
  Exercise: Consider $7n + 2n \cdot log\ n + 3n^2$:
  - ***highest power***? [n^2]
  - ***multiplicative constants***? [7, 2, 3]
  - ***lower terms***? [$7n$, $2n \cdot log\ n$]

Approximating Running Time as a Function of Input Size

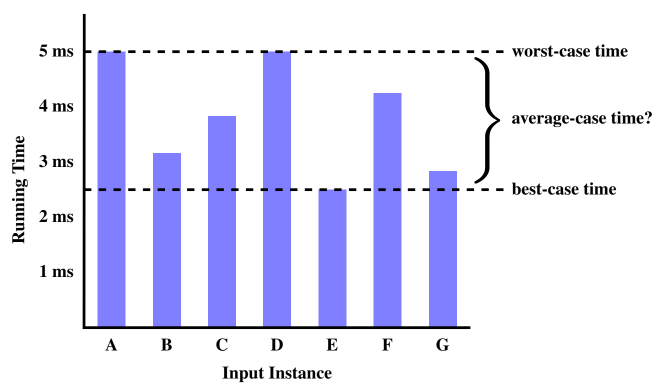
Given the ***high-level description*** of an algorithm, we associate it with a function f, such that $f(n)$ returns the ***number of primitive operations*** that are performed on an ***input of size*** n.

- $f(n) = 5$ [constant]
- $f(n) = log_2 n$ [logarithmic]
- $f(n) = 4 \cdot n$ [linear]
- $f(n) = n^2$ [quadratic]
- $f(n) = n^3$ [cubic]
- $f(n) = 2^n$ [exponential]

Rates of Growth: Comparison



Focusing on the Worst-Case Input



- ***Average-case*** analysis calculates the *expected running time* based on the probability distribution of input values.
- ***worst-case*** analysis or ***best-case*** analysis?

What is Asymptotic Analysis?

Asymptotic analysis

- Is a method of describing ***behaviour towards the limit***:
  - How the ***running time*** of the algorithm under analysis changes as the ***input size*** changes **without** bound
  - e.g., Contrast: $RT_1(n) = n$ vs. $RT_2(n) = n^2$
- Allows us to compare the ***relative performance*** of alternative algorithms:
  - For large enough inputs, the multiplicative constants and lower-order terms of an exact running time can be disregarded.
  - e.g., $RT_1(n) = 3n^2 + 7n + 18$ and $RT_1(n) = 100n^2 + 3n - 100$ are considered **equally efficient**, ***asymptotically***.
  - e.g., $RT_1(n) = n^3 + 7n + 18$ is considered **less efficient** than $RT_1(n) = 100n^2 + 100n + 2000$, ***asymptotically***.

Three Notions of Asymptotic Bounds

We may consider three kinds of ***asymptotic bounds*** for the ***running time*** of an algorithm:

- Asymptotic *upper* bound [O]
- Asymptotic lower bound [Ω]
- Asymptotic tight bound [Θ]

Asymptotic Upper Bound: Definition

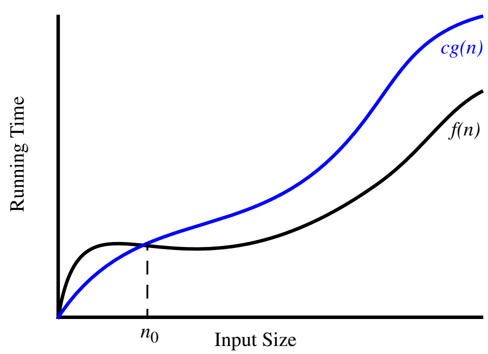
- Let ***f(n)*** and ***g(n)*** be functions mapping pos. integers (input size) to pos. real numbers (running time).
  - ***f(n)*** characterizes the running time of some algorithm.
  - ***O(g(n))***:
    - denotes a collection of functions
    - consists of all functions that can be ***upper bounded by g(n)***, starting at some point, using some constant factor
- ***f(n)*** $\in$ ***O(g(n))*** if there are:
  - A real ***constant*** $c > 0$
  - An integer ***constant*** $n_0 \geq 1$

  such that:

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

- For each member function ***f(n)*** in ***O(g(n))***, we say that:
  - $f(n) \in O(g(n))$ [f(n) is a member of “big-O of g(n)”]
  - $f(n)$ **is** $O(g(n))$ [f(n) is “big-O of g(n)”]
  - $f(n)$ **is order of** $g(n)$

Asymptotic Upper Bound: Visualization



From n_0, ***f(n)*** is ***upper bounded by*** $c \cdot$ ***g(n)***, so ***f(n)*** is $O(g(n))$.

Asymptotic Upper Bound: Proposition

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, then $f(n)$ *is* $O(n^d)$.

- We prove by choosing

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

- We know that for $n \geq 1$: $n^0 \leq n^1 \leq n^2 \leq \cdots \leq n^d$
- Upper-bound effect: $n_0 = 1$? $[f(1) \leq (|a_0| + |a_1| + \cdots + |a_d|) \cdot 1^d]$

$$a_0 \cdot 1^0 + a_1 \cdot 1^1 + \cdots + a_d \cdot 1^d \leq |a_0| \cdot 1^d + |a_1| \cdot 1^d + \cdots + |a_d| \cdot 1^d$$

- Upper-bound effect holds? $[f(n) \leq (|a_0| + |a_1| + \cdots + |a_d|) \cdot n^d]$

$$a_0 \cdot n^0 + a_1 \cdot n^1 + \cdots + a_d \cdot n^d \leq |a_0| \cdot n^d + |a_1| \cdot n^d + \cdots + |a_d| \cdot n^d$$

Asymptotic Upper Bound: Example

Prove: The function $f(n) = 5n^4 - 3n^3 + 2n^2 - 4n + 1$ is $O(n^4)$.

Strategy: Choose a real constant $c > 0$ and an integer constant $n_0 \geq 1$, such that for every integer $n \geq n_0$:

$$5n^4 + 3n^3 + 2n^2 + 4n + 1 \leq c \cdot n^4$$

Using the proven **proposition**, choose:

- $c = |5| + |-3| + |2| + |-4| + |1| = 15$
- $n_0 = 1$

Asymptotic Upper Bound: Families

- If a function $f(n)$ is ***upper bounded by*** another function $g(n)$ of degree d, $d \geq 0$, then $f(n)$ is also ***upper bounded by*** all other functions of a ***strictly higher degree*** (i.e., $d+1$, $d+2$, *etc.*).
  - e.g., Family of $O(n)$ contains all $f(n)$ that can be ***upper bounded by*** $g(n) = n^1$:

    $n, 2n, 3n, \ldots$ [functions with degree 1]
    $n^0, 2n^0, 3n^0, \ldots$ [functions with degree 0]
  - e.g., Family of $O(n^2)$ contains all $f(n)$ that can be ***upper bounded by*** $g(n) = n^2$:

    $n^2, 2n^2, 3n^2, \ldots$ [functions with degree 2]
    $n, 2n, 3n, \ldots$ [functions with degree 1]
    $n^0, 2n^0, 3n^0, \ldots$ [functions with degree 0]
- Consequently:

$$O(n^0) \subset O(n^1) \subset O(n^2) \subset \ldots$$

Asymptotic Upper Bound: More Examples

- $5n^2 + 3n \cdot logn + 2n + 5$ is $O(n^2)$ $[c = 15, n_0 = 1]$
- $20n^3 + 10n \cdot logn + 5$ is $O(n^3)$ $[c = 35, n_0 = 1]$
- $3 \cdot logn + 2$ is $O(logn)$ $[c = 5, n_0 = 2]$
  - Why can't n_0 be 1?
  - Choosing $n_0 = 1$ means $\Rightarrow f(\boxed{1})$ ***is*** upper-bounded by $c \cdot log\boxed{1}$:
    - We have $f(\boxed{1}) = 3 \cdot log1 + 2$, which is 2.
    - We have $c \cdot log\boxed{1}$, which is 0.

    $\Rightarrow f(\boxed{1})$ ***is not*** upper-bounded by $c \cdot log\boxed{1}$ [Contradiction!]
- 2^{n+2} is $O(2^n)$ $[c = 4, n_0 = 1]$
- $2n + 100 \cdot logn$ is $O(n)$ $[c = 102, n_0 = 1]$

Using Asymptotic Upper Bound Accurately

- Use the big-O notation to characterize a function (of an algorithm's running time) ***as closely as possible***.

  For example, say $f(n) = 4n^3 + 3n^2 + 5$:
  - Recall: $O(n^3) \subset O(n^4) \subset O(n^5) \subset \ldots$
  - It is the *most accurate* to say that $f(n)$ is $O(n^3)$.
  - It is ***true***, but not very useful, to say that $f(n)$ is $O(n^4)$ and that $f(n)$ is $O(n^5)$.
  - It is ***false*** to say that $f(n)$ is $O(n^2)$, $O(n)$, or $O(1)$.
- Do **not** include ***constant factors*** and ***lower-order terms*** in the big-O notation.

  For example, say $f(n) = 2n^2$ is $O(n^2)$, do not say $f(n)$ is $O(4n^2 + 6n + 9)$.

Classes of Functions

upper bound	class	cost
$O(1)$	constant	*cheapest*
$O(log(n))$	logarithmic	
$O(n)$	linear	
$O(n \cdot log(n))$	"n-log-n"	
$O(n^2)$	quadratic	
$O(n^3)$	cubic	
$O(n^k), k \geq 1$	polynomial	
$O(a^n), a > 1$	exponential	*most expensive*

Upper Bound of Algorithm: Example (1)

```
1 boolean containsDuplicate (int[] a, int n) {
2   for (int i = 0; i < n; ) {
3     for (int j = 0; j < n; ) {
4       if (i != j && a[i] == a[j]) {
5         return true; }
6       j ++; }
7     i ++; }
8   return false; }
```

- Worst case is when we reach Line 8.
- # of primitive operations $\approx c_1 + n \cdot n \cdot c_2$, where c_1 and c_2 are some constants.
- Therefore, the running time is $O(n^2)$.
- That is, this is a *quadratic* algorithm.

Upper Bound of Algorithm: Example (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; }
```

- # of primitive operations $\approx (c_1 \cdot n + c_2) + (c_3 \cdot n \cdot n + c_4)$, where c_1, c_2, c_3, and c_4 are some constants.
- Therefore, the running time is $O(n + n^2) = O(n^2)$.
- That is, this is a *quadratic* algorithm.

Upper Bound of Algorithm: Example (3)

```
int triangularSum (int[] a, int n) {
  int sum = 0;
  for (int i = 0; i < n; i ++) {
    for (int j = i; j < n; j ++) {
      sum += a[j]; } }
  return sum; }
```

- # of primitive operations $\approx n + (n-1) + \cdots + 2 + 1 = \frac{n \cdot (n+1)}{2}$
- Therefore, the running time is $O(\frac{n^2+n}{2}) = O(n^2)$.
- That is, this is a *quadratic* algorithm.

Array Implementations: Stack and Queue

- When implementing ***stack*** and ***queue*** via ***arrays***, we imposed a maximum capacity:

```
public class ArrayStack<E> implements Stack<E> {
  private final int MAX_CAPACITY = 1000;
  private E[] data;
  ...
  public void push(E e) {
    if (size() == MAX_CAPACITY) { /* Precondition Violated */ }
    else { ... }
  }
  ...
}
```

```
public class ArrayQueue<E> implements Queue<E> {
  private final int MAX_CAPACITY = 1000;
  private E[] data;
  ...
  public void enqueue(E e) {
    if (size() == MAX_CAPACITY) { /* Precondition Violated */ }
    else { ...
  }
  ...
}
```

- This made the ***push*** and ***enqueue*** operations both cost ***O(1)***.

Dynamic Array: Constant Increments

Implement ***stack*** using a ***dynamic array*** resizing itself by a constant increment:

```
1  public class ArrayStack<E> implements Stack<E> {
2    private int I;
3    private int C;
4    private int capacity;
5    private E[] data;
6    public ArrayStack() {
7      I = 1000; /* arbitrary initial size */
8      C = 500; /* arbitrary fixed increment */
9      capacity = I;
10     data = (E[]) new Object[capacity];
11     t = -1;
12   }
13   public void push(E e) {
14     if (size() == capacity) {
15       /* resizing by a fixed constant */
16       E[] temp = (E[]) new Object[capacity + C];
17       for(int i = 0; i < capacity; i ++) {
18         temp[i] = data[i];
19       }
20       data = temp;
21       capacity = capacity + C
22     }
23     t++;
24     data[t] = e;
25   }
26 }
```

- This alternative strategy ***resizes*** the array, whenever needed, by a ***constant*** amount.
- **L17** – **L19** make ***push*** cost ***O(n)***, in the ***worst case***.
- However, given that ***resizing*** only happens rarely, how about the *average* running time?
- We will refer **L14 – L22** as the ***resizing*** part and **L23 – L24** as the ***update*** part.

Dynamic Array: Doubling

Implement ***stack*** using a ***dynamic array*** resizing itself by doubling:

```
1  public class ArrayStack<E> implements Stack<E> {
2    private int I;
3    private int capacity;
4    private E[] data;
5    public ArrayStack() {
6      I = 1000; /* arbitrary initial size */
7      capacity = I;
8      data = (E[]) new Object[capacity];
9      t = -1;
10   }
11   public void push(E e) {
12     if (size() == capacity) {
13       /* resizing by doubling */
14       E[] temp = (E[]) new Object[capacity * 2];
15       for(int i = 0; i < capacity; i ++) {
16         temp[i] = data[i];
17       }
18       data = temp;
19       capacity = capacity * 2;
20     }
21     t++;
22     data[t] = e;
23   }
24 }
```

- This alternative strategy ***resizes*** the array, whenever needed, by ***doubling*** its current size.
- **L15** – **L17** make ***push*** cost ***O(n)***, in the ***worst case***.
- However, given that ***resizing*** only happens rarely, how about the *average* running time?
- We will refer **L12 – L20** as the resizing part and **L21 – L22** as the update part.

Avg. RT: Const. Increment vs. Doubling

- Without loss of generality, assume: There are n ***push*** operations, and the **last push** triggers the **last** ***resizing*** routine.

	Constant Increments	Doubling
RT of exec. update part for n pushes	$O(n)$	
RT of executing 1st resizing	I	
RT of executing 2nd resizing	$I + C$	$2 \cdot I$
RT of executing 3rd resizing	$I + 2 \cdot C$	$4 \cdot I$
RT of executing 4th resizing	$I + 3 \cdot C$	$8 \cdot I$
RT of executing k^{th} resizing	$I + (k - 1) \cdot C$	$2^{k-1} \cdot I$
RT of executing last resizing	n	
# of resizing needed (solve k for $RT = n$)	$O(n)$	$O(log_2 n)$
Total RT for n pushes	$O(n^2)$	$O(n)$
Amortized/Average RT over n pushes	$O(n)$	$O(1)$

- Over n push operations, the *amortized* / *average* running time of the ***doubling*** strategy is more efficient.

Index (1)

Index (2)

Index (3)

Self-Balancing Binary Search Trees

EECS3101 E:
Design and Analysis of Algorithms
Fall 2025

CHEN-WEI WANG

Learning Outcomes of this Lecture

This module is designed to help you understand:

- When the ***Worst-Case RT*** of a ***BST Search*** Occurs
- ***Height-Balance*** Property
- Review: Insertion & Deletion on a BST
- Performing ***Rotations*** to Restore Tree ***Balance***

Implementation: Generic BST 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) { ... }
}
```

