Asymptotic Analysis of Algorithms

EECS2011 N & Z:
Fundamentals of Data Structures
Winter 2022

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 $\langle a_1, a_2, \ldots, a_n \rangle$
  - **Output:** A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$
  - 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*
- Steps in an *algorithm* manipulate well-chosen *data structure(s)*.

Measuring “Goodness” of an Algorithm

1. *Correctness* :
   - Does the algorithm produce the expected output?
   - Use 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 that are meant to be run on a computer should run *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
- *How do you determine the running time of an algorithm?*
  1. Measure time via *experiments*
  2. Characterize time as a *mathematical function* of the input size

Measure Running Time via Experiments

- Once the algorithm is implemented in Java:
  - Execute the program on *test inputs* of various *sizes* and *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 input tests must be “reasonably” *complete*.

Example Experiment

- *Computational Problem*:
  - **Input**: A character *c* and an integer *n*
  - **Output**: A string consisting of *n* repetitions of character *c*
    e.g., Given input `'*'` and `15`, output `***************`.
- *Algorithm 1* using *String* Concatenations:

```
public static String repeat1(char c, int n) {
  String answer = "";
  for (int i = 0; i < n; i ++) { answer += c; }
  return answer; }
```

- *Algorithm 2* using *StringBuilder* `append`’s:

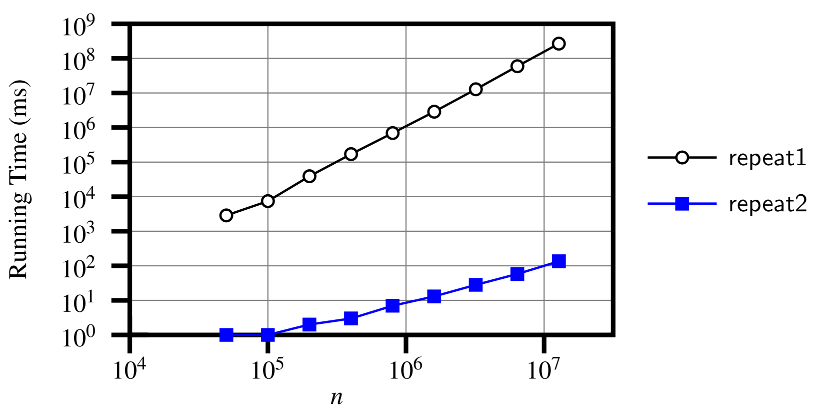
```
public static String repeat2(char c, int n) {
  StringBuilder sb = new StringBuilder();
  for (int i = 0; i < n; i ++) { sb.append(c); }
  return sb.toString(); }
```

Example Experiment: Detailed Statistics

n	`repeat1` (in ms)	`repeat2` (in ms)
50,000	2,884	1
100,000	7,437	1
200,000	39,158	2
400,000	170,173	3
800,000	690,836	7
1,600,000	2,847,968	13
3,200,000	12,809,631	28
6,400,000	59,594,275	58
12,800,000	265,696,421 (≈ 3 days)	135

- As *input size* is doubled, *rates of increase* for both algorithms are *linear*:
  - *Running time* of `repeat1` increases by ≈ 5 times.
  - *Running time* of `repeat2` increases by ≈ 2 times.

Example Experiment: Visualization



Experimental Analysis: Challenges

1. An algorithm must be *fully implemented* (i.e., translated into valid Java syntax) 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 environment of:
   - *Hardware*: CPU, running processes
   - *Software*: OS, JVM version
3. Experiments can be done only on *a limited set of test inputs*.
   - What if "*important*" inputs were not included in the experiments?

Moving Beyond Experimental Analysis

- A better approach to analyzing the **efficiency** (e.g., *running times*) of algorithms should be one that:
  - Allows us to calculate the *relative efficiency* (rather than absolute elapsed time) of algorithms in a ways that is ***independent of*** the hardware and software environment.
  - Can be applied using a *high-level description* of the algorithm (without fully implementing it).
  - 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 the most running time)

Counting Primitive Operations

A ***primitive operation*** corresponds to a low-level instruction with a *constant execution time*.

- 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: Why is a method call in general **not** a primitive operation?
A: 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)

Example: Counting Primitive Operations (1)

```
1 int findMax (int[] a, int n) {
2   currentMax = a[0];
3   for (int i = 1; i < n; ) {
4     if (a[i] > currentMax) {
5       currentMax = a[i]; }
6     i ++ }
7   return currentMax; }
```

\# of times `i < n` in **Line 3** is executed? [n]
\# of times the loop body (**Line 4** to **Line 6**) is executed? [$n-1$]

- **Line 2**: 2 [1 indexing + 1 assignment]
- **Line 3**: $n+1$ [1 assignment + n comparisons]
- **Line 4**: $(n-1)\cdot 2$ [1 indexing + 1 comparison]
- **Line 5**: $(n-1)\cdot 2$ [1 indexing + 1 assignment]
- **Line 6**: $(n-1)\cdot 2$ [1 addition + 1 assignment]
- **Line 7**: 1 [1 return]
- **Total # of Primitive Operations**: 7n - 2

Example: Counting Primitive Operations (2)

Count the number of primitive operations for

```
boolean foundEmptyString = false;
int i = 0;
while (!foundEmptyString && i < names.length) {
  if (names[i].length() == 0) {
    /* set flag for early exit */
    foundEmptyString = true;
  }
  i = i + 1;
}
```

- \# times the stay condition of the `while` loop is checked?
  [between 1 and `names.length` + 1]
  [***worst case***: `names.length` + 1 times]
- \# times the body code of `while` loop is executed?
  [between 0 and `names.length`]
  [***worst case***: `names.length` times]

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*.
- The *number of primitive operations* required by an algorithm should be *proportional* to its *actual running time* on a specific environment.
  e.g., `findMax (int[] a, int n)` has $7n-2$ POs
  $$RT = (7n - 2) \cdot t$$
  Say two algorithms with RT $(7n - 2) \cdot t$ and RT $(10n + 3) \cdot t$.
  ⇒ 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 (considering that input size may be very large):
  - **Only** the ***highest power*** matters.
  - ***multiplicative constants*** and ***lower terms*** can be dropped.

  $\Rightarrow$ $7n - 2$ is approximately n

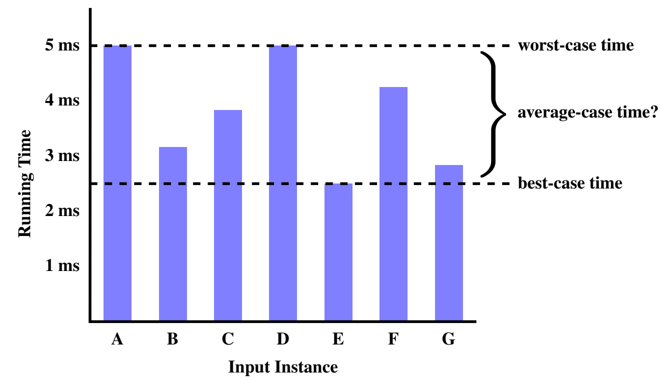
  Exercise: Consider $7n + 2n \cdot log\ n + 3n^2$:
  - ***highest power***? [n^2]
  - ***multiplicative constants***? [7, 2, 3]
  - ***lower terms***? [$7n + 2n \cdot log\ n$]

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]

Focusing on the Worst-Case Input



- *Average-case* analysis calculates the *expected running times* 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 in the limit*:
  - How the *running time* of the algorithm under analysis changes as the *input size* changes without bound
  - e.g., contrast $RT_1(n) = n$ with $RT_2(n) = n^2$
- 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

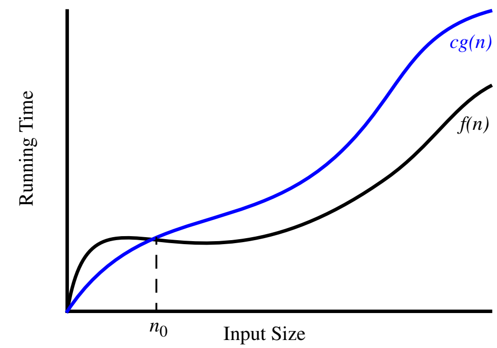
- Let $f(n)$ and $g(n)$ be functions mapping positive integers (input size) to positive real numbers (running time).
  - $f(n)$ characterizes the running time of some algorithm.
  - $O(g(n))$:
    - denotes *a collection of* functions
    - consists of *all* functions that can be upper bounded by $g(n)$, 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: Example (1)

Prove: The function $8n + 5$ is $O(n)$.
Strategy: Choose a real constant $c > 0$ and an integer constant $n_0 \geq 1$, such that for every integer $n \geq n_0$:

$$8n + 5 \leq c \cdot n$$

Can we choose $c = 9$? What should the corresponding n_0 be?

n	8n + 5	9n
1	13	9
2	21	18
3	29	27
4	37	36
5	45	45
6	53	54

...

Therefore, we prove it by choosing $c = 9$ and $n_0 = 5$.
We may also prove it by choosing $c = 13$ and $n_0 = 1$. Why?

Asymptotic Upper Bound: Example (2)

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 $n \geq n_0$:

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

$f(1) = 5 + 3 + 2 + 4 + 1 = 15$

Choose $c = 15$ and $n_0 = 1$!

Asymptotic Upper Bound: Proposition (1)

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

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

- 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: Proposition (2)

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

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:

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

e.g., Family of $O(n^2)$ contains:

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

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*.

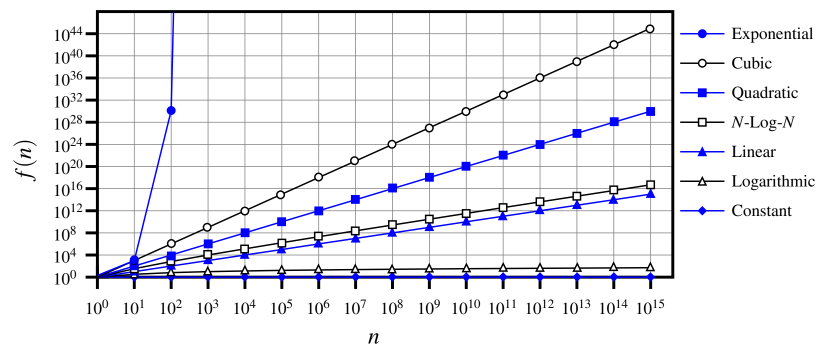
  For example, say $f(n) = 4n^3 + 3n^2 + 5$:
  - Recall: $O(n^3) \subset O(n^4) \subset O(n^5) \subset \ldots$
  - It is the *most accurate* to say that $f(n)$ is $O(n^3)$.
  - It is ***true***, but not very useful, to say that $f(n)$ is $O(n^4)$ and that $f(n)$ is $O(n^5)$.
  - It is ***false*** to say 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*

Rates of Growth: Comparison



Upper Bound of Algorithm: Example (1)

```
int maxOf (int x, int y) {
  int max = x;
  if (y > x) {
    max = y;
  }
  return max;
}
```

- # of primitive operations: 4

  2 assignments + 1 comparison + 1 return = 4
- Therefore, the running time is $O(1)$.
- That is, this is a *constant-time* algorithm.

Upper Bound of Algorithm: Example (2)

```
int findMax (int[] a, int n) {
  currentMax = a[0];
  for (int i = 1; i < n; ) {
    if (a[i] > currentMax) {
      currentMax = a[i]; }
    i ++ }
  return currentMax; }
```

- From last lecture, we calculated that the # of primitive operations is $7n - 2$.
- Therefore, the running time is $O(n)$.
- That is, this is a *linear-time* algorithm.

Upper Bound of Algorithm: Example (3)

```
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 (4)

```
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 (5)

```
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.

Beyond this lecture ...

- 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]

Index (1)

Index (2)

Index (3)

Index (4)

Background Study: Generics in Java

- It is assumed that, in EECS2030, you learned about the basics of Java ***generics***:
  - General collection (e.g., `Object[]`) vs. Generic collection (e.g., `E[]`)
  - How using generics minimizes ***casts*** and ***`instanceof` checks***
  - How to implement and use generic classes
- If needed, review the above assumed basics from the relevant parts of EECS2030 (`https://www.eecs.yorku.ca/~jackie/teaching/lectures/index.html#EECS2030_F21`):
  - Parts A1 – A3, Lecture 7, Week 10
  - Parts B – C, Lecture 7, Week 11

  Tips.
  - Skim the ***slides***: watch lecture videos if needing explanations.
  - Ask questions related to the assumed basics of ***generics***!
- Assuming that know the basics of Java ***generics***, we will implement and use ***generic SLL*** and ***DLL***.

Basic Data Structures: Arrays vs. Linked-Lists

EECS2011 N & Z:
Fundamentals of Data Structures
Winter 2022

CHEN-WEI WANG

Learning Outcomes of this Lecture

This module is designed to help you learn about:

- ***basic data structures***: ***Arrays*** vs. ***Linked Lists***
- Two ***Sorting*** Algorithms: Selection Sort vs. Insertion Sort
- ***Linked Lists***: **Singly**-Linked vs. **Doubly**-Linked
- ***Running Time***: Array vs. Linked-List Operations
- Java ***Implementations***: **`String`** Lists vs. ***Generic*** Lists

Basic Data Structure: Arrays

- An array is a sequence of indexed elements.
- ***Size*** of an array is **fixed** at the time of its construction.
  - e.g., `int[] numbers = new int[10];`
  - **Heads-Up**. Two ***resizing*** strategies: ***increments*** vs. ***doubling***.
- Supported operations on an array:
  - ***Accessing***: e.g., `int max = a[0];`
    Time Complexity: *O(1)* [constant-time op.]
  - ***Updating***: e.g., `a[i] = a[i + 1];`
    Time Complexity: *O(1)* [constant-time op.]
  - ***Inserting/Removing***:

```
String[] insertAt(String[] a, int n, String e, int i)
  String[] result = new String[n + 1];
  for(int j = 0; j <= i - 1; j ++){ result[j] = a[j]; }
  result[i] = e;
  for(int j = i + 1; j <= n; j ++){ result[j] = a[j-1]; }
  return result;
```

Time Complexity: *O(n)* [linear-time op.]

Array Case Study: Comparing Two Sorting Strategies

- ***The Sorting Problem***:

  Input: An array a of n numbers $\langle a_1, a_2, \dots, a_n \rangle$ (e.g., $\langle 3, 4, 1, 3, 2 \rangle$)

  Output: A permutation/reordering $\langle a'_1, a'_2, \dots, a'_n \rangle$ of the input sequence s.t. elements are arranged in a ***non-descending*** order (e.g., $\langle 1, 2, 3, 3, 4 \rangle$): $a'_1 \le a'_2 \le \dots \le a'_n$

  Remark. Variants of the ***sorting problem*** may require different ***orderings***:
  - non-descending
  - ascending/increasing
  - non-ascending
  - descending/decreasing
- Two alternative ***implementation strategies*** for solving this problem
- At the end, choose one based on their ***time complexities***.

Sorting: Strategy 1 – Selection Sort

- Maintain a (initially empty) ***sorted portion*** of array *a*.
- From left to right in array *a*, select and insert the ***minimum*** element to the ***end*** of this sorted portion, so it remains sorted.

```
1 void selectionSort(int[] a, int n)
2   for (int i = 0; i <= (n - 2); i ++)
3     int minIndex = i;
4     for (int j = i; j <= (n - 1); j ++)
5       if (a[j] < a[minIndex]) { minIndex = j; }
6     int temp = a[i];
7     a[i] = a[minIndex];
8     a[minIndex] = temp;
```

- How many times does the body of **for-loop** (L4) run? [(n - 1)]
- Running time? [$O(n^2)$]

$$\underbrace{n}_{\text{find } \{a[0], \dots, a[n-1]\}} + \underbrace{(n-1)}_{\text{find } \{a[1], \dots, a[n-1]\}} + \dots + \underbrace{2}_{\text{find } \{a[n-2], a[a[n-1]]\}}$$

- So ***selection sort*** is a ***quadratic-time algorithm***.

Sorting: Strategy 2 – Insertion Sort

- Maintain a (initially empty) ***sorted portion*** of array *a*.
- From left to right in array *a*, insert ***one element*** at a time into the ***"correct" spot*** in this sorted portion, so it remains sorted.

```
1 void insertionSort(int[] a, int n)
2   for (int i = 1; i < n; i ++)
3     int current = a[i];
4     int j = i;
5     while (j > 0 && a[j - 1] > current)
6       a[j] = a[j - 1];
7       j --;
8     a[j] = current;
```

- **while-loop** (L5) exits when? [`j <= 0` or `a[j - 1] <= current`]
- Running time? [$O(n^2)$]

$$O(\underbrace{1}_{\text{insert into } \{a[0]\}} + \underbrace{2}_{\text{insert into } \{a[0], a[1]\}} + \dots + \underbrace{(n-1)}_{\text{insert into } \{a[0], \dots, a[n-2]\}})$$

- So ***insertion sort*** is a ***quadratic-time algorithm***.

Sorting: Alternative Implementations?

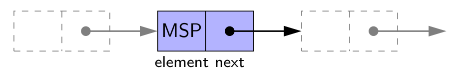
- In the Java implementations of ***selection sort*** and ***insertion sort***, we maintain the ***"sorted portion"*** from the ***left*** end.
  - For ***selection sort***, we select the ***minimum*** element from the ***"unsorted portion"*** and insert it to the ***end*** of the ***"sorted portion"***.
  - For ***insertion sort***, we choose the ***left-most*** element from the ***"unsorted portion"*** and insert it at the ***"correct spot"*** in the ***"sorted portion"***.
- **Exercise:** Modify the Java implementations, so that the ***"sorted portion"*** is:
  - arranged in a ***non-ascending*** order (e.g., $\langle 5, 4, 3, 2, 1 \rangle$); and
  - maintained and grown from the ***right*** end instead.

Comparing Insertion & Selection Sorts

- *Asymptotically*, running times of ***selection sort*** and ***insertion sort*** are both $O(n^2)$.
- We will later see that there exist better algorithms that can perform better than quadratic: $O(n \cdot logn)$.

Basic Data Structure: Singly-Linked Lists

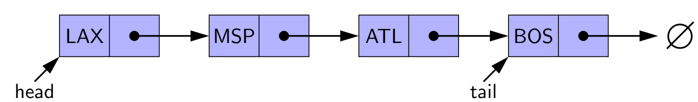
- We know that ***arrays*** perform:
  - ***well*** in indexing
  - ***badly*** in inserting and deleting
- We now introduce an alternative data structure to arrays.
- A ***linked list*** is a series of **connected** ***nodes***, forming a ***linear sequence***.
  Remark. At ***runtime***, node ***connections*** are through ***reference aliasing***.
- Each ***node*** in a *singly-linked list (SLL)* stores:
  - ***reference*** to a ***data object***; and
  - ***reference*** to the ***next node*** in the list.
  Contrast. *relative* positioning of LL vs. **absolute** indexing of arrays



- The ***last node*** in a singly-linked list is different from others. How so? Its reference to the ***next node*** is simply `null`.

Singly-Linked List: How to Keep Track?

- Due to its ***"chained" structure***, a SLL, when first being created, does **not** need to be specified with a **fixed length**.
- We can use a SLL to ***dynamically*** store and manipulate as many elements as we desire **without** the need to ***resize*** by:
  - e.g., ***creating*** a new node and setting the relevant ***references***.
  - e.g., ***inserting*** some node to the ***beginning***/***middle***/***end*** of a SLL
  - e.g., ***deleting*** some node from the ***beginning***/***middle***/***end*** of a SLL
- **Contrary to arrays**, we do **not** keep track of all nodes in a SLL **directly** by indexing the ***nodes***.
- Instead, we only store a ***reference*** to the ***head*** (i.e., ***first node***), and find other parts of the list ***indirectly***.



- **Exercise**: Given the ***head*** reference of a SLL, describe how we may:
  - Count the number of nodes currently in the list. [Running Time?]
  - Find the reference to its ***tail*** (i.e., ***last node***) [Running Time?]

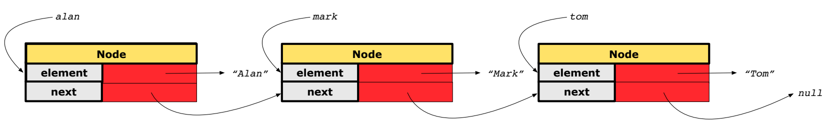
Singly-Linked List: Java Implementation

We first implement a ***SLL*** storing strings only.

```
public class Node {
  private String element;
  private Node next;
  public Node(String e, Node n) { element = e; next = n; }
  public String getElement() { return element; }
  public void setElement(String e) { element = e; }
  public Node getNext() { return next; }
  public void setNext(Node n) { next = n; }
}
```

```
public class SinglyLinkedList {
  private Node head;
  public void setHead(Node n) { head = n; }
  public int getSize() { ... }
  public Node getTail() { ... }
  public void addFirst(String e) { ... }
  public Node getNodeAt(int i) { ... }
  public void addAt(int i, String e) { ... }
  public void removeLast() { ... }
}
```


Singly-Linked List: Constructing a Chain of Nodes



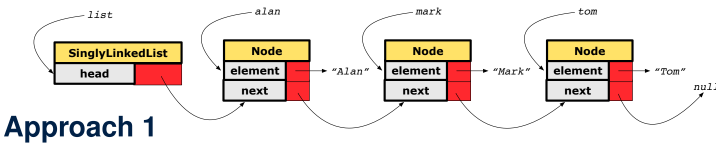
Approach 1

```
Node tom = new Node("Tom", null);
Node mark = new Node("Mark", tom);
Node alan = new Node("Alan", mark);
```

Approach 2

```
Node alan = new Node("Alan", null);
Node mark = new Node("Mark", null);
Node tom = new Node("Tom", null);
alan.setNext(mark);
mark.setNext(tom);
```


Singly-Linked List: Setting a List's Head



Approach 1

```
Node tom = new Node("Tom", null);
Node mark = new Node("Mark", tom);
Node alan = new Node("Alan", mark);
SinglyLinkedList list = new SinglyLinkedList();
list.setHead(alan);
```

Approach 2

```
Node alan = new Node("Alan", null);
Node mark = new Node("Mark", null);
Node tom = new Node("Tom", null);
alan.setNext(mark);
mark.setNext(tom);
SinglyLinkedList list = new SinglyLinkedList();
list.setHead(alan);
```


Singly-Linked List: Counting # of Nodes (1)

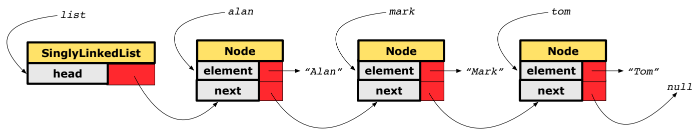
Problem: Return the number of nodes currently stored in a SLL.

- **Hint**. Only the ***last node*** has a ***null next*** reference.
- Assume we are in the context of class `SinglyLinkedList`.

