EECS2101 Fundamentals of Data Structures

Lecture Notes

Winter 2025 (Section X)

Jackie Wang

Lecture 1 - January 7

Syllabus
Introduction to the Course

Solving Problems via Data Structures

Course Learning Outcomes (CLOs)

API + pre/post cond. inv.

CLO1 Instantiate a range of standard abstract data types (ADT) as data structures.

CLO2 Implement these data structures and associated operations and check that they satisfy the properties of the ADT.

JUnit testing (regression)

CLO3 Apply best practice software engineering principles in the design of new data structures.

↳ make decisions among alternative DS.

CLO4 Demonstrate the ability to reason about data structures using contracts, assertions, and invariants.

↳ ① correctness ② efficiency (running time)

CLO5 Analyse the asymptotic run times of standard operations for a broad range of common data structures.

CLO6 Select the most appropriate data structures for novel applications.

Sorting

1. Insertion Sort
2. Selection Sort

) nested loops

3. Merge Sort
4. Quick Sort

) recursively.

5. Heap Sort

→ balanced binary search tree

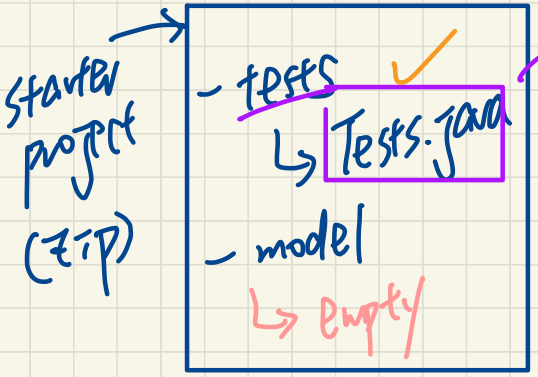
Written Test

~ section-specific
~ eClass (in-person)
~ multiple choice qs. (one or multiple correct ans.)

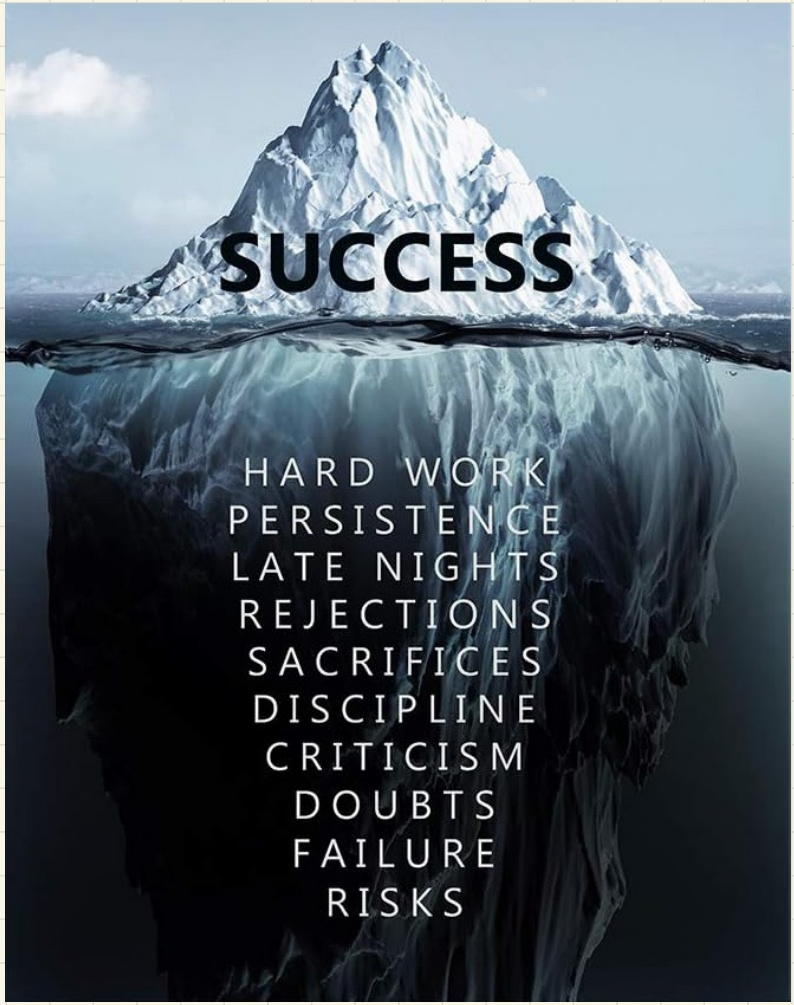
Exam

- in-person
- 3 hours
- mostly written questions.

Programming Tests



1. example usages of methods
2. meant to be incomplete
↳ you've expected:
(1) not to make your code work only for the starter tests.
(2) write additional tests.



General Tips about Success

Source: https://a.co/d/aQ13fR1

Lecture 2 - January 9

Introduction to the Course
Recursion: Part 1

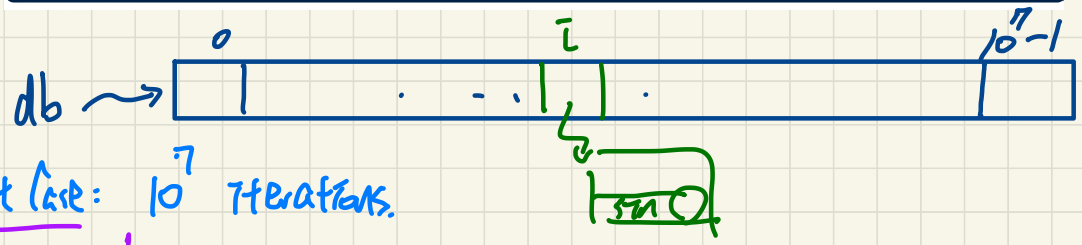
Solving Problems via Data Structures
References to Recursion Basics
More Advanced Recursion: splitArray

Announcements/Reminders

- **Assignment 1** released
- Office Hours: 3pm to 4pm, Mon/Tue/Wed/Thu
- Trial attendance check via iClicker today!

A Searching Problem

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



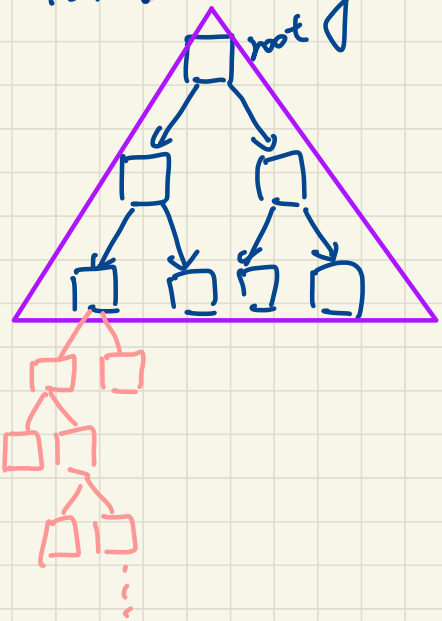
Worst Case: 10^7 iterations.