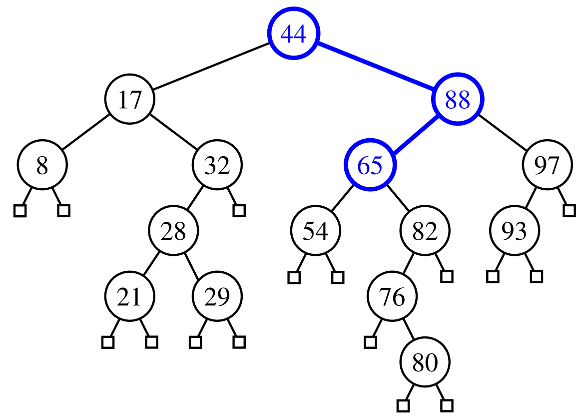

Implementing BST Operation: Searching

Given a *BST* rooted at node ***p***, to locate a particular ***node*** whose ***key*** matches ***k***, we may view it as a ***decision tree***.

```java
public BSTNode<E> search(BSTNode<E> p, int k) {
  BSTNode<E> result = null;
  if(p.isExternal()) {
    result = p; /* unsuccessful search */
  }
  else if(p.getKey() == k) {
    result = p; /* successful search */
  }
  else if(k < p.getKey()) {
    result = search(p.getLeft(), k); /* recur on LST */
  }
  else if(k > p.getKey()) {
    result = search(p.getRight(), k); /* recur on RST */
  }
  return result;
}
```


Visualizing BST Operation: Searching (1)

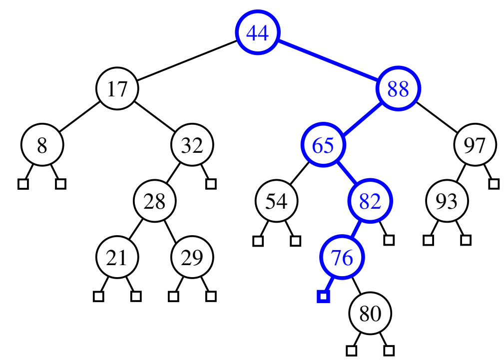
A ***successful*** search for ***key 65***:



The ***internal node*** storing key 65 is **returned**.

Visualizing BST Operation: Searching (2)

- An ***unsuccessful*** search for ***key 68***:



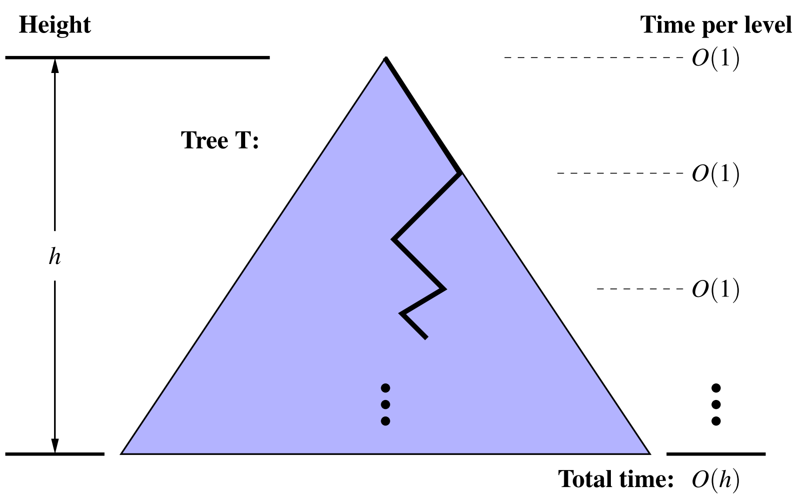
The ***external, left child node*** of the ***internal node*** storing ***key 76*** is **returned**.

- *Exercise* : Provide ***keys*** for different ***external nodes*** to be **returned**.

Testing BST Operation: Searching

```
@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* */
}
```


RT of BST Operation: Searching (1)



RT of BST Operation: Searching (2)

- Recursive calls of `search` are made on a ***path*** which
  - Starts from the **root**
  - Goes down one ***level*** at a time

    RT of deciding from each node to go to LST or RST? [*O(1)*]
  - Stops when the key is found or when a ***leaf*** is reached

    Maximum number of nodes visited by the search? [$h + 1$]

  ∴ RT of ***search on a BST*** is $O(h)$
- Recall: Given a BT with ***n*** nodes, the ***height h*** is bounded as:

  $$log(n + 1) - 1 \le h \le n - 1$$

  - ***Best*** RT of a ***binary search*** is ***O(log(n))*** [***balanced*** BST]
  - ***Worst*** RT of a ***binary search*** is ***O(n)*** [***ill-balanced*** BST]
- ***Binary search*** on non-linear vs. linear structures:

	Search on a **BST**	Binary Search on a **Sorted Array**
START	Root of BST	Middle of Array
PROGRESS	LST or RST	Left Half or Right Half of Array
BEST RT	*O(log(n))*	*O(log(n))*
WORST RT	*O(n)*	

Sketch of BST Operation: Insertion

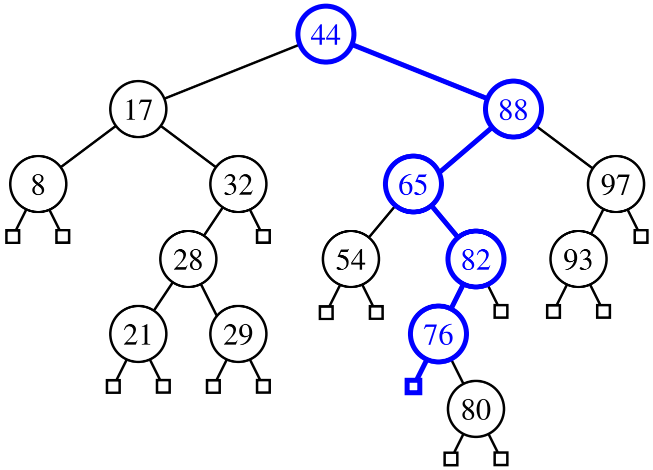
To *insert* an ***entry*** (with **key** k & **value** v) into a BST rooted at ***node*** n:

- Let node ***p*** be the return value from `search(n, k)`.
- If ***p*** is an ***internal node***
  - ⇒ Key k exists in the BST.
  - ⇒ Set p's value to v.
- If ***p*** is an ***external node***
  - ⇒ Key k deos **not** exist in the BST.
  - ⇒ Set p's key and value to k and v.

Running time? [$O(h)$]

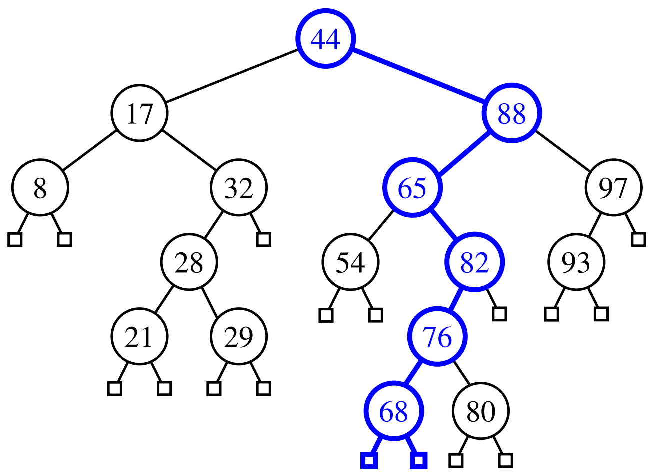
Visualizing BST Operation: Insertion (1)

Before *inserting* an entry with ***key 68*** into the following BST:



Visualizing BST Operation: Insertion (2)

After *inserting* an entry with ***key 68*** into the following BST:



Exercise on BST Operation: Insertion

Exercise: In `BSTUtilities` class, ***implement*** and ***test*** the `void insert(BSTNode<E> p, int k, E v)` method.

Sketch of BST Operation: Deletion

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

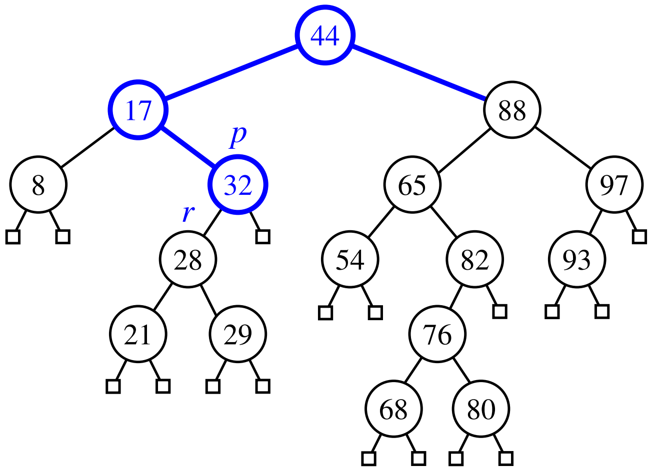
Let node ***p*** be the return value from `search(n, k)`.

- **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***.
  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**.

Running time? [$O(h)$]

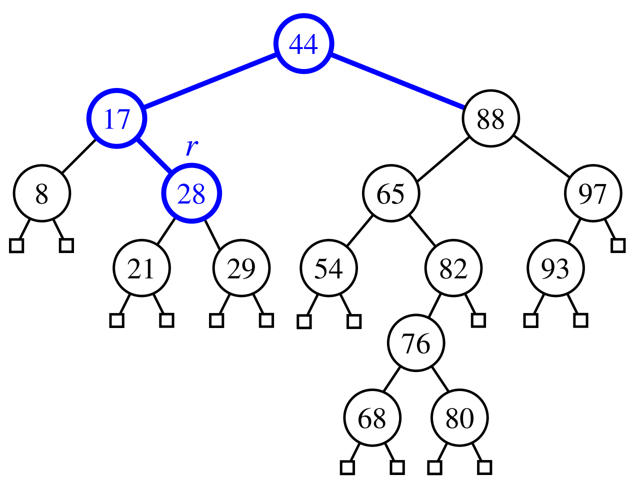
Visualizing BST Operation: Deletion (1.1)

(**Case 3**) Before *deleting* the node storing ***key 32***:



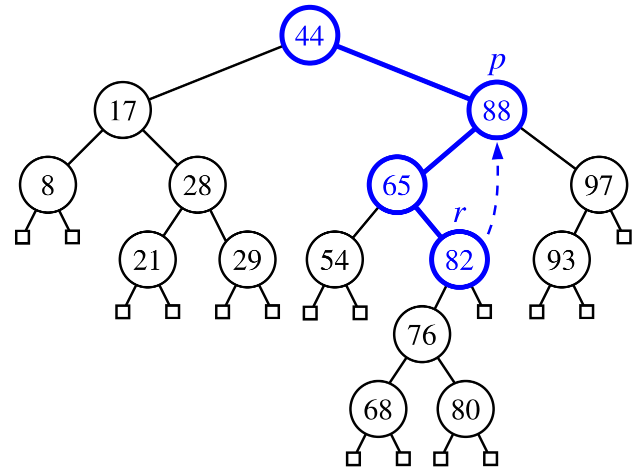
Visualizing BST Operation: Deletion (1.2)

(**Case 3**) After *deleting* the node storing ***key 32***:



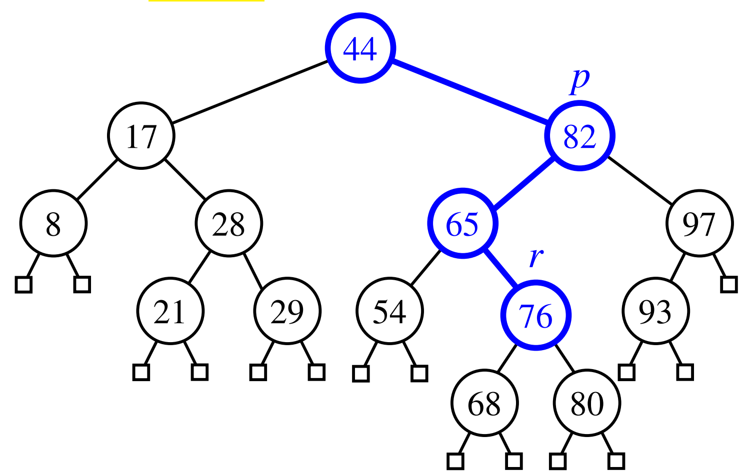
Visualizing BST Operation: Deletion (2.1)

(**Case 4**) Before *deleting* the node storing ***key 88***:



Visualizing BST Operation: Deletion (2.2)

(**Case 4**) After *deleting* the node storing ***key 88***:



Exercise on BST Operation: Deletion

Exercise: In `BSTUtilities` class, ***implement*** and ***test*** the `void delete(BSTNode<E> p, int k)` method.

Balanced Binary Search Trees: Motivation

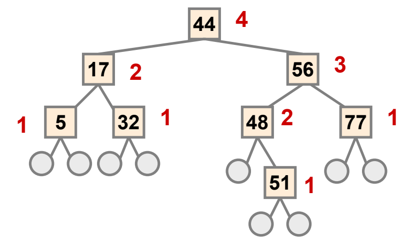
- After ***insertions*** into a BST, the ***worst-case RT*** of a ***search*** occurs when the ***height*** h is at its ***maximum***: ***O(n)***:
  - e.g., Entries were inserted in an decreasing order of their keys
    $$\langle 100, 75, 68, 60, 50, 1 \rangle$$
    $\Rightarrow$ ***One-path***, ***left-slanted*** BST
  - e.g., Entries were inserted in an increasing order of their keys
    $$\langle 1, 50, 60, 68, 75, 100 \rangle$$
    $\Rightarrow$ ***One-path***, ***right-slanted*** BST
  - e.g., Last entry's key is in-between keys of the previous two entries
    $$\langle 1, 100, 50, 75, 60, 68 \rangle$$
    $\Rightarrow$ ***One-path***, ***side-alternating*** BST
- To avoid the worst-case RT ($\because$ a ***ill-balanced tree***), we need to take actions ***as soon as*** the tree becomes ***unbalanced***.

Balanced Binary Search Trees: Definition

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

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

- A ***balanced*** BST T satisfies the ***height-balance property***:
  For every ***internal node*** n, ***heights*** of n's child nodes differ ≤ 1.