```
1 int getSize() {
2   int size = 0;
3   Node current = head;
4   while (current != null) {
5     current = current.getNext();
6     size ++;
7   }
8   return size;
9 }
```

- When does the while-loop (L4) exit? [`current == null`]
- RT of `getSize`: *O(n)* [linear-time op.]
- **Contrast**: RT of `a.length`: *O(1)* [constant-time op.]

Singly-Linked List: Counting # of Nodes (2)




```
int getSize() {
  int size = 0;
  Node current = head;
  while (current != null) { /* exit when current == null */
    current = current.getNext();
    size ++;
  }
  return size;
}
```

Let's now consider `list.getSize()`:

`current`	`current != null`	End of Iteration	`size`
alan	*true*	1	1
mark	*true*	2	2
tom	*true*	3	*3*
`null`	*false*	–	–

Singly-Linked List: Finding the Tail (1)

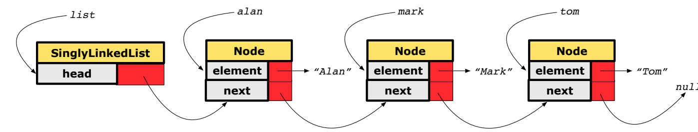
Problem: Retrieved the tail (i.e., last node) in a SLL.

- **Hint**. Only the ***last node*** has a ***null next*** reference.
- Assume we are in the context of class `SinglyLinkedList`.

```
1 Node getTail() {
2   Node current = head;
3   Node tail = null;
4   while (current != null) {
5     tail = current;
6     current = current.getNext();
7   }
8   return tail;
9 }
```

- When does the while-loop (L4) exit? [`current == null`]
- RT of `getTail`: *O(n)* [linear-time op.]
- **Contrast**: RT of `a[a.length - 1]`: *O(1)* [constant-time op.]

Singly-Linked List: Finding the Tail (2)




```
Node getTail() {
  Node current = head;
  Node tail = null;
  while (current != null) { /* exit when current == null */
    tail = current;
    current = current.getNext();
  }
  return tail;
}
```

Let's now consider `list.getTail()`:

`current`	`current != null`	End of Iteration	`tail`
alan	*true*	1	alan
mark	*true*	2	mark
tom	*true*	3	*tom*
`null`	*false*	–	–

Singly-Linked List: Can We Do Better?

- In practice, we may frequently need to:
  - Access the ***tail*** of a list. [e.g., customers joining a service queue]
  - Inquire the ***size*** of a list. [e.g., the service queue full?]

  Both operations cost $O(n)$ to run (with only ***head*** available).
- We may improve the ***RT*** of these two operations.

  Principle. Trade ***space*** for ***time***.

  - Declare a new attribute ***tail*** pointing to the end of the list.
  - Declare a new attribute ***size*** denoting the number of stored nodes.
  - ***RT*** of these operations, accessing attribute values, are $O(1)$!
- Why not declare attributes to store references of ***all nodes*** between ***head*** and ***tail*** (e.g., `secondNode`, `thirdNode`)?
  - No – at the ***time of declarations***, we simply do **not** know how many nodes there will be at ***runtime***.

Singly-Linked List: Inserting to the Front (1)

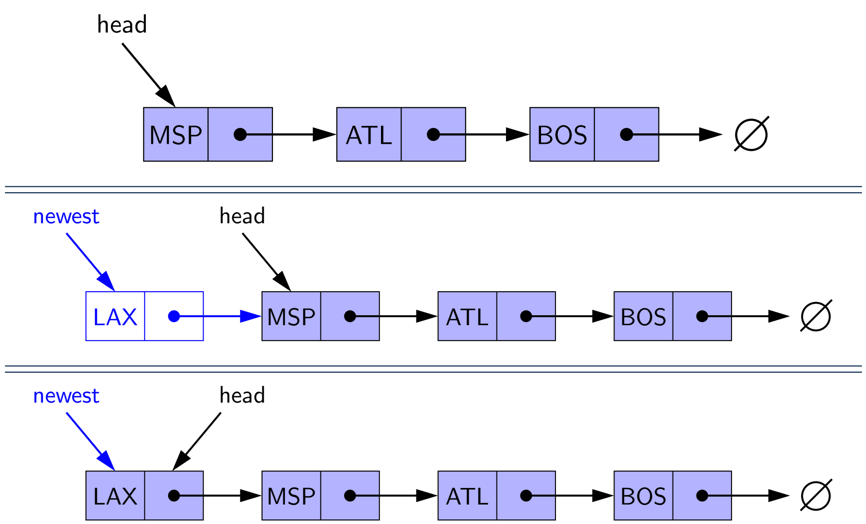
Problem: Insert a new string *e* to the front of the list.

- **Hint.** The list's new head should store *e* and point to the old head.
- Assume we are in the context of class `SinglyLinkedList`.

```
void addFirst (String e) {
  head = new Node(e, head);
  if (size == 0) {
    tail = head;
  }
  size ++;
}
```

- Remember that RT of accessing *head* or *tail* is $O(1)$
- RT of `addFirst` is *O(1)* [constant-time op.]
- **Contrast**: Inserting into an array costs *O(n)* [linear-time op.]

Singly-Linked List: Inserting to the Front (2)



Exercise

See `ExampleStringLinkedLists.zip`.

Compare and contrast two alternative ways to constructing a SLL: `testSLL_01` vs. `testSLL_02`.

Exercise

- Complete the Java ***implementations***, ***tests***, and ***running time analysis*** for:
  - `void removeFirst()`
  - `void addLast(String e)`
- **Question:** The `removeLast()` method may not be completed in the same way as is `void addLast(String e)`. Why?

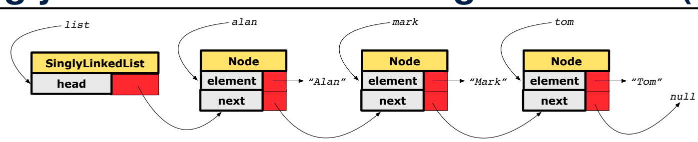
Singly-Linked List: Accessing the Middle (1)

Problem: Return the node at index *i* in the list.

- **Hint**. $0 \leq i <$ `list.getSize()`
- Assume we are in the context of class `SinglyLinkedList`.

```
Node getNodeAt (int i) {
  if (i < 0 || i >= size) {
    throw new IllegalArgumentException("Invalid Index");
  }
  else {
    int index = 0;
    Node current = head;
    while (index < i) {  /* exit when index == i */
      index ++;
      /* current is set to node at index i
       * last iteration: index incremented from i - 1 to i
       */
      current = current.getNext();
    }
    return current;
  }
}
```


Singly-Linked List: Accessing the Middle (2)




```
Node getNodeAt (int i) {
  if (i < 0 || i >= size) { /* error */ }
  else {
    int index = 0;
    Node current = head;
    while (index < i) { /* exit when index == i */
      index ++;
      current = current.getNext();
    }
    return current;
  }
}
```

Let's now consider `list.getNodeAt(2)`:

`current`	`index`	`index < 2`	Beginning of Iteration
alan	0	*true*	1
mark	1	*true*	2
tom	2	*false*	–

Singly-Linked List: Accessing the Middle (3)

- What is the ***worst case*** of the index `i` for `getNodeAt(i)`?
  - Worst case: `list.getNodeAt(list.size - 1)`
  - RT of `getNodeAt` is *O(n)* [linear-time op.]
- **Contrast**: Accessing an array element costs *O(1)* [constant-time op.]

Singly-Linked List: Inserting to the Middle (1)

Problem: Insert a new element at index *i* in the list.

- **Hint 1**. $0 \leq i \leq$ `list.getSize()`
- **Hint 2**. Use `getNodeAt(?)` as a helper method.

```
void addAt (int i, String e) {
  if (i < 0 || i > size) {
    throw new IllegalArgumentException("Invalid Index.");
  }
  else {
    if (i == 0) {
      addFirst(e);
    }
    else {
      Node nodeBefore = getNodeAt(i - 1);
      Node newNode = new Node(e, nodeBefore.getNext());
      nodeBefore.setNext(newNode);
      size ++;
    }
  }
}
```

Example. See `testSLL_addAt` in `ExampleStringLinkedLists.zip`.

Singly-Linked List: Inserting to the Middle (2)

- A call to `addAt(i, e)` may end up executing:
  - Line 3 (throw exception) [$O(1)$]
  - Line 7 (addFirst) [$O(1)$]
  - Lines 10 (getNodeAt) [$O(n)$]
  - Lines 11 – 13 (setting references) [$O(1)$]
- What is the ***worst case*** of the index `i` for `addAt(i, e)`?

  A. `list.addAt(list.getSize(), e)`

  which requires list.getNodeAt(list.getSize() - 1)
- RT of `addAt` is *O(n)* [linear-time op.]
- **Contrast**: Inserting into an array costs *O(n)* [linear-time op.]

  For arrays, when given the *index* to an element, the RT of inserting an element is always $O(n)$!

Singly-Linked List: Removing from the End

__Problem__: Remove the last node (i.e., tail) of the list.

__Hint__. Using ***tail*** sufficient? Use `getNodeAt(?)` as a helper?

- Assume we are in the context of class `SinglyLinkedList`.

```
void removeLast () {
  if (size == 0) {
    throw new IllegalArgumentException("Empty List.");
  }
  else if (size == 1) {
    removeFirst();
  }
  else {
    Node secondLastNode = getNodeAt(size - 2);
    secondLastNode.setNext(null);
    tail = secondLastNode;
    size --;
  }
}
```

Running time? $O(n)$

Singly-Linked List: Exercises

Consider the following two linked-list operations, where a *reference node* is given as an input parameter:

- `void insertAfter(Node n, String e)`
  - Steps?
    - *Create a new node `nn`.*
    - *Set `nn`'s next to `n`'s next.*
    - *Set `n`'s next to `nn`.*
  - Running time? [*O(1)*]
- `void insertBefore(Node n, String e)`
  - Steps?
    - *Iterate from the `head`, until `current.next == n`.*
    - *Create a new node `nn`.*
    - *Set `nn`'s next to `current`'s next (which is `n`).*
    - *Set `current`'s next to `nn`.*
  - Running time? [*O(n)*]

Exercise

- Complete the Java ***implementation***, ***tests***, and ***running time analysis*** for `void removeAt(int i)`.

Arrays vs. Singly-Linked Lists

Operation \ Data Structure		Array	Singly-Linked List
get size		O(1)	O(1)
get first/last element		O(1)	O(1)
get element at index i		O(1)	O(n)
remove last element		O(1)	O(n)
add/remove first element, add last element		O(n)	O(1)
add/remove i^{th} element	given reference to $(i-1)^{th}$ element	O(n)	O(1)
	not given	O(n)	O(n)

Background Study: Generics in Java

- It is assumed that, in EECS2030, you learned about the basics of Java ***generics***:
  - General collection (e.g., `Object[]`) vs. Generic collection (e.g., `E[]`)
  - How using generics minimizes ***casts*** and ***`instanceof` checks***
  - How to implement and use generic classes
- If needed, review the above assumed basics from the relevant parts of EECS2030 (`https://www.eecs.yorku.ca/~jackie/teaching/lectures/index.html#EECS2030_F21`):
  - Parts A1 – A3, Lecture 7, Week 10
  - Parts B – C, Lecture 7, Week 11

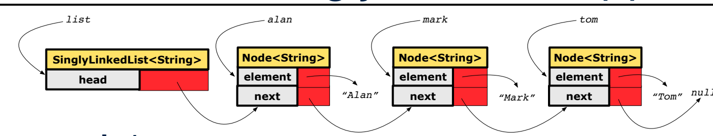
  Tips.
  - Skim the ***slides***: watch lecture videos if needing explanations.
  - Ask questions related to the assumed basics of ***generics***!
- Assuming that know the basics of Java ***generics***, we will implement and use ***generic SLL*** and ***DLL***.

Generic Classes: Singly-Linked List (1)

```
public class Node< E > {
  private E element;
  private Node< E > next;
  public Node( E e, Node< E > n) { element = e; next = n; }
  public E getElement() { return element; }
  public void setElement( E e) { element = e; }
  public Node< E > getNext() { return next; }
  public void setNext(Node< E > n) { next = n; }
}
```

```
public class SinglyLinkedList< E > {
  private Node< E > head;
  private Node< E > tail;
  private int size;
  public void setHead(Node< E > n) { head = n; }
  public void addFirst( E e) { ... }
  Node< E > getNodeAt (int i) { ... }
  void addAt (int i, E e) { ... }
}
```


Generic Classes: Singly-Linked List (2)



Approach 1

```
Node<String> tom = new Node<String>("Tom", null);
Node<String> mark = new Node<>("Mark", tom);
Node<String> alan = new Node<>("Alan", mark);
SinglyLinkedList<String> list = new SinglyLinkedList<>();
list.setHead(alan);
```

Approach 2

```
Node<String> alan = new Node<String>("Alan", null);
Node<String> mark = new Node<>("Mark", null);
Node<String> tom = new Node<>("Tom", null);
alan.setNext(mark);
mark.setNext(tom);
SinglyLinkedList<String> list = new SinglyLinkedList<>();
list.setHead(alan);
```

Generic Classes: Singly-Linked List (3)

Assume we are in the context of class `SinglyLinkedList`.

```
void addFirst (E e) {
  head = new Node<E>(e, head);
  if (size == 0) { tail = head; }
  size ++;
}
```

```
Node<E> getNodeAt (int i) {
  if (i < 0 || i >= size) {
   throw new IllegalArgumentException("Invalid Index"); }
  else {
    int index = 0;
    Node<E> current = head;
    while (index < i) {
      index ++;
      current = current.getNext();
    }
    return current;
  }
}
```


Singly-Linked Lists: Handling Edge Cases

```
void addFirst (E e) {
  head = new Node<E>(e, head);
  if (size == 0) {
    tail = head; } size ++; }
```

```
void removeFirst () {
  if (size == 0) { /* error */ }
  else if (size == 1) {
    head = null; tail = null; size --; }
  else {
    Node<E> oldHead = head;
    head = oldHead.getNext();
    oldHead.setNext(null); size --;
  } }
```

- We have to ***explicitly*** deal with special cases where the ***current list*** or ***resulting list*** is empty.
- We can actually resolve this issue via a ***small extension***!

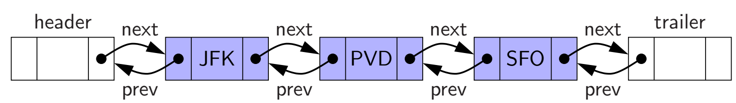
Basic Data Structure: Doubly-Linked Lists (1)

- We know that *singly-linked* lists perform:
  - **WELL**: [*O(1)*]
    - inserting to the front/end [***head***/***tail***]
    - removing from the front [***head***]
    - inserting/deleting the middle [given ref. to previous node]
  - **POORLY**: [*O(n)*]
    - accessing the middle [`getNodeAt(i)`]
    - removing from the end [`getNodeAt(list.getSize() - 2)`]
- We may again improve the performance by

  trading space for time

  just like how attributes ***size*** and ***tail*** were introduced.

Basic Data Structure: Doubly-Linked Lists (2)



- Each ***node*** in a *doubly-linked list (DLL)* stores:
  - A ***reference*** to an element of the sequence
  - A ***reference*** to the next node in the list
  - A ***reference*** to the *previous node* in the list [SYMMETRY]
- Each ***DLL*** stores:
  - A ***reference*** to a dedicated *header node* in the list
  - A ***reference*** to a dedicated *trailer node* in the list

  Remark. Unlike SLL, ***DLL*** does not store refs. to ***head*** and ***tail***.
- These two special nodes are called ***sentinels*** or ***guards***:
  - They do **not** store data, but store node references:
    - The ***header node*** stores the **next** reference only
    - The ***trailer node*** stores **previous** reference only
  - They **always** exist, even in the case of empty lists.

Generic Doubly-Linked Lists in Java (1)

```java
public class Node<E> {
  private E element;
  private Node<E> next;
  public E getElement() { return element; }
  public void setElement(E e) { element = e; }
  public Node<E> getNext() { return next; }
  public void setNext(Node<E> n) { next = n; }
  private Node<E> prev;
  public Node<E> getPrev() { return prev; }
  public void setPrev(Node<E> p) { prev = p; }
  public Node(E e, Node<E> p, Node<E> n) {
    element = e;
    prev = p;
    next = n;
  }
}
```


Generic Doubly-Linked Lists in Java (2)

```java
1  public class DoublyLinkedList<E> {
2    private int size = 0;
3    public void addFirst(E e) { ... }
4    public void removeLast() { ... }
5    public void addAt(int i, E e) { ... }
6    private Node<E> header;
7    private Node<E> trailer;
8    public DoublyLinkedList() {
9      header = new Node<>(null, null, null);
10     trailer = new Node<>(null, header, null);
11     header.setNext(trailer);
12   }
13 }
```

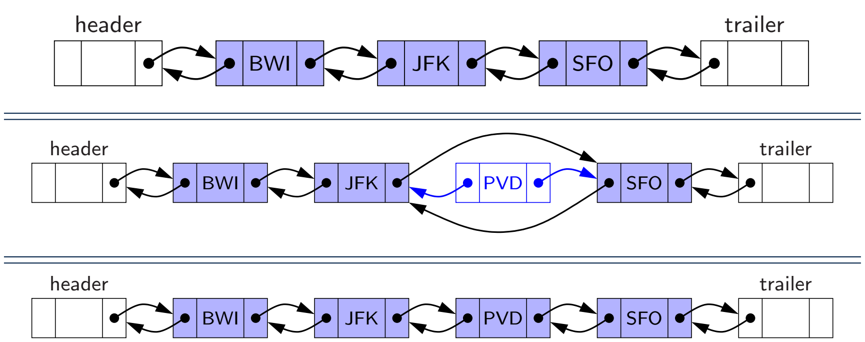
Lines 8 to 10 are equivalent to:

```java
header = new Node<>(null, null, null);
trailer = new Node<>(null, null, null);
header.setNext(trailer);
trailer.setPrev(header);
```


Header, Trailer, and `prev` Reference

- The ***`prev` reference*** helps ***improve the performance*** of `removeLast()`.
  - ∵ The ***second last node*** can be accessed in *constant time*. [`trailer.getPrev().getPrev()`]
- The two ***sentinel/guard*** nodes (***header*** and ***trailer***) do **not** help improve the performance.
  - Instead, they help *simplify the logic* of your code.
  - Each insertion/deletion can be treated
    - *Uniformly* : a node is always inserted/deleted in-between two nodes
    - Without worrying about re-setting the ***head*** and ***tail*** of list

Doubly-Linked List: Insertions



Doubly-Linked List: Inserting to Front/End

```
void addBetween(E e, Node<E> pred, Node<E> succ) {
  Node<E> newNode = new Node<>(e, pred, succ);
  pred.setNext(newNode);
  succ.setPrev(newNode);
  size ++;
}
```

Running Time? $O(1)$

```
void addFirst(E e) {
  addBetween(e, header, header.getNext())
}
```

Running Time? $O(1)$

```
void addLast(E e) {
  addBetween(e, trailer.getPrev(), trailer)
}
```

Running Time? $O(1)$

Doubly-Linked List: Inserting to Middle

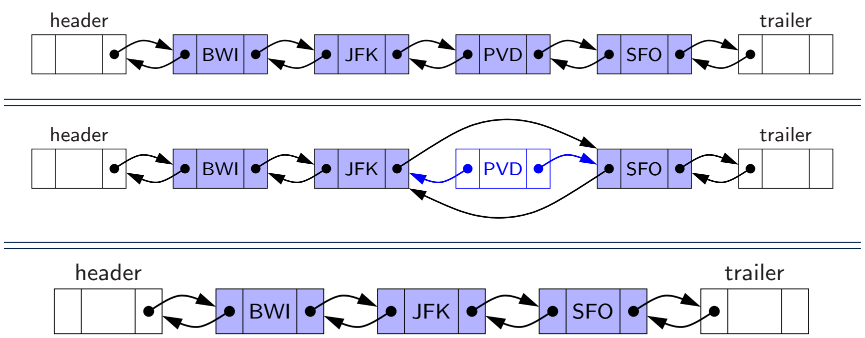
```
void addBetween(E e, Node<E> pred, Node<E> succ) {
  Node<E> newNode = new Node<>(e, pred, succ);
  pred.setNext(newNode);
  succ.setPrev(newNode);
  size ++;
}
```

Running Time? $O(1)$

```
addAt(int i, E e) {
  if (i < 0 || i > size) {
    throw new IllegalArgumentException("Invalid Index."); }
  else {
    Node<E> pred = getNodeAt(i - 1);
    Node<E> succ = pred.getNext();
    addBetween(e, pred, succ);
  }
}
```

Running Time? Still $O(n)$!!!

Doubly-Linked List: Removals



Doubly-Linked List: Removing from Front/End

```
void remove(Node<E> node) {
  Node<E> pred = node.getPrev();
  Node<E> succ = node.getNext();
  pred.setNext(succ); succ.setPrev(pred);
  node.setNext(null); node.setPrev(null);
  size --;
}
```

Running Time? $O(1)$

```
void removeFirst() {
  if (size == 0) { throw new IllegalArgumentException("Empty"); }
  else { remove(header.getNext()); }
}
```

Running Time? $O(1)$

```
void removeLast() {
  if (size == 0) { throw new IllegalArgumentException("Empty"); }
  else { remove(trailer.getPrev()); }
}
```

Running Time? Now $O(1)$!!!

Doubly-Linked List: Removing from Middle

```
void remove (Node<E> node) {
  Node<E> pred = node.getPrev();
  Node<E> succ = node.getNext();
  pred.setNext(succ); succ.setPrev(pred);
  node.setNext(null); node.setPrev(null);
  size --;
}
```

Running Time? $O(1)$

```
removeAt (int i) {
  if (i < 0 || i >= size) {
    throw new IllegalArgumentException("Invalid Index."); }
  else {
    Node<E> node = getNodeAt(i);
    remove(node);
  }
}
```

Running Time? Still $O(n)$!!!

Reference Node: To be Given or Not to be Given

Exercise 1: Compare the steps and running times of:

- *Not given* a reference node:
  - `addNodeAt(int i, E e)` [$O(n)$]
- *Given* a reference node:
  - `addNodeBefore(Node<E> n, E e)` [SLL: $O(n)$; DLL: $O(1)$]
  - `addNodeAfter(Node<E> n, E e)` [$O(1)$]

Exercise 2: Compare the steps and running times of:

- *Not given* a reference node:
  - `removeNodeAt(int i)` [$O(n)$]
- *Given* a reference node:
  - `removeNodeBefore(Node<E> n)` [SLL: $O(n)$; DLL: $O(1)$]
  - `removeNodeAfter(Node<E> n)` [$O(1)$]
  - `removvNode(Node<E> n)` [SLL: $O(n)$; DLL: $O(1)$]

Arrays vs. (Singly- and Doubly-Linked) Lists

<table>
<tr><th colspan="2">DATA STRUCTURE / OPERATION</th><th>ARRAY</th><th>SINGLY-LINKED LIST</th><th>DOUBLY-LINKED LIST</th></tr>
<tr><td colspan="2">size</td><td colspan="3" rowspan="2">O(1)</td></tr>
<tr><td colspan="2">first/last element</td></tr>
<tr><td colspan="2">element at index i</td><td>O(1)</td><td rowspan="2">O(n)</td><td>O(n)</td></tr>
<tr><td colspan="2">remove last element</td><td rowspan="4">O(n)</td><td rowspan="3">O(1)</td></tr>
<tr><td colspan="2">add/remove first element, add last element</td><td rowspan="2">O(1)</td></tr>
<tr><td rowspan="2">add/remove i^{th} element</td><td>given reference to $(i-1)^{th}$ element</td></tr>
<tr><td>not given</td><td colspan="2">O(n)</td></tr>
</table>

Beyond this lecture . . .

- In Eclipse, ***implement*** and ***test*** the assigned methods in `SinglyLinkedList` class and `DoublyLinkedList` class.
- Modify the ***insertion sort*** and ***selection sort*** implementations using a SLL or DLL.

Index (1)

Index (2)

Index (3)

Index (4)

Index (5)

Abstract Data Types (ADTs), Stacks, Queues

EECS2011 N & Z:
Fundamentals of Data Structures
Winter 2022

CHEN-WEI WANG

YORK UNIVERSITÉ UNIVERSITY

Learning Outcomes of this Lecture

This module is designed to help you learn about:

- The notion of *Abstract Data Types (**ADTs**)*
- The obligations of an ADT's ***supplier***
- The benefits of an ADT's ***client***
- Criterion of *Modularity*, Modular Design
- ***ADTs***: Stack vs. Queue
- Implementing Stack and Queue in Java [interface, classes]
- Applications of Stack

Background Study: Interfaces in Java

- It is assumed that, in EECS2030, you learned about the basics of Java ***interfaces***:
  - How to declare an interface
  - How to create a class implementing an interface
  - How ***polymorphism*** and ***dynamic binding*** work
- If needed, review the above assumed basics from the relevant parts of EECS2030 (`https://www.eecs.yorku.ca/~jackie/teaching/lectures/index.html#EECS2030_F21`):
  - Parts B1 – B3, Lecture 6, Week 10

  Tips.
  - Skim the ***slides***: watch lecture videos if needing explanations.
  - Ask questions related to the assumed basics of ***interfaces***!
- Assuming that know the basics of Java ***interfaces***, we will implement and use ***generic Stack*** and ***Queue***.

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 are the ***benefits*** and ***obligations*** of 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)