linear search

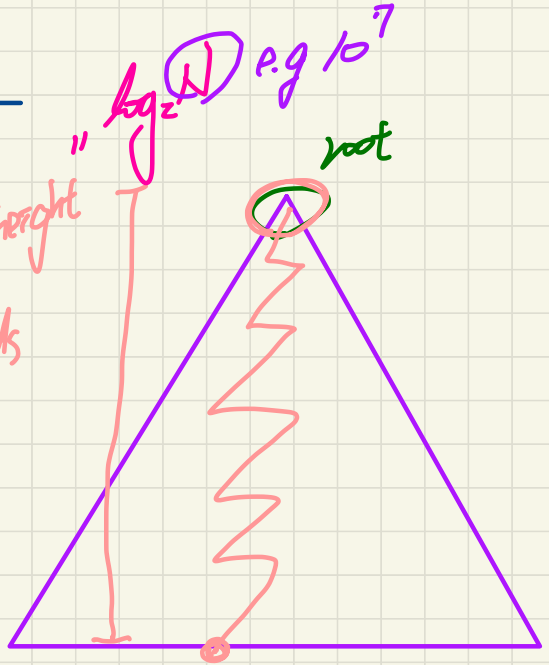
BBST : $\log_2 10^7 \approx 23.3$

Solution for Efficient Searching

✓ Balanced ✓ Binary Search Tree

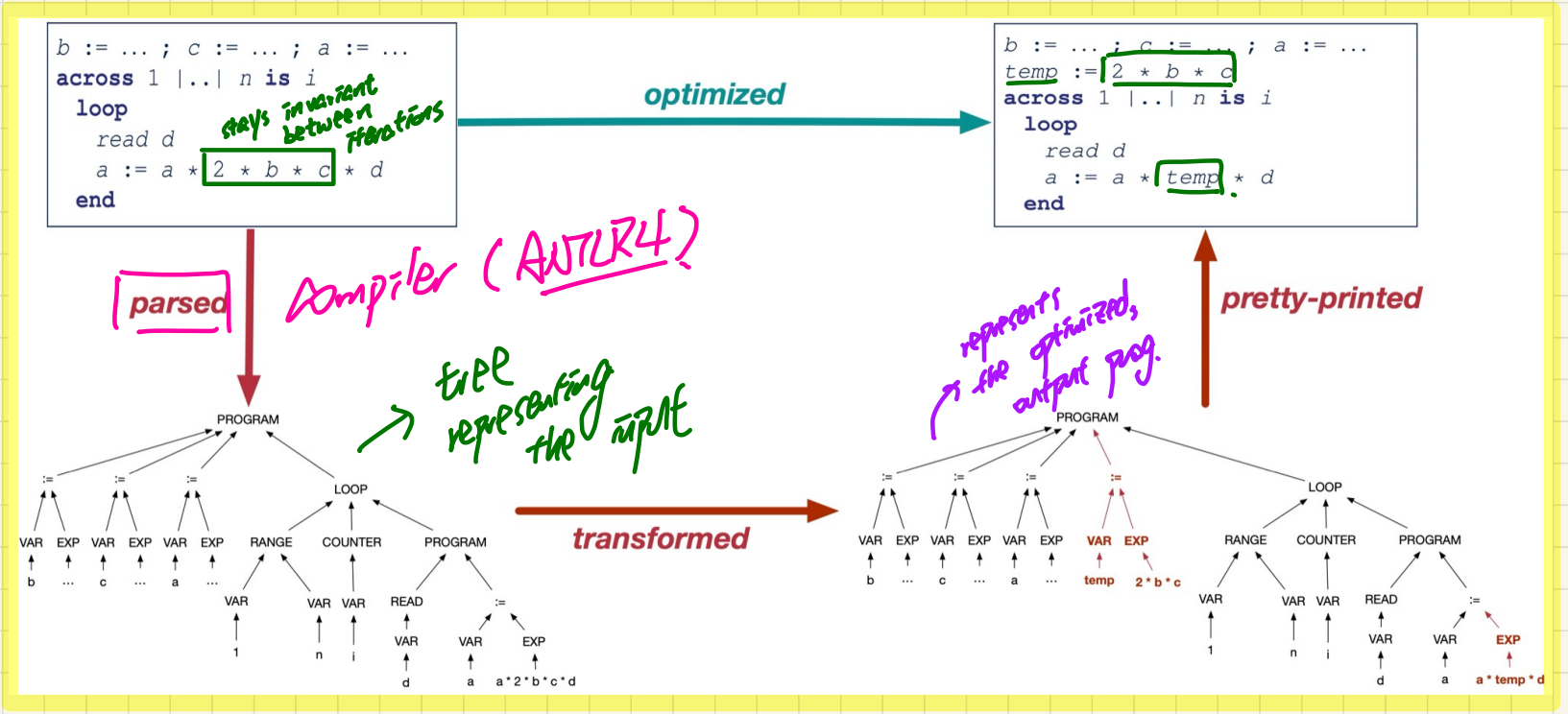


height ≤ N records



leaves (bottom level)

Program Optimization Problem



Program Translation Problem

object-relational bridge

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

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

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

translated

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

parsed

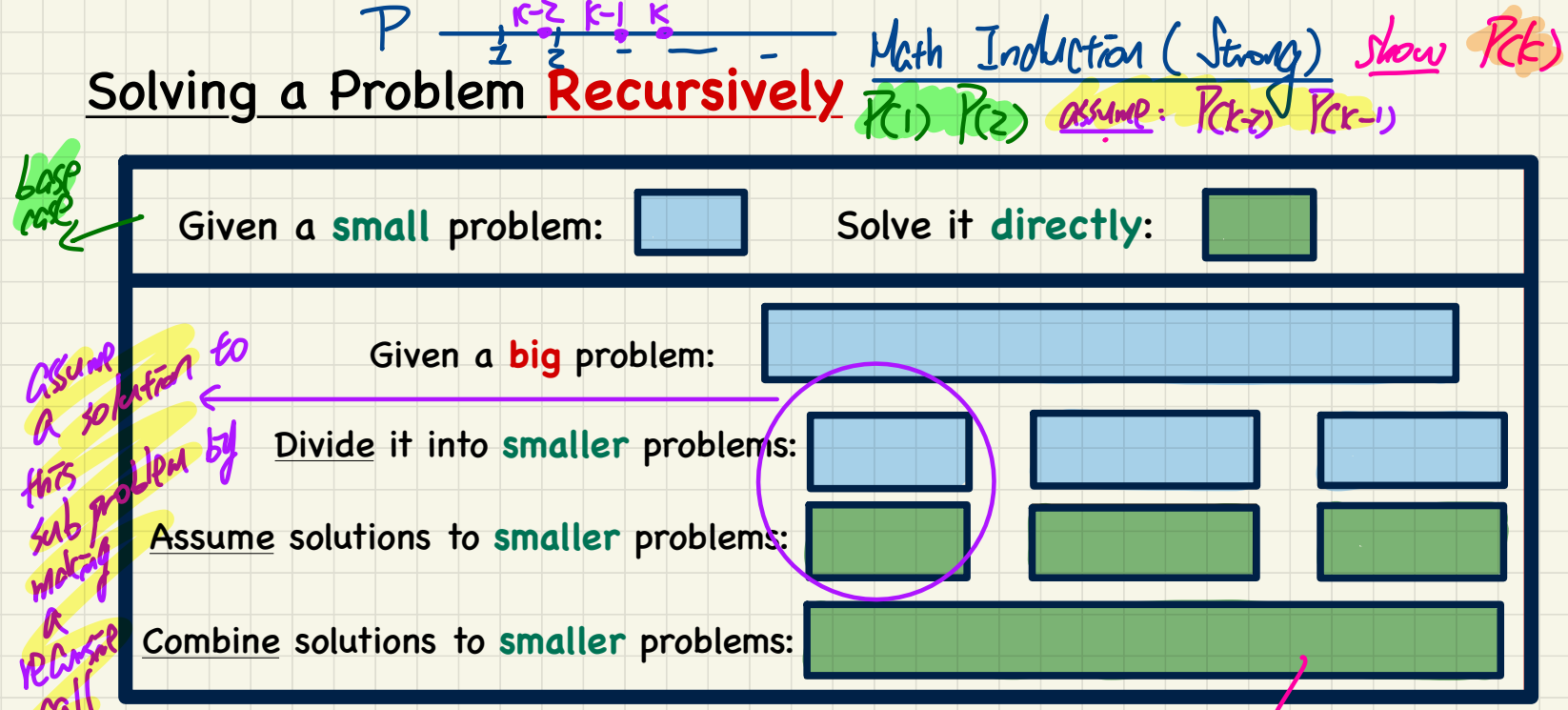
pretty-printed

Abstract Syntax Tree of **Source** Object-Oriented Program

transformed

Abstract Syntax Tree of **Target** Relational DB Queries

Solving a Problem Recursively



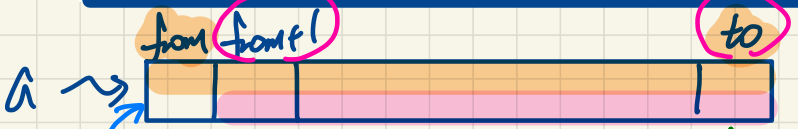

```
m (i) {
  if(i == ...) { /* base case: do something directly */ }
  else {
    m (j);/* recursive call with strictly smaller value */
  }
}
```

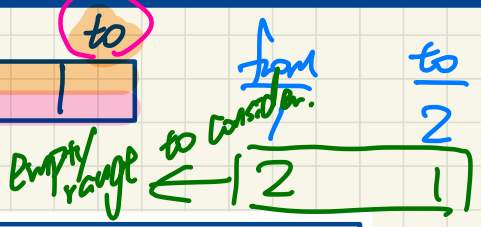
Recursion on an Array: Passing Same Array Reference

the value of a's address will be passed between recursive calls

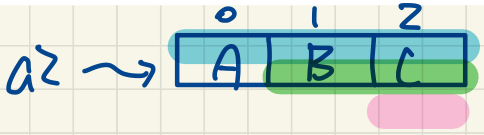
```
void m(int[] a, int from, int to) {
 if(from > to) { /* base case */ }
 else if(from == to) { /* base case */ }
  else m(a, from + 1, to) } }
```

specify the range of elements in 'a' to look at in a specific recursive call (without the need to create a new sub-array)





Say a2 = {A, B, C}, consider m(a2, 0, a2.length - 1)



m(a2, 0, 2)
→ m(a2, 1, 2)
→ m(a2, 2, 2)

Problem on Recursion

https://codingbat.com/prob/p185204

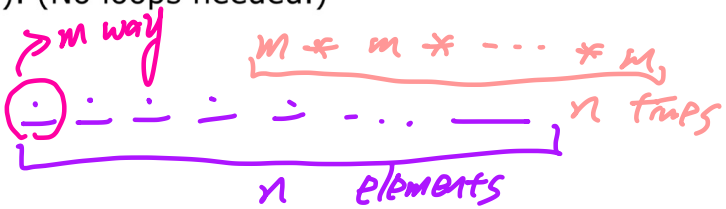
Given an array of ints, is it possible to divide the ints into two groups, so that the sums of the two groups are the same. Every int must be in one group or the other. Write a recursive helper method that takes whatever arguments you like, and make the initial call to your recursive helper from splitArray(). (No loops needed.)

splitArray([2, 5, 3]) → true

splitArray([2, 2]) → true

splitArray([2, 3]) → false

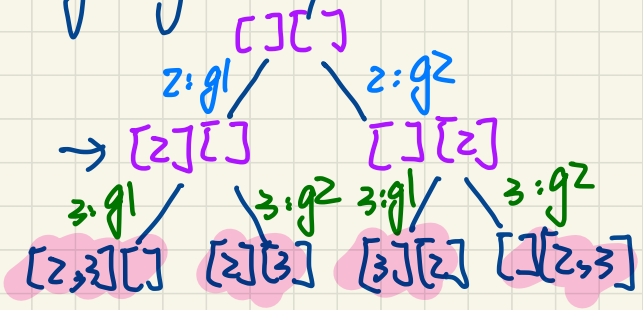
splitArray([5, 2, 3]) → true



Intuition

MATH 2030 : How many ways to put n elements into m groups?

n^m (1) m^n (2)



Lecture 3 - January 14

Recursion: Part 1
Asymptotic Analysis of Algorithms

splitArray: Implementation and Tracing

Announcements/Reminders

- Assignment 1 released
- Office Hours: 3pm to 4pm, Mon/Tue/Wed/Thu
- Contact Information of TAs on common eClass site

Problem on Recursion

https://codingbat.com/prob/p185204

Given an array of ints, is it possible to divide the ints into two groups, so that the sums of the two groups are the same. Every int must be in one group or the other. Write a recursive helper method that takes whatever arguments you like, and make the initial call to your recursive helper from splitArray(). (No loops needed.)

splitArray([2, 5, 3]) → time.