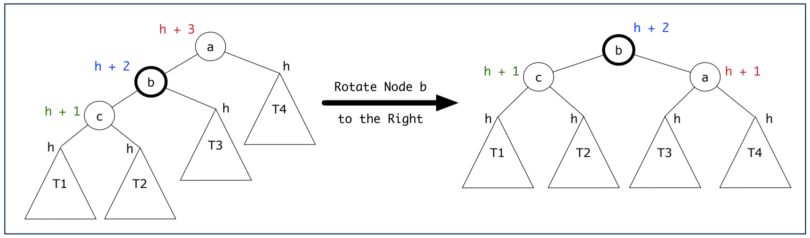


Q: Is the above tree a ***balanced BST***? ✓
Q: Will the tree remain ***balanced*** after inserting 55? ×
Q: Will the tree remain ***balanced*** after inserting 63? ✓

Fixing Unbalanced BST: Rotations

A tree ***rotation*** is performed:

- When the latest **insertion**/**deletion** creates ***unbalanced*** nodes, along the ***ancestor path*** of the node being inserted/deleted.
- To change the ***shape*** of tree, ***restoring*** the ***height-balance property***



Q. An ***in-order traversal*** on the resulting tree?
A. Still produces a sequence of ***sorted keys*** $\langle T_1,\ c,\ T_2,\ b,\ T_3,\ a,\ T_4 \rangle$

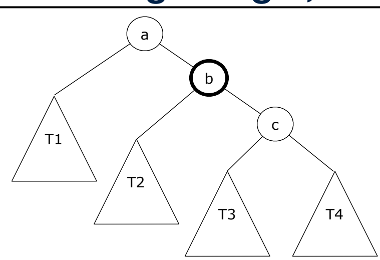
- After ***rotating*** node b to the right:
  - Heights of ***descendants*** (b, c, T_1, T_2, T_3) and ***sibling*** (T_4) stay ***unchanged***.
  - Height of ***parent*** (a) is ***decreased by 1***.
    $\Rightarrow$ ***Balance*** of node a was ***restored*** by the ***rotation***.

After Insertions: Trinode Restructuring via Rotation(s)

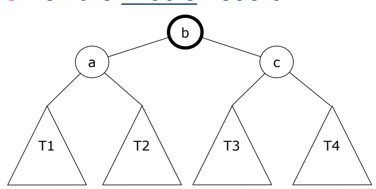
After ***inserting*** a new node n:

- **Case 1**: Nodes on n's ***ancestor path*** remain ***balanced***.
  $\Rightarrow$ No rotations needed
- **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 child node b in n's ***ancestor path***.
  3. Get b's child node c in n's ***ancestor path***.
  4. Perform rotation(s) based on the ***alignment*** of a, b, and c:
     - Slanted the ***same*** way $\Rightarrow$ ***single** rotation* on the **middle** node b
     - Slanted ***different*** ways $\Rightarrow$ ***double** rotations* on the **lower** node c

Trinode Restructuring: Single, Left Rotation



After a ***left rotation*** on the middle node b:

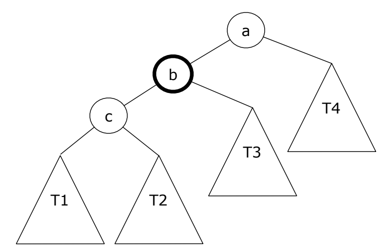


BST property maintained? $\langle T_1,\ a,\ T_2,\ b,\ T_3,\ c,\ T_4 \rangle$

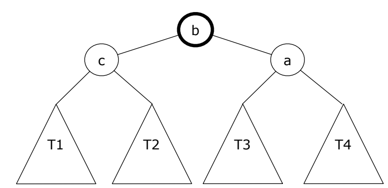
Left Rotation

- ***Insert*** the following sequence of nodes into an empty BST:
  $$\langle 44, 17, 78, 32, 50, 88, 95 \rangle$$
- Is the BST now ***balanced***?
- ***Insert*** 100 into the BST.
- Is the BST still ***balanced***?
- Perform a ***left rotation*** on the appropriate node.
- Is the BST again ***balanced***?

Trinode Restructuring: Single, Right Rotation



After a ***right rotation*** on the middle node b:



BST property maintained? $\langle T_1,\ a,\ T_2,\ b,\ T_3,\ c,\ T_4 \rangle$

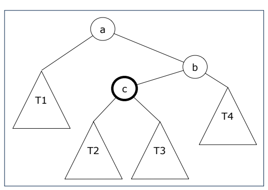
Right Rotation

- ***Insert*** the following sequence of nodes into an empty BST:

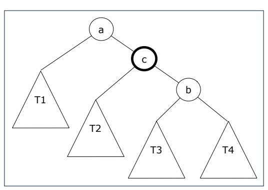
  $$\langle 44, 17, 78, 32, 50, 88, 48 \rangle$$

- Is the BST now ***balanced***?
- ***Insert*** 46 into the BST.
- Is the BST still ***balanced***?
- Perform a ***right rotation*** on the appropriate node.
- Is the BST again ***balanced***?

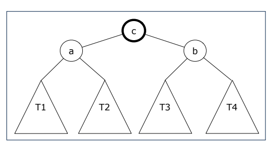
Trinode Restructuring: Double, R-L Rotations



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



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



After Right-Left Rotations

BST property maintained? $\langle T_1,\ a,\ T_2,\ c,\ T_3,\ b,\ T_4 \rangle$

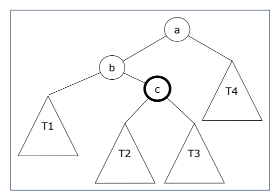
R-L Rotations

- ***Insert*** the following sequence of nodes into an empty BST:

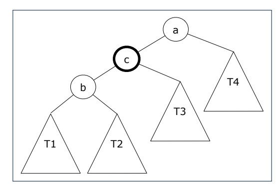
  $$\langle 44, 17, 78, 32, 50, 88, 82, 95 \rangle$$

- Is the BST now ***balanced***?
- ***Insert*** 85 into the BST.
- Is the BST still ***balanced***?
- Perform the ***R-L rotations*** on the appropriate node.
- Is the BST again ***balanced***?

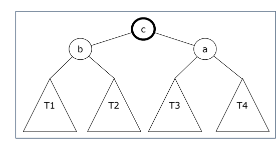
Trinode Restructuring: Double, L-R Rotations



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



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



After Left-Right Rotations

BST property maintained? $\langle T_1,\ b,\ T_2,\ c,\ T_3,\ a,\ T_4 \rangle$

L-R Rotations

- ***Insert*** the following sequence of nodes into an empty BST:

  $$\langle 44, 17, 78, 32, 50, 88, 48, 62 \rangle$$
- Is the BST now ***balanced***?
- ***Insert*** 54 into the BST.
- Is the BST still ***balanced***?
- Perform the ***L-R rotations*** on the appropriate node.
- Is the BST again ***balanced***?

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***]

Single Trinode Restructuring Step

- ***Insert*** the following sequence of nodes into an empty BST:

  $$\langle 44, 17, 62, 32, 50, 78, 48, 54, 88 \rangle$$
- Is the BST now ***balanced***?
- ***Delete*** 32 from the BST.
- Is the BST still ***balanced***?
- Perform a ***left rotation*** on the appropriate node.
- Is the BST again ***balanced***?

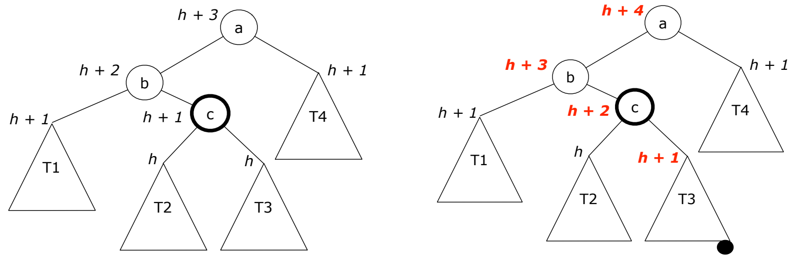
Multiple Trinode Restructuring Steps

- ***Insert*** the following sequence of nodes into an empty BST:

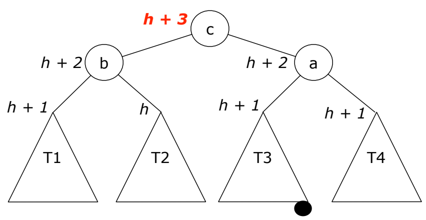
  $$\langle 50, 25, 10, 30, 5, 15, 27, 1, 75, 60, 80, 55 \rangle$$
- Is the BST now ***balanced***?
- ***Delete*** 80 from the BST.
- Is the BST still ***balanced***?
- Perform a ***right rotation*** on the appropriate node.
- Is the BST now ***balanced***?
- Perform another ***right rotation*** on the appropriate node.
- Is the BST again ***balanced***?

Restoring Balance from Insertions

Before Insertion into T3 *After Insertion into T3*

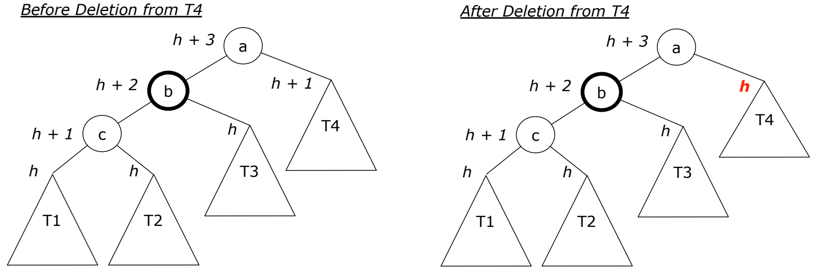


*After Performing L-R Rotations on Node c: Height of Subtree Being Fixed Remains **h + 3***

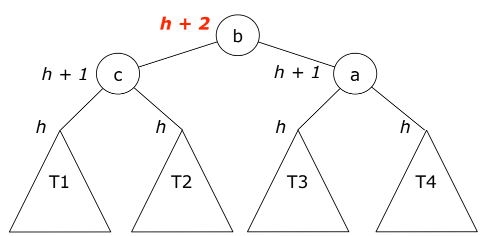


Restoring Balance from Deletions

Before Deletion from T4 *After Deletion from T4*



*After Performing Right Rotation on Node b: Height of Subtree Being Fixed **Reduces its Height by 1**!*



Restoring Balance: Insertions vs. Deletions

- Each ***rotation*** involves only ***POs*** of setting parent-child references.
  - ⇒ ***O(1)*** running time for each tree ***rotation***
- After each ***insertion***, a ***trinode restructuring*** step can ***restore the balance*** of the subtree rooted at the first ***unbalanced*** node.
  - ⇒ ***O(1)*** rotations suffices to restore the balance of tree
- After each ***deletion***, one or more ***trinode restructuring*** steps may ***restore the balance*** of the subtree rooted at the first ***unbalanced*** node.
  - ⇒ May take ***O(log n)*** rotations to restore the balance of tree

Index (1)

Index (2)

Index (3)

Index (4)

Graphs

EECS3101 E:
Design and Analysis of Algorithms
Fall 2025

CHEN-WEI WANG

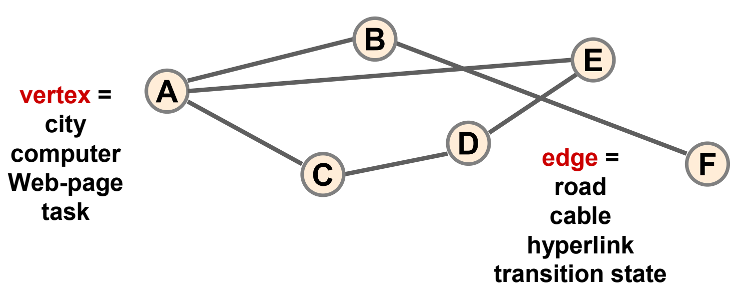
Learning Outcomes of this Lecture

This module is designed to help you understand:

- Vocabulary of the ***Graph*** ADT
- ***Properties*** of Graphs
- ***Algorithms*** on Graphs
  - Traversals: **Depth-**First Search vs. **Breadth-**First Search
  - Topological Sort
  - Minimum Spanning Trees (MST)
  - Dijkstra's Shortest Path Algorithm
- ***Proving*** Properties of Graphs
- ***Implementing*** Graphs in Java

Graphs: Definition

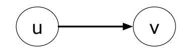
A *graph* $G = (V, E)$ represents ***relations*** that exist between **pairs** of objects.



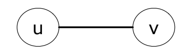
- A set V of ***objects***: ***vertices*** (***nodes***)
- A set E of ***connections*** between objects: ***edges*** (***arcs***)
  - Each ***edge*** (from E) is an **ordered pair** of ***vertices*** (from V).
- e.g., $G = (\{A, B, C, D, E, F\}, \{(A, B), (A, C), (A, E), (C, D), (D, E), (B, F)\})$

Directed vs. Undirected Edges

- An ***edge*** (u, v) connects two ***vertices*** u and v in the graph.
- ***Edge*** (u, v) is *directed* if it indicates the direction of travel.



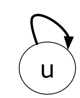
  - Vertex u is the ***origin***.
  - Vertex v is the ***destination***.
  - $(u, v) \neq (v, u)$
- ***Edge*** (u, v) is *undirected* if it does not indicate a direction.



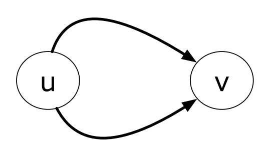
  - $(u, v) = (v, u)$
- 1 undirected edge $(u, v) \equiv$ 2 directed edges (u, v) and (v, u).
- ***Directions*** of ***edges*** represent dependency, order, or flow.

Self vs. Parallel Edges

- An edge (u, u), either directed or undirected, is called a ***self-edge*** (or a ***self-loop***).



- Edges that have the same two end vertices are ***parallel edges*** or ***multiple edges***.



  e.g., In a flight network graph, there are more than one airlines flying between two Seoul and Vancouver.
- A ***simple graph*** has **no** ***self-loops*** and ***parallel edges***.

Vertices

Given an ***edge*** (u, v):

- Vertices u and v are its two ***End vertices*** (***Endpoints***).
- The two end vertices u and v is said to be ***adjacent***.
- Edge (u, v) is ***incident on*** the two end vertices u and v.
- When edge (u, v) is directed:
  - u is **origin** and v is **destination**
  - Edge (u, v) is an ***outgoing edge*** of the origin u
  - Edge (u, v) is an ***incoming edge*** of the destination u
- The ***degree*** of a vertex v is the number of edges ***incident on*** v.

Directed vs. Undirected Graphs

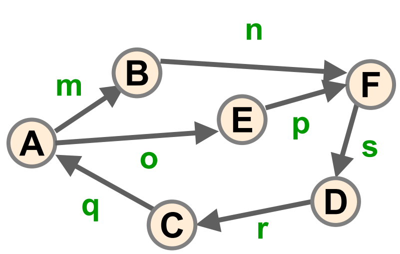
- In a ***directed graph***, **all** edges are directed.

  e.g., dependency graphs (inheritance relationships, method calls, *etc.*)
- In an ***undirected graph***, all edges are undirected.

  e.g., Subway map of Young-University Line
- In a ***mixed graph***, **some** edges directed; **some** undirected.