```
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);
} }
```

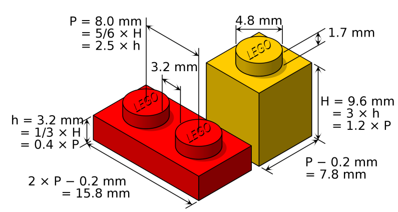
- The *contract* is *honoured* if:

  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

Modularity (1): Childhood Activity

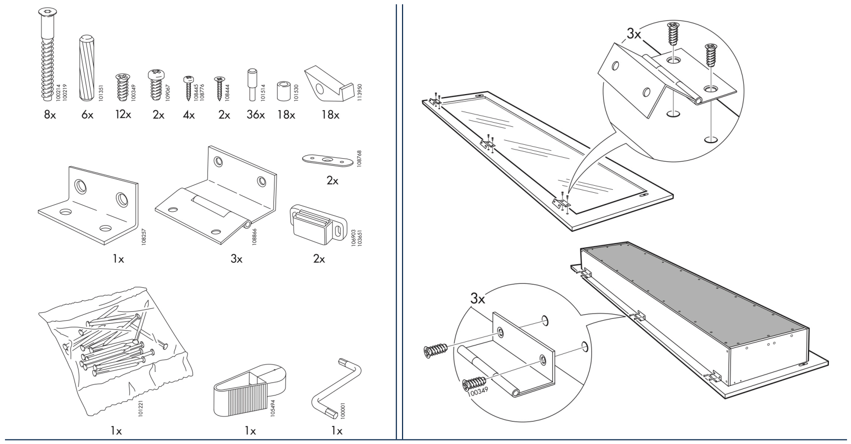


(INTERFACE) SPECIFICATION

(ASSEMBLY) ARCHITECTURE

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

Modularity (2): Daily Construction

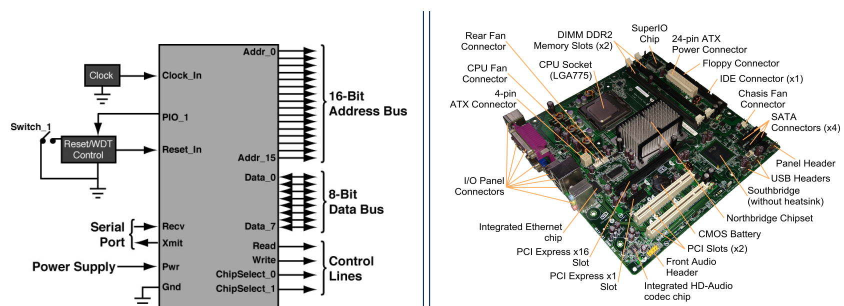


(INTERFACE) SPECIFICATION | (ASSEMBLY) ARCHITECTURE

Source: https://usermanual.wiki/

Modularity (3): Computer Architecture

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

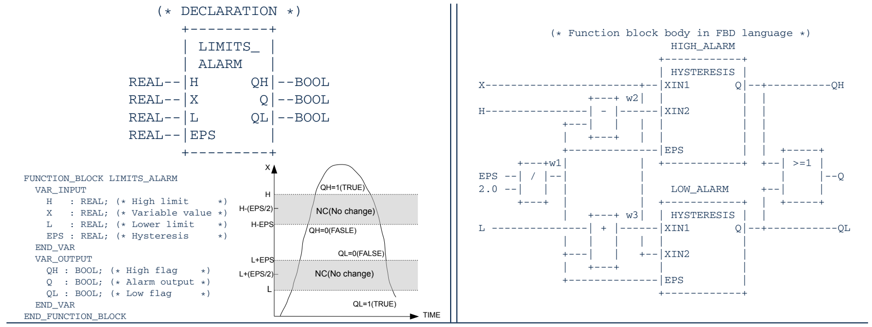


(INTERFACE) SPECIFICATION | (ASSEMBLY) ARCHITECTURE

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*.

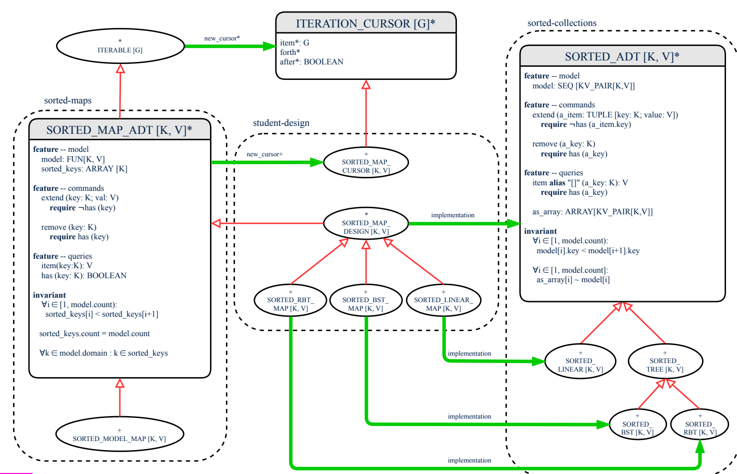


(INTERFACE) SPECIFICATION | (ASSEMBLY) ARCHITECTURE

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

Modularity (5): Software Design

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

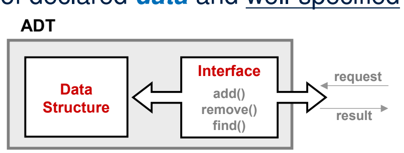


Design Principle: Modularity

- *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, decompose 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***



- Supplier's ***Obligations***:
  - Implement all operations
  - Choose the "**right**" data structure [e.g., arrays vs. SLL vs. DLL]
  - The internal details of an implemented *ADT* should be **hidden**.
- Client's ***Benefits***:
  - ***Correct*** output
  - ***Efficient*** performance

Java API Approximates 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

Java API Approximates ADTs (2)

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())`

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

Building ADTs for Reusability

- *ADTs* are ***reusable software components*** that are common for solving many real-world problems.
  e.g., Stacks, Queues, Lists, Tables, Trees, Graphs
- An *ADT*, once thoroughly tested, can be reused by:
  - ***Clients*** of Applications
  - ***Suppliers*** of other ADTs
- As a supplier, you are obliged to:
  - ***Implement*** standard ADTs [≈ lego building bricks]
    Note. Recall the basic data structures: arrays vs. SLLs vs. DLLs
  - ***Design*** algorithms using standard ADTs [≈ lego houses, ships]
- For each standard *ADT*, you should know its *interface*:
  - Stored ***data***
  - For each ***operation*** manipulating the stored data
    - How are ***clients*** supposed to use the method? [*preconditions*]
    - What are the services provided by ***suppliers***? [*postconditions*]
    - Time (and sometimes space) ***complexity***

What is a Stack?

- A ***stack*** is a collection of objects.
- Objects in a ***stack*** are inserted and removed according to the *last-in, first-out (**LIFO**)* principle.
  - ***Cannot*** access arbitrary elements of a stack
  - ***Can*** only access or remove the *most-recently added* element

The Stack ADT

- ***top***
  [***precondition***: stack is not empty]
  [***postcondition***: return item **last** pushed to the stack]
- ***size***
  [***precondition***: **none**]
  [***postcondition***: return number of items pushed to the stack]
- ***isEmpty***
  [***precondition***: **none**]
  [***postcondition***: return whether there is no item in the stack]
- ***push(item)***
  [***precondition***: stack is not full]
  [***postcondition***: push the input item onto the top of the stack]
- ***pop***
  [***precondition***: stack is not empty]
  [***postcondition***: remove and return the top of stack]

Stack: Illustration

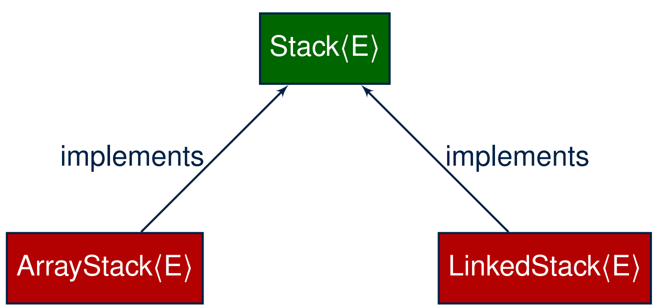
OPERATION	RETURN VALUE	STACK CONTENTS
–	–	∅
isEmpty	*true*	∅
push(5)	–	5
push(3)	–	3 5
push(1)	–	1 3 5
size	3	1 3 5
top	1	1 3 5
pop	1	3 5
pop	3	5
pop	5	∅

Generic Stack: Interface

```
public interface Stack< E > {
  public int size();
  public boolean isEmpty();
  public  E  top();
  public void push( E  e);
  public  E  pop();
}
```

The ***Stack*** ADT, declared as an ***interface***, allows ***alternative implementations*** to conform to its method headers.

Generic Stack: Architecture



Implementing Stack: Array (1)

```
public class ArrayStack<E> implements Stack<E> {
  private final int MAX_CAPACITY = 1000;
  private E[] data;
  private int t; /* index of top */
  public ArrayStack() {
    data = (E[]) new Object[MAX_CAPACITY];
    t = -1;
  }

  public int size() { return (t + 1); }
  public boolean isEmpty() { return (t == -1); }

  public E top() {
    if (isEmpty()) { /* Precondition Violated */ }
    else { return data[t]; }
  }
  public void push(E e) {
    if (size() == MAX_CAPACITY) { /* Precondition Violated */ }
    else { t ++; data[t] = e; }
  }
  public E pop() {
    E result;
    if (isEmpty()) { /* Precondition Violated */ }
    else { result = data[t]; data[t] = null; t --; }
    return result;
  }
}
```


Implementing Stack: Array (2)

- Running Times of *Array*-Based *Stack* Operations?

***ArrayStack* Method**	**Running Time**
size	O(1)
isEmpty	O(1)
top	O(1)
push	O(1)
pop	O(1)

- **Exercise** This version of implementation treats the ***end*** of array as the ***top*** of stack. Would the RTs of operations change if we treated the ***beginning*** of array as the ***top*** of stack?
- **Q**. What if the preset capacity turns out to be insufficient?

  A. `IllegalArgumentException` occurs and it takes ***O(1)*** time to respond.
- At the end, we will explore the alternative of a ***dynamic array***.

Implementing Stack: Singly-Linked List (1)

```
public class LinkedStack<E> implements Stack<E> {
  private SinglyLinkedList<E> list;
  ...
}
```

Question:

Stack Method	Singly-Linked List Method	
	Strategy 1	Strategy 2
size	list.size	
isEmpty	list.isEmpty	
top	list.first	list.last
push	list.addFirst	list.addLast
pop	list.removeFirst	list.removeLast

Which ***implementation strategy*** should be chosen?

Implementing Stack: Singly-Linked List (2)

- If the ***front of list*** is treated as the ***top of stack***, then:
  - All stack operations remain ***O(1)*** [∵ removeFirst takes ***O(1)***]
- If the ***end of list*** is treated as the ***top of stack***, then:
  - The ***pop*** operation takes ***O(n)*** [∵ removeLast takes ***O(n)***]
- But in both cases, given that a linked, ***dynamic*** structure is used, ***no resizing*** is necessary!

Generic Stack: Testing Implementations

```
@Test
public void testPolymorphicStacks() {
  Stack<String> s = new ArrayStack<>();
  s.push("Alan"); /* dynamic binding */
  s.push("Mark"); /* dynamic binding */
  s.push("Tom"); /* dynamic binding */
  assertTrue(s.size() == 3 && !s.isEmpty());
  assertEquals("Tom", s.top());

  s = new LinkedStack<>();
  s.push("Alan"); /* dynamic binding */
  s.push("Mark"); /* dynamic binding */
  s.push("Tom"); /* dynamic binding */
  assertTrue(s.size() == 3 && !s.isEmpty());
  assertEquals("Tom", s.top());
}
```


Polymorphism & Dynamic Binding

```
1  Stack<String> myStack;
2  myStack = new ArrayStack<String>();
3  myStack.push("Alan");
4  myStack = new LinkedStack<String>();
5  myStack.push("Alan");
```

- *Polymorphism*

  An object may change its ***"shape"*** (i.e., ***dynamic type***) at runtime.

  Which lines? 2, 4
- *Dynamic Binding*

  Effect of a method call depends on the ***"current shape"*** of the target object.

  Which lines? 3, 5

Stack Application: Reversing an Array

- ***Implementing*** a ***generic*** algorithm:

```
public static <E> void reverse(E[] a) {
  Stack<E> buffer = new ArrayStack<E>();
  for (int i = 0; i < a.length; i ++) {
    buffer.push(a[i]);
  }
  for (int i = 0; i < a.length; i ++) {
    a[i] = buffer.pop();
  }
}
```

- ***Testing*** the ***generic*** algorithm:

```
@Test
public void testReverseViaStack() {
   String[] names = {"Alan", "Mark", "Tom"};
   String[] expectedReverseOfNames = {"Tom", "Mark", "Alan"};
   StackUtilities.reverse(names);
   assertArrayEquals(expectedReverseOfNames, names);

   Integer[] numbers = {46, 23, 68};
   Integer[] expectedReverseOfNumbers= {68, 23, 46};
   StackUtilities.reverse(numbers);
   assertArrayEquals(expectedReverseOfNumbers, numbers);
}
```


Stack Application: Matching Delimiters (1)

- **Problem**

  Opening delimiters: `(`, `[`, `{`

  Closing delimiters: `)`, `]`, `}`

  e.g., **Correct**: `() (()){([()])}`

  e.g., **Incorrect**: `({[])}`

- **Sketch of Solution**
  - When a new ***opening*** delimiter is found, ***push*** it to the stack.
  - *Most-recently* found delimiter should be matched first.
  - When a new ***closing*** delimiter is found:
    - If it matches the ***top*** of the stack, then ***pop*** off the stack.
    - Otherwise, an error is found!
  - Finishing reading the input, an empty stack means a success!

Stack Application: Matching Delimiters (2)

- ***Implementing*** the algorithm:

```
public static boolean isMatched(String expression) {
  final String opening = "([{";
  final String closing = ")]}";
  Stack<Character> openings = new LinkedStack<Character>();
  int i = 0;
  boolean foundError = false;
  while (!foundError && i < expression.length()) {
    char c = expression.charAt(i);
    if(opening.indexOf(c) != -1) { openings.push(c); }
    else if (closing.indexOf(c) != -1) {
      if(openings.isEmpty()) { foundError = true; }
      else {
        if (opening.indexOf(openings.top()) == closing.indexOf(c)) { openings.pop(); }
        else { foundError = true; } } }
    i ++; }
  return !foundError && openings.isEmpty(); }
```

- ***Testing*** the algorithm:

```
@Test
public void testMatchingDelimiters() {
  assertTrue(StackUtilities.isMatched(""));
  assertTrue(StackUtilities.isMatched("{[]}({})"));
  assertFalse(StackUtilities.isMatched("{[])"));
  assertFalse(StackUtilities.isMatched("{[]})"));
  assertFalse(StackUtilities.isMatched("({[]}"));
}
```


Stack Application: Postfix Notations (1)

Problem: Given a postfix expression, calculate its value.

Infix Notation	**Postfix** Notation
Operator ***in-between*** Operands	Operator ***follows*** Operands
Parentheses force precedence	Order of evaluation embedded
`3`	`3`
`3 + 4`	`3 4 +`
`3 + 4 + 5`	`3 4 + 5 +`
`3 + (4 + 5)`	`3 4 5 + +`
`3 - 4 * 5`	`3 4 5 * -`
`(3 - 4) * 5`	`3 4 - 5 *`

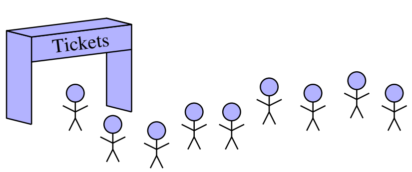
Stack Application: Postfix Notations (2)

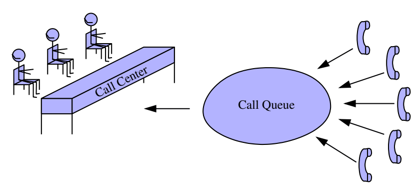
Sketch of Solution

- When input is an ***operand*** (i.e., a number), ***push*** it to the stack.
- When input is an ***operator***, obtain its two ***operands*** by ***popping*** off the stack **twice**, evaluate, then ***push*** the result back to stack.
- When finishing reading the input, there should be **only one** number left in the stack.
- **Error** if:
  - Not enough items left in the stack for the operator [e.g., `523+*+`]
  - When finished, two or more numbers left in stack [e.g., `53+6`]

What is a Queue?

- A ***queue*** is a collection of objects.
- Objects in a ***queue*** are inserted and removed according to the *first-in, first-out (**FIFO**)* principle.
  - Each new element joins at the ***back***/***end*** of the queue.
  - ***Cannot*** access arbitrary elements of a queue
  - ***Can*** only access or remove the *least-recently inserted (or longest-waiting)* element





The Queue ADT

- ***first*** ≈ ***top*** of stack
  [***precondition***: queue is not empty]
  [***postcondition***: return item **first** enqueued]
- ***size***
  [***precondition***: **none**]
  [***postcondition***: return number of items enqueued]
- ***isEmpty***
  [***precondition***: **none**]
  [***postcondition***: return whether there is no item in the queue]
- ***enqueue(item)*** ≈ ***push*** of stack
  [***precondition***: queue is not full]
  [***postcondition***: enqueue item as the "last" of the queue]
- ***dequeue*** ≈ ***pop*** of stack
  [***precondition***: queue is not empty]
  [***postcondition***: remove and return the first of the queue]

Queue: Illustration

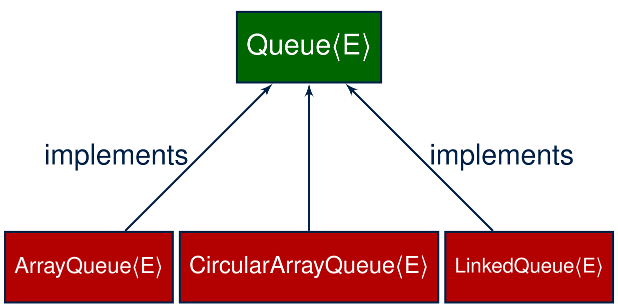
Operation	Return Value	Queue Contents
–	–	∅
isEmpty	*true*	∅
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
enqueue(1)	–	(5, 3, 1)
size	3	(5, 3, 1)
dequeue	5	(3, 1)
dequeue	3	1
dequeue	1	∅

Generic Queue: Interface

```
public interface Queue< E > {
  public int size();
  public boolean isEmpty();
  public E first();
  public void enqueue(E e);
  public E dequeue();
}
```

The ***Queue*** ADT, declared as an ***interface***, allows ***alternative implementations*** to conform to its method headers.

Generic Queue: Architecture



Implementing Queue ADT: Array (1)

```
public class ArrayQueue<E> implements Queue<E> {
  private final int MAX_CAPACITY = 1000;
  private E[] data;
  private int r; /* rear index */
  public ArrayQueue() {
    data = (E[]) new Object[MAX_CAPACITY];
    r = -1;
  }
  public int size() { return (r + 1); }
  public boolean isEmpty() { return (r == -1); }
  public E first() {
    if (isEmpty()) { /* Precondition Violated */ }
    else { return data[0]; }
  }
  public void enqueue(E e) {
    if (size() == MAX_CAPACITY) { /* Precondition Violated */ }
    else { r ++; data[r] = e; }
  }
  public E dequeue() {
    if (isEmpty()) { /* Precondition Violated */ }
    else {
      E result = data[0];
      for (int i = 0; i < r; i ++) { data[i] = data[i + 1]; }
      data[r] = null; r --;
      return result;
    }
  }
}
```


Implementing Queue ADT: Array (2)

- Running Times of *Array*-Based *Queue* Operations?

ArrayQueue **Method**	**Running Time**
size	O(1)
isEmpty	O(1)
first	O(1)
enqueue	O(1)
dequeue	$O(n)$

- **Exercise** This version of implementation treats the ***beginning*** of array as the ***first*** of queue. Would the RTs of operations change if we treated the ***end*** of array as the ***first*** of queue?
- **Q**. What if the preset capacity turns out to be insufficient?

  A. `IllegalArgumentException` occurs and it takes ***O(1)*** time to respond.
- At the end, we will explore the alternative of a ***dynamic array***.

Implementing Queue: Singly-Linked List (1)

```
public class LinkedQueue<E> implements Queue<E> {
  private SinglyLinkedList<E> list;
  ...
}
```

Question:

Queue Method	**Singly-Linked List Method**	
	Strategy 1	Strategy 2
size	list.size	
isEmpty	list.isEmpty	
first	list.first	list.last
enqueue	list.addLast	list.addFirst
dequeue	list.removeFirst	list.removeLast

Which ***implementation strategy*** should be chosen?

Implementing Queue: Singly-Linked List (2)

- If the ***front of list*** is treated as the ***first of queue***, then:
  - All queue operations remain ***O(1)*** [∵ removeFirst takes ***O(1)***]
- If the ***end of list*** is treated as the ***first of queue***, then:
  - The ***dequeue*** operation takes ***O(n)*** [∵ removeLast takes ***O(n)***]
- But in both cases, given that a linked, ***dynamic*** structure is used, ***no resizing*** is necessary!

Generic Queue: Testing Implementations

```
@Test
public void testPolymorphicQueues() {
  Queue<String> q = new ArrayQueue<>();
  q.enqueue("Alan"); /* dynamic binding */
  q.enqueue("Mark"); /* dynamic binding */
  q.enqueue("Tom"); /* dynamic binding */
  assertTrue(q.size() == 3 && !q.isEmpty());
  assertEquals("Alan", q.first());

  q = new LinkedQueue<>();
  q.enqueue("Alan"); /* dynamic binding */
  q.enqueue("Mark"); /* dynamic binding */
  q.enqueue("Tom"); /* dynamic binding */
  assertTrue(q.size() == 3 && !q.isEmpty());
  assertEquals("Alan", q.first());
}
```


Polymorphism & Dynamic Binding

```
1 Queue<String> myQueue;
2 myQueue = new CircularArrayQueue<String>();
3 myQueue.enqueue("Alan");
4 myQueue = new LinkedQueue<String>();
5 myQueue.enqueue("Alan");
```

- *Polymorphism*

  An object may change its ***"shape"*** (i.e., *dynamic type*) at runtime.

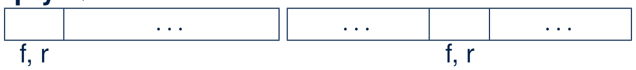
  Which lines? 2, 4
- *Dynamic Binding*

  Effect of a method call depends on the *"current shape"* of the target object.

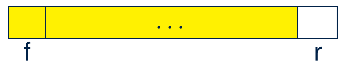
  Which lines? 3, 5

Implementing Queue ADT: Circular Array (1)

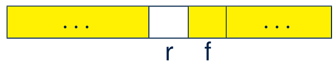
- Maintain two indices: ***f*** for ***front***; ***r*** for ***next available slot***.
- **Maximum size**: $N - 1$ [N = `data.length`]
- **Empty Queue**: when ***r = f***



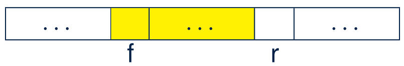
- **Full Queue**: when ***((r + 1) % N) = f***
  - When $r > f$:



  - When $r < f$:



- **Size of Queue**:
  - If $r = f$: ***0***
  - If $r > f$: ***r - f***



  - If $r < f$: ***r + (N - f)***

Implementing Queue ADT: Circular Array (2)

Running Times of *CircularArray*-Based *Queue* Operations?

***CircularArrayQueue* Method**	**Running Time**
size	O(1)
isEmpty	O(1)
first	O(1)
enqueue	O(1)
dequeue	*O*(1)

Exercise: Create a Java class `CircularArrayQueue` that `implements` the `Queue` interface using a ***circular array***.

Exercise: Implementing a Queue using Two Stacks

```
public class StackQueue<E> implements Queue<E> {
  private Stack<E> inStack;
  private Stack<E> outStack;
  ...
}
```

- For `size`, add up sizes of `inStack` and `outStack`.
- For `isEmpty`, are `inStack` and `outStack` both empty?
- For `enqueue`, ***push*** to `inStack`.
- For `dequeue`:
  - ***pop*** from `outStack`

    If `outStack` is empty, we need to first ***pop*** all items from `inStack` and ***push*** them to `outStack`.

Exercise: Why does this work? [***implement*** and ***test***]

Exercise: Running Time? [see analysis on ***dynamic arrays***]

Limitations of Queue

- Say we use a ***queue*** to implement a ***waiting list***.
  - What if we `dequeue` the front customer, but find that we need to ***put them back to the front*** (e.g., seat is still not available, the table assigned is not satisfactory, *etc.*)?
  - What if the customer at the end of the queue decides not to wait and leave, how do we ***remove them from the end of the queue***?
- **Solution:** A new ADT extending the ***Queue*** by supporting:
  - ***insertion*** to the ***front***
  - ***deletion*** from the ***end***

The Double-Ended Queue ADT

- ***Double-Ended Queue*** (or ***Deque***) is a queue-like data structure that supports ***insertion*** and ***deletion*** at both the ***front*** and the ***end*** of the queue.