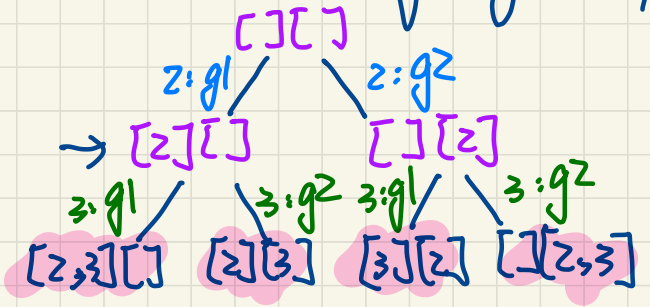
splitArray([2, 2]) → true
splitArray([2, 3]) → false
splitArray([5, 2, 3]) → true

Insight: The recursive and base cases are defined s.t. the tree of runtime recursive calls represents an exhaustive search on all the input scenarios.

worst case

Intuition

MATH 2030: How many ways to put n elements into m groups?



splitArray: Java Implementation

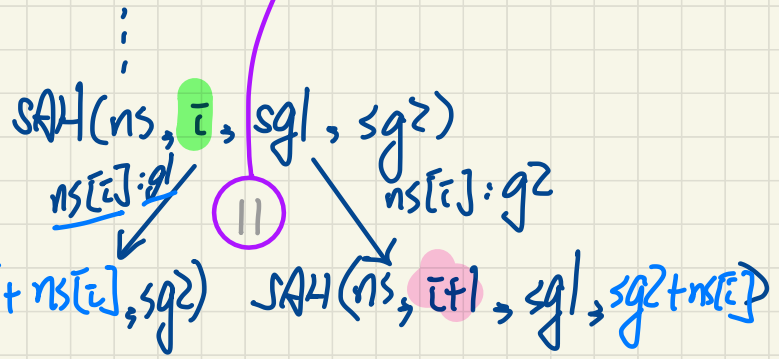
empty groups.

```java
public boolean splitArray(int[] ns) {
    return splitArrayHelper(ns, 0, 0, 0);
}
```

starting to group elements from index 0

```java
private boolean splitArrayHelper(int[] ns, int i, int sumOfGroup1, int sumOfGroup2) {
    if(i == ns.length) {
        return sumOfGroup1 == sumOfGroup2;
    }
    else {
        return
            splitArrayHelper(ns, i + 1, sumOfGroup1 + ns[i], sumOfGroup2)
            ||
            splitArrayHelper(ns, i + 1, sumOfGroup1, sumOfGroup2 + ns[i]);
    }
}
```

short-circuit eval.



Math

Commutativity:

$$p \wedge q \equiv q \wedge p$$
$$p \vee q \equiv q \vee p$$

Java

Evaluation orders matter

&& || do not commute

(2.1) $p \;\&\&\; q \neq q \;\&\&\; p$

(2.2) $p \;||\; q \neq q \;||\; p$

① Evaluation : left to right

(2.1) p eval. to T → still eval q
p eval. to F → skip eval of q → overall F

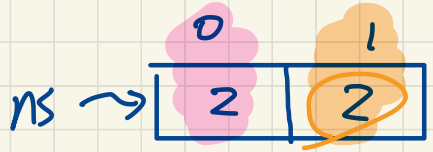
(2.2) p eval. to F → still eval q
p eval. to T → skip eval. of q → overall T

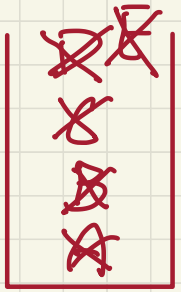
splitArray: Tracing (1)

```java
public boolean splitArray(int[] ns) {
    return splitArrayHelper(ns, 0, 0, 0);
}
```

```java
@Test
public void testSplitArray_01() {
    RecursiveMethods rm = new RecursiveMethods();
    int[] input = {2, 2};
    assertEquals(true, rm.splitArray(input));
}
```

```java
private boolean splitArrayHelper(int[] ns, int i, int sumOfGroup1, int sumOfGroup2) {
    if(i == ns.length) {
        return sumOfGroup1 == sumOfGroup2;
    }
    else {
        return
            splitArrayHelper(ns, i + 1, sumOfGroup1 + ns[i], sumOfGroup2)
            ||
            splitArrayHelper(ns, i + 1, sumOfGroup1, sumOfGroup2 + ns[i]);
    }
}
```





splitArray: Tracing (2)

```java
public boolean splitArray(int[] ns) {
    return splitArrayHelper(ns, 0, 0, 0);
}
```

```java
private boolean splitArrayHelper(int[] ns, int i, int sumOfGroup1, int sumOfGroup2) {
    if(i == ns.length) {
        return sumOfGroup1 == sumOfGroup2;
    }
    else {
        return
            splitArrayHelper(ns, i + 1, sumOfGroup1 + ns[i], sumOfGroup2)
            ||
            splitArrayHelper(ns, i + 1, sumOfGroup1, sumOfGroup2 + ns[i]);
    }
}
```

```java
@Test
public void testSplitArray_04() {
    RecursiveMethods rm = new RecursiveMethods();
    int[] input = {5, 2, 2};
    assertEquals(false, rm.splitArray(input));
}
```

Exercise

(1) Trace (recursion tree)

(2) Trace (debugger)

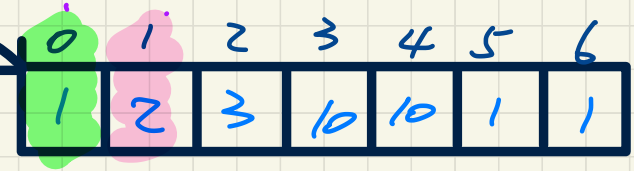
splitArray: Tracing (3)

```
@Test
public void testSplitArray_13() {
    RecursiveMethods rm = new RecursiveMethods();
    int[] input = {1, 2, 3, 10, 10, 1, 1};
    assertEquals(true, rm.splitArray(input));
}
```

```
public boolean splitArray(int[] ns) {
    return splitArrayHelper(ns, 0, 0, 0);
}
```

```
private boolean splitArrayHelper(int[] ns, int i, int sumOfGroup1, int sumOfGroup2) {
    if(i == ns.length) {
        return sumOfGroup1 == sumOfGroup2;
    }
    else {
        boolean possibility1 = splitArrayHelper(ns, i + 1, sumOfGroup1 + ns[i], sumOfGroup2);
        boolean possibility2 = splitArrayHelper(ns, i + 1, sumOfGroup1, sumOfGroup2 + ns[i]);
        return possibility1 || possibility2;
    }
}
```

splitArray: Tracing (3)



Level 0 (0, 0) $1 = 2^0$

Level 1 after insert. 1st elem. at index 0: 1:g1 (1, 0), 1:g2 (0, 1) $2 = 2^1$

Level 2 after insp. 2nd elem. at index 1: 2:g1, 2:g1, 2:g1, 2:g2 $4 = 2^2$

geometric seq. sum.

Level 7 after insp. 7th elem. at index 6: /\ /\ /\ . . . /\ /\ . . . /\ $= 2^7 = 128$

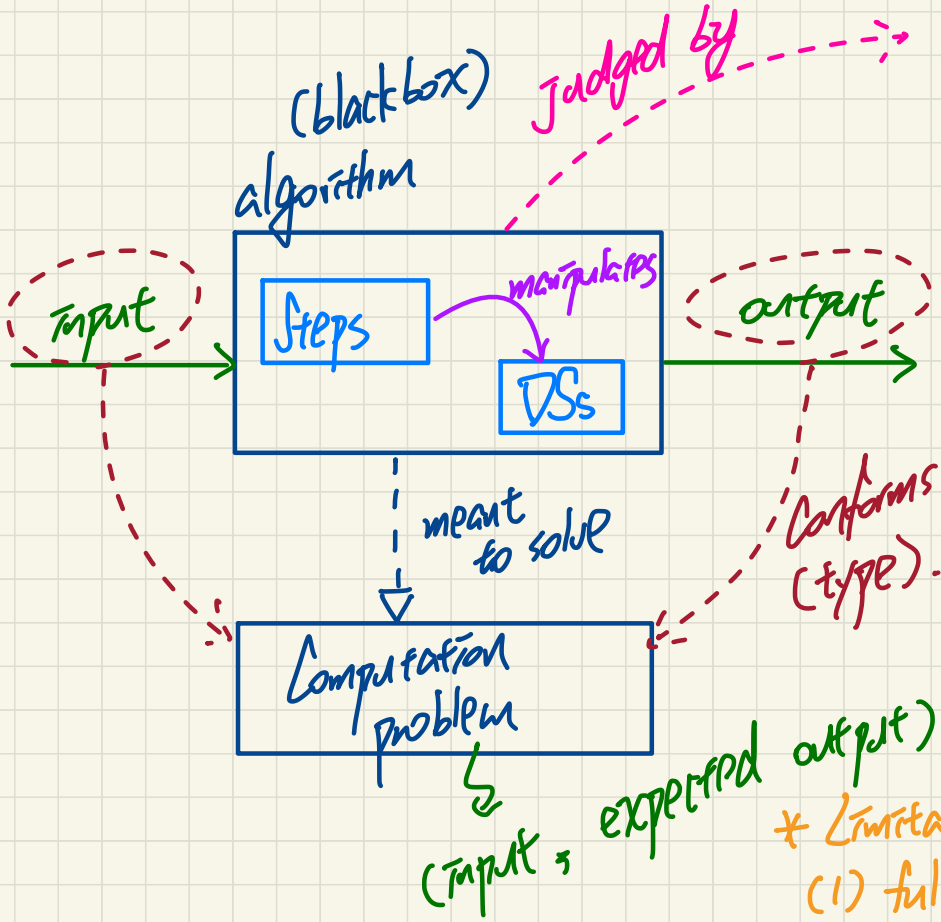
Lecture 4 - January 16

Asymptotic Analysis of Algorithms

Limitations of Experiments
Primitive Operations (POs)
Counting POs: findMax

Announcements/Reminders

- **Assignment 1** released
- Office Hours: 3pm to 4pm, Mon/Tue/Wed/Thu
- Contact Information of TAs on common eClass site
- ***splitArrayHarder***: an **extended** version coming soon



1. Correctness
   - ↳ testing (dynamic)
   - ↳ formal method (static)
     - ↳ 1090 (3342, 4315)

2. Efficiency
   - ↳ Time
     - Experiments
     - Math Func. (big-O).
   - ↳ Space
     - ↳ may impact RT

* Limitations:
(1) full, working implementation
(2) exec. environment
(3) input representativeness

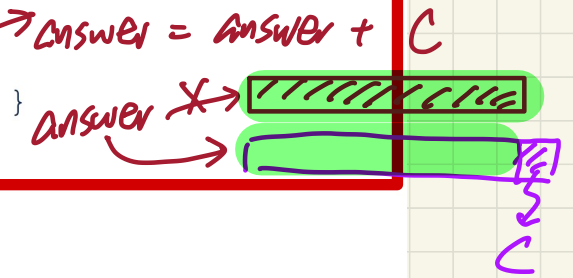
Example Experiment

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 `***************`.