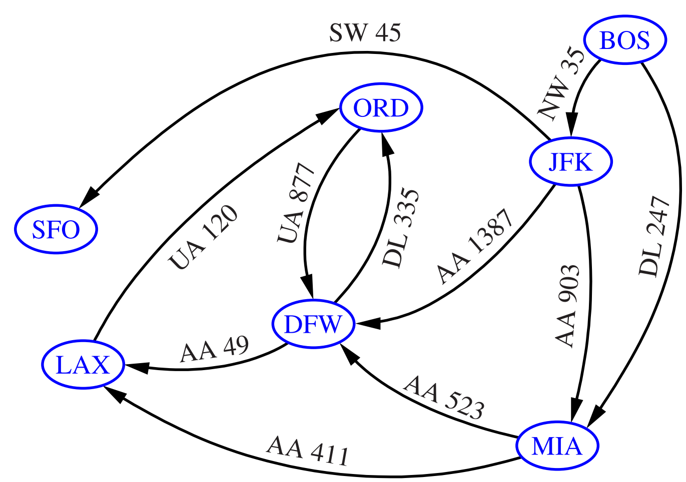
  e.g., A city map has street intersections as vertices and streets as edges: each street may be one-way (a directed edge) or both-way (an undirected edge).

Exercise (1)

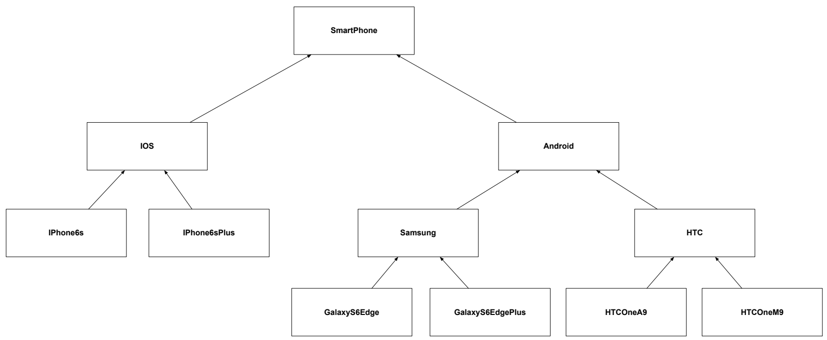


- *End vertices* of edge *m*? [*A*, *B*]
- *Outgoing edges* of vertex *A*? [*m*, *o*]
- *Incoming edges* of vertex *A*? [*q*]
- Edges *incident* on vertex *A*? [*m*, *o*, *q*]
- *Degree* of vertex *A*? [3]

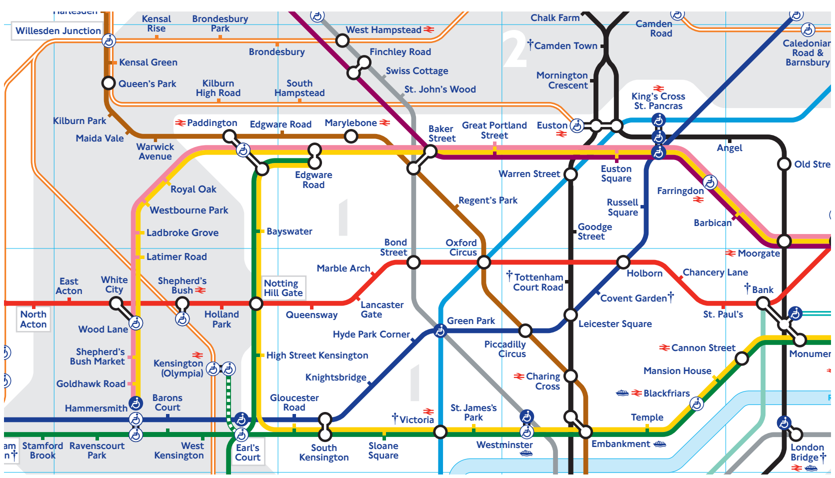
Directed Graph Example (1): A Flight Network



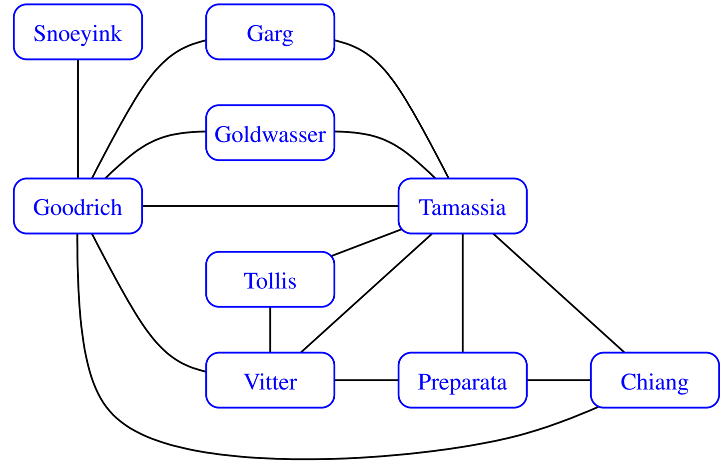
Directed Graph Example (2): Class Inheritance



Undirected Graph Example (1): London Tube



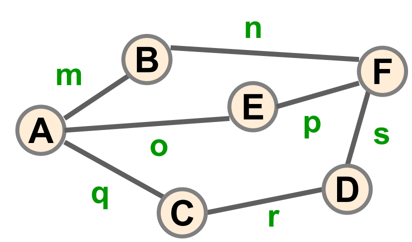
Undirected Graph Example (2): Co-authorship



Basic Properties of Graphs (1)

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

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

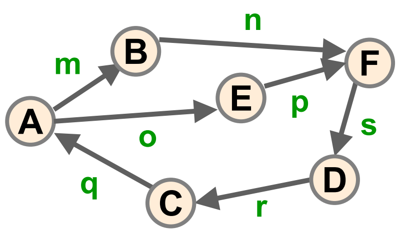


  - ***Intuition***: Each edge (u, v) contributes to degrees of both u and v.
  - ***Formal Proof***: ***Mathematical inductoin*** on $|V|$.
- Prove that the claim still holds on graphs that are **not simple**.

Basic Properties of Graphs (2)

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

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

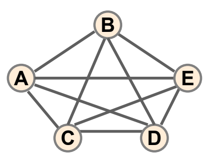


  - ***Intuition***: Each directed edge (u, v) contributes to the `out-degree` of origin u and the `in-degree` of destination v.
  - ***Formal Proof***: ***Mathematical inductoin*** on $|V|$.
- Prove that the claim still holds on graphs that are **not simple**.

Basic Properties of Graphs (3)

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

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



- **Intuition**: Say $V = \{v_1, v_2, \dots, v_n\}$
  - ***Maximum*** value of m is obtained when **each** vertex is connected to **all other** $n-1$ vertices: $n \cdot (n-1)$
  - Since G is **undirected**, for each pair of vertices v_i and v_j, we have **double-counted** (v_i, v_j) and (v_j, v_i): $\frac{n \cdot (n-1)}{2}$
- G is a ***complete graph*** when $m = \frac{n \cdot (n-1)}{2}$

Paths and Cycles (1)

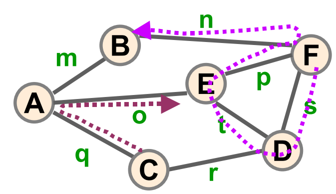
Given a graph $G = (V, E)$:

- A ***path*** of G is a sequence of **alternating** vertices and edges, which **starts** and **ends** at vertices:

$$\langle v_1, e_1, v_2, e_2, \dots, v_{n-1}, e_{n-1}, v_n \rangle \quad v_i \in V, 1 \leq i \leq n, e_j \in E, 1 \leq j < n$$

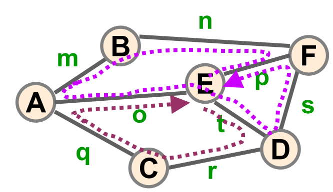
- A ***cycle*** of G is a ***path*** of G with the **same** vertex appearing more than once.
- A ***simple path*** of G is a ***path*** of G with **distinct** vertices.
- A ***simple cycle*** of G is a ***cycle*** of G with **distinct** vertices (except the beginning and end vertices that form the cycle).
- Given two vertices u and v in G, vertex v is ***reachable*** from vertex u if there exists a ***path*** of G such that its ***start vertex*** is u and ***end vertex*** is v.
  - Vertex v may be reachable from vertex u via more than one paths.
  - Any of the ***reachable paths*** from u to v contains a cycle $\Rightarrow$ An **infinite** number of reachable paths from u to v.

Paths and Cycles (2)



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

Simple Path = (C, q, A, o, E)



Cycle = (E, p, F, n, B, m, A, o, E, t, D, s, F, p, E)

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

Vertex F is ***reachable*** from vertex A via:

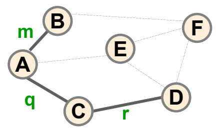
- (A, m, B, n, F)
- (A, o, E, p, F)
- (A, o, E, t, D, s, F)

...

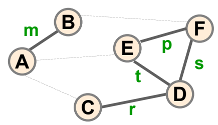
Subgraphs vs. Spanning Subgraphs

Given a graph $G = (V, E)$:

- A ***subgraph*** of G is another graph $G' = (\boldsymbol{V'}, \boldsymbol{E'})$ such that $\boxed{V' \subseteq V}$ and that $E' \subseteq E$.
  e.g., $G_1 = (\{A, B, C, D, E, F\}, \{m, q, r\})$



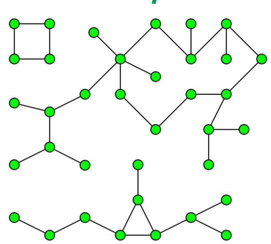
- A ***spanning subgraph*** of G is another graph $G' = (V', E')$ s.t. $\boxed{V' = V}$ and that $E' \subseteq E$.
  e.g., $G_2 = (\{A, B, C, D, E, F\}, \{m, p, s, t, r\})$



Connected Graph vs. Connected Components

Given a graph $G = (V, E)$:

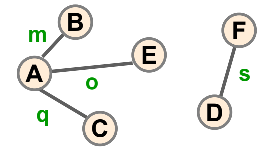
- G is ***connected***: there is a ***path*** between any two vertices of G.
  e.g., Spanning subgraph G_2 extended with the edge n, o, or q
- G's ***connected components***: G's **maximal** ***connected subgraphs***.
  A ***CC*** is **maximal** in that it **cannot** be expanded any further.
  e.g., How many ***connected components*** does the following graph have?



Answer: 3

Forests vs. Trees

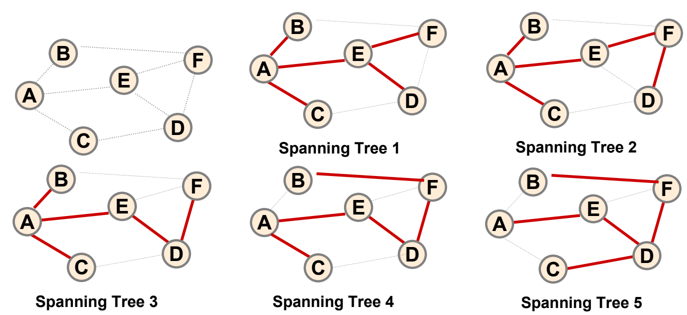
- A ***forest*** is an ***undirected*** graph **without** ***cycles***.



  - $\boxed{\textit{Acyclic}}$:
    Any two ***vertices*** are connected via **at most one** ***path***.
  - A ***forest*** may or may **not** be ***connected***.
    $(\exists v_1, v_2 \bullet \{v_1, v_2\} \subseteq V \land \neg\texttt{connected}(v_1, v_2)) \Rightarrow \neg\texttt{connected}(\textit{Forest}\ G)$
- A ***tree*** is a **connected** ***forest***.
  - $\boxed{\textit{Acyclic}\ \&\ \textit{Connected}}$:
    Any two ***vertices*** are connected via **exactly one** path.
  - e.g., Add either edge (E, F) or (E, D) to the above forest.

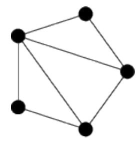
Spanning Trees

- A ***spanning tree*** of graph G: a ***spanning subgraph*** that is also a ***tree***
  - $\Rightarrow$ A ***spanning tree*** of G is a **connected** ***spanning subgraph*** of G that contains **no** ***cycles***.
  - $\Rightarrow \neg\texttt{connected}(G) \Rightarrow \neg(\exists G' \bullet G'$ is a *spanning tree* of G)

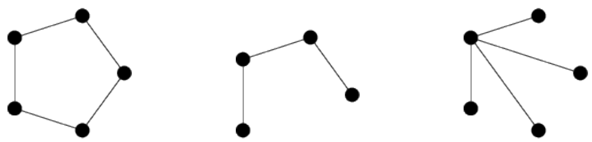


Exercise (2)

Given a graph



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



(a) (b) (c)

- **(a)**: ***spanning subgraph*** containing a ***cycle*** ($\therefore$ not a ***tree***).
- **(b)**: ***tree*** but **not** ***spanning***.

Basic Properties of Graphs (4)

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

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

- Prove the ***spanning tree*** case via *mathematical induction on n*:
  - **Base Cases**: $n = 1 \Rightarrow m = 0$, $n = 2 \Rightarrow m = 1$, $n = 3 \Rightarrow m = 2$
  - **Inductive Cases**: Assume that a spanning tree has n vertices and $n - 1$ edges.
  - When adding a new vertex v' into the existing graph, we may only expand the **existing** ***spanning tree*** by connecting v' to **exactly one** of the existing vertices; otherwise there will be a ***cycle***.
  - This makes the new spanning tree contains $n + 1$ vertices and n edges.
- When G is a ***forest***, it may be **unconnected** $\Rightarrow m < n - 1$
- When G is **connected**, it may contain ***cycles*** $\Rightarrow m \geq n$

Graph Traversals: Definition

Given a graph G = (**V**, **E**):

- A ***traversal*** of G is a systematic procedure for examining **all** its vertices **V** and edges **E**.
- A ***traversal*** of G is considered ***efficient*** if its ***running time*** is ***linear*** on |**V**| and/or |**E**|. [e.g., $O(|V| + |E|)$]

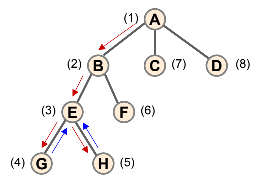
Graph Traversals: Applications

Fundamental questions about graphs involve ***reachability***.
Given a graph G = (**V**, **E**) (directed or undirected):

- Given a vertex **u**, find **all** other vertices in G ***reachable*** from **u**.
- Given a vertex **u** and a vertex **v**:
  - compute a ***path*** from **u** to **v**, or report that there is **no** such a path.
  - compute a ***path*** from **u** to **v** that involves the ***minimum*** number of edges, or report that there is **no** such a path.
- Determine whether or not G is ***connected***.
- Given that G is ***connected***, compute a ***spanning tree*** of G.
- Compute the ***connected components*** of G.
- Identify a ***cycle*** in G, or report that G is ***acyclic***.

Depth-First Search (DFS)

- A ***Depth-First Search*** (***DFS***) of graph G = (V, E), starting from some vertex $v \in V$, 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.



- DFS resembles the ***preorder traversal*** in trees.
- Use a ***LIFO stack*** to keep track of the nodes to be visited.