```
public interface Deque<E> {
  /* Queue operations */
  public int size();
  public boolean isEmpty();
  public E first();
  public void addLast(E e); /* enqueue */
  public E removeFirst(); /* dequeue */
  /* Extended operations */
  public void addFirst(E e);
  public E removeLast();
}
```

- **Exercise**: Implement *Deque* using a ***circular array***.
- **Exercise**: Implement *Deque* using a ***SLL*** and/or ***DLL***.

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.

Beyond this lecture . . .

- Attempt the exercises throughout the lecture.
- Implement the ***Postfix Calculator*** using a stack.

Index (1)

Index (2)

Index (3)

Index (4)

Index (5)

Recursion

EECS2011 N & Z:
Fundamentals of Data Structures
Winter 2022

Chen-Wei Wang

Background Study: Basic Recursion

- It is assumed that, in EECS2030, you learned about the basics of ***recursion*** in Java:
  - What makes a method recursive?
  - How to trace recursion using a ***call stack***?
  - How to define and use ***recursive helper methods*** on arrays?
- If needed, review the above assumed basics from the relevant parts of EECS2030 (`https://www.eecs.yorku.ca/~jackie/teaching/lectures/index.html#EECS2030_F21`):
  - Parts A – C, Lecture 8, Week 12

  Tips.
  - Skim the ***slides***: watch lecture videos if needing explanations.
  - Recursion lab from EECS2030-F19: **here** [Solution: **here**]
  - Ask questions related to the assumed basics of ***recursion***!
- Assuming that you know the basics of ***recursion*** in Java, we will proceed with more advanced examples.

Extra Challenging Recursion Problems

1. groupSum
   - Problem Specification: ***here*** Solution: ***here***
   - Solution Walkthrough: ***here***
   - Notes: ***here [pp. 7–10]*** & ***here***
2. parenBit
   - Problem Specification: ***here*** Solution: ***here***
   - Solution Walkthrough: ***here***
   - Notes: ***here [pp. 4–5]***
3. climb
   - Problem Specification: ***here*** Solution: ***here***
   - Solution Walkthrough: ***here*** & ***here***
   - Notes: ***here [pp. 7–8]*** & ***here [p. 4]***
4. climbStrategies
   - Problem Specification: ***here*** Solution: ***here***
   - Solution Walkthrough: ***here***
   - Notes: ***here [pp. 5 – 6]***

Learning Outcomes of this Lecture

This module is designed to help you:

- Know about the resources on ***recursion basics***.
- Learn about the more intermediate ***recursive algorithms***:
  - Binary Search
  - Merge Sort
  - Quick Sort
  - Tower of Hanoi
- Explore extra, ***challenging*** recursive problems.

Recursion: Binary Search (1)

- **Searching Problem**

  Given a numerical key k and an array a of n numbers:

  Precondition: Input array a ***sorted*** in a non-descending order

  i.e., $a[0] \leq a[1] \leq \ldots \leq a[n-1]$

  Postcondition: Return whether or not k exists in the input array a.
- **Q.** RT of a search on an ***unsorted*** array?

  A. ***O(n)*** (despite being iterative or recursive)
- **A Recursive Solution**

  Base Case: Empty array $\longrightarrow$ ***false***.

  Recursive Case: Array of size ≥ 1 $\longrightarrow$
  - Compare the ***middle*** element of array a against key k.
    - All elements to the left of ***middle*** are $\leq k$
    - All elements to the right of ***middle*** are $\geq k$
  - If the ***middle*** element ***is*** equal to key k $\longrightarrow$ ***true***
  - If the ***middle*** element ***is not*** equal to key k:
    - If $k <$ ***middle***, **recursively** *search* key k on the left half.
    - If $k >$ ***middle***, **recursively** *search* key k on the right half.

Recursion: Binary Search (2)

```
boolean binarySearch(int[] sorted, int key) {
  return binarySearchH(sorted, 0, sorted.length - 1, key);
}
boolean binarySearchH(int[] sorted, int from, int to, int key) {
  if (from > to) { /* base case 1: empty range */
    return false; }
  else if(from == to) { /* base case 2: range of one element */
    return sorted[from] == key; }
  else {
    int middle = (from + to) / 2;
    int middleValue = sorted[middle];
    if(key < middleValue) {
      return binarySearchH(sorted, from, middle - 1, key);
    }
    else if (key > middleValue) {
      return binarySearchH(sorted, middle + 1, to, key);
    }
    else { return true; }
  }
}
```


Running Time: Binary Search (1)

We define ***T(n)*** as the ***running time function*** of a *binary search*, where ***n*** is the size of the input array.

$$\begin{cases} T(0) &= 1 \\ T(1) &= 1 \\ T(n) &= T(\frac{n}{2}) + 1 \quad \text{where } n \geq 2 \end{cases}$$

To solve this recurrence relation, we study the pattern of ***T(n)*** and observe how it reaches the ***base case(s)***.

Running Time: Binary Search (2)

Without loss of generality, assume $n = 2^i$ for some $i \geq 0$.

$$\begin{aligned} T(n) &= T(\tfrac{n}{2}) + 1 \\ &= \underbrace{(T(\frac{n}{4}) + 1)}_{T(\frac{n}{2})} + \underbrace{1}_{1 \text{ time}} \\ &= \underbrace{((T(\frac{n}{8}) + 1)}_{T(\frac{n}{4})} + \underbrace{1) + 1}_{2 \text{ times}} \\ &= \dots \\ &= (((\underbrace{1}_{T(\frac{n}{2^{\log n}}) = T(1)}) + \underbrace{1) \dots) + 1}_{\log n \text{ times}} \end{aligned}$$

$\therefore$ ***T(n)*** is ***O(log n)***

Recursion: Merge Sort

- **Sorting Problem**
  Given a list of **n** numbers $\langle a_1, a_2, \dots, a_n \rangle$:
  Precondition: NONE
  Postcondition: A permutation of the input list $\langle a'_1, a'_2, \dots, a'_n \rangle$ ***sorted*** in a non-descending order (i.e., $a'_1 \leq a'_2 \leq \dots \leq a'_n$)
- **A Recursive Algorithm**
  Base Case 1: Empty list $\longrightarrow$ Automatically sorted.
  Base Case 2: List of size 1 $\longrightarrow$ Automatically sorted.
  Recursive Case: List of size ≥ 2 $\longrightarrow$
  1. ***Split*** the list into two (***unsorted***) halves: ***L*** and ***R***.
  2. **Recursively** *sort* ***L*** and ***R***, resulting in: ***sortedL*** and ***sortedR***.
  3. Return the ***merge*** of ***sortedL*** and ***sortedR***.

Recursion: Merge Sort in Java (1)

```
/* Assumption:  L and R are both already sorted. */
private List<Integer> merge(List<Integer> L, List<Integer> R) {
  List<Integer> merge = new ArrayList<>();
  if(L.isEmpty()||R.isEmpty()) { merge.addAll(L); merge.addAll(R); }
  else {
    int i = 0;
    int j = 0;
    while(i < L.size() && j < R.size()) {
      if(L.get(i) <= R.get(j)) { merge.add(L.get(i)); i ++; }
      else { merge.add(R.get(j)); j ++; }
    }
    /* If i >= L.size(), then this for loop is skipped. */
    for(int k = i; k < L.size(); k ++) { merge.add(L.get(k)); }
    /* If j >= R.size(), then this for loop is skipped. */
    for(int k = j; k < R.size(); k ++) { merge.add(R.get(k)); }
  }
  return merge;
}
```

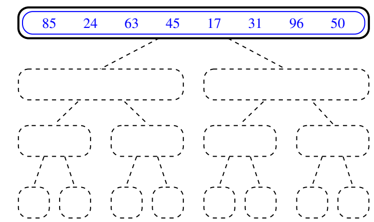
RT(`merge`)? [***O(L.size() + R.size())***]

Recursion: Merge Sort in Java (2)

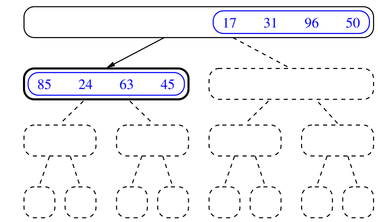
```
public List<Integer> sort(List<Integer> list) {
  List<Integer> sortedList;
  if(list.size() == 0) { sortedList = new ArrayList<>(); }
  else if(list.size() == 1) {
    sortedList = new ArrayList<>();
    sortedList.add(list.get(0));
  }
  else {
    int middle = list.size() / 2;
    List<Integer> left = list.subList(0, middle);
    List<Integer> right = list.subList(middle, list.size());
    List<Integer> sortedLeft = sort(left);
    List<Integer> sortedRight = sort(right);
    sortedList = merge(sortedLeft, sortedRight);
  }
  return sortedList;
}
```


Recursion: Merge Sort Example (1)

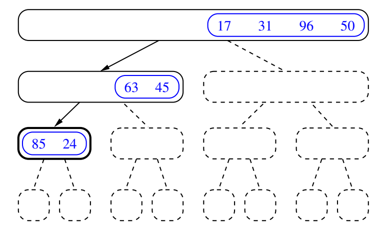
(1) Start with input list of size 8



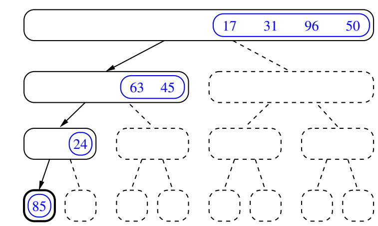
(2) Split and recur on L of size 4



(3) Split and recur on L of size 2

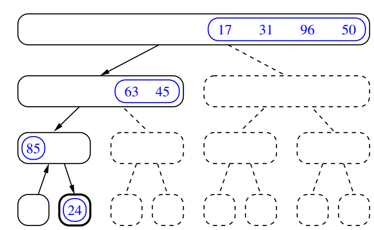


(4) Split and recur on L of size 1, *return*

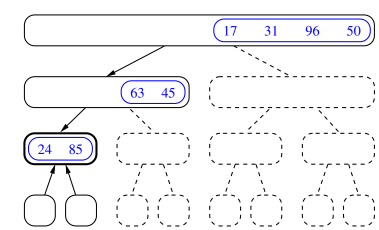


Recursion: Merge Sort Example (2)

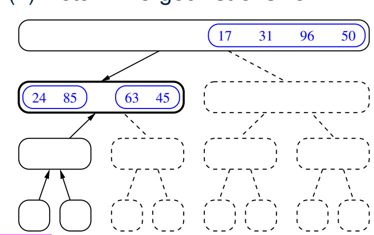
(5) Recur on R of size 1 and *return*



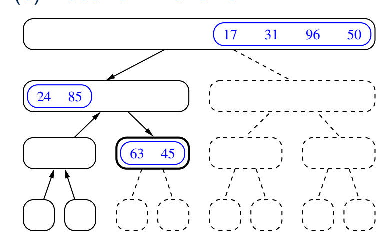
(6) Merge sorted L and R of sizes 1



(7) Return merged list of size 2

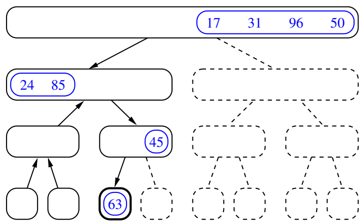


(8) Recur on R of size 2

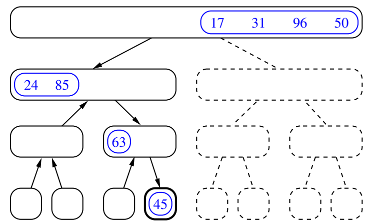


Recursion: Merge Sort Example (3)

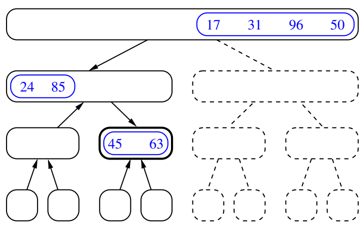
(9) Split and recur on L of size 1, *return*



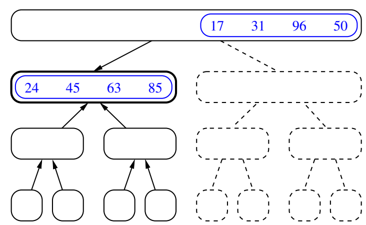
(10) Recur on R of size 1, *return*



(11) Merge sorted L and R of sizes 1, *return*

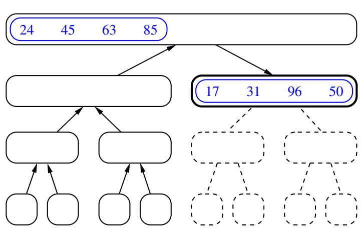


(12) Merge sorted L and R of sizes 2

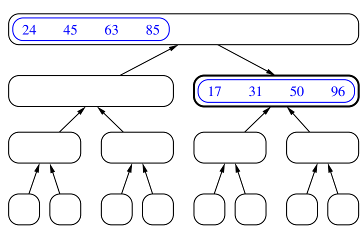


Recursion: Merge Sort Example (4)

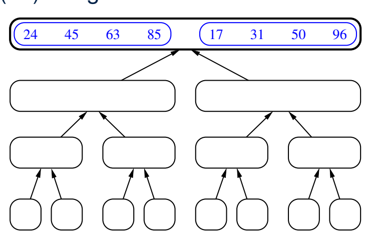
(13) Recur on R of size 4



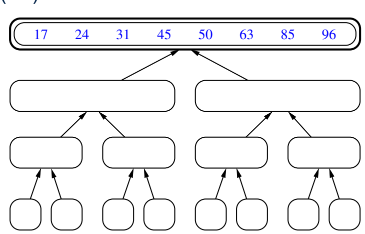
(14) *Return* a sorted list of size 4



(15) Merge sorted L and R of sizes 4



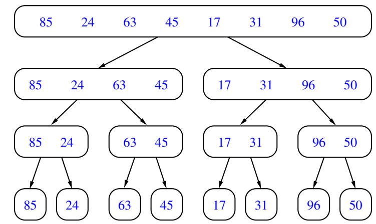
(16) *Return* a sorted list of size 8



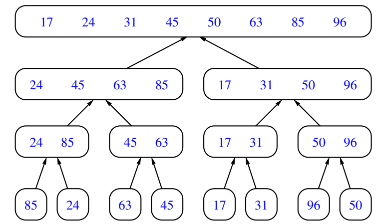
Recursion: Merge Sort Example (5)

Let's visualize the two ***critical phases*** of *merge sort*:

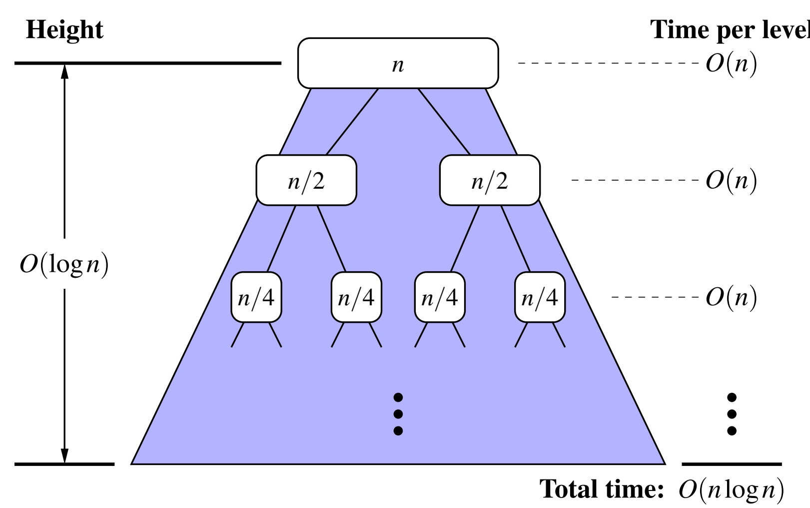
(1) After ***Splitting Unsorted*** Lists



(2) After ***Merging Sorted*** Lists



Recursion: Merge Sort Running Time (1)



Recursion: Merge Sort Running Time (2)

- **Base Case 1**: Empty list ⟶ Automatically sorted. [***O(1)***]
- **Base Case 2**: List of size 1 ⟶ Automatically sorted. [***O(1)***]
- **Recursive Case**: List of size ≥ 2 ⟶
  1. ***Split*** the list into two (***unsorted***) halves: ***L*** and ***R***; [***O(1)***]
  2. **Recursively** *sort* ***L*** and ***R***, resulting in: ***sortedL*** and ***sortedR***
     Q. # times to ***split*** until ***L*** and ***R*** have size 0 or 1?
     A. [***O(log n)***]
  3. Return the *merge* of ***sortedL*** and ***sortedR***. [***O(n)***]
-

$\boxed{\textbf{Running Time of Merge Sort}}$

= (**RT** each RC) × (# **RC**s)
= (**RT** merging ***sortedL*** and ***sortedR***) × (# splits until bases)
= $O(n \cdot log\ n)$

Recursion: Merge Sort Running Time (3)

We define ***T(n)*** as the ***running time function*** of a *merge sort*, where ***n*** is the size of the input array.

$$\begin{cases} T(0) & = & 1 \\ T(1) & = & 1 \\ T(n) & = & 2 \cdot T(\frac{n}{2}) + n \quad \texttt{where}\ n \geq 2 \end{cases}$$

To solve this recurrence relation, we study the pattern of ***T(n)*** and observe how it reaches the ***base case(s)***.

Recursion: Merge Sort Running Time (4)

Without loss of generality, assume $n = 2^i$ for some $i \geq 0$.

$$\begin{aligned}
T(n) &= 2 \times T(\tfrac{n}{2}) + n \\
&= \underbrace{2 \times (2}_{\texttt{2 terms}} \times T(\tfrac{n}{4}) + \underbrace{\tfrac{n}{2}) + n}_{\texttt{2 terms}} \\
&= \underbrace{2 \times (2 \times (2}_{\texttt{3 terms}} \times T(\tfrac{n}{8}) + \underbrace{\tfrac{n}{4}) + \tfrac{n}{2}) + n}_{\texttt{3 terms}} \\
&= \ldots \\
&= \underbrace{2 \times (2 \times (2 \times \cdots \times (2}_{log\ n\ \texttt{terms}} \times T(\tfrac{n}{2^{log\ n}}) + \underbrace{\tfrac{n}{2^{(log\ n)-1}}) + \cdots + \tfrac{n}{4}) + \tfrac{n}{2}) + n}_{log\ n\ \texttt{terms}} \\
&= \underbrace{2 \cdot \tfrac{n}{2} + 2^2 \cdot \tfrac{n}{4} + \cdots + 2^{(log\ n)-1} \cdot \tfrac{n}{2^{(log\ n)-1}} + \underbrace{n}_{2^{log\ n} \cdot \frac{n}{2^{log\ n}}}}_{log\ n\ \texttt{terms}} \\
&= \underbrace{n + n + \cdots + n + n}_{log\ n\ \texttt{terms}}
\end{aligned}$$

∴ ***T(n)*** is ***O(n · log n)***

Recursion: Quick Sort

- **Sorting Problem**
  Given a list of **n** numbers $\langle a_1, a_2, \ldots, a_n \rangle$:
  Precondition: **None**
  Postcondition: A permutation of the input list $\langle a'_1, a'_2, \ldots, a'_n \rangle$ ***sorted*** in a non-descending order (i.e., $a'_1 \leq a'_2 \leq \ldots \leq a'_n$)
- **A Recursive Algorithm**
  Base Case 1: Empty list ⟶ Automatically sorted.
  Base Case 2: List of size 1 ⟶ Automatically sorted.
  Recursive Case: List of size ≥ 2 ⟶
  1. Choose a ***pivot*** element. [ideally the ***median***]
  2. ***Split*** the list into two (***unsorted***) halves: ***L*** and ***R***, s.t.:
     All elements in ***L*** are less than or equal to (≤) the ***pivot***.
     All elements in ***R*** are greater than (>) the ***pivot***.
  3. **Recursively** *sort* ***L*** and ***R***: ***sortedL*** and ***sortedR***;
  4. Return the ***concatenation*** of: ***sortedL*** + ***pivot*** + ***sortedR***.

Recursion: Quick Sort in Java (1)