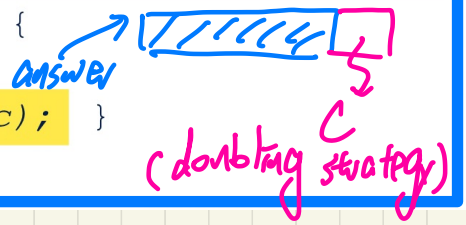
Algorithm 1 using *String* Concatenations:

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



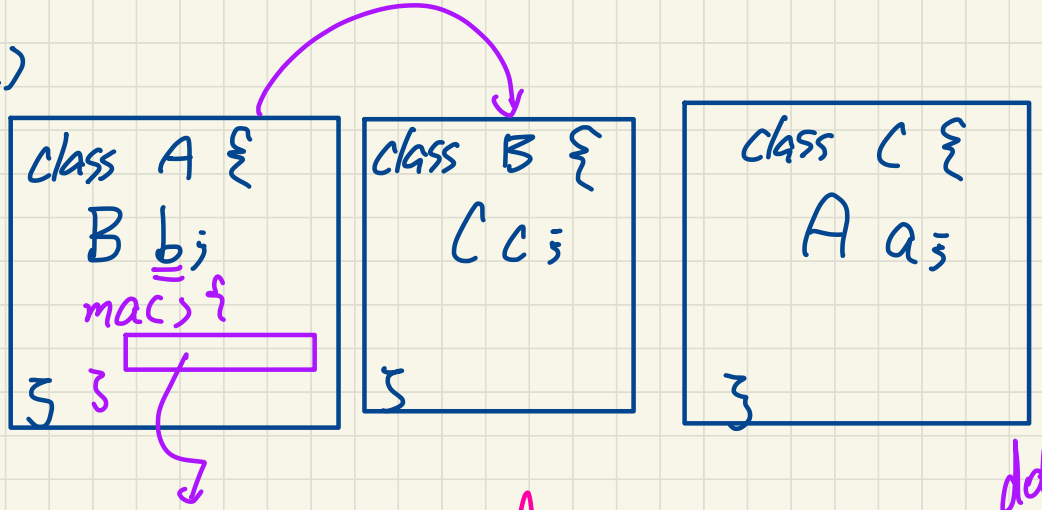
Algorithm 2 using *StringBuilder* `append`'s:

```
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(); }
```



Accessing an object's attribute

(aggregation)



arbitrarily long (as long as refs valid)

- b
- b.C
- b.C.a

Assumption

Length of dot notation expression is always a small constant.

does not depend on input size (n).

Method Call : may or may not be a PO

obj.m(...)

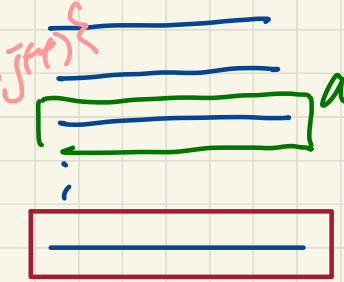
fixed, not dependent on input size

Case 1

m considered as PO

```
m() {
    for(int i=0; i<100; i++) {
        for(int j=0; j<100; j++) {
            ...
        }
    }
}
```

100 * 100 = 10000



can be:
(1) a method call (that's considered a PO)
(2) a loop (with a fixed # of iterations)

Case 2 m not PO

```
m() {
    for(int i=0; i<a.length; i++) {
        ...
    }
}
```

size of input

findMax(a, a.length)

Example 1: Counting Number of Primitive Operations

```
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; }
```

2

$n \cdot 1 + 1$

$(n-1) \cdot 2$

$(n-1) \cdot 2$

$(n-1) \cdot 2$

1

i	i < n
1	T
2	T
⋮	⋮
n-1	T
n	F

i = i + 1

Q. # of times i < n in Line 3 is executed?

n times (n-1 times i<n (T) ; 1 time i<n (F))

Q. # of times loop body (Lines 4 to 6) is executed?

n-1 times

n-2

↳ size of input

Lecture 5 - January 21

Asymptotic Analysis of Algorithms

From Absolute RT to Relative RT
Approximating RT Functions
Asymptotic Upper Bound (Big-O): Def.

Announcements/Reminders

- Assignment 1 released
- ***splitArrayHarder***: an **extended** version released
- Office Hours: 3pm to 4pm, Mon/Tue/Wed/Thu
- Contact Information of TAs on common eClass site

Example 2: Counting Number of Primitive Operations

(Exercise)

```
1  boolean foundEmptyString = false;
2  int i = 0;
3  while (!foundEmptyString && i < names.length) {
4    if (names[i].length() = 0) {
5      /* set flag for early exit */
6      foundEmptyString = true;
7    }
8    i = i + 1;
9  }
```

Q. # of times Line 3 is executed?

Q. # of times loop body (Lines 4 to 8) is executed?

Q. # of POs in the loop body (Lines 4 to 8)?

Comparing Algorithms: From Absolute RT to Relative RT

t
exact/absolute time taken to execute a P.O. in some exec. environment

e.g. Mac M1 $t = 2$ (ms)

e.g. Mac'14 $t = 4$ (ms)

abs. RT	Mac M1	Mac'14
$7n-2$	$(100 \cdot 7 - 2) \cdot 2$	$(100 \cdot 7 - 2) \cdot 4$
$10n+3$	$(100 \cdot 10 + 3) \cdot 2$	

absolute RTs of two alt. algorithms (solving the same problem)

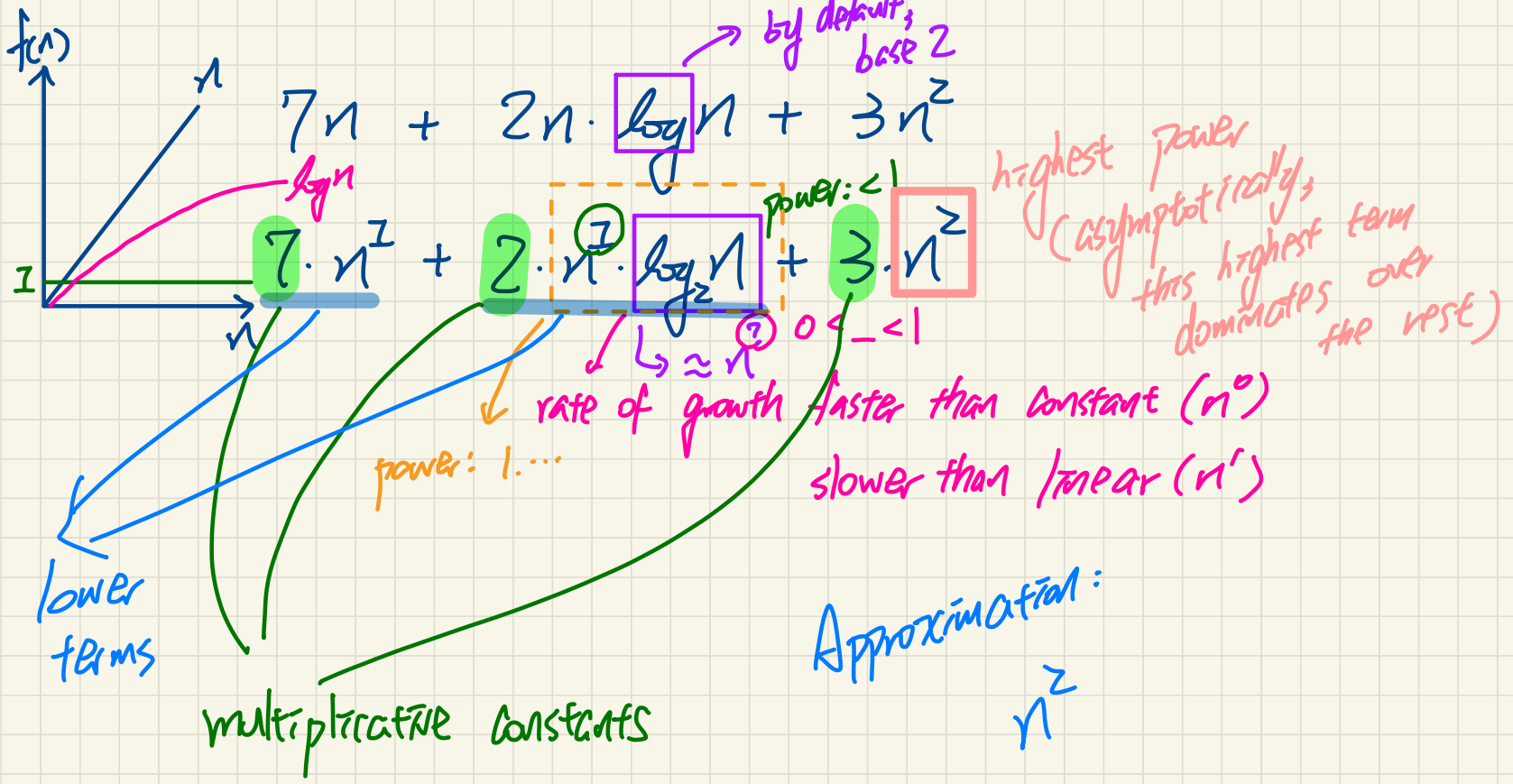
input size (assume $n = 100$)