DFS: Marking Vertices and Edges

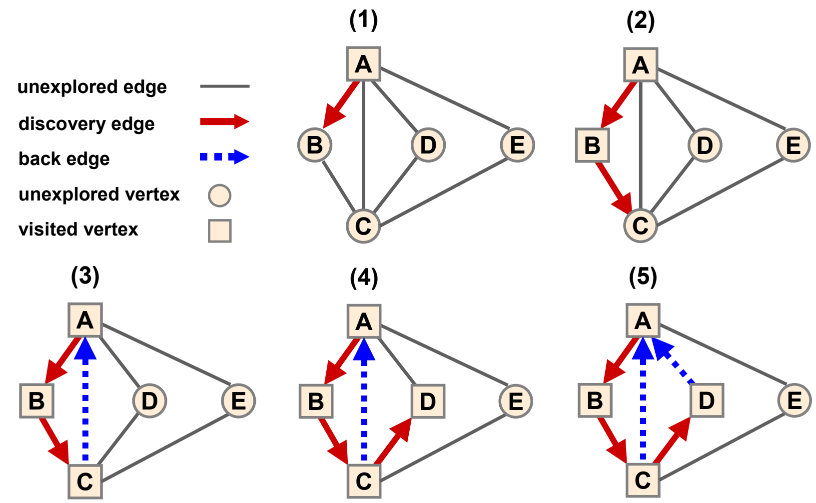
Before the ***DFS*** starts:

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

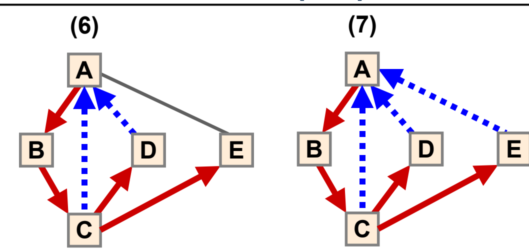
Over the course of a ***DFS***, we **mark** vertices and edges:

- 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)

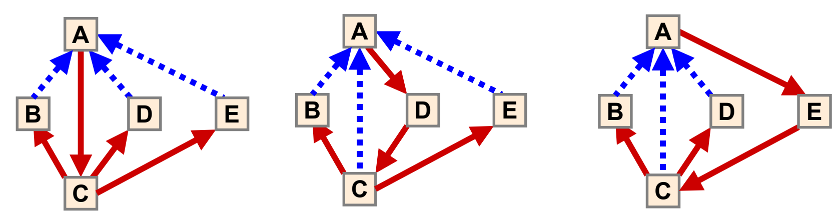
DFS: Illustration (1.1)



DFS: Illustration (1.2)

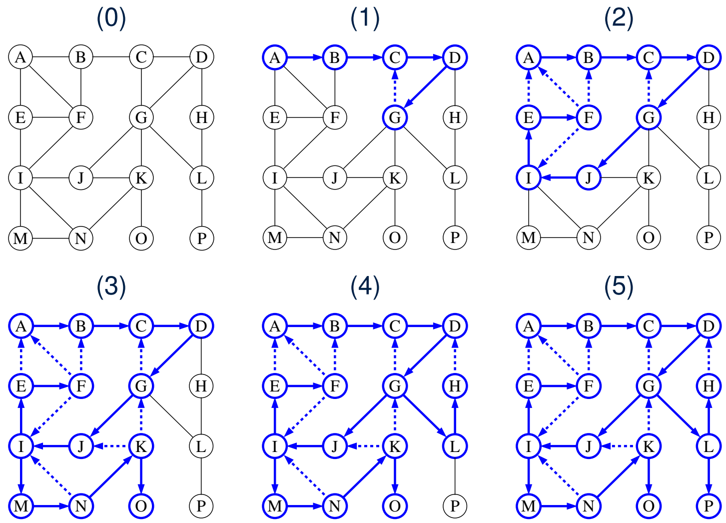


Other solutions (different ***incident edges*** on vertex ***A*** to get started):



edge AC visited first   edge AD visited first   edge AE visited first

DFS: Illustration (2)



DFS: Properties

1. Running Time?
   - Every **vertex** is set as ***visited*** **at most once**.
   - Each **edge** is set as either ***DISCOVERY*** or ***BACK*** **at most once**.

   $\Rightarrow$ $O(m+n)$
2. For a ***DFS*** starting from vertex u in a graph G = (V, E):

   2.1 $|visited\ nodes| = |V| \Rightarrow G$ is ***connected***

   2.2 $|visited\ nodes| < |V| \Rightarrow G$ has > 1 ***connected components***

   2.3 There are **no** ***back edges*** $\Rightarrow G$ is ***acyclic***
3. For a ***DFS*** starting from vertex u in an undirected graph G:

   3.1 The traversal visits **all** nodes in the ***connected component*** containing u.

   3.2 ***Discovery edges*** form a ***spanning tree*** (with $|V|-1$ edges) of the ***connected component*** containing u.
4. If a graph G is **not** ***connected***, then it takes **multiple** runs of ***DFS*** to identify **all** G's ***connected components***.

Graph Questions: Adapting DFS

- Given a (directed or undirected) graph $G = (V, E)$:
  - Find a ***path*** between vertex u and vertex v.

    Start a DFS from u and stop as soon as v is encountered.
  - Is vertex v ***reachable*** from vertex u?

    No if a DFS starting from u never encounters v.
  - Find all ***connected components*** of G.
    - Continuously apply ***DFS***'s until the entire set V is visited.
    - Each ***DFS*** produces a **subgraph** representing a new ***CC***.
  - Given that G is ***connected***, find a ***spanning tree*** of it.

    G is **connected**. $\Rightarrow$ G's only ***CC*** is its ***spanning tree***.
- Given an undirected graph $G = (V, E)$:
  - Is G ***connected***?
    - Start a ***DFS*** from an arbitrary vertex, and count # of visited nodes.
    - When the traversal completes, compare the counter value against $|V|$.
  - Is G ***acyclic***?
    - Start a ***DFS*** from an arbitrary vertex.
    - Return **no** (i.e., a ***cycle*** exists) as soon as a ***back edge*** is found.

Graphs in Java: DL Node and List

For each graph, maintain two ***doubly-linked lists*** for ***vertices*** and ***edges***.

```
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 */
}
```

```
public class DoublyLinkedList<E> {
 private int size;
 private DLNode<E> header; private DLNode<E> trailer;
 public void remove (DLNode<E> node) {
  DLNode<E> pred = node.getPrev();
  DLNode<E> succ = node.getSucc();
  pred.setNext(succ); succ.setPrev(pred);
  node.setNext(null); node.setPrev(null);
  size --;
 }
}
```

Graphs in Java: Vertex and Edge

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

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


Graphs in Java: Interface (1)

```java
public interface Graph<V,E> {
  /* Number of vertices of the graph */
  public int getNumberOfVertices();

  /* Number of edges of the graph */
  public int getNumberOfEdges();

  /* Vertices of the graph */
  public Iterable<EdgeListVertex<V>> getVertices();

  /** Edges of the graph */
  public Iterable<EdgeListEdge<E, V>> getEdges();

  /* Number of edges leaving vertex v. */
  public int getOutDegreeOf(EdgeListVertex<V> v);

  /* Number of edges for which vertex v is the destination. */
  public int getInDegreeOf(EdgeListVertex<V> v);

  public int getDegreeOf(EdgeListVertex<V> v);
```


Graphs in Java: Interface (2)

```java
  /* Edges for which vertex v is the origin. */
  public Iterable<Edge<E, V>> getOutgoingEdgesOf(Vertex<V> v);

  /* Edges for which vertex v is the destination. */
  public Iterable<Edge<E, V>> getIncomingEdgesOf(Vertex<V> v);

  /* The edge from u to v, or null if they are not adjacent. */
  public Edge<E, V> getEdgeBetween(Vertex<V> u, Vertex<V> v);
```


Graphs in Java: Interface (3)

```java
  /* Inserts a new vertex, storing given element. */
  public Vertex<V> addVertex(V element);

  /* Inserts a new edge between vertices u and v,
   * storing given element.
   */
  public Edge<E, V> addEdge(Vertex<V> u, Vertex<V> v, E element);

  /* Removes a vertex and all its incident edges from the graph. */
  public void removeVertex(Vertex<V> v);

  /* Removes an edge from the graph. */
  public void removeEdge(Edge<E, V> e);
} /* end Graph */
```

Graphs in Java: Edge List (1)

Each ***vertex*** or ***edge*** stores a ***reference*** to its ***position*** in the respective vertex or edge list.

⇒ ***O(1)*** **deletion** of the vertex or edge from the list.

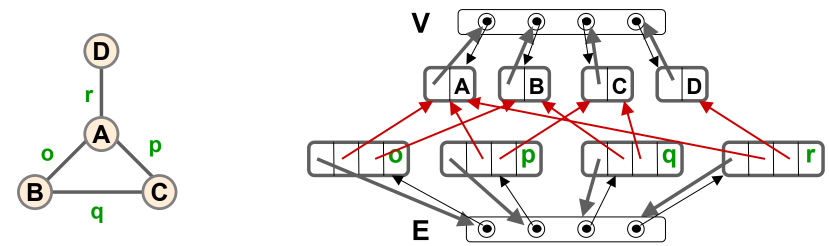
```
public class EdgeListVertex<V> extends Vertex<V> {
  private DLNode<Vertex<V>> vertexListPosition;
  /* setter and getter for vertexListPosition */
}
```

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


Graphs in Java: Edge List (2)

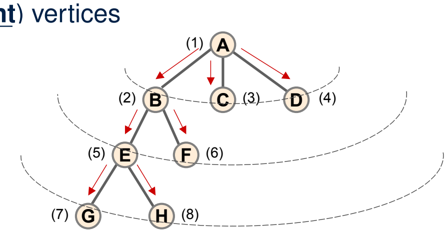
```
public class EdgeListGraph<V, E> implements Graph<V, E> {
  private DoublyLinkedList<EdgeListVertex<V>> vertices;
  private DoublyLinkedList<EdgeListEdge<E, V>> edges;
  private boolean isDirected;

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


Graphs in Java: Edge List (3)



Breadth-First Search (BFS)

- A ***breadth-first search (BFS)*** of graph G = (V, E), starting from some vertex $v \in V$:
  - 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.
- Use a ***FIFO queue*** to keep track of the nodes to be visited.

BFS in Java: Marking Vertices and Edges

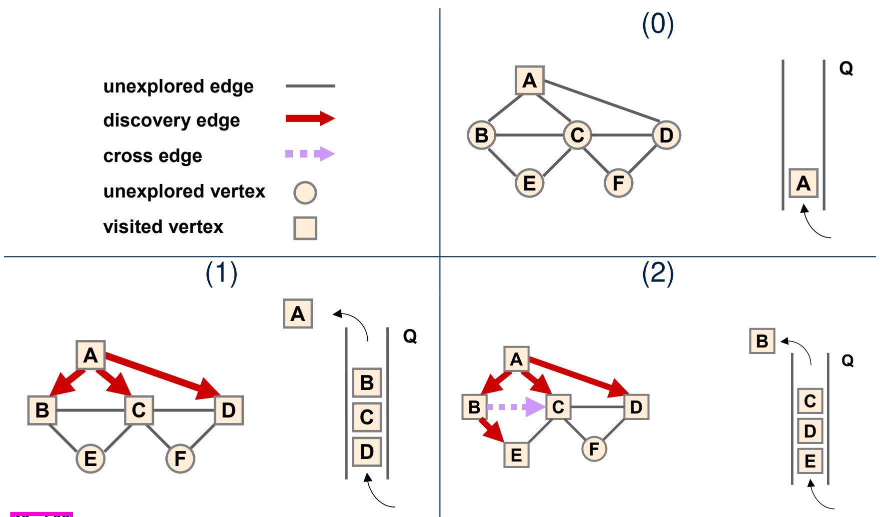
Before the ***BFS*** starts:

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

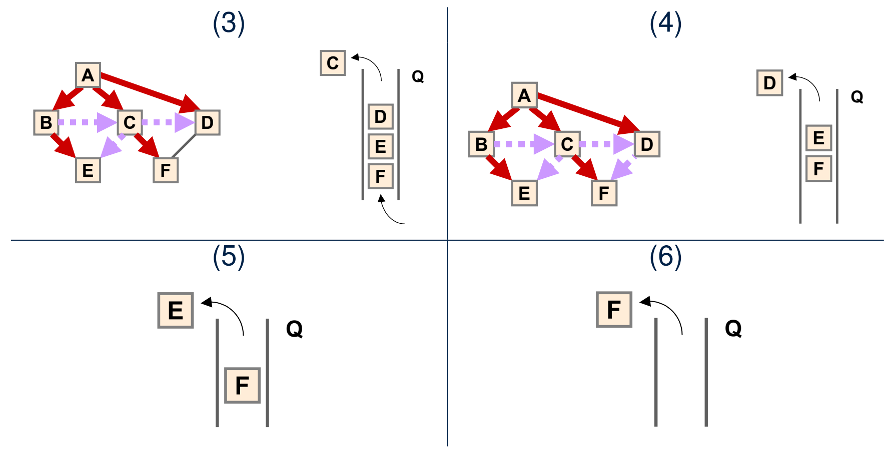
Over the course of a ***BFS***, we **mark** vertices and edges:

- 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***).
- A ***cross*** edge:
  - Always connects to vertices *at the same level*
  - Can not connect to vertices at an upper or a lower level
    ∵ It would've been or will be marked as a ***discovery edge***.

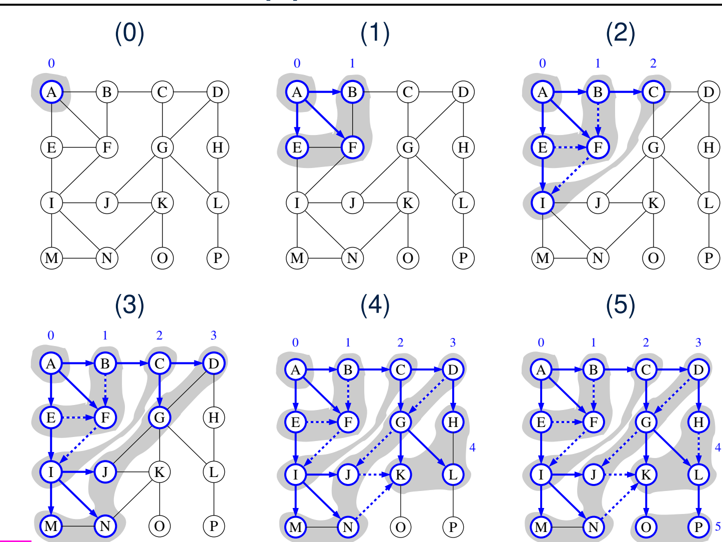
Iterative BFS: Illustration (1.1)



Iterative BFS: Illustration (1.2)



BFS: Illustration (2)



BFS: Properties

1. Running Time?
   - Every **vertex** is set as ***visited*** **at most once**.
   - Each **edge** is set as either ***DISCOVERY*** or ***CROSS*** **at most once**.
   - $\Rightarrow$ $O(m+n)$
2. For a ***BFS*** starting from vertex u in a graph G = (V, E):
   - 2.1 $|visited\ nodes| = |V| \Rightarrow G$ is ***connected***
   - 2.2 $|visited\ nodes| < |V| \Rightarrow G$ has > 1 ***connected components***
   - 2.3 A ***cross edge*** connects vertices that in different ***branches***. In a directed graph, this does **not** necessarily creates a **cycle**.
3. For a ***BFS*** starting from vertex u in an undirected graph G:
   - 3.1 The traversal visits **all** nodes in the ***connected component*** containing u.
   - 3.2 ***Discovery edges*** form a ***spanning tree*** or ***level tree*** (with $|V|-1$ edges) of the ***connected component*** containing u.
4. If a graph G is **not** ***connected***, then it takes **multiple** runs of ***BFS*** to identify **all** G's ***connected components***.

Graph Questions: Adapting BFS

- Given a (directed or undirected) graph $G = (V, E)$:
  - Find a ***shortest path*** (by **edges**) between vertex u and vertex v.
    Start a BFS from u and stop as soon as v is encountered.
  - Is vertex v ***reachable*** from vertex u?
    No if a BFS starting from u never encounters v.
  - Find all ***connected components*** of G.
    - Continuously apply ***BFS***'s until the entire set V is visited.
    - Each ***BFS*** produces a **subgraph** representing a new ***CC***.
  - Given that G is ***connected***, find a ***spanning tree*** of it.
    G is **connected**. $\Rightarrow$ G's only ***CC*** is its ***spanning tree***.
- Given an undirected graph $G = (V, E)$:
  - Is G ***connected***?
    - Start a ***BFS*** from an arbitrary vertex, and count # of visited nodes.
    - When the traversal completes, compare the counter value against $|V|$.
  - Is an undirected G ***acyclic***?
    - Start a ***BFS*** from an arbitrary vertex.
    - Return **no** (i.e., a ***cycle*** exists) as soon as a ***cross edge*** is found.

Graphs in Java: Adjacency List (1)

- Extends the ***edge list*** structure
- Each ***vertex*** v also stores a list of ***incident edges***.
  $\Rightarrow$ vertex-based methods such as `outgoingEdges` and `removeVertex` takes **O(d_v)** rather than $O(|E|)$
- Each ***edge*** also stores **references** to its ***positions*** in both ***lists of incident edges*** of its two end vertices.
  $\Rightarrow$ **O(1)** deletion of the ***edge*** from the ***incident edges list***.

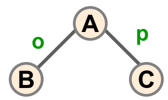
```
class AdjacencyListVertex<V> extends EdgeListVertex<V> {
 private DoublyLinkedList<AdjacencyListEdge<E, V>> incidentEdges;
 /* getter for incidentEdges */
}
```

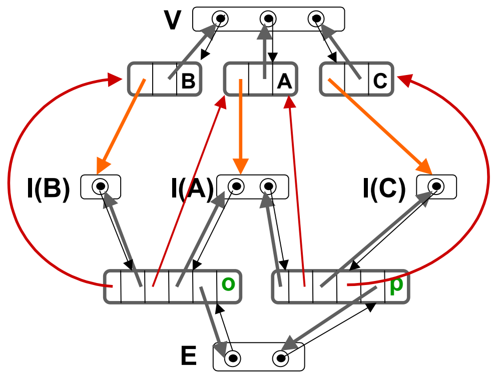
```
class AdjacencyListEdge<V> extends EdgeListEdge<V> {
 DLNode<Edge<E, V>> originIncidentListPos;
 DLNode<Edge<E, V>> destIncidentListPos;
}
```


Graphs in Java: Adjacency List (2)