```
List<Integer> allLessThanOrEqualTo(int pivotIndex, List<Integer> list) {
  List<Integer> sublist = new ArrayList<>();
  int pivotValue = list.get(pivotIndex);
  for(int i = 0; i < list.size(); i ++) {
    int v = list.get(i);
    if(i != pivotIndex && v <= pivotValue) { sublist.add(v); }
  }
  return sublist;
}
List<Integer> allLargerThan(int pivotIndex, List<Integer> list) {
  List<Integer> sublist = new ArrayList<>();
  int pivotValue = list.get(pivotIndex);
  for(int i = 0; i < list.size(); i ++) {
    int v = list.get(i);
    if(i != pivotIndex && v > pivotValue) { sublist.add(v); }
  }
  return sublist;
}
```

RT(`allLessThanOrEqualTo`)? [***O(n)***]

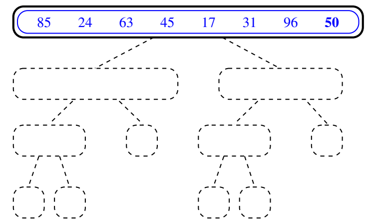
RT(`allLargerThan`)? [***O(n)***]

Recursion: Quick Sort in Java (2)

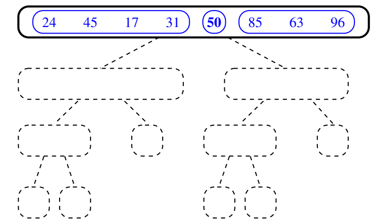
```
public List<Integer> sort(List<Integer> list) {
  List<Integer> sortedList;
  if(list.size() == 0) { sortedList = new ArrayList<>(); }
  else if(list.size() == 1) {
    sortedList = new ArrayList<>(); sortedList.add(list.get(0)); }
  else {
    int pivotIndex = list.size() - 1;
    int pivotValue = list.get(pivotIndex);
    List<Integer> left = allLessThanOrEqualTo (pivotIndex, list);
    List<Integer> right = allLargerThan (pivotIndex, list);
    List<Integer> sortedLeft = sort(left);
    List<Integer> sortedRight = sort(right);
    sortedList = new ArrayList<>();
    sortedList.addAll(sortedLeft);
    sortedList.add(pivotValue);
    sortedList.addAll(sortedRight);
  }
  return sortedList;
}
```


Recursion: Quick Sort Example (1)

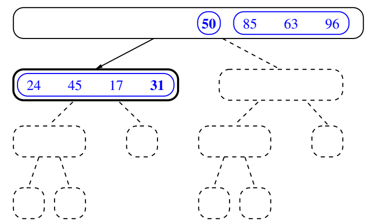
(1) Choose pivot 50 from list of size 8



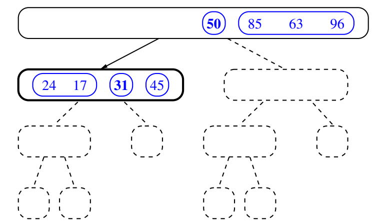
(2) Split w.r.t. the chosen pivot 50



(3) Recur on L of size 4, choose pivot 31

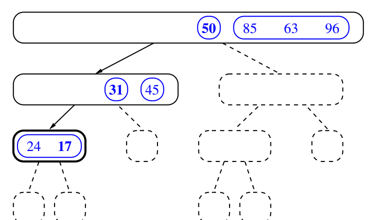


(4) Split w.r.t. the chosen pivot 31

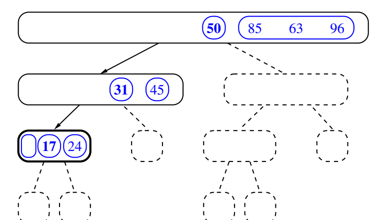


Recursion: Quick Sort Example (2)

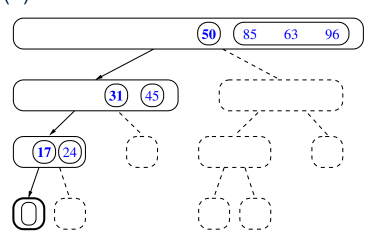
(5) Recur on L of size 2, choose pivot 17



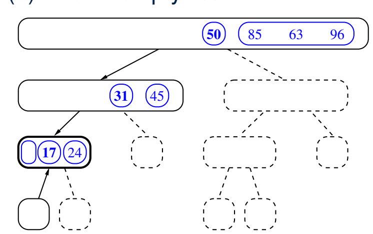
(6) Split w.r.t. the chosen pivot 17



(7) Recur on L of size 0

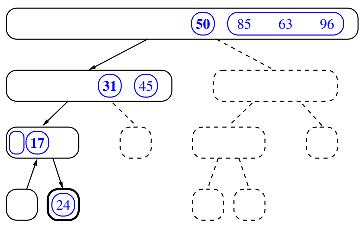


(8) *Return* empty list

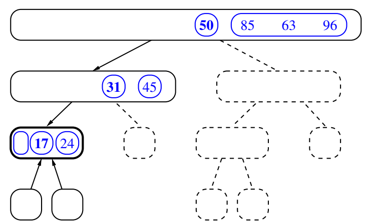


Recursion: Quick Sort Example (3)

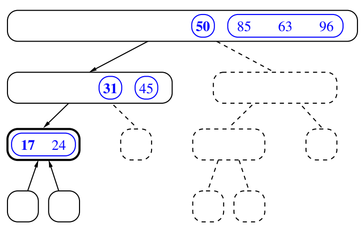
(9) Recur on R of size 1



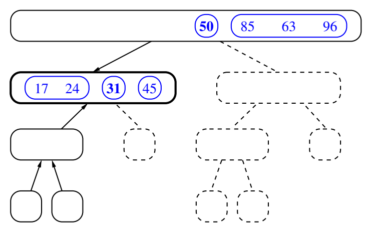
(10) *Return* singleton list ⟨24⟩



(11) Concatenate ⟨⟩, ⟨17⟩, and ⟨24⟩

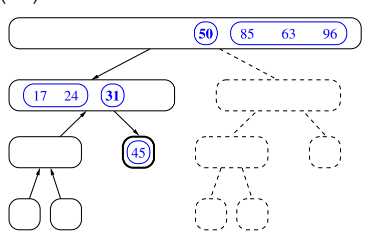


(12) *Return* concatenated list of size 2

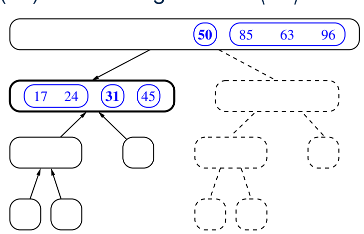


Recursion: Quick Sort Example (4)

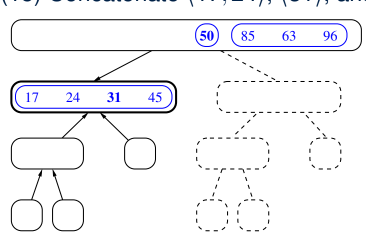
(13) Recur on R of size 1



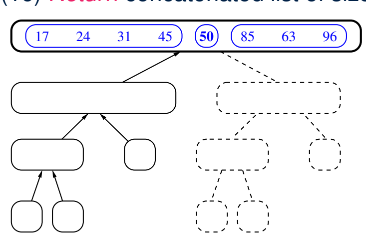
(14) *Return* singleton list ⟨45⟩



(15) Concatenate ⟨17, 24⟩, ⟨31⟩, and ⟨45⟩

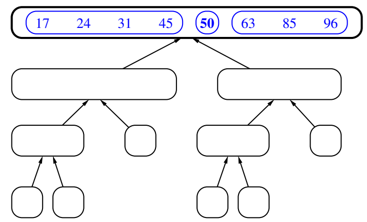


(16) *Return* concatenated list of size 4

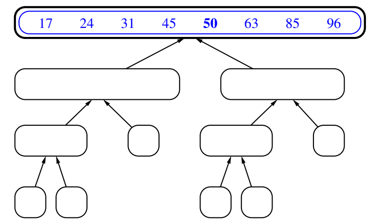


Recursion: Quick Sort Example (5)

(15) Recur on R of size 3



(16) *Return* sorted list of size 3

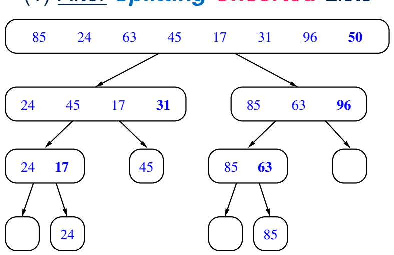


(17) Concatenate ⟨17, 24, 31, 45⟩, ⟨50⟩, and ⟨63, 85, 96⟩, then *return*

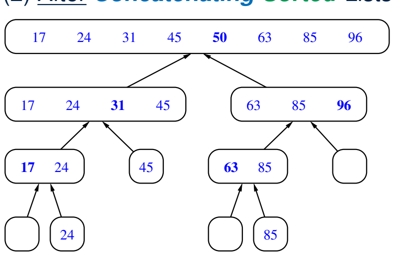
Recursion: Quick Sort Example (6)

Let's visualize the two ***critical phases*** of *quick sort* :

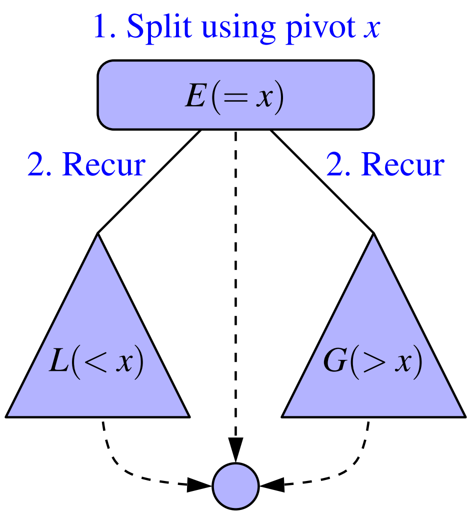
(1) After ***Splitting Unsorted*** Lists



(2) After ***Concatenating Sorted*** Lists



Recursion: Quick Sort Running Time (1)



Recursion: Quick Sort Running Time (2)

- **Base Case 1**: Empty list $\longrightarrow$ Automatically sorted. [***O(1)***]
- **Base Case 2**: List of size 1 $\longrightarrow$ Automatically sorted. [***O(1)***]
- **Recursive Case**: List of size ≥ 2 $\longrightarrow$
  1. Choose a ***pivot*** element (e.g., rightmost element) [***O(1)***]
  2. ***Split*** the list into two (***unsorted***) halves: ***L*** and ***R***, s.t.:
     All elements in ***L*** are less than or equal to ($\leq$) the ***pivot***. [***O(n)***]
     All elements in ***R*** are greater than ($>$) the ***pivot***. [***O(n)***]
  3. **Recursively** *sort* ***L*** and ***R***: ***sortedL*** and ***sortedR***;
     Q. # times to ***split*** until ***L*** and ***R*** have size 0 or 1?
     A. ***O(log n)*** [**if** pivots chosen are close to ***median values***]
  4. Return the ***concatenation*** of: ***sortedL*** + ***pivot*** + ***sortedR***. [***O(1)***]
-

Running Time of Quick Sort

$$
\begin{aligned}
&= (\textbf{RT}\ \texttt{each RC}) \times (\texttt{\#}\ \textbf{RC}\texttt{s}) \\
&= (\textbf{RT}\ \texttt{splitting into}\ \boldsymbol{L}\ \texttt{and}\ \boldsymbol{R}) \times (\texttt{\# splits until bases}) \\
&= O(n \cdot \log n)
\end{aligned}
$$

Recursion: Quick Sort Running Time (3)

- We define ***T(n)*** as the ***running time function*** of a *quick sort*, where ***n*** is the size of the input array.
- ***Worst Case***
  - If the pivot is s.t. the two sub-arrays are "***unbalanced***" in sizes:
    e.g., rightmost element in a reverse-sorted array
    ("***unbalanced***" splits/partitions: 0 vs. $n-1$ elements)

$$
\begin{cases}
T(0) = 1 \\
T(1) = 1 \\
T(n) = T(n-1) + n \quad \texttt{where}\ n \geq 2
\end{cases}
$$

  - As efficient as Selection/Insertion Sorts: ***O(n²)*** [EXERCISE]
- ***Best Case***
  If the pivot is s.t. it is close to the ***median*** value:

$$
\begin{cases}
T(0) = 1 \\
T(1) = 1 \\
T(n) = 2 \cdot T(\frac{n}{2}) + n \quad \texttt{where}\ n \geq 2
\end{cases}
$$

  - As efficient as Merge Sort: ***O(n · log n)***
  - Even with partitions such as $\frac{n}{10}$ vs. $\frac{9 \cdot n}{10}$ elements, RT remains ***O(n · log n)***.

Beyond this lecture ...

- Notes on Recursion:
  https://www.eecs.yorku.ca/~jackie/teaching/lectures/2021/F/EECS2030/notes/EECS2030_F21_Notes_Recursion.pdf
- The *best* approach to learning about recursion is via a functional programming language:
  Haskell Tutorial: https://www.haskell.org/tutorial/

Index (1)

Index (2)

Index (3)

General Trees and Binary Trees

EECS2011 N & Z:
Fundamentals of Data Structures
Winter 2022

CHEN-WEI WANG

Learning Outcomes of this Lecture

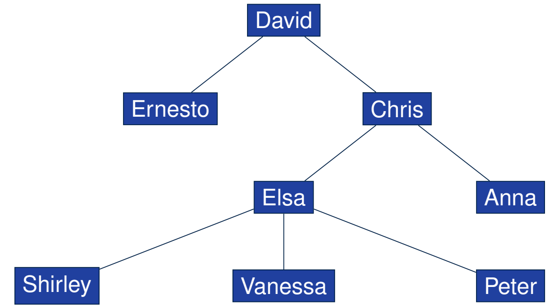
This module is designed to help you understand:

- ***Linar*** DS (e.g., arrays, LLs) vs. ***Non-Linear*** DS (e.g., trees)
- Terminologies: ***General*** Trees vs. ***Binary*** Trees
- Implementation of a ***Generic*** Tree
- Mathematical Properties of Binary Trees
- Tree ***Traversals***

General Trees

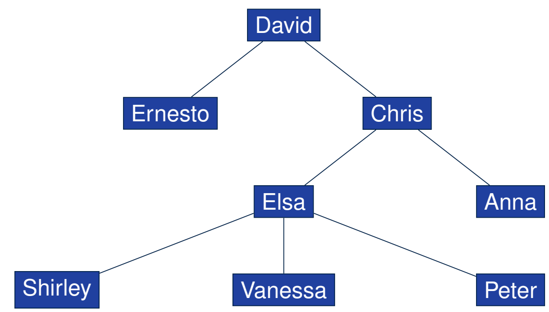
- A ***linear*** data structure is a sequence, where stored objects can be related via notions of "**predecessor**" and "**successor**".
  - e.g., arrays
  - e.g., Singly-Linked Lists (SLLs)
  - e.g., Doubly-Linked Lists (DLLs)
- The *Tree ADT* is a ***non-linear*** collection of nodes/positions.
  - Each node stores some data object.
  - ***Nodes*** in a ***tree*** are organized into ***levels***: some nodes are "**above**" others, and some are "**below**" others.
  - Think of a ***tree*** forming a ***hierarchy*** among the stored ***nodes***.
- Terminology of the *Tree ADT* borrows that of ***family trees***:
  - e.g., root
  - e.g., parents, siblings, children
  - e.g., ancestors, descendants

General Trees: Terminology (1)



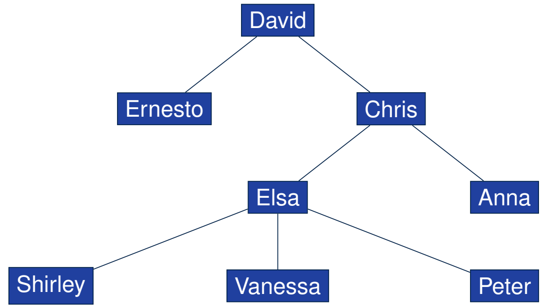
- ***top*** element of the tree [***root of tree***]
  e.g., root of the above family tree: David
- **the** node ***immediately above*** node *n* [***parent of*** *n*]
  e.g., parent of Vanessa: Elsa
- **all** nodes ***immediately below*** node *n* [***children of*** *n*]
  e.g., children of Elsa: Shirley, Vanessa, and Peter
  e.g., children of Ernesto: $\varnothing$

General Trees: Terminology (2)



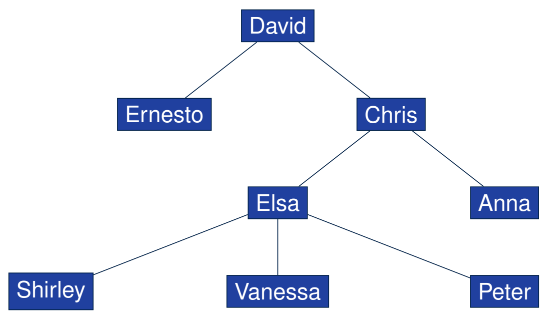
- Union of *n*, *n*'s ***parent***, *n*'s ***grand parent***, ..., ***root*** [*n's* ***ancestors***]
  e.g., ancestors of Vanessa: Vanessa, Elsa, Chris, and David
  e.g., ancestors of David: David
- Union of *n*, *n*'s ***children***, *n*'s ***grand children***, ... [*n's* ***descendants***]
  e.g., descendants of Vanessa: Vanessa
  e.g., descendants of David: the entire family tree
- By the above definitions, a ***node*** is both its ***ancestor*** and ***descendant***.

General Trees: Terminology (3)



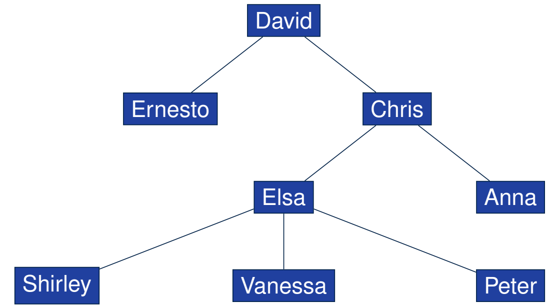
- **all** nodes with the ***same parent*** as *n*'s [***siblings of node*** *n*]
  e.g., siblings of Vanessa: Shirley and Peter
- **the** tree formed by ***descendants*** of *n* [***subtree rooted at*** *n*]
- nodes with ***no children*** [***external nodes (leaves)***]
  e.g., leaves of the above tree: Ernesto, Anna, Shirley, Vanessa, Peter
- nodes with ***at least one child*** [***internal nodes***]
  e.g., non-leaves of the above tree: David, Chris, Elsa

General Trees: Terminology (4)



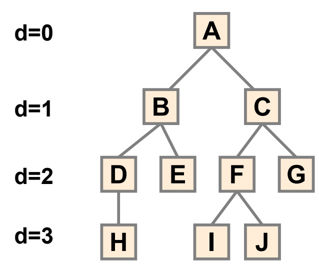
- a pair of ***parent*** and ***child*** nodes [***an edge of tree***]
  e.g., (David, Chris), (Chris, Elsa), (Elsa, Peter) are three edges
- a sequence of nodes where any two consecutive nodes form an ***edge*** [***a path of tree***]
  e.g., ⟨ David, Chris, Elsa, Peter ⟩ is a path
  e.g., Elsa's ***ancestor path***: ⟨ Elsa, Chris, David ⟩

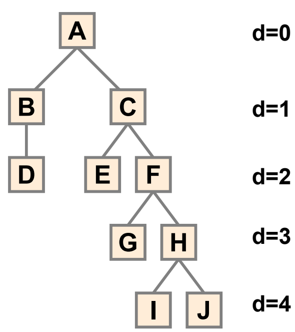
General Trees: Terminology (5)



- number of ***edges*** from the ***root*** to node *n* [***depth of*** *n*]
  alternatively: number of *n*'s ***ancestors*** of *n* minus one
  e.g., depth of David (root): 0
  e.g., depth of Shirley, Vanessa, and Peter: 3
- ***maximum depth*** among all nodes [***height of tree***]
  e.g., Shirley, Vanessa, and Peter have the maximum depth

General Trees: Example Node Depths





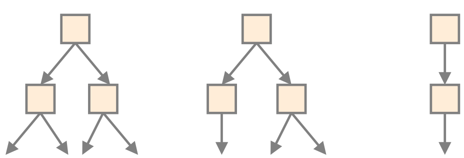
General Tree: Definition

A *tree T* is a set of ***nodes*** satisfying **parent-child** properties:

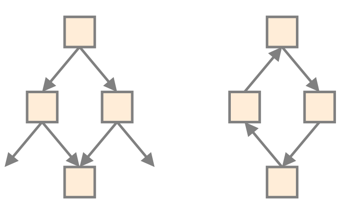
1. If *T* is ***empty***, then it does not contain any nodes.
2. If *T* is ***nonempty***, then:
   - *T* contains at least its ***root*** (a special node with no parent).
   - Each node *n* of *T* that is not the root has ***a unique parent node*** *w*.
   - Given two nodes *n* and *w*,
     if *w* is the ***parent*** of *n*, then **symmetrically**, *n* is one of *w*'s ***children***.

General Tree: Important Characteristics

There is a ***single, unique path*** from the ***root*** to any particular node in the same tree.



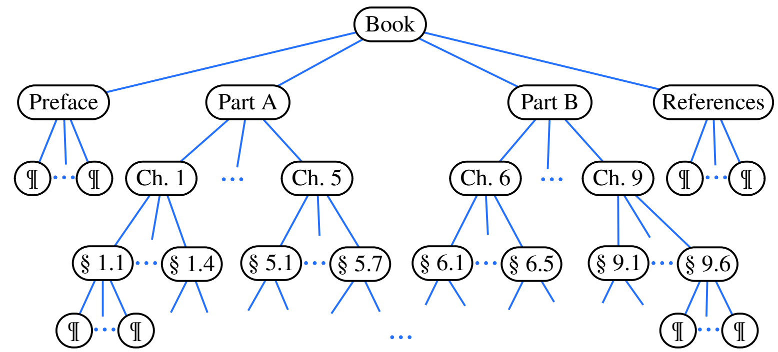
legal tree organization



illegal tree organization (nontrees)

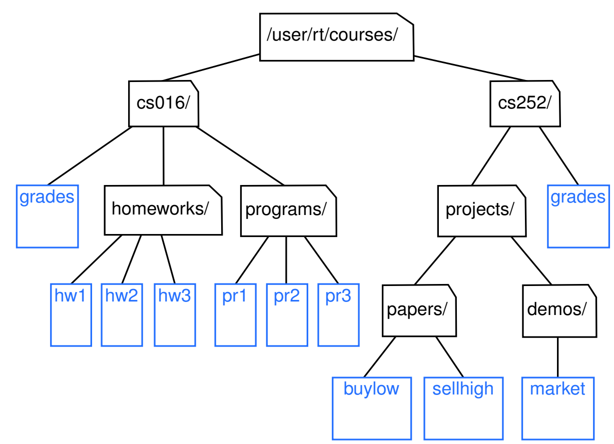
General Trees: Ordered Trees

A tree is *ordered* if there is a meaningful ***linear order*** among the ***children*** of each ***internal node***.



General Trees: Unordered Trees

A tree is *unordered* if the order among the ***children*** of each ***internal node*** does **not** matter.



Implementation: Generic Tree Nodes (1)