when compared the same alg., it's not necessary to know the precise RT

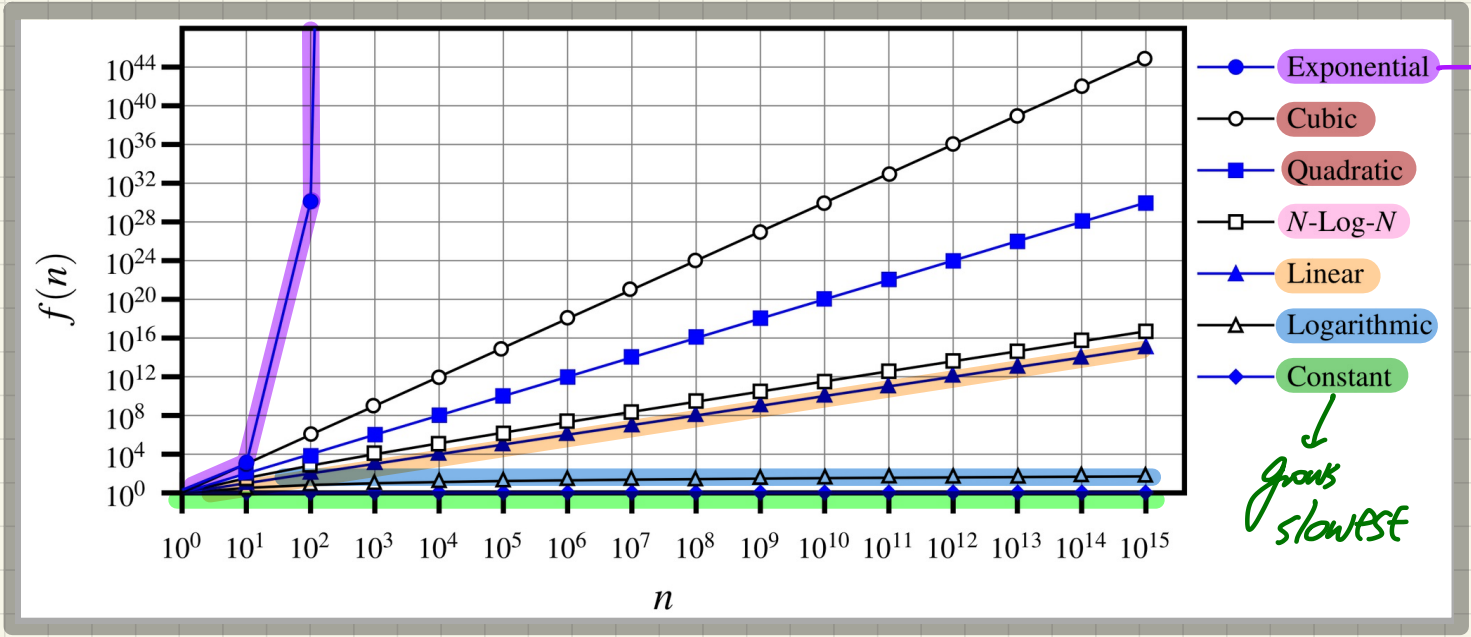
when comparing two diff. alg.'s not necessary to consider the extra, common factor. ↳ t diff.

$n^0 \equiv 1$ $n^1 \equiv n$ $\log n$

Exercise: Approximating $f(n) = 7n + 2n \cdot \log n + 3n^2$



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



RT

$f(n) = 1000 = 1000 \cdot n^0$

$f(1) = 1000$

$f(1000) = 1000$

$f(1M) = 1000$

Comparing Relative, Asymptotic RTs of Algorithms

Q1. Compare:

$RT_1(n) = 3n^2 + 7n + 18 \approx n^2$

$RT_2(n) = 100n^2 + 3n - 100 \approx n^2$

↳ equally efficient, asymptotically.

Q2: Compare:

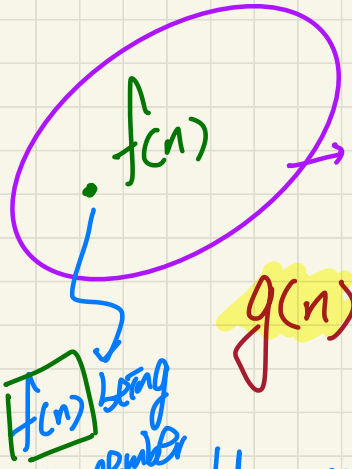
$RT_1(n) = n^3 + 7n + 18 \approx n^3$

$RT_2(n) = 100n^2 + 100n + 2000 \approx n^2$

↳ RT_2 more efficient (taking less time), asymptotically.

$f(n)$: RT function

↳ input size ⇝ relative RT



e.g. $f(n) = 7n - 2$

$g(n)$: reference function

$g(n) \rightsquigarrow c \cdot g(n)$

(further manipulation on $g(n)$ expected)

$f(n)$ being a member in the family means that it can somehow be upper-bounded by $g(n)$

Goal: Prove $f(n)$ is $O(g(n))$

u.b.e. : upper-bound effect

Asymptotic Upper Bound: Big-O

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

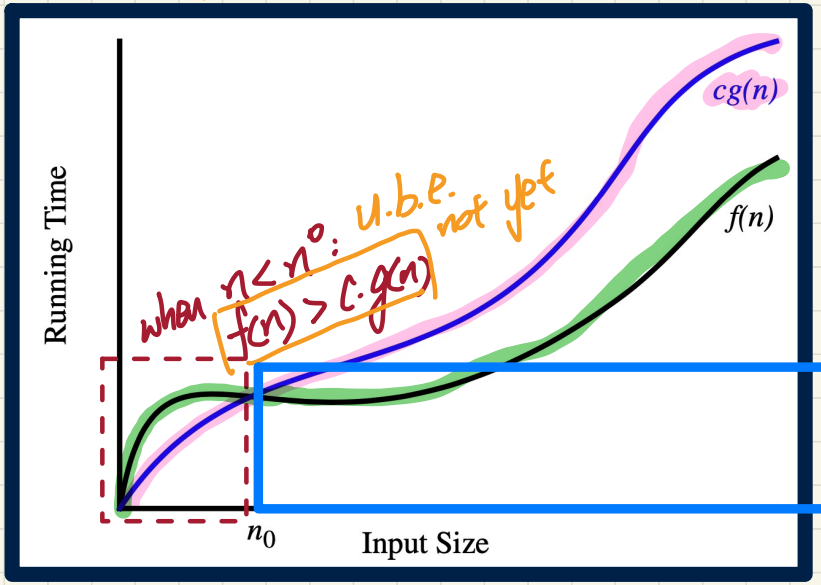
- A real *constant* $c > 0$
- An integer *constant* $n_0 \geq 1$

such that:

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

multiplicative constant applied to g(n) to change its slope

starting point of the u.b.e.



Example:

$f(n) = 8n + 5$

$g(n) = n$

Prove:

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

Choose c = 9

What about n0?

Can n_0 be 1?

$f(1) \overset{?}{\leq} 9 \cdot 1$

8+5=13

False

↳ u.b.e. not here.

→ starting from $n = n_0$

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

u.b.e. is here!

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

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

We prove by choosing

$c = |a_0| + |a_1| + \cdots + |a_d|$

$n_0 = 1$

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 (i.e., negative, zero, or positive),

then $f(n)$ is $O(n^d)$. → $g(n)$

(1) $f(1) \le c \cdot 1^d$

(2) $f(n) \le c \cdot n^d$ $(n > 1)$

Upper-bound effect: $n_0 = 1$? $[f(1) \le (|a_0| + |a_1| + \cdots + |a_d|) \cdot 1^d]$

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

Lecture 6 - January 23

Asymptotic Analysis of Algorithms

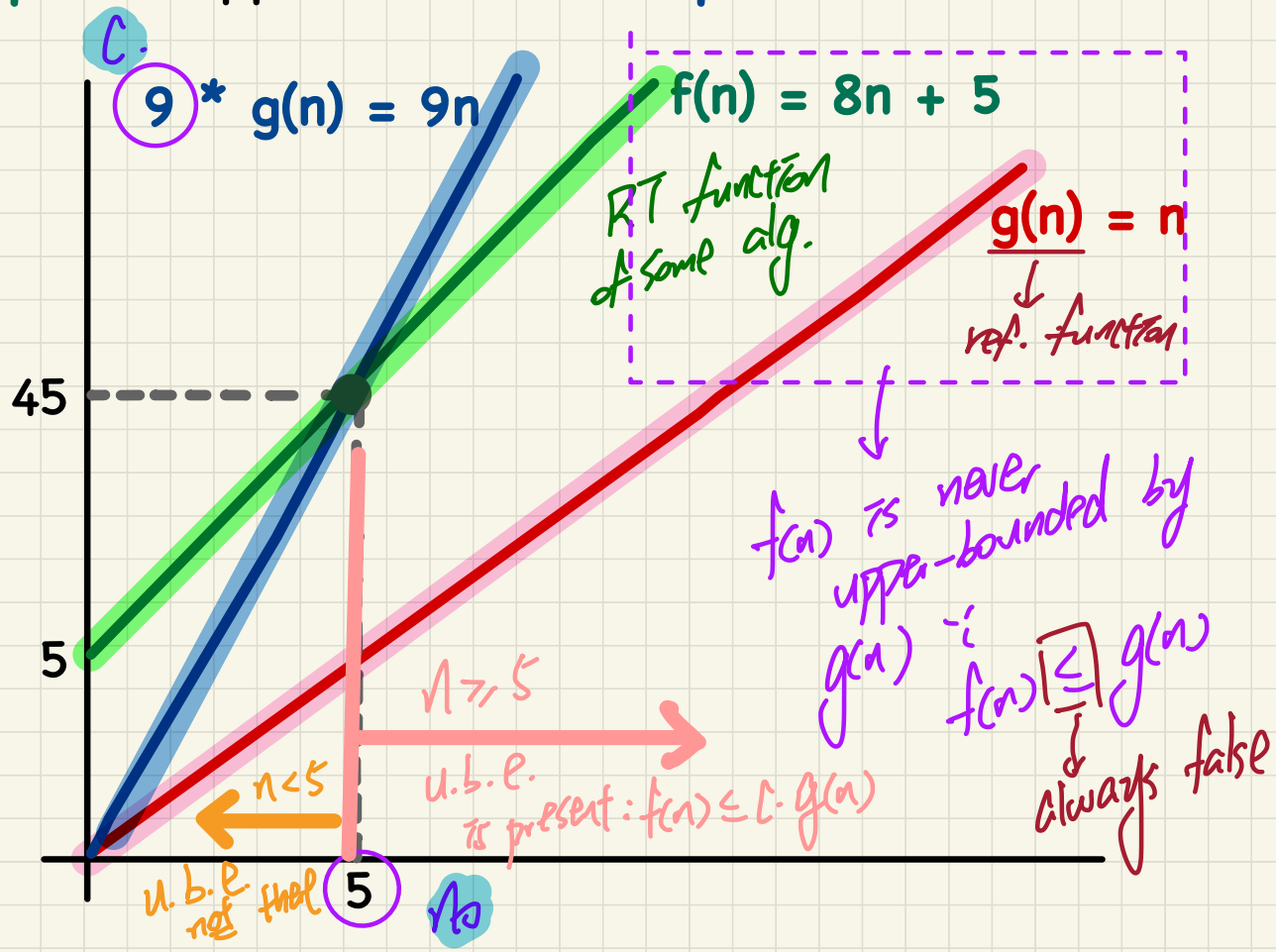
Big-O: Pred. Def., Properties, Examples
Correct vs. Accurate Asymptotic U.B.
Deriving U.B. from Code: Basic Examples