```
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) {
  this.vertices = new DoublyLinkedList<>();
  this.edges = new DoublyLinkedList<>();
  this.isDirected = isDirected;
 }
```

Graphs in Java: Adjacency List (3)

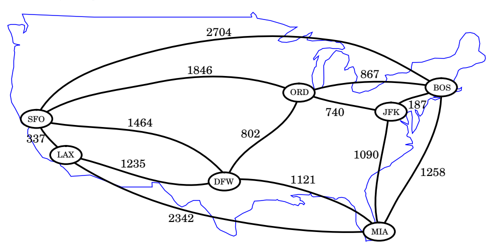




Weighted Graphs

A graph may usefully carry ***weights*** on its ***edges***:

- e.g., Edges of a graph of cities denote distances.



- In a ***weighted graph***, for each *edge* $e = (u, v)$, we write $w(u, v)$ to denote the numerical value of *edge e's weight*.
  e.g., $w(JFK, ORD) = 740$, $w(ORD, DFW) = 802$, $w(DFW, LAX) = 1235$
- ***Weights*** on edges may be considered as "***cost***".
  ⇒ When there are **more than one** ***paths*** existing between two ***vertices***, choose one whose "***total cost***" is the ***minimum***.
- We assume that all edge weights are **non-negative** (i.e., ≥ 0).

Shortest Paths in Weighted Graphs

- Given a ***path P*** $= (v_0, v_1, \ldots, v_k)$, with $k + 1$ ***vertices*** and k ***edges***, we define $\boldsymbol{w(P)}$ as the ***length*** (or ***weight***) of ***P***:

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

e.g., $w(JFK, ORD, DFW, LAX) = 2777$, $w(JFK, MIA, DFW, LAX) = 3446$

- *d(u, v)* denotes the ***distance*** or ***shortest path*** between ***u*** and ***v***: **minimum** weight sum of a path between ***u*** and ***v***.
  e.g.

$$\begin{array}{rcl} d(JFK, LAX) & = & w(JFK, ORD, DFW, LAX) \\ & > & w(JFK, MIA, DFW, LAX) \end{array}$$

- If there is no path existing between u and v, then $d(u, v) = \infty$

Dijkstra's Shortest Path Algorithm

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]
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**:
   $v \in Q \land \boxed{D(u) + w(u, v) < D(v)}$, then:
   - Set $D(v) = D(u) + w(u, v)$
   - Set $a(v) = u$

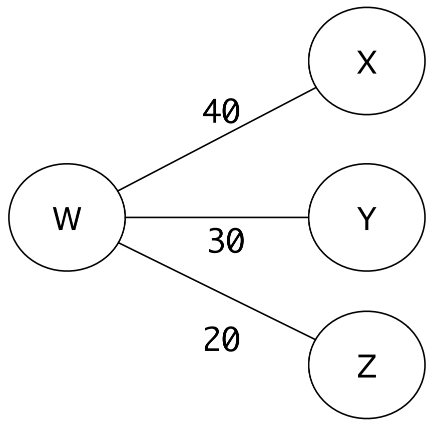
   2.3 Remove vertex u from Q.

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

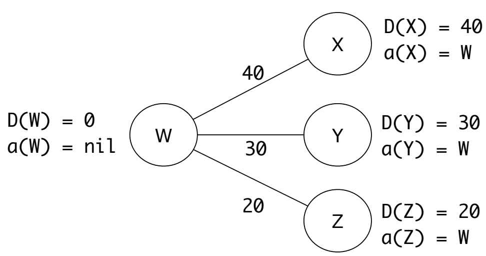
- $\boxed{D(t) = d(s, t)}$ (i.e., weight of *shortest path* from s to t).
- **Reversing** t's ***ancestor path*** → *shortest path*: $\langle s, \ldots, a(t), t\rangle$

Dijkstra's Algorithm: Example (1) Input

Perform Dijkstra's algorithm on the following graph, starting with a ***source vertex*** W:



Dijkstra's Algorithm: Example (1) Output



History of Q's contents:

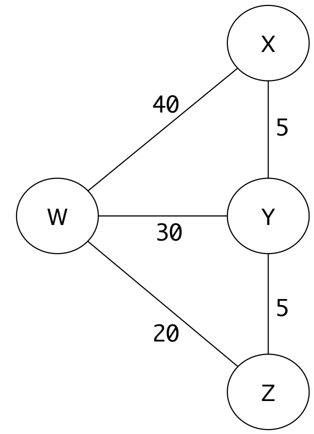
W	X	Y	Z

Shortest paths and distances:

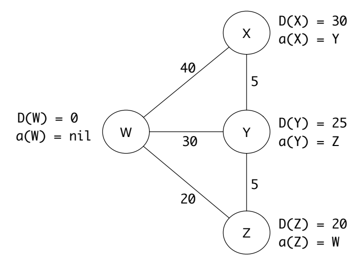
- W to X: $\langle a(X),\ X\rangle = \langle W,\ X\rangle$; $d(W, X) = D(X) = 40$
- W to Y: $\langle a(Y),\ Y\rangle = \langle W,\ Y\rangle$; $d(W, Y) = D(Y) = 30$
- W to Z: $\langle a(Z),\ Z\rangle = \langle W,\ Z\rangle$; $d(W, Z) = D(Z) = 20$

Dijkstra's Algorithm: Example (2) Input

Perform Dijkstra's algorithm on the following graph, starting with a ***source vertex*** W:



Dijkstra's Algorithm: Example (2) Output



History of Q's contents:

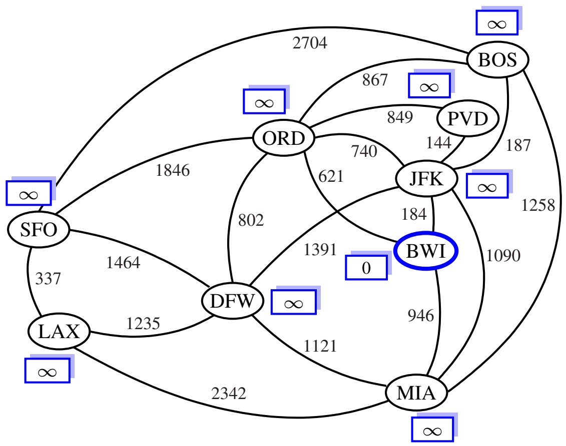
W	X	Y	Z

Shortest paths and distances:

- W to X: $\langle a(Z),\ a(Y),\ a(X),\ X\rangle = \langle W,\ Z,\ Y,\ X\rangle$; $d(W, X) = D(X) = 30$
- W to Y: $\langle a(Z),\ a(Y),\ Y\rangle = \langle W,\ Z,\ Y\rangle$; $d(W, Y) = D(Y) = 25$
- W to Z: $\langle a(Z),\ Z\rangle = \langle W,\ Z\rangle$; $d(W, Z) = D(Z) = 20$

Dijkstra’s Algorithm: Exercise

Perform Dijkstra’s algorithm on the following initial configuration, starting with a ***source vertex*** *BWI*:



Correctness of Loops

How do we prove that the following loop is correct?

```
{Q}
S_init
while(B) {
  S_body
}
{R}
```

In case of C/Java, $\boxed{B}$ denotes the ***stay condition***.

- In C/Java, there is **not** native, syntactic support for checking the ***correctness*** of loops.
- Instead, we have to **manually** add assertions to encode:
  - LOOP INVARIANT [for establishing ***partial correctness***]
  - LOOP VARIANT [for ensuring ***termination***]

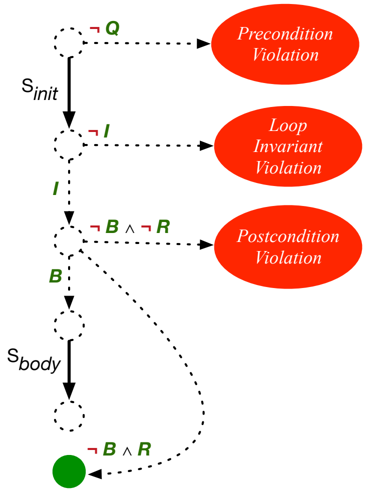
Specifying Loops

- ***Loop Invariant*** (***LI***): Boolean expression for measuring/proving ***partial*** *correctness*
  - **Established** before the very first iteration.
  - **Maintained** TRUE after each iteration.

Specifying Loops: Syntax

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

Specifying Loops: Runtime Checks (1)

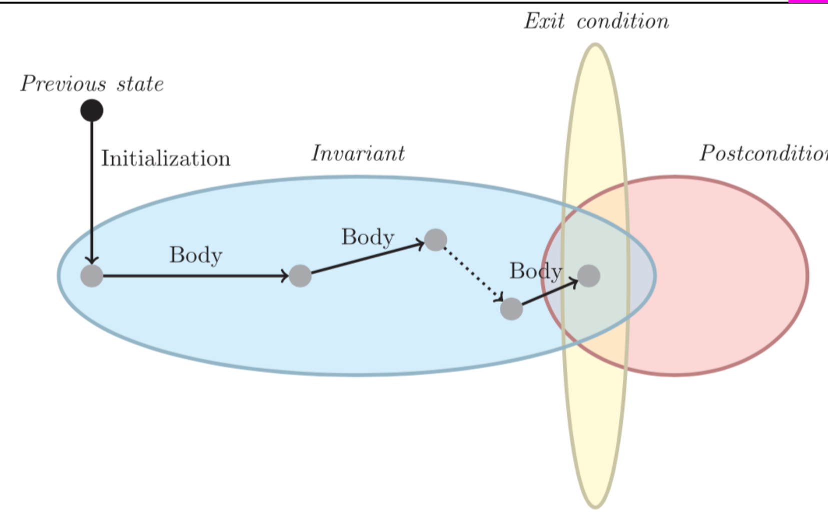


Specifying Loops: Runtime Checks (2)

```
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 */
}
```

L1: Change to `1 <= i && i <= 5` for a ***Loop Invariant Violation***.

Specifying Loops: Visualization



Digram Source: page 5 in *Loop Invariants: Analysis, Classification, and Examples*

Dijkstra's SP Algorithm: Loop Invariant

- 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.

- Formally:

$$\forall u \bullet u \in V \wedge u \notin Q \Rightarrow D(u) = d(s, u)$$

- Important assumption: weights are ***non-negative***.
- To relax this assumption, update **visited** nodes.

  [ref: Bellman-Ford]

  $\Rightarrow$ Worse running time: $O(|V|^3)$ (rather than $O(|V|^2 \cdot log\ |V|)$)

Running Time of Dijkstra's Algorithm (1)