```
1  public class TreeNode<E> {
2    private E element; /* data object */
3    private TreeNode<E> parent; /* unique parent node */
4    private TreeNode<E>[] children; /* list of child nodes */
5
6    private final int MAX_NUM_CHILDREN = 10; /* fixed max */
7    private int noc; /* number of child nodes */
8
9    public TreeNode(E element) {
10     this.element = element;
11     this.parent = null;
12     this.children = (TreeNode<E>[])
13       Array.newInstance(this.getClass(), MAX_NUM_CHILDREN);
14     this.noc = 0;
15   }
16 ...
17 }
```

Replacing **L13** with the following results in a ***`ClassCastException`***:

```
this.children = (TreeNode<E>[]) new Object[MAX_NUM_CHILDREN];
```


Implementation: Generic Tree Nodes (2)

```
public class TreeNode<E> {
  private E element; /* data object */
  private TreeNode<E> parent; /* unique parent node */
  private TreeNode<E>[] children; /* list of child nodes */

  private final int MAX_NUM_CHILDREN = 10; /* fixed max */
  private int noc; /* number of child nodes */

  public E getElement() { ... }
  public TreeNode<E> getParent() { ... }
  public TreeNode<E>[] getChildren() { ... }

  public void setElement(E element) { ... }
  public void setParent(TreeNode<E> parent) { ... }
  public void addChild(TreeNode<E> child) { ... }
  public void removeChildAt(int i) { ... }
}
```

Exercise: Implement `void removeChildAt(int i)`.

Testing: Connected Tree Nodes

Constructing a *tree* is similar to constructing a *SLL*:

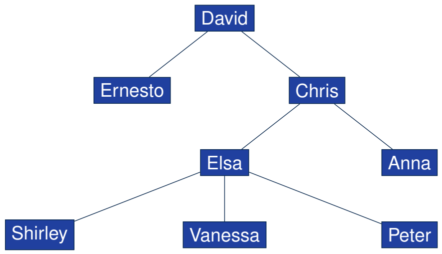
```
@Test
public void test_general_trees_construction() {
  TreeNode<String> agnarr = new TreeNode<>("Agnarr");
  TreeNode<String> elsa = new TreeNode<>("Elsa");
  TreeNode<String> anna = new TreeNode<>("Anna");

  agnarr.addChild(elsa);
  agnarr.addChild(anna);
  elsa.setParent(agnarr);
  anna.setParent(agnarr);

  assertNull(agnarr.getParent());
  assertTrue(agnarr == elsa.getParent());
  assertTrue(agnarr == anna.getParent());
  assertTrue(agnarr.getChildren().length == 2);
  assertTrue(agnarr.getChildren()[0] == elsa);
  assertTrue(agnarr.getChildren()[1] == anna);
}
```


Problem: Computing a Node's Depth

- Given a node *n*, its ***depth*** is defined as:
  - If *n* is the ***root***, then *n*'s depth is 0.
  - Otherwise, *n*'s ***depth*** is the ***depth*** of *n*'s parent plus one.
- Assuming under a ***generic*** class `TreeUtilities<E>`:

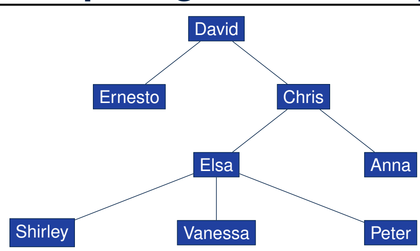
```
public int depth(TreeNode<E> n) {
  if(n.getParent() == null) {
    return 0;
  }
  else {
    return 1 + depth(n.getParent());
  }
}
```

Testing: Computing a Node's Depth




```java
@Test
public void test_general_trees_depths() {
  ... /* constructing a tree as shown above */
  TreeUtilities<String> u = new TreeUtilities<>();
  assertEquals(0, u.depth(david));
  assertEquals(1, u.depth(ernesto));
  assertEquals(1, u.depth(chris));
  assertEquals(2, u.depth(elsa));
  assertEquals(2, u.depth(anna));
  assertEquals(3, u.depth(shirley));
  assertEquals(3, u.depth(vanessa));
  assertEquals(3, u.depth(peter));
}
```


Unfolding: Computing a Node's Depth



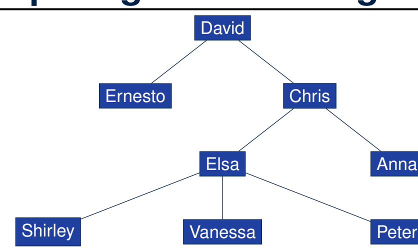
depth(`vanessa`)
= { `vanessa.getParent() == elsa` }
1 + ***depth***(`elsa`)
= { `elsa.getParent() == chris` }
1 + 1 + ***depth***(`chris`)
= { `chris.getParent() == david` }
1 + 1 + 1 + ***depth***(`David`)
= { `David is the root` }
1 + 1 + 1 + 0
= 3

Problem: Computing a Tree's Height

- Given node *n*, the ***height*** of subtree <u>rooted at *n*</u> is defined as:
  - If *n* is a ***leaf***, then the ***height*** of subtree rooted at *n* is 0.
  - Otherwise, the height of subtree rooted at *n* is one plus the ***maximum height*** of <u>all</u> subtrees rooted at *n*'s children.
- Assuming under a ***generic*** class `TreeUtilities<E>`:

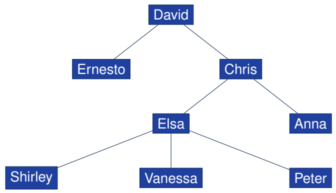
```java
public int height(TreeNode<E> n) {
  TreeNode<E>[] children = n.getChildren();
  if(children.length == 0) { return 0; }
  else {
    int max = 0;
    for(int i = 0; i < children.length; i ++) {
      int h = 1 + height(children[i]);
      max = h > max ? h : max;
    }
    return max;
  }
}
```


Testing: Computing a Tree's Height




```java
@Test
public void test_general_trees_heights() {
  ... /* constructing a tree as shown above */
  TreeUtilities<String> u = new TreeUtilities<>();
  /* internal nodes */
  assertEquals(3, u.height(david));
  assertEquals(2, u.height(chris));
  assertEquals(1, u.height(elsa));
  /* external nodes */
  assertEquals(0, u.height(ernesto));
  assertEquals(0, u.height(anna));
  assertEquals(0, u.height(shirley));
  assertEquals(0, u.height(vanessa));
  assertEquals(0, u.height(peter));
}
```

Unfolding: Computing a Tree's Height



$$
\begin{array}{ll}
& \textit{height}(\texttt{subtree rooted at chris}) \\
= & \{\ \textit{chris is not a leaf}\ \} \\
& MAX\begin{pmatrix} 1 + \textit{height}(\texttt{subtree rooted at elsa}), \\ 1 + \textit{height}(\texttt{subtree rooted at anna}) \end{pmatrix} \\
= & \{\ \textit{elsa is not a leaf, anna is a leaf}\ \} \\
& MAX\begin{pmatrix} 1 + MAX\begin{pmatrix} 1 + \textit{height}(\texttt{subtree rooted at shirley}), \\ 1 + \textit{height}(\texttt{subtree rooted at vanessa}), \\ 1 + \textit{height}(\texttt{subtree rooted at peter}) \end{pmatrix}, \\ 1 + 0 \end{pmatrix} \\
= & \{\ \textit{shirley, vanessa, and peter are all leaves}\ \} \\
& MAX\begin{pmatrix} 1 + MAX\begin{pmatrix} 1 + 0, \\ 1 + 0, \\ 1 + 0 \end{pmatrix}, \\ 1 + 0 \end{pmatrix} \\
= & 2
\end{array}
$$

Exercises on General Trees

- Implement and test the following ***recursive*** algorithm:

```
public TreeNode<E>[] ancestors(TreeNode<E> n)
```

  which returns the list of ***ancestors*** of a given node `n`.

- Implement and test the following ***recursive*** algorithm:

```
public TreeNode<E>[] descendants(TreeNode<E> n)
```

  which returns the list of ***descendants*** of a given node `n`.

Binary Trees (BTs): Definitions

A *binary tree (BT)* is an ***ordered tree*** satisfying the following:

1. Each node has **at most two** (≤ 2) children.
2. Each ***child node*** is labeled as either a ***left child*** or a ***right child***.
3. A ***left child*** precedes a ***right child***.

A *binary tree (BT)* is either:

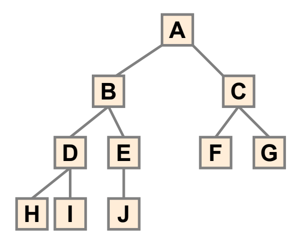
- An ***empty*** tree; or
- A ***nonempty*** tree with a ***root*** node *r* which has:
  - a *left subtree* rooted at its ***left child***, if any
  - a *right subtree* rooted at its ***right child***, if any

BT Terminology: LST vs. RST

For an ***internal*** node (with at least one child):

- Subtree rooted at its ***left child***, if any, is called ***left subtree***.
- Subtree rooted at its ***right child***, if any, is called ***right subtree***.

  e.g.,

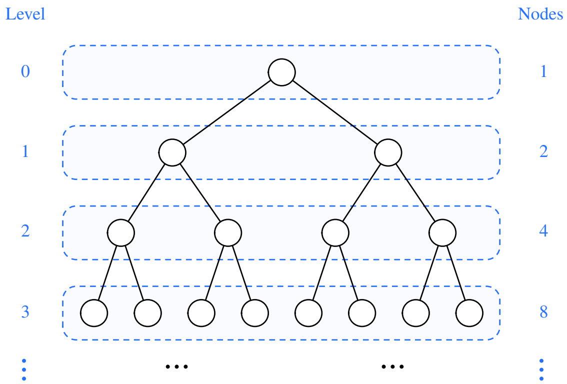


Node A has:

- a ***left subtree*** rooted at node B
- a ***right subtree*** rooted at node C

BT Terminology: Depths, Levels

The set of nodes with the same ***depth*** d are said to be at the same *level d*.



Background: Sum of Geometric Sequence

- Given a *geometric sequence* of n terms, where the initial term is a and the common factor is r, the ***sum*** of all its terms is:

$$\sum_{k=0}^{n-1}(a \cdot r^k) = a \cdot r^0 + a \cdot r^1 + a \cdot r^2 + \cdots + a \cdot r^{n-1} = a \cdot \left(\frac{r^n - 1}{r - 1}\right)$$

[See *here* to see how the formula is derived.]

- For the purpose of ***binary trees***, ***maximum*** numbers of nodes at all ***levels*** form a *geometric sequence*:
  - $a = 1$ [the ***root*** at **Level 0**]
  - $r = 2$ [≤ 2 children for each ***internal*** node]
  - e.g., *Max* total # of nodes at *levels* 0 to 4 = $1 + 2 + 4 + 8 + 16 = 1 \cdot (\frac{2^5 - 1}{2 - 1}) = 31$

BT Properties: Max # Nodes at Levels

Given a *binary tree* with ***height*** h:

- At each level:
  - ***Maximum*** number of nodes at ***Level 0***: $2^0 = 1$
  - ***Maximum*** number of nodes at ***Level 1***: $2^1 = 2$
  - ***Maximum*** number of nodes at ***Level 2***: $2^2 = 4$

  . . .
  - ***Maximum*** number of nodes at ***Level*** h: 2^h
- Summing all levels:

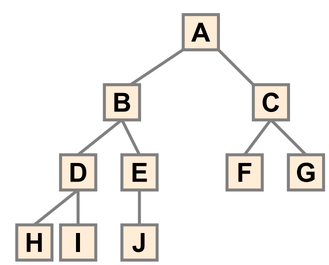
  Maximum total number of nodes:

$$\underbrace{2^0 + 2^1 + 2^2 + \cdots + 2^h}_{h+1 \text{ terms}} = 1 \cdot \left(\frac{2^{h+1} - 1}{2 - 1}\right) = 2^{h+1} - 1$$

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? $(2^h - 1) + 1 = 2^h$

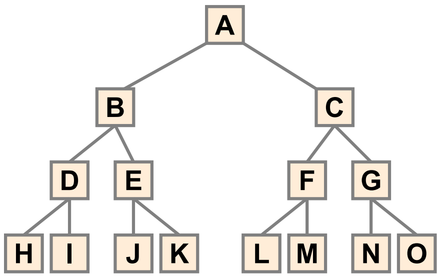
Q2: ***Maximum*** # of nodes of a ***complete*** BT? $2^{h+1} - 1$

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? $2^{h+1} - 1$

BT Properties: Bounding # of Nodes

Given a *binary tree* with ***height*** h, the ***number of nodes*** n is bounded as:

$$h + 1 \leq n \leq 2^{h+1} - 1$$

- Shape of BT with ***minimum*** # of nodes?
  A "one-path" tree (each ***internal node*** has exactly one child)
- Shape of BT with ***maximum*** # of nodes?
  A tree completely filled at each level

BT Properties: Bounding Height of Tree

Given a *binary tree* with n ***nodes***, the ***height*** h is bounded as:

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

- Shape of BT with ***minimum*** height?
  A tree completely filled at each level
  $$\begin{array}{lll} & n & = 2^{h+1} - 1 \\ \iff & n + 1 & = 2^{h+1} \\ \iff & log(n+1) & = h + 1 \\ \iff & log(n+1) - 1 & = h \end{array}$$
- Shape of BT with ***maximum*** height?
  A "one-path" tree (each ***internal node*** has exactly one child)

BT Properties: Bounding # of Ext. Nodes

Given a *binary tree* with ***height*** h, the ***number of external nodes*** n_E is bounded as:

$$1 \leq n_E \leq 2^h$$

- Shape of BT with ***minimum*** # of external nodes?
  A tree with only one node (i.e., the ***root***)
- Shape of BT with ***maximum*** # of external nodes?
  A tree whose bottom level (with ***depth*** h) is completely filled

BT Properties: Bounding # of Int. Nodes

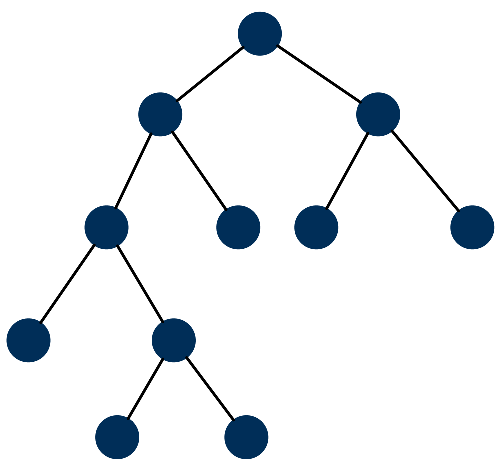
Given a *binary tree* with ***height*** h, the ***number of internal nodes*** n_I is bounded as:

$$h \leq n_I \leq 2^h - 1$$

- Shape of BT with ***minimum*** # of internal nodes?
  - Number of nodes in a "one-path" tree $(h + 1)$ minus one
  - That is, the "deepest" leaf node excluded
- Shape of BT with ***maximum*** # of internal nodes?
  - A tree whose $\leq h - 1$ ***levels*** are all completely filled
  - That is: $\underbrace{2^0 + 2^1 + \cdots + 2^{h-1}}_{n \text{ terms}} = 2^h - 1$

BT Terminology: Proper BT

A *binary tree* is ***proper*** if each ***internal node*** has two children.



BT Properties: #s of Ext. and Int. Nodes

Given a *binary tree* that is:

- ***nonempty*** and ***proper***
- with n_I ***internal nodes*** and n_E ***external nodes***

We can then expect that: $\boxed{\mathbf{n_E = n_I + 1}}$

Proof by *mathematical induction*:

- **Base Case:**

  A ***proper*** BT with only the ***root*** (an ***external node***): $\mathbf{n_E} = 1$ and $\mathbf{n_I} = 0$.

- **Inductive Case:**
  - Assume a ***proper*** BT with n nodes $(n > 1)$ with $\mathbf{n_I}$ ***internal nodes*** and $\mathbf{n_E}$ ***external nodes*** such that $\mathbf{n_E} = \mathbf{n_I} + 1$.
  - Only one way to create a **larger** BT (with $n + 2$ nodes) that is still ***proper*** (with $\mathbf{n_E'}$ ***external nodes*** and $\mathbf{n_I'}$ ***internal nodes***):

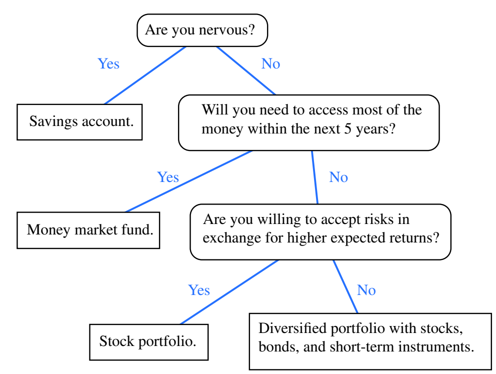
    Convert an external node into an ***internal*** node.

    $\mathbf{n_E'} = (n_E - 1) + 2 = n_E + 1 \wedge \mathbf{n_I'} = n_I + 1 \Rightarrow \mathbf{n_E'} = \mathbf{n_E'} + 1$

Binary Trees: Application (1)

A ***decision tree*** is a **proper** binary tree used to to express the decision-making process:

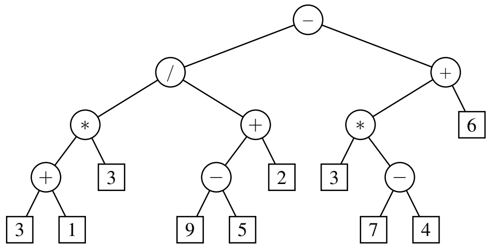
- Each ***internal node*** denotes a decision point: yes or no.
- Each ***external node*** denotes the consequence of a decision.



Binary Trees: Application (2)

An ***infix arithmetic expression*** can be represented using a binary tree:

- Each ***internal node*** denotes an operator (unary or binary).
- Each ***external node*** denotes an operand (i.e., a number).



- To evaluate the expression that is represented by a binary tree, certain *traversal* over the entire tree is required.

Tree Traversal Algorithms: Definition

- A *traversal* of a ***tree*** *T* systematically ***visits*** **all** *T*'s nodes.
- Visiting each ***node*** may be associated with an ***action***: e.g.,
  - ***Print*** the node element.
  - ***Determine*** if the node element satisfies certain property (e.g., positive, matching a key).
  - ***Accumulate*** the node element values for some global result.

Tree Traversal Algorithms: Common Types

Three common traversal orders:

- **Preorder**: Visit parent, then visit child subtrees.

```
preorder (n)
  visit and act on position n
  for child c: children(n) { preorder (c) }
```

- **Postorder**: Visit child subtrees, then visit parent.

```
postorder (n)
  for child c: children(n) { postorder (c) }
  visit and act on position n
```

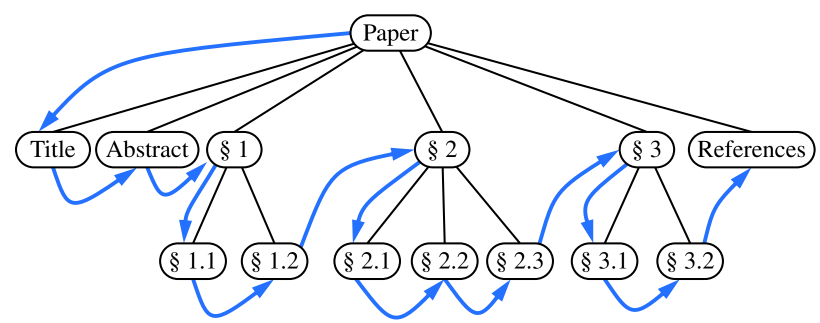
- **Inorder** (for ***BT***): Visit left subtree, then parent, then right subtree.

```
inorder (n)
  if (n has a left child lc) { inorder (lc) }
  visit and act on position n
  if (n has a right child rc) { inorder (rc) }
```


Tree Traversal Algorithms: Preorder

Preorder: Visit parent, then visit child subtrees.

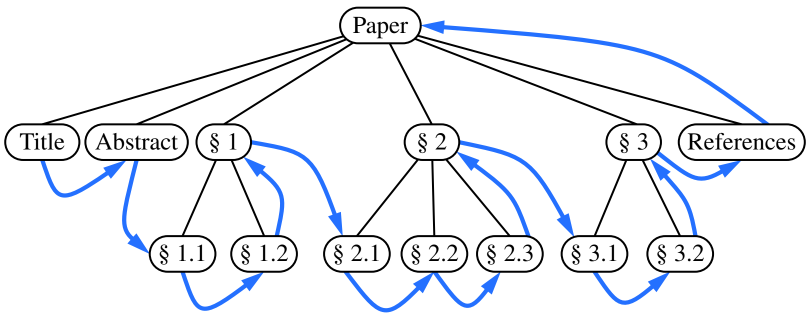
```
preorder (n)
  visit and act on position n
  for child c: children(n) { preorder (c) }
```



Tree Traversal Algorithms: Postorder

Postorder: Visit child subtrees, then visit parent.

```
postorder (n)
  for child c: children(n) { postorder (c) }
  visit and act on position n
```

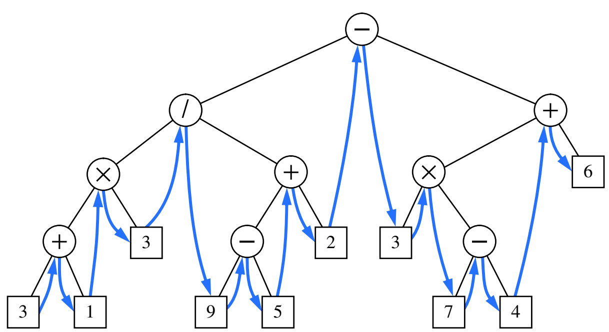


Index (1)

Tree Traversal Algorithms: Inorder

Inorder (for BT): Visit left subtree, then parent, then right subtree.