Announcements/Reminders

- **Assignment 1** due next Monday
- ***splitArrayHarder***: an **extended** version released
- Office Hours: 3pm to 4pm, Mon/Tue/Wed/Thu
- Contact Information of TAs on common eClass site

Asymptotic Upper Bound: Example

f(n) is order of O(g(n))



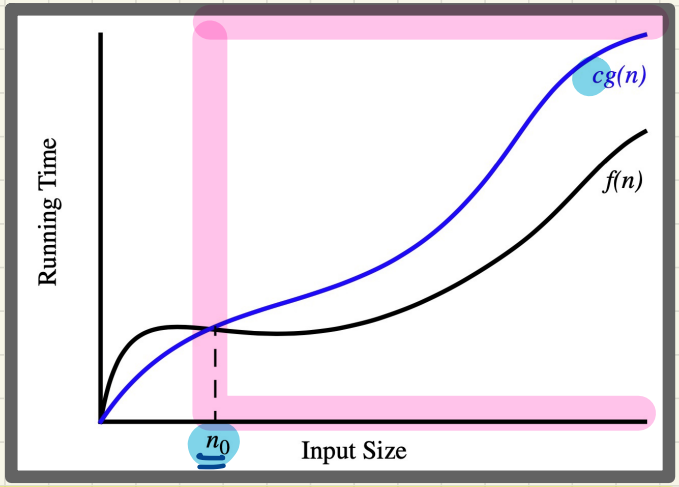
Asymptotic Upper Bound (Big-O): Alternative Formulation

Known:

$f(n) \in O(g(n))$ if there are:
- A real *constant* $c > 0$
- An integer *constant* $n_0 \geq 1$

such that:

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



Q. Formulate the definition of "f(n) is order of O(g(n))" using **logical** operator(s): $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\forall$, $\exists$

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

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

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

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

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

We prove by choosing

$$c = |a_0| + |a_1| + \cdots + |a_d|$$
$$n_0 = 1$$

(1) $f(1) \le c \cdot 1^d$

(2) $f(n) \le c \cdot n^d$ $(n > 1)$

Upper-bound effect: $n_0 = 1$?

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

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

($a_0 \le |a_0|$, $a_1 \le |a_1|$, $1^0 = 1^1 = \cdots = 1^d = 1$; c; $g(1)$)

Upper-bound effect holds?

$$[f(n) \le (|a_0| + |a_1| + \cdots + |a_d|) \cdot n^d] \quad (n > 1)$$

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

($n^0 \le n^d$, $n^1 \le n^d$, $n^d \le n^d$; c; $g(n)$)

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

$5 \cdot n^4 - 3n^3 + 2n^2 - 4n + 1 \cdot n^0$

(1) Guess: $O(n^4)$ ↓ $g(n)$

(2) Prove

choose $C = |5| + |(-3)| + |2| + |(-4)| + |1| = 15$

$n_0 = 1$

Verify:

$f(1) \leq 15 \cdot g(1)$

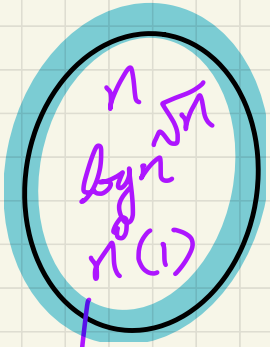
$5 - 3 + 2 - 4 + 1 \leq 15 \cdot 1$ ✓

Big-O Properties (1): Members in a Family

Each member f(n) in O(g(n)) is such that:

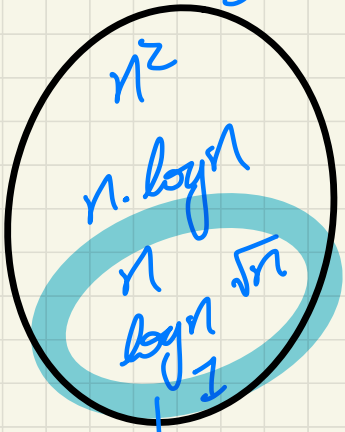
Higest Power of f(n) <= Highest Power of g(n)

O(n)



functions growing "no faster" than n should be included

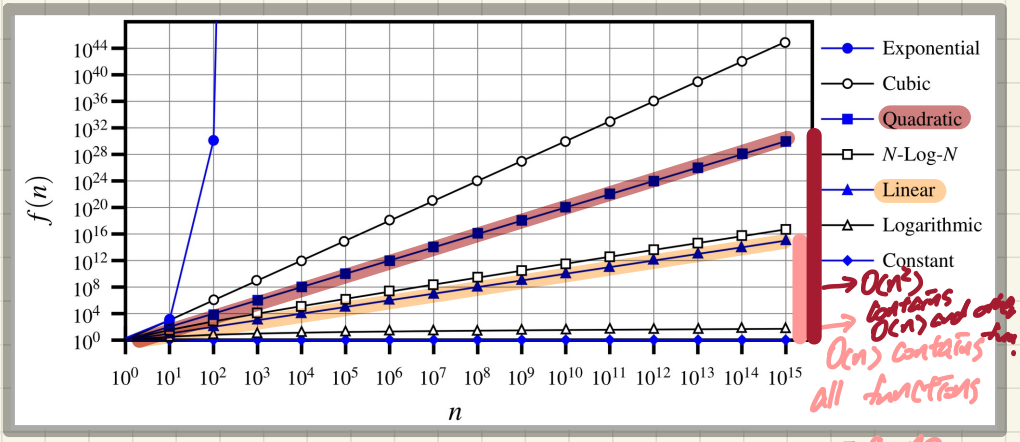
O(n^2)



O(n) is "contained" within $O(n^2)$

$2^n \notin O(n^2)$

Functions: Rates of Growth



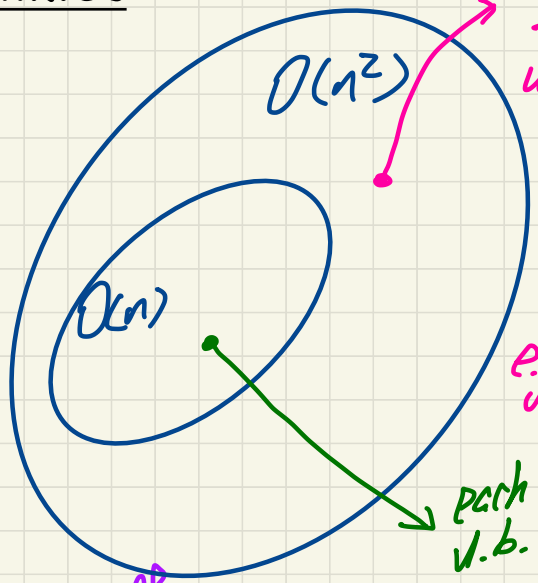
$O(n^2)$ contains $O(n)$ and others

$O(n)$ contains the all functions growing no faster than n

Big-O Properties (2): Relating Families

$O(n)$ $\subseteq$ $O(n^2)$

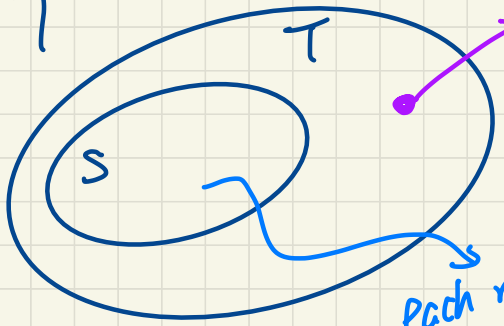
Correct
(1) $\subseteq$ → it's possible $O(n) = O(n^2)$
(2) $\subset$ correct & accurate
(3) $\supseteq$
(4) $\supset$



at least one function u.b. by n^2 but not u.b. by n.
e.g. n^2, $n \log n$

each function u.b. by n can also be u.b. by n^2

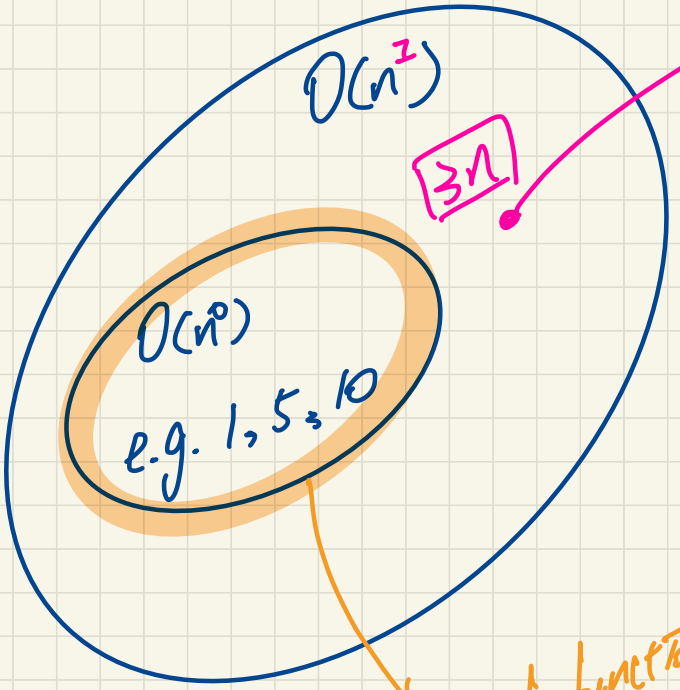
$S \subset T$



at least one member in T that's not a member in S.