```
1  ALGORITHM: Dijkstra-Shortest-Path
2    INPUT: Graph G = (V, E); Source Vertex s ∈ V
3    OUTPUT: For t ∈ V (t ≠ s),
4      • D(t) := d(s,t)
5      • Shortest Path: ⟨s,...,a(a(t)),a(t),t⟩
6  PROCEDURE:
7    D(s) = 0
8    for(t ∈ (V \ {s})): D(t) := ∞
9    for(v ∈ V): a(v) := nil
10   for(v ∈ V): Q.insert(v) -- Q is a PQ keyed by D
11   while ( ¬Q.isEmpty() ):
12     u := Q.min()
13     for(v adjacent to u):
14       if(v ∈ Q ∧ D(u) + w(u, v) < D(v)):
15         D(v) := D(u) + w(u, v)
16         a(v) := u
17       else:
18         skip
19     Q.removeMin()
```


Running Time of Dijkstra's Algorithm (2)

- When implemented using a ***heap***, the ***priority queue*** Q can perform each insertion and deletion in **O(*log n*)** time.
- Given $|V| = n$ and $|E| = m$, ***time compexity*** breaks down to:
  - **L7 – L9**: initializing ***D*** and ***a*** for all vertices [$O(n)$]
  - **L10**: n insertions to Q [$O(n \cdot log\ n)$]
    L11: `while` loop has n iterations (**L12 – L19**).
  - **L12**: retrieving the root of heap n times [$O(n)$]
    Q. How many iterations for **L14 – L18**?
    A. # adjacency edges across all vertices: $\sum_{u \in V} \mathtt{degree}(u) = m$
  - **L15**: ***upward bubbling*** to restore relational property of Q [$O(m \cdot log\ n)$]
  - **L14,L16-18**: constant operations [$O(m)$]
  - **L19**: removing min-root of heap n times [$O(n \cdot log\ n)$]
- Efficient implementation of Dijkstra Algorithm: $O((n + m) \cdot log\ n)$
- G almost ***complete*** (i.e., $m = O(n^2)$) $\Rightarrow$ RT is $O(n^2 \cdot logn)$

Graphs in Java: Adjacency Matrix (1)

- Extends the ***edge list*** structure
- Each ***vertex*** v also stores an ***integer index*** that is used to index into a 2-dimensional ***adjacency matrix***.

  $\Rightarrow$ locating an edge between two vertices takes **O(1)**

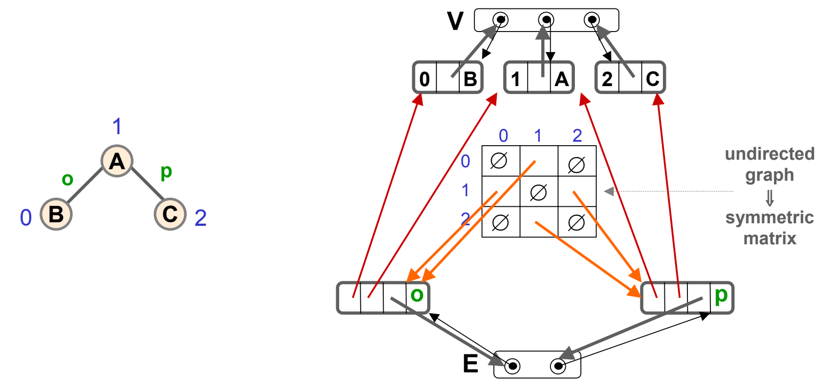
```java
class AdjacencyMatrixVertex<V> extends EdgeListVertex<V> {
  private int index;
  /* getter and setter for index */
}
```


Graphs in Java: Adjacency Matrix (2)

```java
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;
  }
```

Space Requirements? $O(|V|^2)$

Graphs in Java: Adjacency Matrix (3)



- Each ***row index*** in `matrix` represents an ***origin vertex***.
- Each ***column index*** in `matrix` represents a ***destination vertex***.
  e.g., edge (B, A): `matrix[0][1]`
- For an undirected graph, cells are **symmetric**.
  e.g., `matrix[0][1] == matrix[1][0]`

Graphs in Java: Comparing Strategies (1)

	VERTEX	EDGE	GRAPH
ADJACENCY LIST	incidentEdges	originIncidentListPos destIncidentListPos	isDirected vertices edges
EDGE LIST	vertexListPosition	edgeListPosition	isDirected vertices edges
ADJACENCY MATRIX	index	edgeListPosition	matrix

Graphs in Java: Comparing Strategies (2)

	EDGE LIST	ADJACENCY LIST	ADJACENCY MATRIX
numVertices() numEdges()	O(1)		
vertices()	O(n)		
edges()	O(m)		
getEdge(u, v)	O(m)	***O(min(d_u, d_v))***	***O(1)***
outDegree(v) inDegree(v)	O(m)	***O(d_v)***	O(n)
outgoingEdges(v) incomingEdges(v)	O(m)	***O(d_v)***	O(n)
insertVertex(x)	***O(1)***		***O(n^2)***
removeVertex(v)	O(m)	***O(d_v)***	***O(n^2)***
insertEdge(u, v, x) removeEdge(e)	O(1)		

Directed Acyclic Graph (DAG)

- ***Directed Acyclic Graph*** (***DAG***): **directed** graph with **no cycle**.
- A ***DAG*** has many applications where ***dependency*** exists between ***vertices***:
  - e.g., Prerequisites between courses of the undergrad program
  - e.g., Inheritance hierarchy among Java classes
  - e.g., Scheduling constraints between tasks
  - e.g., Dependency betwen variables in transactional updates
- In a ***DAG***, an ***edge*** (v_i, v_j) means v_i "occurs **before**" v_j
  - e.g., $(eecs2101, eecs3101)$
  - e.g., (`int x = 0`, `println(x)`)
- Given a ***DAG*** $G = (V, E)$, where $|V| = n$, a ***topological ordering*** of G is a sequence of n vertices

$$v_1, v_2, \ldots, v_n$$

such that

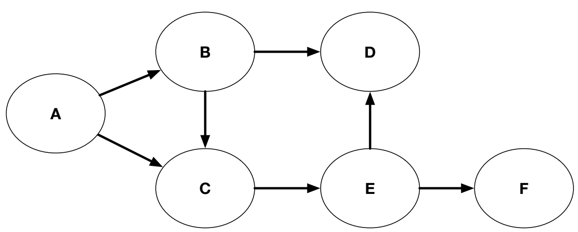
$$\forall i, j \bullet 1 \leq i, j \leq n \land (v_i, v_j) \in E \Rightarrow i < j$$

Using DFS for Topological Sort

- Given a ***DAG***, the process of computing a ***topological ordering*** of G is called performing a ***topological sort***.
- Initialize an empty ***sorted list*** of vertices.
- Repetitively perform an extended verison of **DFS**, say DFS_{topo}, until all vertices in the ***DAG*** are visited.
- Each DFS_{topo}:
  - Starts with an arbitrary unvisited vertex v
  - Returns a ***sorted list*** that corresponds to the **reverse order** in which vertices ***backtracked***.
    - When visiting v, only **push** v's ***adjacent*** vertices that are unvisited.
    - When **popping** a vertex v, add v to the front of the ***sorted list***.

    $\Rightarrow$ The ***sorted list*** contains all vertices **reachable** from v, within the current DFS run (i.e., all vertices that must "occur after" v).
  - Add the produced ***sorted list*** to the front of the list accumulated from the previous DFS_{topo}.

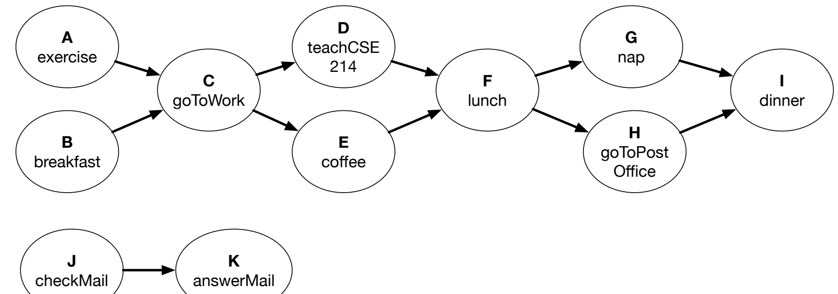
DAG: Illustration (1)



Topologically Sorted lists produced by extended DFS:

- List from $DFS_{topo}(F)$ — $\langle F \rangle$
- List from $DFS_{topo}(D)$ — $\langle D \rangle$
- List from $DFS_{topo}(E)$ — $\langle E, D, F \rangle$ or $\langle E, F, D \rangle$
- List from $DFS_{topo}(C)$ — $\langle C, E, D, F \rangle$ or $\langle C, E, F, D \rangle$
- List from $DFS_{topo}(B)$ — $\langle B, C, E, D, F \rangle$ or $\langle B, C, E, F, D \rangle$
- List from $DFS_{topo}(A)$ — $\langle A, B, C, E, D, F \rangle$ or $\langle A, B, C, E, F, D \rangle$

DAG: Illustration (2)



Possible ***topological orderings*** after a ***topological sort***:

- $\langle A, B, C, D, E, F, G, H, I, J, K \rangle$
- $\langle B, A, C, E, D, F, H, G, I, J, K \rangle$
- $\langle J, B, A, C, E, D, F, H, G, I, K \rangle$
- ...

DAG: Topological Sort in Java (1)

```
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;
}
```

DAG: Topological Sort in Java (2)

```
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();}
 }
}
```


Minimum Spanning Trees (MSTs): Problem

- ***Minimum Spanning Tree (MST) problem***:

  Given a **simple**, and **undirected**, **weighted** graph $G = (V, E)$, find a ***spanning tree*** T with the ***minimum*** total weight (over all spanning trees). More precisely:

  $$\boxed{w(T) = \sum_{(u,v) \in T} w(u,v)}$$

  is the ***minimum*** among all ***spanning trees***.
- Solving the ***MST problem*** in practice:
  - e.g., **Telecom network design**: Build a network connecting all cell towers via fiber links with the minimum installation cost or distance.
  - e.g., **Chip design**: Connect all ground pins through wiring backbone with minimal wire length or delay.

MST Problem: The Greedy Method

- Intuitively, to design using the ***Greedy Method***:
  - Build an **iterative** solution.
  - At each iteration:
    - Multiple **feasible** choices exist to keep the partial solution valid.
    - Pick the one that looks best **right now** w.r.t. a simple ***cost function***.
    - The choice made is **not** necessarily the **globally** best-looking.
    - After a choice is made, **never** undo it (i.e., **no** ***backtracking***).
- A ***cost/score function*** assigns a number:
  - to ***rank*** possible choices; and
  - to ***measure*** the **partial** solution constructed so far.

  We may either minimize the cost (e.g., smallest edge weight, smallest distance), or maximize the score (e.g., largest profit).
- A ***greedy algorithm*** builds the solution **incrementally**, always picking the ***locally-optimal choice*** (i.e., a **feasible** option with the **best** score at the moment by the ***cost function***).

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 ∈ V: C(v) := {v} -- build |V| elementary clusters
  Initialize a priority queue Q containing E -- keyed by weights
  T := ∅
  while |T| ≠ 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) ≠ C(v) then
      T := T ∪ {(u, v)}
      Merge C(u) and C(v) into one cluster
```

MST Problem: Tracing Kruskal's (1)

Apply Kruskal's algorithm on this graph:

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

edge	weight
(A, B)	1
(A, D)	3
(B, C)	2
(B, H)	10
(C, D)	3
(C, F)	6
(D, E)	5
(E, F)	4
(E, H)	8
(F, G)	7
(G, H)	9

MST Problem: Tracing Kruskal's (2)

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

MST Problem: From Clusters to Cuts

- ***Partition*** (of V) [V broken into one or more pieces]
  - A set P of non-empty, ***disjoint*** vertex sets whose **union** equals V.

$$\bigcup_{x \in P} x = V$$
$$\wedge \quad (\forall x_1, x_2 \bullet x_1 \in P \wedge x_2 \in P \wedge x_1 \neq \varnothing \wedge x_2 \neq \varnothing \wedge x_1 \neq x_2 \Rightarrow x_1 \cap x_2 = \varnothing)$$

- ***Cluster*** [one piece of the current partition]
  - In each iteration of executing Kruskal's algorithm (say $x, y \in V$):
    - A ***cluster*** is a set $C(x)$ that's a member of ***partition*** P (i.e., $C(x) \in P$) e.g., **L10** and **L11** in Kruskal's algorithm
    - $C(x) = C(y) \Rightarrow x$ and y in the **same** ***connected component***.
    - $C(x) \neq C(y) \Rightarrow x$ and y in **different** ***connected components***.
- ***Cut*** (of V) [V broken into two pieces]
  - A ***partition*** of V into **two** non-empty, ***disjoint*** sets:

$$P = \{S, V \setminus S\}$$

  - An edge ***crosses the cut*** with its endpoints in S and $V \setminus S$.

MST Problem: Cut Property of Safe Edges

Cut Property. Given:
- $G = (V, E)$ a weighted, ***connected*** graph
- Any ***cut*** $\{S, V \setminus S\}$ of the vertices
- e a ***minimum-weight edge*** among all edges that ***cross this cut***

$\Rightarrow$ There exists **an** ***MST*** that contains e.

We say: e is a ***safe edge*** for some ***MST***.

In Kruskal's algorithm:
- **L10**: ***Cluster*** $C(u)$ helps form a ***cut***: $\{C(u), V \setminus C(u)\}$
  - $C(u) \neq \varnothing$
  - $V \setminus C(u) \neq \varnothing$ [i.e., not in the **last** iteration]
- **L12**: $C(u) \neq C(v)$ means $C(u) \cap C(v) = \varnothing$ and $C(v) \subseteq V \setminus C(u)$.
  - Recall: Edges extracted from Q in a non-decreasing order on weights
  - ***Clusters*** only **merge**; never split.
  - Any edge (x, y) with $x \in C(u)$ and $w(x, y) < w(u, v)$ would've been processed (as a **tree** edge or as an **internal** edge), so now $x, y \in C(u)$.

    $\Rightarrow \nexists$ edge "cheaper" than (u, v) ***crossing the cut*** $\{C(u), V \setminus C(u)\}$

  $\therefore (u, v)$ from **L9** is a ***min-weight edge*** ***crossing the cut***
- By the ***Cut Property***, **L13** is justified:
  - (u, v) is a ***safe edge***: $\exists$ some ***MST*** containing $T \cup \{(u, v)\}$
  - Adding (u, v) keeps us **on the right track** to reach some ***MST***.

Index (1)

Index (2)

Index (3)

Index (4)

Index (5)

Index (6)

Index (7)

Index (8)

Priority Queues ADT and Heaps

EECS3101 E:
Design and Analysis of Algorithms
Fall 2025

CHEN-WEI WANG

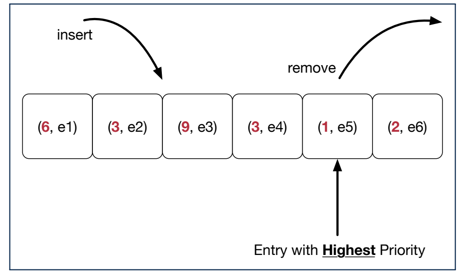
Learning Outcomes of this Lecture

This module is designed to help you understand:

- The ***Priority Queue*** (***PQ***) ADT
- The ***Heap*** Data Structure (Properties & Operations)
- Time Complexities of ***Heap***-Based ***PQ***

What is a Priority Queue?

- A ***Priority Queue (PQ)*** stores a collection of ***entries***.



- Each ***entry*** is a pair: an ***element*** and its ***key***.
- The ***key*** of each ***entry*** denotes its ***element***'s "priority".
- ***Keys*** in a Priority Queue (PQ) are **not** used for uniquely identifying an entry.