```
inorder (n)
  if (n has a left child lc) { inorder (lc) }
  visit and act on position n
  if (n has a right child rc) { inorder (rc) }
```



Index (2)

Index (3)

Binary Search Trees

EECS2011 N & Z:
Fundamentals of Data Structures
Winter 2022

CHEN-WEI WANG

Index (4)

Learning Outcomes of this Lecture

This module is designed to help you understand:

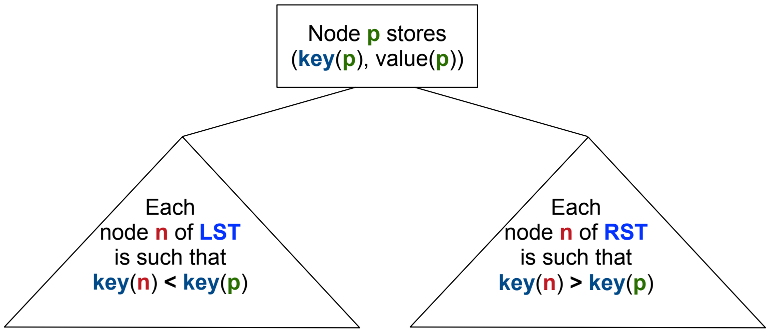
- ***Binary Search Trees* (*BSTs*)** = ***BTs*** + ***Search Property***
- Implementing a ***Generic*** BST in Java
- BST Operations:
  - ***Searching***: Implementation, Visualization, RT
  - ***Insertion***: (Sketch of) Implementation, Visualization, RT
  - ***Deletion***: (Sketch of) Implementation, Visualization, RT

Binary Search Tree: Recursive Definition

A *Binary Search Tree* (*BST*) is a ***BT*** satisfying the ***search property***:

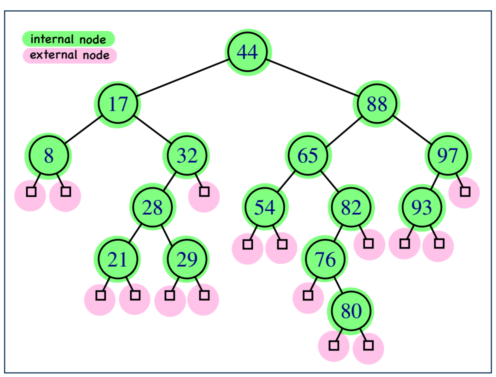
Each ***internal node*** *p* stores an ***entry***, a key-value pair (k, v), such that:

- For each node ***n*** in the ***LST*** of ***p***: $key(n) < key(p)$
- For each node ***n*** in the ***RST*** of ***p***: $key(n) > key(p)$



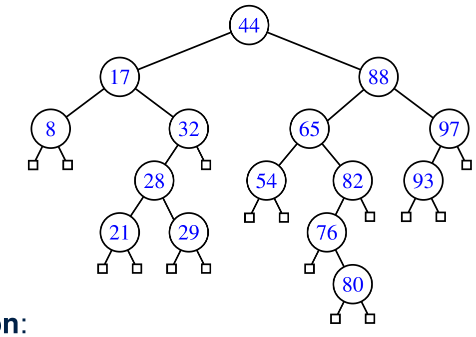
BST: Internal Nodes vs. External Nodes

- We store key-value pairs only in ***internal nodes***.
- Recall how we treat ***header*** and ***trailer*** in a DLL.
- We treat ***external nodes*** as ***sentinels***, in order to **simplify** the ***coding logic*** of BST algorithms.



BST: Sorting Property

- An ***in-order traversal*** of a *BST* will result in a sequence of nodes whose ***keys*** are arranged in an ***ascending order***.
- Unless necessary, we may only show ***keys*** in BST nodes:



Justification:

- ***In-Order Traversal***: Visit ***LST***, then ***root***, then ***RST***.
- ***Search Property of BST***: keys in ***LST***/***RST*** $<$ / $>$ ***root***'s key

Implementation: Generic BST Nodes

```
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) { ... }
}
```

Implementation: BST Utilities – Traversal

```
import java.util.ArrayList;
public class BSTUtilities<E> {
  public ArrayList<BSTNode<E>> inOrderTraversal(BSTNode<E> root) {
    ArrayList<BSTNode<E>> result = null;
    if(root.isInternal()) {
      result = new ArrayList<>();

      if(root.getLeft().isInternal) {
        result.addAll(inOrderTraversal(root.getLeft()));
      }

      result.add(root);

      if(root.getRight().isInternal) {
        result.addAll(inOrderTraversal(root.getRight()));
      }
    }
    return result;
  }
}
```


Testing: Connected BST Nodes

Constructing a *BST* is similar to constructing a *General Tree*:

```
@Test
public void test_binary_search_trees_construction() {
  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<>();
  ArrayList<BSTNode<String>> inOrderList = u.inOrderTraversal(n28);
  assertTrue(inOrderList.size() == 3);
  assertEquals(21, inOrderList.get(0).getKey());
  assertEquals("mark", inOrderList.get(0).getValue());
  assertEquals(28, inOrderList.get(1).getKey());
  assertEquals("alan", inOrderList.get(1).getValue());
  assertEquals(35, inOrderList.get(2).getKey());
  assertEquals("tom", inOrderList.get(2).getValue());
}
```

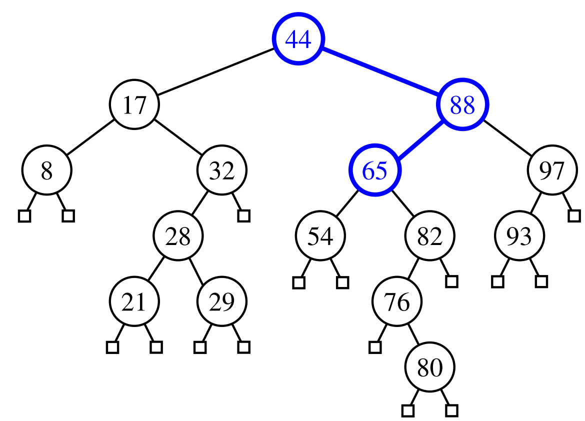

Implementing BST Operation: Searching

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

```
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)

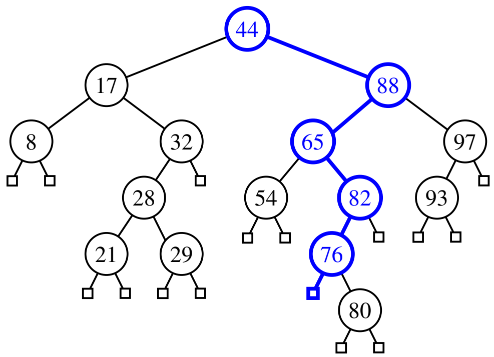
A ***successful*** search for ***key 65***:



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

Visualizing BST Operation: Searching (2)

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



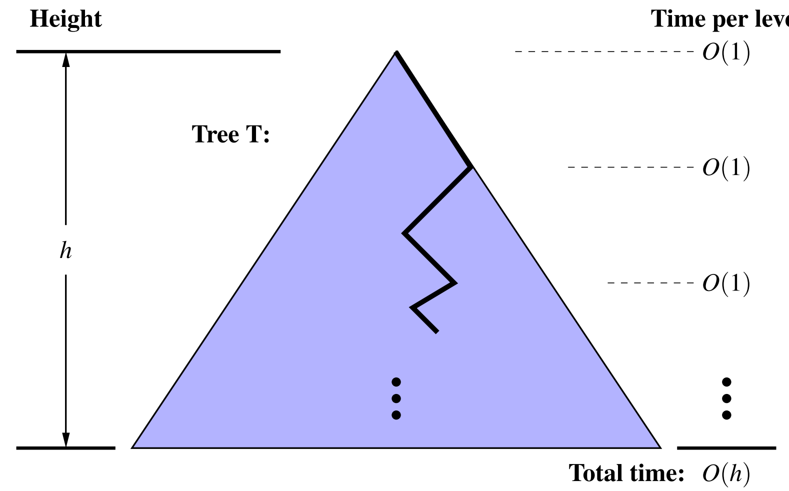
The ***external, left child node*** of the ***internal node*** storing ***key 76*** is **returned**.

- *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:

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

  - ***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

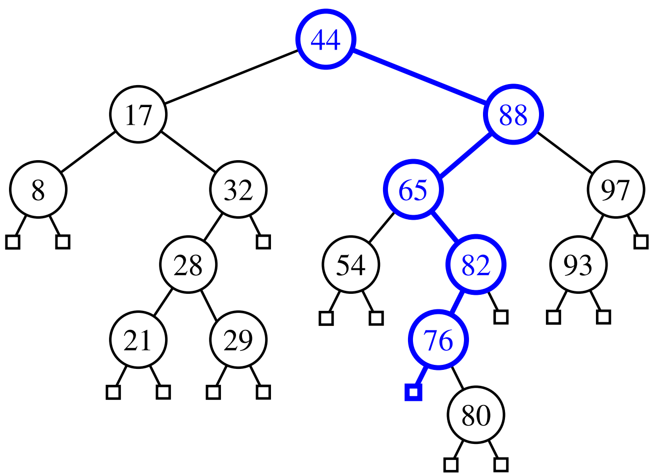
To *insert* an ***entry*** (with **key** k & **value** v) into a BST rooted at ***node*** n:

- 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? [$O(h)$]

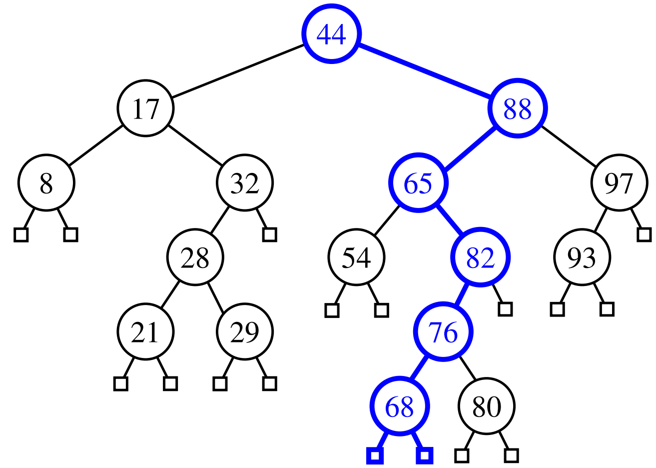
Visualizing BST Operation: Insertion (1)

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:

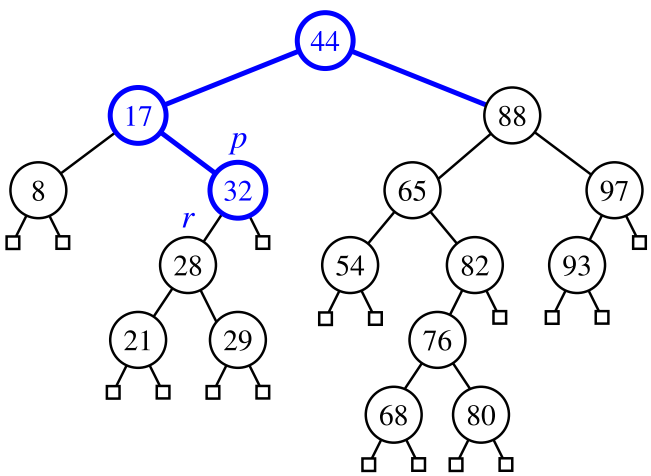
Let node ***p*** be the return value from `search(n, k)`.

- **Case 1**: Node ***p*** is ***external***.
  k is not an existing key $\Rightarrow$ Nothing to remove
- **Case 2**: Both of node ***p***'s child nodes are ***external***.
  No "orphan" subtrees to be handled $\Rightarrow$ Remove ***p*** [Still BST?]
- **Case 3**: One of the node ***p***'s children, say r, is ***internal***.
  - r's sibling is ***external*** $\Rightarrow$ 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***.
    $\Rightarrow$ 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*** $\Rightarrow$ Remove r as in **Case 2**.
    - An ***external, RC*** & an ***internal LC*** $\Rightarrow$ Remove r as in **Case 3**.

Running time? [$O(h)$]

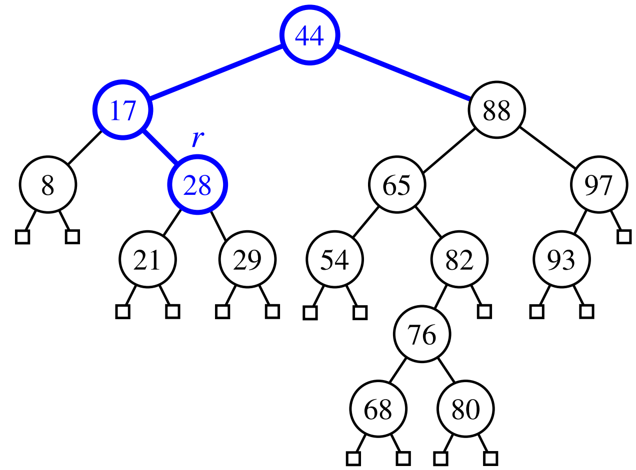
Visualizing BST Operation: Deletion (1.1)

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



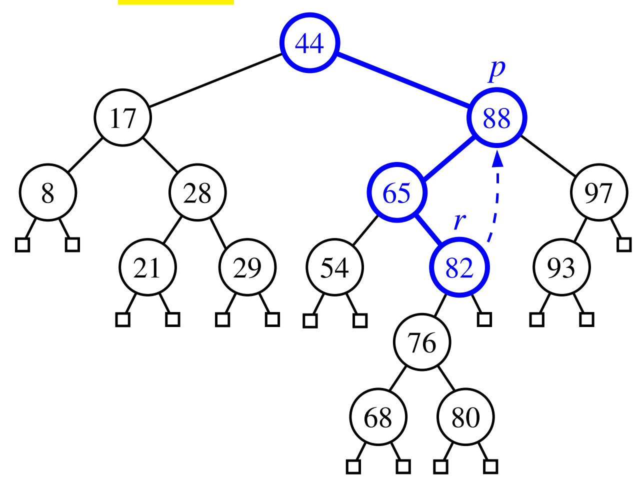
Visualizing BST Operation: Deletion (1.2)

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



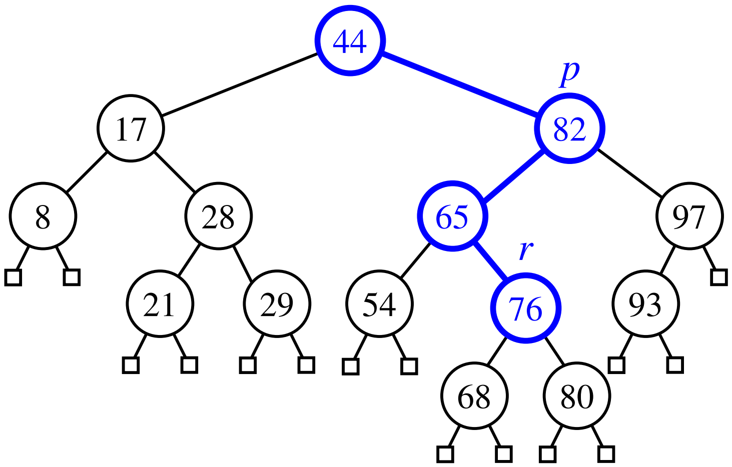
Visualizing BST Operation: Deletion (2.1)

(**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.

Index (1)

Index (2)

Index (3)

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
- Performing ***Rotations*** to Restore Tree ***Balance***

Balanced Binary Search Trees

EECS2011 N & Z:
Fundamentals of Data Structures
Winter 2022

CHEN-WEI WANG

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***.

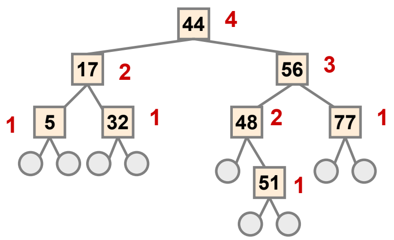
Balanced Binary Search Trees: Definition

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

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

- A ***balanced*** BST T satisfies the ***height-balance property*** :

  For every ***internal node*** n, ***heights*** of n's child nodes differ $\leq$ *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? ✓

After Insertions: Trinode Restructuring via Rotation(s)

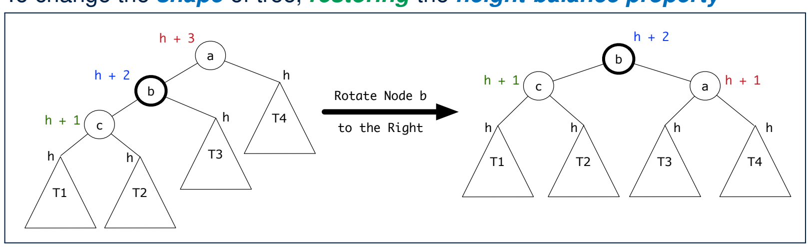
After ***inserting*** a new node n:

- **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

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***

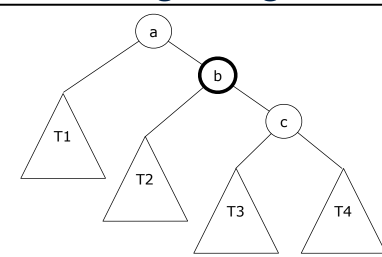


Q. An ***in-order traversal*** on the resulting tree?
A. Still produces a sequence of ***sorted keys*** $\langle T_1,\ c,\ T_2,\ b,\ T_3,\ a,\ T_4 \rangle$

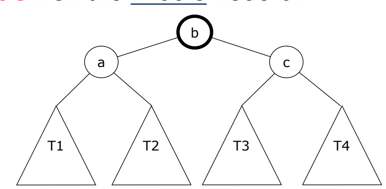
- After ***rotating*** node b to the right:
  - Heights of ***descendants*** (b, c, T_1, T_2, T_3) and ***sibling*** (T_4) stay ***unchanged***.
  - Height of ***parent*** (a) is ***decreased by 1***.

    $\Rightarrow$ ***Balance*** of node a was ***restored*** by the ***rotation***.

Trinode Restructuring: Single, Left Rotation



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

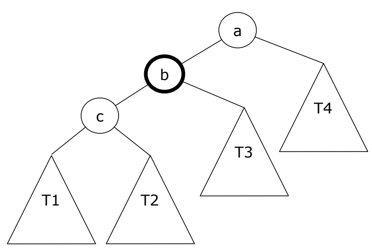


BST property maintained? $\langle T_1,\ a,\ T_2,\ b,\ T_3,\ c,\ T_4 \rangle$

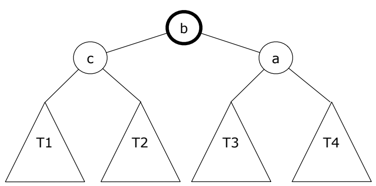
Left Rotation

- ***Insert*** the following sequence of nodes into an <u>empty</u> BST:
  $$\langle 44, 17, 78, 32, 50, 88, 95 \rangle$$
- Is the BST now ***balanced***?
- ***Insert*** <u>100</u> 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 <u>middle</u> node b:

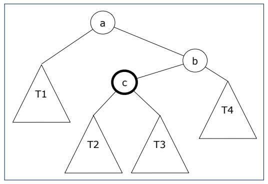


BST property maintained? $\langle T_1,\ a,\ T_2,\ b,\ T_3,\ c,\ T_4 \rangle$

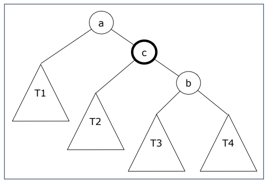
Right Rotation

- ***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*** <u>46</u> into the BST.
- Is the BST still ***balanced***?
- Perform a ***right rotation*** on the appropriate node.
- Is the BST again ***balanced***?

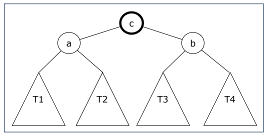
Trinode Restructuring: Double, R-L Rotations



<u>Perform</u> a ***Right Rotation*** on Node *c*



<u>Perform</u> a ***Left Rotation*** on Node *c*



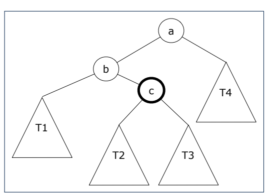
<u>After</u> Right-Left Rotations

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

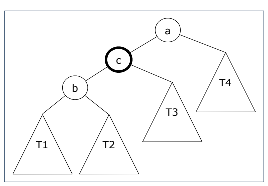
R-L Rotations

- ***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***?

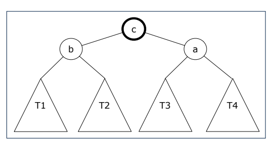
Trinode Restructuring: Double, L-R Rotations



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 <u>empty</u> BST:

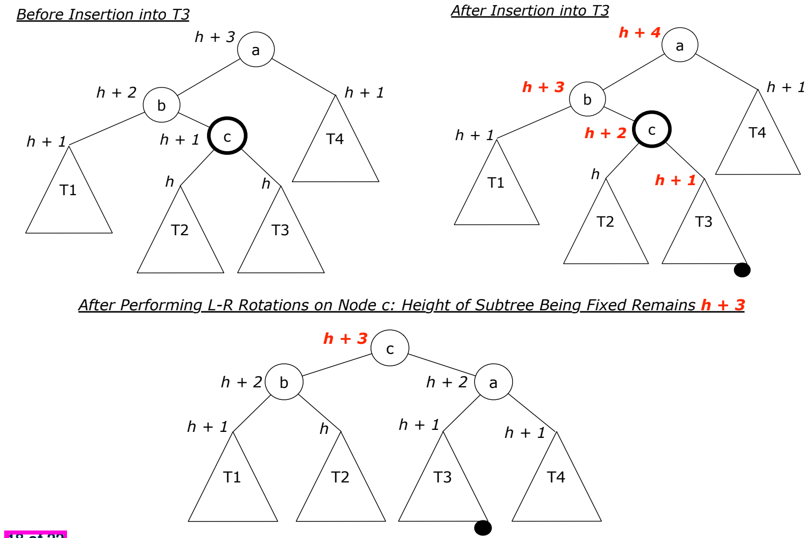
  $\langle 44, 17, 62, 32, 50, 78, 48, 54, 88 \rangle$
- Is the BST now ***balanced***?
- ***Delete*** <u>32</u> from the BST.
- Is the BST still ***balanced***?
- Perform a ***left rotation*** on the appropriate node.
- Is the BST again ***balanced***?

Multiple Trinode Restructuring Steps

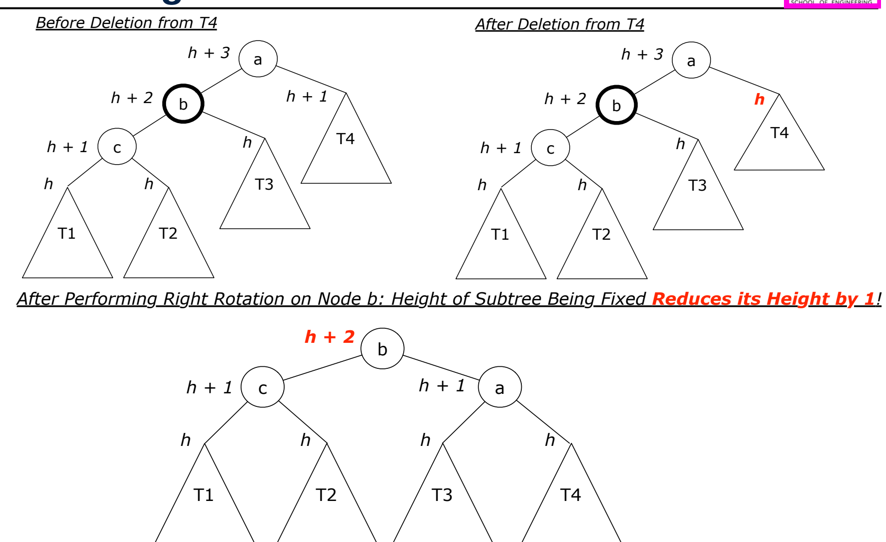
- ***Insert*** the following sequence of nodes into an <u>empty</u> BST:

  $\langle 50, 25, 10, 30, 5, 15, 27, 1, 75, 60, 80, 55 \rangle$
- Is the BST now ***balanced***?
- ***Delete*** <u>80</u> 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



Restoring Balance from Deletions



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)

Priority Queues, Heaps, and Heap Sort

YORK UNIVERSITÉ UNIVERSITY

EECS2011 N & Z:
Fundamentals of Data Structures
Winter 2022

CHEN-WEI WANG

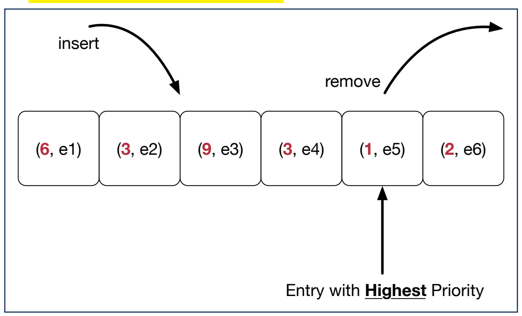
Learning Outcomes of this Lecture

This module is designed to help you understand:

- The ***Priority Queue*** (***PQ***) ADT
- Time Complexities of ***List***-Based ***PQ***
- The ***Heap*** Data Structure (Properties & Operations)
- ***Heap Sort***
- Time Complexities of ***Heap***-Based ***PQ***
- ***Heap*** Construction Methods: Top-Down vs. Bottom-Up
- ***Array***-Based Representation of a ***Heap***

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.