each member in S is also a member in T

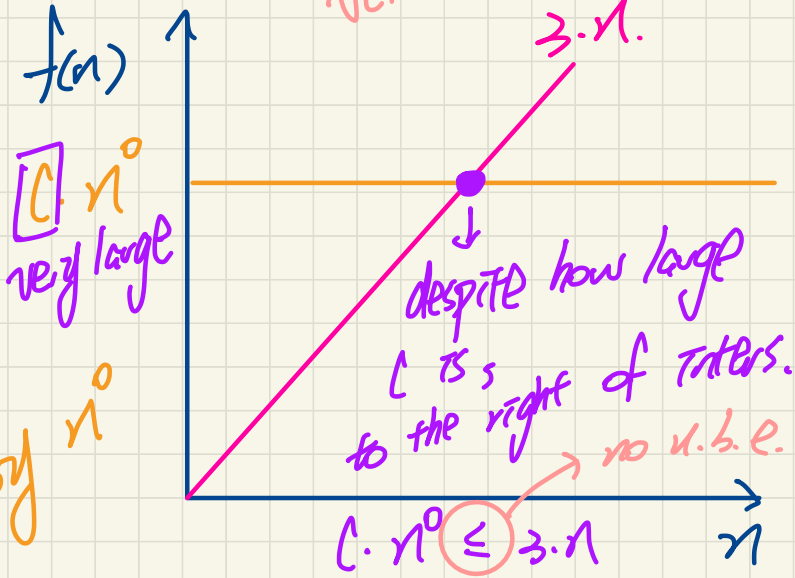
$O(n^0) \subset O(n)$



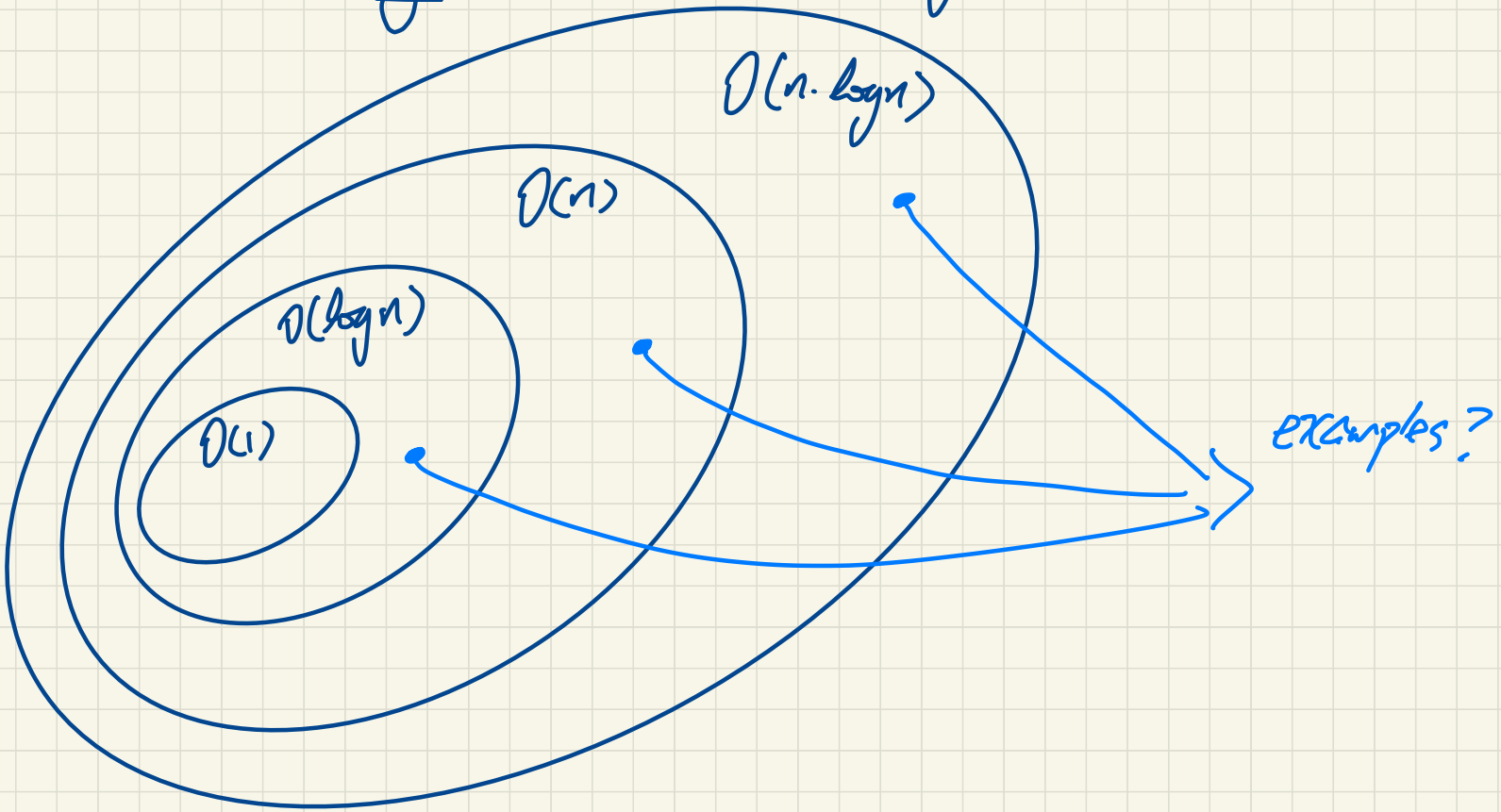
at least a function u.b. by n^1 but not u.b. by n^0

even with a VERY LARGE C chosen

set of functions that can be u.b. by n^0



$O(n^0) \subset O(\log n) \subset O(n) \subset O(n \cdot \log n) \subset \dots$



Big-O Properties (3): Deciding Correct & Accurate Bound

e.g. $f_1(n) = 7n - 2$

$f_2(n) = 4n^2 - 3n + 6$

$7n-2$ is order of ($\in$)

- $O(n)$ → most accurate (tightest U.b.)
- $O(n^2)$
- $O(n^3)$
- $O(2^n)$

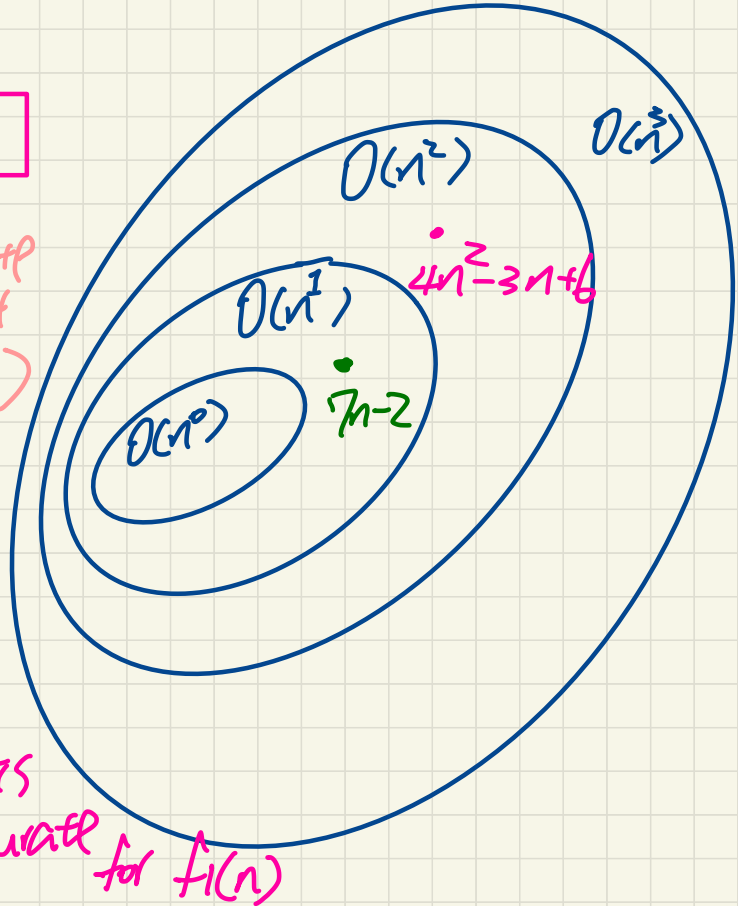
all correct

misleading.

$f_1(n)$ is $O(n^3)$ → $f_2(n)$ more efficient?

$f_2(n)$ is $O(n^2)$

No! ∵ the $O(n^3)$ is not accurate for $f_1(n)$



$O(7n^2 + 4n - 2)$ X not appropriate as the final answer

$O(n^2)$ ✓

Asymptotic Upper Bounds: Example (1)

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

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

(2) **Prove** your claim.

(1) $O(n^2)$ — $g(n)$

(2) Choose $c = |5| + |3| + |2| + |5| = 15$

$n_0 = 1$

↳ verify $f(1) \leq 15 \cdot 1^2$

$5 \cdot 1^2 + 3 \cdot 1 \cdot \log 1 + 2 \cdot 1 + 5 \leq 15$ (true!)

($\log 1 = 0$, $2^0 = 1$)

Asymptotic Upper Bounds: Example (2) (Exercise)

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

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

(2) **Prove** your claim.

Asymptotic Upper Bounds: Example (3)

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

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

(2) **Prove** your claim.

(1) $O(\log n)$

(2) choose $C = |3| + |2| = 5$

$n_0 = 1$ → verify: $f(1) \leq 5 \cdot \log 1$

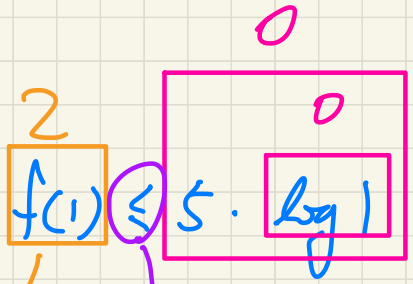
$f(1) = 3 \cdot \log 1 + 2$

fake

What if $n_0 = 2$

↳ $f(2) \leq 5 \cdot \log 2$

(Exercise) u.b.e?