- In a PQ, the next entry to remove has the "***highest***" priority.
  - e.g., In the stand-by queue of a fully-booked flight, ***frequent flyers*** get the higher priority to replace any cancelled seats.
  - e.g., A network router, faced with insufficient bandwidth, may only handle ***real-time tasks*** (e.g., streaming) with highest priorities.
  - e.g., When performing Dijkstra's ***shortest path algorithm*** on a weighted graph, the vertex with the minimum *D* value gets the highest priority to be visited next.

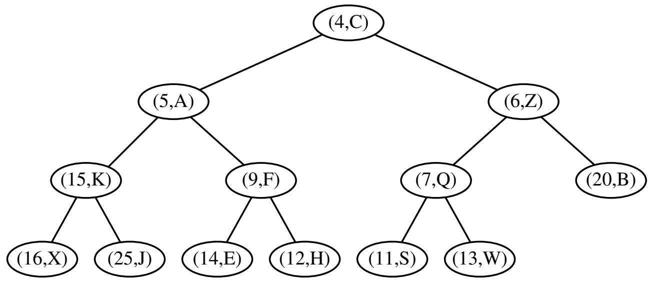
The Priority Queue (PQ) ADT

- ***min***
  [***precondition***: PQ is not empty]
  [***postcondition***: return entry with **highest priority** in PQ]
- ***size***
  [***precondition***: **none**]
  [***postcondition***: return number of entries inserted to PQ]
- ***isEmpty***
  [***precondition***: **none**]
  [***postcondition***: return whether there is no entry in PQ]
- ***insert(k, v)***
  [***precondition***: PQ is not full]
  [***postcondition***: insert the input entry into PQ]
- ***removeMin***
  [***precondition***: PQ is not empty]
  [***postcondition***: remove and return a **min** entry in PQ]

Heaps

A **heap** is a ***binary tree*** which:

1. Stores in each node an ***entry*** (i.e., ***key*** and ***value***).

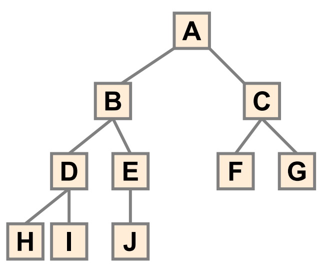


2. Satisfies a ***structural*** property of tree **organization**
3. Satisfies a ***relational*** property of stored **keys**

BT Terminology: Complete BTs

A *binary tree* with ***height*** h is considered as ***complete*** if:

- Nodes with ***depth*** $\leq h - 2$ has two children.
- Nodes with ***depth*** $h - 1$ may have zero, one, or two child nodes.
- ***Children*** of nodes with ***depth*** $h - 1$ are filled from left to right.



Q1: ***Minimum*** # of nodes of a ***complete*** BT? $(2^h - 1) + 1 = 2^h$

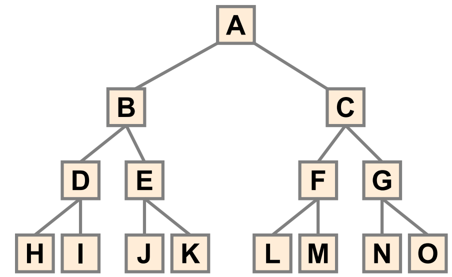
Q2: ***Maximum*** # of nodes of a ***complete*** BT? $2^{h+1} - 1$

BT Terminology: Full BTs

A *binary tree* with ***height*** h is considered as ***full*** if:

Each node with ***depth*** $\leq h - 1$ has two child nodes.

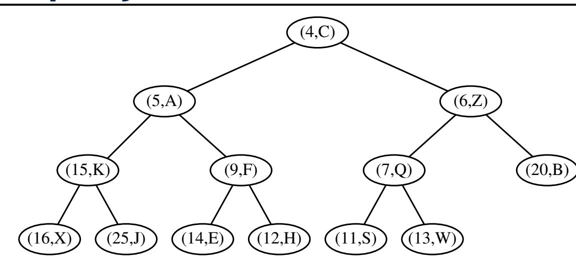
That is, all ***leaves*** are with the same ***depth*** h.



Q1: ***Minimum*** # of nodes of a complete BT? $2^{h+1} - 1$

Q2: ***Maximum*** # of nodes of a complete BT? $2^{h+1} - 1$

Heap Property 1: Structural



A ***heap*** with ***height h*** satisfies the ***Complete BT Property***:

- Nodes with ***depth*** $\leq$ **h – 2** has two child nodes.
- Nodes with ***depth h - 1*** may have zero, one, or two child nodes.
- Nodes with ***depth h*** are filled from left to right.

Q. When the # of nodes is n, what is h? $\lfloor \log_2 n \rfloor$

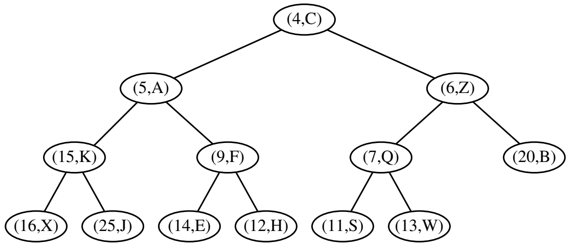
Q. # of nodes from Level 0 through Level $h - 1$? $2^h - 1$

Q. # of nodes at Level h? $n - (2^h - 1)$

Q. **Minimum** # of nodes of a complete BT? 2^h

Q. **Maximum** # of nodes of a complete BT? $2^{h+1} - 1$

Heap Property 2: Relational



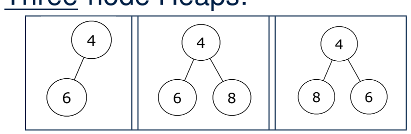
Keys in a **heap** satisfy the ***Heap-Order Property*** :

- Every node n (other than the root) is s.t. $key(n) \geq key(\textbf{parent}(n))$
  - ⇒ ***Keys*** in a ***root-to-leaf path*** are sorted in a non-descending order.
    e.g., Keys in entry path $\langle (4, C), (5, A), (9, F), (14, E) \rangle$ are sorted.
  - ⇒ The ***minimal key*** is stored in the ***root***.
    e.g., Root $(4, C)$ stores the minimal key 4.
- ***Keys*** of nodes from ***different subtrees*** are **not** constrained at all.
  e.g., For node $(5, A)$, key of its ***LST***'s root (15) is not minimal for its ***RST***.

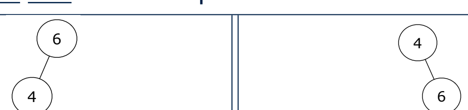
Heaps: More Examples

- The ***smallest*** *heap* is just an empty binary tree.
- The ***smallest*** **non-empty** *heap* is a one-node heap.
  e.g.,

- Two-node and Three-node Heaps:



- These are **not** two-node heaps:



relational property violated | ***structural*** property violated

Heap Operations

- There are three main operations for a ***heap*** :
  1. **Extract the Entry with Minimal Key**:
     Return the stored entry of the ***root***. [***O(1)***]
  2. **Insert a New Entry**:
     A single ***root-to-leaf path*** is affected. [***O(h)*** or ***O(log n)***]
  3. **Delete the Entry with Minimal Key**:
     A single ***root-to-leaf path*** is affected. [***O(h)*** or ***O(log n)***]
- After performing each operation,
  both ***relational*** and ***structural*** properties must be maintained.

Updating a Heap: Insertion

To insert a new entry (k, v) into a heap with ***height h***:

1. Insert (k, v), possibly **temporarily** breaking the ***relational property***.
   - **1.1** Create a new entry $\mathbf{e} = (k, v)$.
   - **1.2** Create a new ***right-most*** node ***n*** at ***Level h***.
   - **1.3** Store entry **e** in node ***n***.
     After steps **1.1** and **1.2**, the ***structural property*** is maintained.
2. Restore the **heap-order property** (**HOP**) using *Up-Heap Bubbling* :
   - **2.1** Let $c = n$.
   - **2.2** While **HOP** is not restored and c is not the root:
     - **2.2.1** Let p be c's parent.
     - **2.2.2** If $key(p) \leq key(c)$, then **HOP** is restored.
       Else, swap nodes c and p. ["upwards" along n's ***ancestor path***]

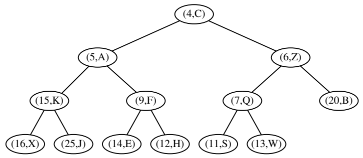
Running Time?

- All sub-steps in **1**, as well as steps **2.1**, **2.2.1**, and **2.2.2** take ***O(1)***.
- Step **2.2** may be executed up to ***O(h)*** (or ***O(log n)***) times.

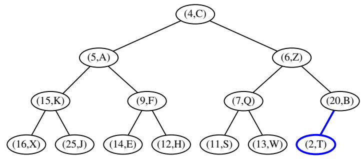
[*O(log n)*]

Updating a Heap: Insertion Example (1.1)

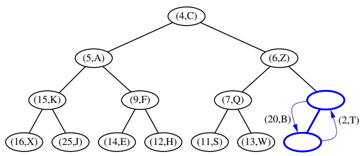
(0) A heap with height 3.



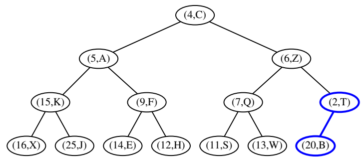
(1) Insert a new entry $(2, T)$ as the ***right-most*** node at Level 3. Perform *up-heap bubbling* from here.



(2) **HOP** violated $\because 2 < 20 \therefore$ Swap.

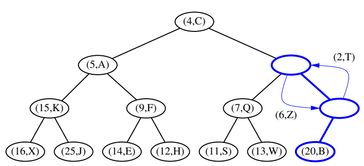


(3) After swap, entry $(2, T)$ prompted up.

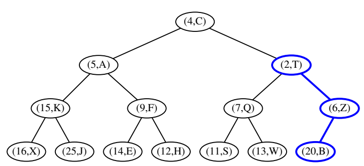


Updating a Heap: Insertion Example (1.2)

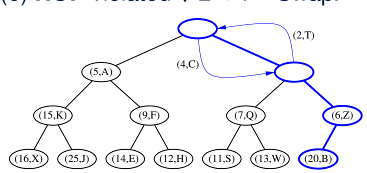
(4) **HOP** violated $\because 2 < 6 \therefore$ Swap.



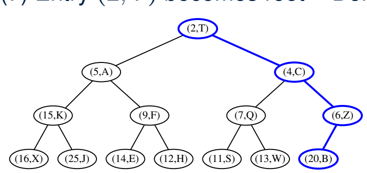
(5) After swap, entry $(2, T)$ prompted up.



(6) **HOP** violated $\because 2 < 4 \therefore$ Swap.



(7) Entry $(2, T)$ becomes root $\therefore$ Done.



Updating a Heap: Deletion

To delete the **root** (with the ***minimal*** key) from a heap with ***height h***:

1. Delete the **root**, possibly <u>**temporarily**</u> breaking **HOP**.
   - **1.1** Let the ***right-most*** node at ***Level h*** be ***n***.
   - **1.2** Replace the **root**'s entry by ***n***'s entry.
   - **1.3** Delete ***n***.

     After steps **1.1** – **1.3**, the ***structural property*** is maintained.
2. Restore **HOP** using *Down-Heap Bubbling*:
   - **2.1** Let ***p*** be the **root**.
   - **2.2** While **HOP** is not restored <u>and</u> ***p*** is <u>not</u> external:
     - **2.2.1** <u>**IF**</u> ***p*** has <u>no</u> **right child**, let ***c*** be ***p***'s ***left child***.

       <u>**Else**</u>, let ***c*** be ***p***'s child with a ***smaller key value***.
     - **2.2.2** <u>**If**</u> $key(p) \leq key(c)$, then **HOP** is restored.

       <u>**Else**</u>, swap nodes ***p*** and ***c***. ["downwards" along a ***root-to-leaf path***]

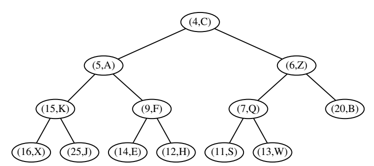
<u>Running Time?</u>

- All sub-steps in **1**, as well as steps **2.1**, **2.2.1**, and **2.2.2** take ***O(1)***.
- Step **2.2** may be executed up to ***O(h)*** (or ***O(log n)***) times.

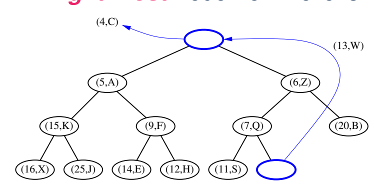
[*O(log n)*]

Updating a Heap: Deletion Example (1.1)

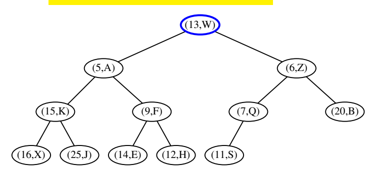
(0) Start with a heap with height 3.



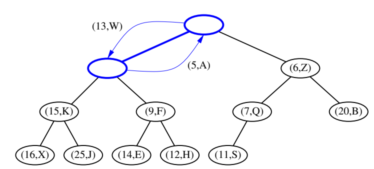
(1) Replace root with $(13, W)$ and delete ***right-most*** node from Level 3.



(2) $(13, W)$ becomes the root. Perform *down-heap bubbling* from here.

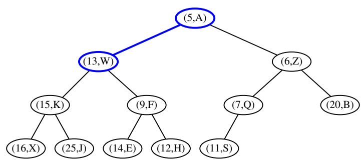


(3) Child with smaller key is $(5, A)$. **HOP** violated $\because 13 > 5 \therefore$ Swap.

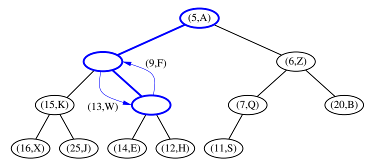


Updating a Heap: Deletion Example (1.2)

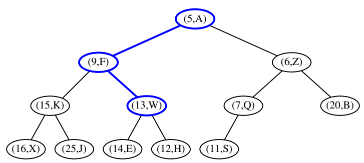
(4) After swap, entry $(13, W)$ demoted down.



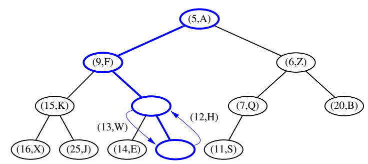
(5) Child with smaller key is $(9, F)$. **HOP** violated $\because 13 > 9 \therefore$ Swap.



(6) After swap, entry $(13, W)$ demoted down.

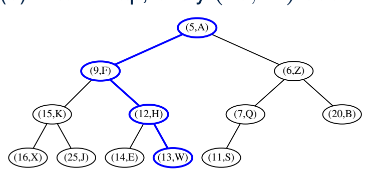


(7) Child with smaller key is $(12, H)$. **HOP** violated $\because 13 > 12 \therefore$ Swap.



Updating a Heap: Deletion Example (1.3)

(8) After swap, entry $(13, W)$ becomes an external node $\therefore$ Done.



Heap-Based Implementation of a PQ

PQ Method	Heap Operation	RT
min	root	***O(1)***
insert	insert then up-heap bubbling	***O(log n)***
removeMin	delete then down-heap bubbling	***O(log n)***

Index (1)

Index (2)