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]

Two List-Based Implementations of a PQ

Consider two strategies for implementing a *PQ*, where we maintain:

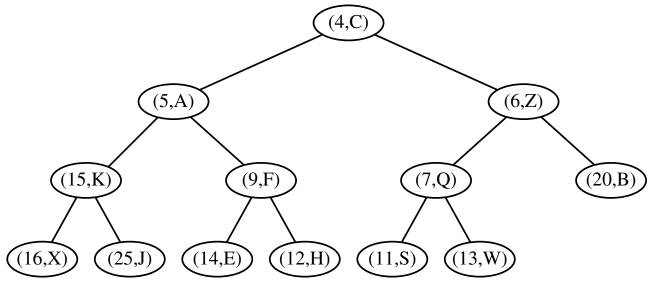
1. A list ***always sorted*** in a non-descending order [≈ INSERTIONSORT]
2. An ***unsorted*** list [≈ SELECTIONSORT]

PQ Method	List Method			
	SORTED LIST		UNSORTED LIST	
size	list.size *O(1)*			
isEmpty	list.isEmpty *O(1)*			
min	list.first	*O(1)*	search min	*O(n)*
insert	insert to "right" spot	*O(n)*	insert to front	*O(1)*
removeMin	list.removeFirst	*O(1)*	search min and remove	*O(n)*

Heaps

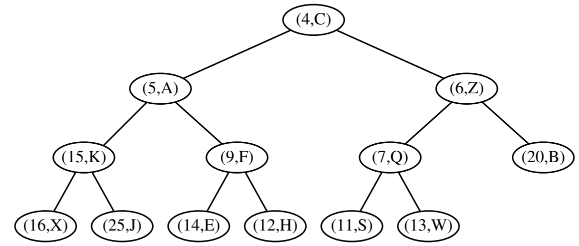
A **heap** is a ***binary tree*** which:

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



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

Heap Property 2: 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? $\lfloor \log_2 n \rfloor$

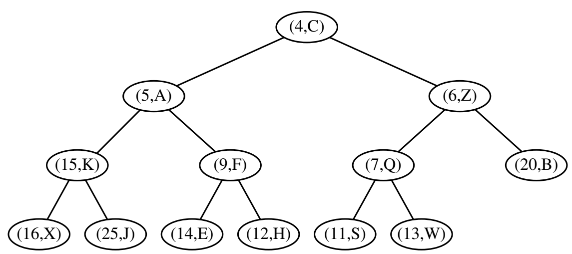
Q. # of nodes from Level 0 through Level $h - 1$? $2^h - 1$

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 1: Relational



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

- Every node ***n*** (other than the root) is s.t. ***key***(***n***) $\geq$ ***key***(**parent**(***n***))
  - $\Rightarrow$ ***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.
  - $\Rightarrow$ The ***minimal key*** is stored in the ***root***.

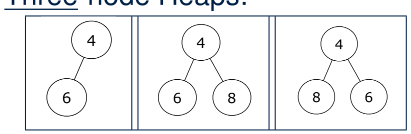
    e.g., Root $(4, C)$ stores the minimal key 4.
- ***Keys*** of nodes from ***different subtrees*** are **not** constrained at all.

  e.g., For node $(5, A)$, key of its ***LST***'s root (15) is not minimal for its ***RST***.

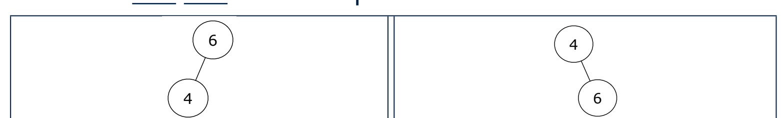
Heaps: More Examples

- 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 **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 $key(p) \le key(c)$, then **HOP** is restored.
       Else, swap nodes ***c*** and ***p***. ["upwards" along ***n***'s ***ancestor path***]

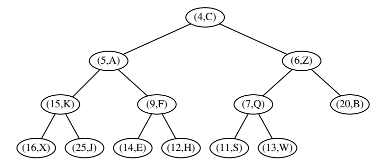
Running Time?

- 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.

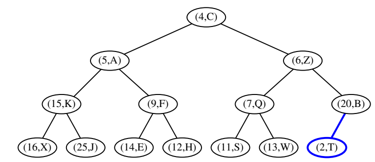
[*O(log n)*]

Updating a Heap: Insertion Example (1.1)

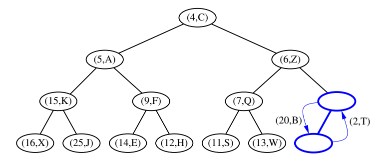
(0) A heap with height 3.



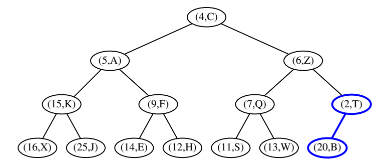
(1) Insert a new entry $(2, T)$ as the ***right-most*** node at Level 3. Perform *up-heap bubbling* from here.



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

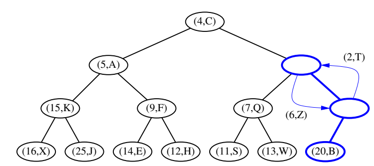


(3) After swap, entry $(2, T)$ prompted up.

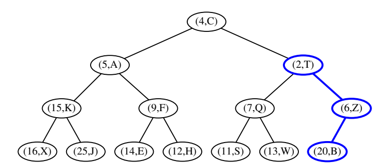


Updating a Heap: Insertion Example (1.2)

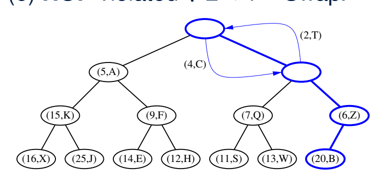
(4) **HOP** violated $\because 2 < 6 \therefore$ Swap.



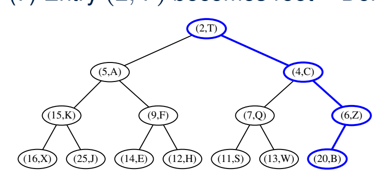
(5) After swap, entry $(2, T)$ prompted up.



(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 **temporarily** 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 and ***p*** is not external:
     - **2.2.1** **IF** ***p*** has no **right child**, let ***c*** be ***p***'s ***left child***.
       Else, let ***c*** be ***p***'s child with a ***smaller key value***.
     - **2.2.2** **If** $key(p) \leq key(c)$, then **HOP** is restored.
       Else, swap nodes ***p*** and ***c***. ["downwards" along a ***root-to-leaf path***]

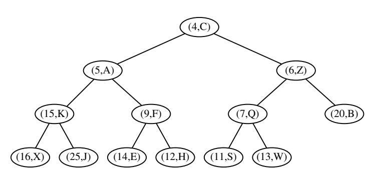
Running Time?

- 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.

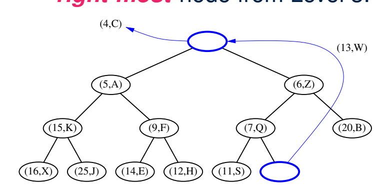
[*O(log n)*]

Updating a Heap: Deletion Example (1.1)

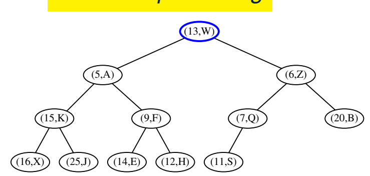
(0) Start with a heap with height 3.



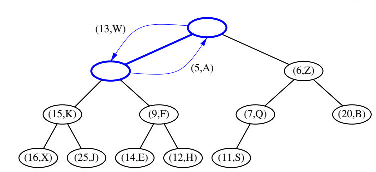
(1) Replace root with $(13, W)$ and delete ***right-most*** node from Level 3.



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

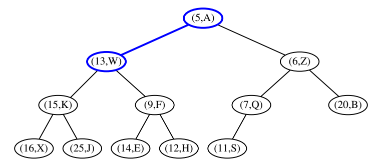


(3) Child with smaller key is $(5, A)$.
HOP violated $\because 13 > 5 \therefore$ Swap.

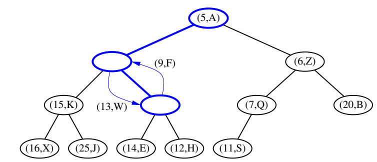


Updating a Heap: Deletion Example (1.2)

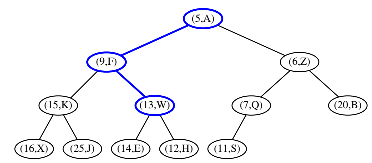
(4) After swap, entry $(13, W)$ demoted down.



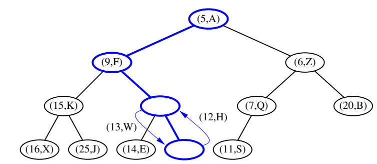
(5) Child with smaller key is $(9, F)$.
HOP violated $\because 13 > 9 \therefore$ Swap.



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

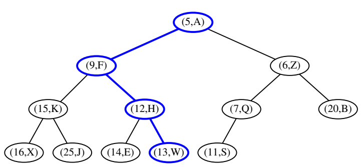


(7) Child with smaller key is $(12, H)$.
HOP violated $\because 13 > 12 \therefore$ Swap.



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)*

Top-Down Heap Construction: List of Entries is Not Known in Advance

Problem: Build a heap out of N entires, supplied one at a time.

- Initialize an ***empty heap h***. [*O(1)*]
- As each new entry $\mathbf{e} = (k, v)$ is supplied, **insert e** into ***h***.
  - Each insertion triggers an ***up-heap bubbling*** step, which takes *O(log n)* time. [$n = 0, 1, 2, \ldots, N - 1$]
  - There are N insertions.

$\therefore$ Running time is $O(N \cdot log\ N)$

Bottom-Up Heap Construction: List of Entries is Known in Advance

Problem: Build a heap out of N entires, supplied all at once.

- **Assume**: The resulting heap will be ***completely filled*** at **all** levels.
  $\Rightarrow N = 2^{h+1} - 1$ for some ***height*** $h \geq 1$ [$h = (log\ (N + 1)) - 1$]
- Perform the following steps called *Bottom-Up Heap Construction*:

  Step 1: Treat the first $\frac{N+1}{2^1}$ list entries as heap roots.
  $\therefore \frac{N+1}{2^1}$ heaps with height 0 and size $2^1 - 1$ constructed.

  Step 2: Treat the next $\frac{N+1}{2^2}$ list entries as heap roots.
  - Each ***root*** sets two heaps from **Step 1** as its ***LST*** and ***RST***.
  - Perform ***down-heap bubbling*** to restore **HOP** if necessary.

  $\therefore \frac{N+1}{2^2}$ heaps, each with height 1 and size $2^2 - 1$, constructed.

  ...

  Step $h + 1$: Treat next $\frac{N+1}{2^{h+1}} = \frac{(2^{h+1}-1)+1}{2^{h+1}} = 1$ list entry as heap root.
  - Each ***root*** sets two heaps from **Step h** as its ***LST*** and ***RST***.
  - Perform ***down-heap bubbling*** to restore **HOP** if necessary.

  $\therefore \frac{N+1}{2^{h+1}} = 1$ heap, each with height h and size $2^{h+1} - 1$, constructed.

Bottom-Up Heap Construction: Example (1.1)

- Build a heap from the following list of 15 keys:

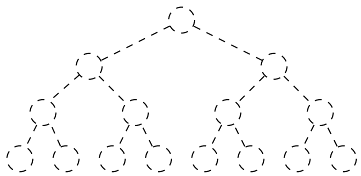
$$\langle 16, 15, 4, 12, 6, 7, 23, 20, 25, 9, 11, 17, 5, 8, 14 \rangle$$

- The resulting heap has:
  - ***Size N*** is 15
  - ***Height h*** is $(log(15 + 1)) - 1 = 3$
- According to the ***bottom-up heap construction*** technique, we will need to perform $h + 1 = 4$ steps, utilizing 4 sublists:

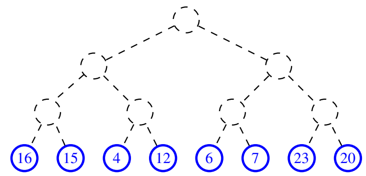
$$\langle \underbrace{16, 15, 4, 12, 6, 7, 23, 20}_{\frac{15+1}{2^1} = 8}, \underbrace{25, 9, 11, 17}_{\frac{15+1}{2^2} = 4}, \underbrace{5, 8}_{\frac{15+1}{2^3} = 2}, \underbrace{14}_{\frac{15+1}{2^4} = 1} \rangle$$

Bottom-Up Heap Construction: Example (1.2)

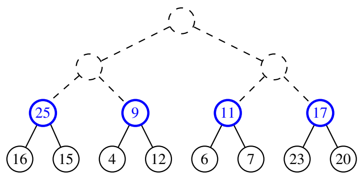
We know in advance to build a heap with height 3 and size $2^{3+1} - 1 = 15$



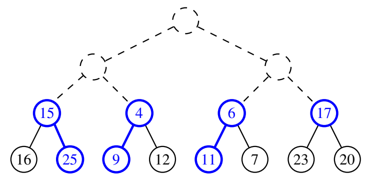
(**Step 1**) Treat first $\frac{15+1}{2^1}$ entries as roots.
∴ 8 one-node heaps.



(**Step 2**) Treat next $\frac{15+1}{2^2}$ entries as roots.
Set LST and RST of each root.

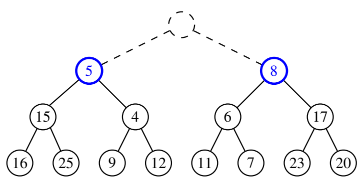


(**Step 2** cont.) Down-heap bubbling.
∴ 4 three-node heaps.

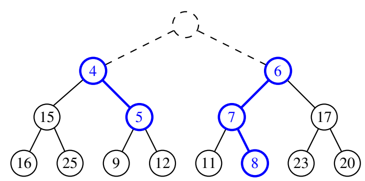


Bottom-Up Heap Construction: Example (1.3)

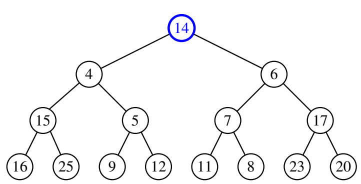
(**Step 3**) Treat next $\frac{15+1}{2^3}$ entries as roots.
Set LST and RST of each root.



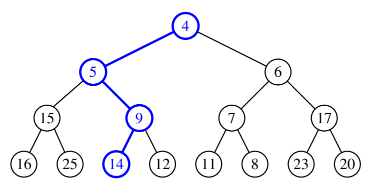
(**Step 3** cont.) Down-heap bubbling.
∴ 2 three-node heaps.



(**Step 4**) Treat next $\frac{15+1}{2^4}$ entry as roots.
Set LST and RST of each root.



(**Step 4** cont.) Down-heap bubbling.
∴ 1 fifteen-node heap.



RT of Bottom-Up Heap Construction

- Intuitively, the majority of the intermediate roots from which we perform ***down-heap bubbling*** are of very ***small height values***:
  - The first $\frac{n+1}{2}$ *1*-node heaps with ***height 0*** require **no** down-heap bubbling.
    [About 50% of the list entries processed]
  - Next $\frac{n+1}{4}$ *3*-node heaps with ***height 1*** require down-heap bubbling.
    [Another 25% of the list entries processed]
  - Next $\frac{n+1}{8}$ *7*-node heaps with ***height 2*** require down-heap bubbling.
    [Another 12.5% of the list entries processed]
  - . . .
  - Next two $\frac{N-1}{2}$-node heaps with ***height (h - 1)*** require down-heap
  - Final one ***N***-node heaps with ***height h*** requires down-heap bubbling.
- Running Time of the ***Bottom-Up Heap Construction*** takes only $O(n)$.

The Heap Sort Algorithm

Sorting Problem:

Given a list of **n** numbers $\langle a_1, a_2, \ldots, a_n \rangle$:
Precondition: NONE
Postcondition: A permutation of the input list $\langle a'_1, a'_2, \ldots, a'_n \rangle$ sorted in a non-descending order (i.e., $a'_1 \leq a'_2 \leq \ldots \leq a'_n$)

The *Heap Sort* algorithm consists of two phases:

1. ***Construct*** a ***heap*** of size ***N*** out of the input array.
   - Approach 1: Top-Down "Continuous-Insertions" [$O(N \cdot \log N)$]
   - Approach 2: Bottom-Up Heap Construction [$O(N)$]
2. ***Delete*** ***N*** entries from the heap.
   - Each deletion takes $O(\log N)$ time.
   - 1st deletion extracts the ***minimum***, 2nd deletion the 2nd ***minimum***, . . .
     ⇒ Extracted ***minimums*** from ***N*** deletions form a *sorted* sequence.

∴ Running time of the *Heap Sort* algorithm is $O(N \cdot \log N)$.

The Heap Sort Algorithm: Exercise

LASSONDE

Sort the following array of integers

$$\langle 16, 15, 4, 12, 6, 7, 23, 20, 25, 9, 11, 17, 5, 8, 14 \rangle$$

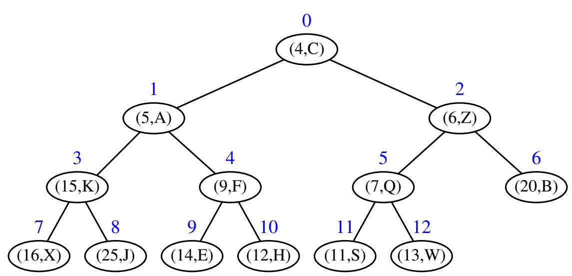
into a ***non-descending*** order using the *Heap Sort Algorithm*.

Demonstrate:

1. Both top-down and bottom-up heap constructions in Phase 1
2. Extractions of minimums in Phase 2

Array-Based Representation of a CBT (1)

LASSONDE

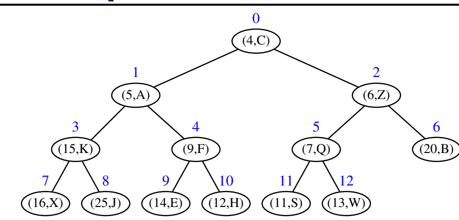


$$index(x) = \begin{cases} 0 & \texttt{if } x \texttt{ is the root} \\ 2 \cdot index(\ parent(x)\ ) + 1 & \texttt{if } x \texttt{ is a left child} \\ 2 \cdot index(\ parent(x)\ ) + 2 & \texttt{if } x \texttt{ is a right child} \end{cases}$$

(4,C)	(5,A)	(6,Z)	(15,K)	(9,F)	(7,Q)	(20,B)	(16,X)	(25,J)	(14,E)	(12,H)	(11,S)	(13,W)
0	1	2	3	4	5	6	7	8	9	10	11	12

Array-Based Representation of a CBT (2)

LASSONDE



(4,C)	(5,A)	(6,Z)	(15,K)	(9,F)	(7,Q)	(20,B)	(16,X)	(25,J)	(14,E)	(12,H)	(11,S)	(13,W)
0	1	2	3	4	5	6	7	8	9	10	11	12

- **Q1:** Where are nodes at ***Levels*** $0 \,..\, h-1$ stored in the array?
  Indices $0 \,..\, (2^h - 2) \equiv 0 \,..\, (2^{\lfloor log_2 N \rfloor} - 2)$ [e.g., Indices $0 \,..\, 2^3 - 2$]
- **Q2:** Where are nodes at ***Level*** h stored in the array?
  Indices $2^h - 1 \,..\, (N-1) \equiv 2^{\lfloor log_2 N \rfloor} - 1 \,..\, (N-1)$ [e.g., Indices 7 .. 12]
- **Q3:** How do we determine if a non-root node x is a ***left or right child***?
  IF $index(x)\ \%\ 2 == 1$ **THEN** *left* **ELSE** *right*
- **Q4:** Given a non-root node x, how do we determine the ***index of x's parent***?
  IF $index(x)\ \%\ 2 == 1$ **THEN** $\frac{index(x)-1}{2}$ **ELSE** $\frac{index(x)-2}{2}$

Index (1)

LASSONDE

Index (2)

Index (3)

Wrap-Up

EECS2011 N & Z:
Fundamentals of Data Structures
Winter 2022

CHEN-WEI WANG

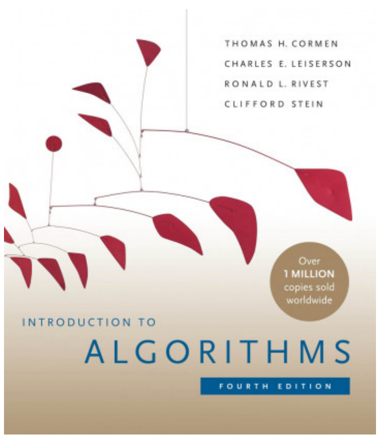
What You Learned (1)

- *Java Programming*
  - JUnit
  - Recursion
  - Generics

What You Learned (2)

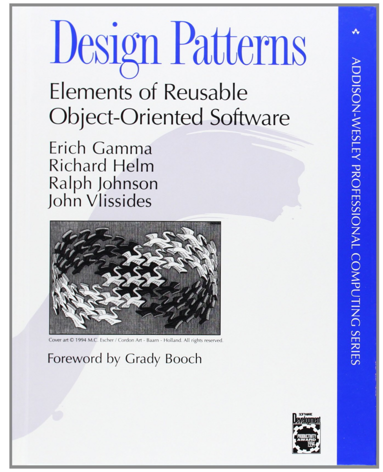
- ***Data Structures***
  - Arrays, Circular Arrays, Dynamic Arrays, Amortized RT Anaylsis
  - Singly-Linked Lists and Doubly-Linked Lists
  - Stacks, Queues, Double-Ended Queues
  - Trees, Binary Trees, Binary Search Trees, Balanced BSTs
  - Priority Queues and Heaps
- ***Algorithms***
  - Asymptotic Analysis
  - Binary Search
  - Trinode Restructuring Steps
  - Insertion Sort, Selection Sort, Merge Sort, Quick Sort, Heap Sort
  - Pre-order, in-order, and post-order traversals

Beyond this course. . . (1)



- *Introduction to Algorithms (4th Ed.)* by Cormen, *etc.*
- DS by DS, Algo. by Algo.:
  - *Understand* math analysis
  - *Read* pseudo code
  - ***Implement*** in Java
  - ***Test*** in JUnit

Beyond this course. . . (2)



- *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, *etc.*
- Pattern by Pattern:
  - *Understand* the problem
  - *Read* the solution (not in Java)
  - ***Implement*** in Java
  - ***Test*** in JUnit

Beyond this course. . . (3)

A tutorial on building a language compiler using Java (from ***EECS4302-W20***):

Using the ANTLR4 Parser Generator to Develop a Compiler

- Trees
- Recursion
- Visitor Design Pattern

Wish You All the Best

- What you have learned will be **assumed** in the third year.
- Some topics we did not cover:
  - Hash table [See Weeks 10 – 11 of EECS2030-F19]
  - Graphs [EECS3101]
- Logic is your friend: Learn/Review `EECS1019/EECS1090`.
- Do **not** abandon Java during the break!!
- Feel free to get in touch and let me know how you're doing :D