Asymptotic Upper Bounds: Example (4) (Exercise)

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

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

(2) **Prove** your claim.

Asymptotic Upper Bounds: Example (5)

(Exercise)

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

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

(2) **Prove** your claim.

Lecture 7 - January 28

Asymptotic Analysis of Algorithms, Arrays and Linked Lists

Deriving Upper Bounds from Code
Inserting into an Array
Sorting Orders

Announcements/Reminders

- **Assignment 1** solution released
- ***splitArrayHarder***: an **extended** version released
- Office Hours: 3pm to 4pm, Mon/Tue/Wed/Thu
- Contact Information of TAs on common eClass site

Determining the Asymptotic Upper Bound (1)

approx.

↳ vs. Counting # B_3

```
1  int maxOf (int x, int y) {
2    int max = x;
3    if (y > x) {
4      max = y;
5    }
6    return max;
7  }
```

$O(\underbrace{1}_{\ell 2} + \underbrace{1}_{\ell 3} + \underbrace{1}_{\ell 4} + \underbrace{1}_{\ell 6}) = O(4 \cdot n^0)$

$= O(n^0)$

$O(1)$

Determining the Asymptotic Upper Bound (2)

```
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; }
```

1 n Iterations

1

1

$$O(1 + n \cdot 1 + 1) = O(n+2)$$

$$= O(n)$$

L2 L3 L4~L6 L7

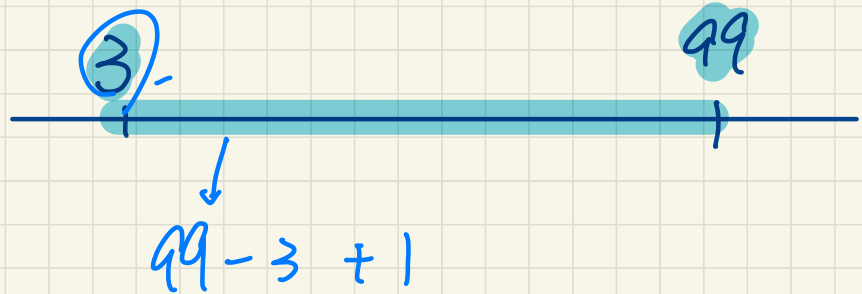
$[a, b]$

↳ $b - a + 1$. integers in the closed interval

$[0, n-1] \rightarrow (n-1) - 0 + 1 = n$.

$[3, 99]$

$99 - 3$



Determining the Asymptotic Upper Bound (3.1)

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

P1: lines 4–6 — O(1) ↳ each exec

P2: line 7 — O(1)

line 8 — O(1)

* P1 gets executed for every combination of (i, j)
* P2 gets exec. for every value of i

Patterns of Loop Counters

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

outer loop runs $[0, n-1]$ ↓ n iterations

$(2, 1) \rightarrow$ P1 exec once

$n \cdot n$ cells in the matrix

i values

j values

$$O(n^2 \cdot 1 + n \cdot 1 + 1)$$

n^2: # (i, j) combinations; 1: L4–L6; n: # i values; 1: L7

$$= O(n^2 + n + 1)$$

$$= O(n^2)$$

Determining the Asymptotic Upper Bound (3.2)

```
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; }
```

$O(10{,}000 \cdot n)$

$= O(n)$

```
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; }
```

$O(1M \cdot 1M)$

$= O(n \cdot n^0)$

$= O(1)$

a.length

b.length

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

$O(n \cdot m) = O(n^2)$

Determining the Asymptotic Upper Bound (4)

```
1  int sumMaxAndCrossProducts (int[] a, int n) {
2    int max = a[0];                                  1
3    for(int i = 1; i < n; i ++) {                    n
4      if (a[i] > max) { max = a[i]; }
5    }
6    int sum = max;                                   1
7    for (int j = 0; j < n; j ++) {                   n^2
8      for (int k = 0; k < n; k ++) {
9        sum += a[j] * a[k]; } }
10   return sum; }                                    1
```

$$O(1 + n + 1 + n^2 + 1)$$

$$= O(n^2 + n + 3) = O(n^2)$$

Determining the Asymptotic Upper Bound (5)

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

exec for each (i, j)

Pattern of (i, j)

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

$[0, n-1] \to n$ pts

$[1, n-1] \to n-1$

$[2, n-1] \to n-2$

$[n-1, n-1] \to 1$

$$\#(i,j) = n + (n-1) + (n-2) + \cdots + 1 = \frac{(n+1) \cdot n}{2} \quad O(n^2)$$

$$O\left(\underbrace{1}_{\sim 2} + \underbrace{n^2}_{\#(i,j)} \cdot \underbrace{1}_{\sim 5} + \underbrace{1}_{\sim 6}\right)$$

$$= O(n^2 + 2) = O(n^2)$$

Asymptotic Upper Bound: Arithmetic Sequence/Progression

$$\underbrace{\overbrace{t}^{t+0\cdot c}}_{\text{start term}} + \overbrace{(t+c)}^{\text{common difference}} + (t+2\cdot c) + \dots + (t+(n-1)\cdot c)$$

$$\#\ \text{terms} : n$$

$$\text{Sum} : \frac{[t + (t+(n-1)\cdot c)]\cdot n}{2} = \frac{c\cdot n^2 + (2t-c)\cdot n}{2}$$

$$\hookrightarrow O(n^2)$$

Inserting into an Array

object creation: O(1)

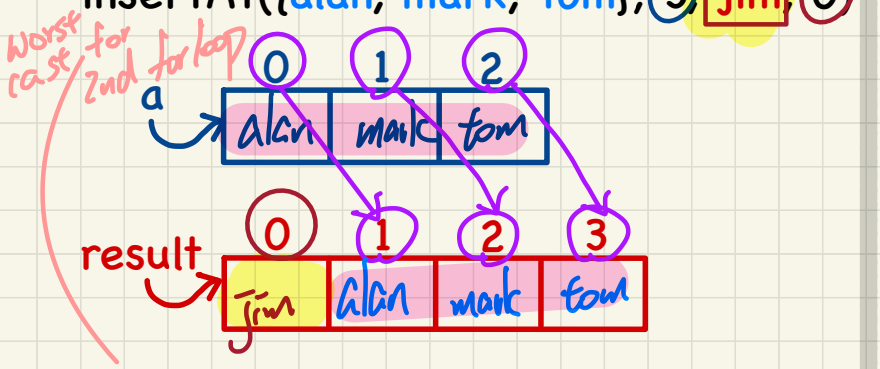
a.length

insert "e" to index "i" → 0 ≤ i ≤ n (insert first / insert last)

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

Example:

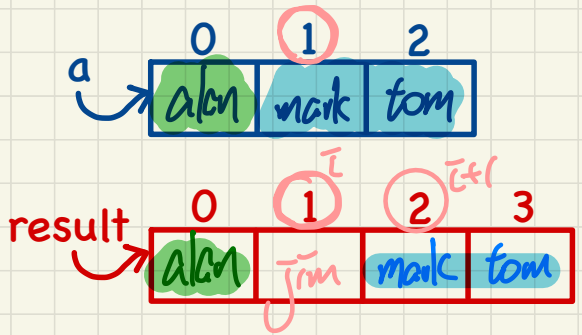
insertAt({alan, mark, tom}, 3, jim, 0)



Example:

insertAt({alan, mark, tom}, 3, jim, 1)

O(1 + n·1 + 1 + (n-i)·1+1) = O(n)



Exercise: insertAt({alan, mark, tom}, 3, jim, 3) → worst case for 1st for loop

Lecture 8 - January 30

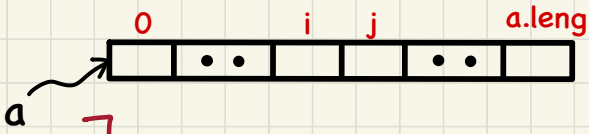
Arrays and Linked Lists

Exercise: Relating Sorting Orders
Selection vs. Insertion Sorts

Announcements/Reminders

- Assignment 1 solution released
- ***splitArrayHarder***: an **extended** version released
- Lecture notes template available
- Office Hours: 3pm to 4pm, Mon/Tue/Wed/Thu
- Contact Information of TAs on common eClass site

Sorting Orders of Arrays

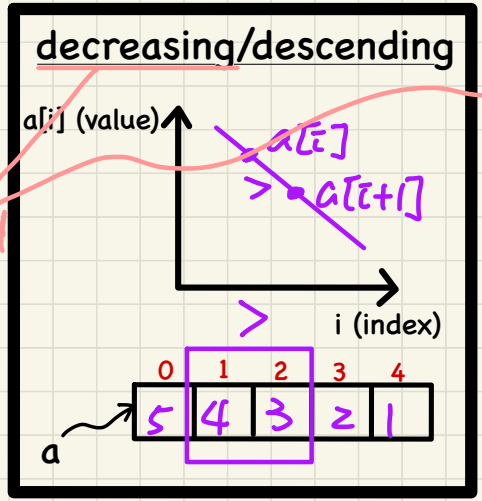


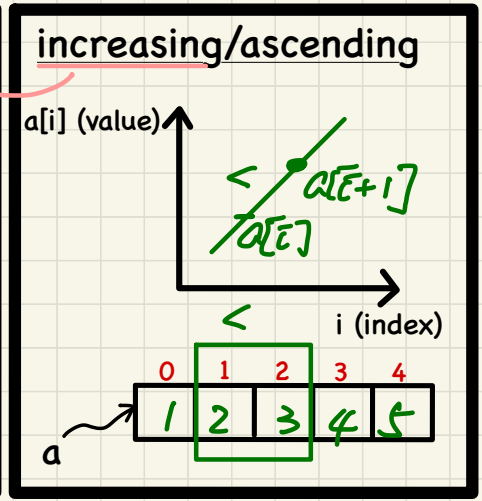
non-descending

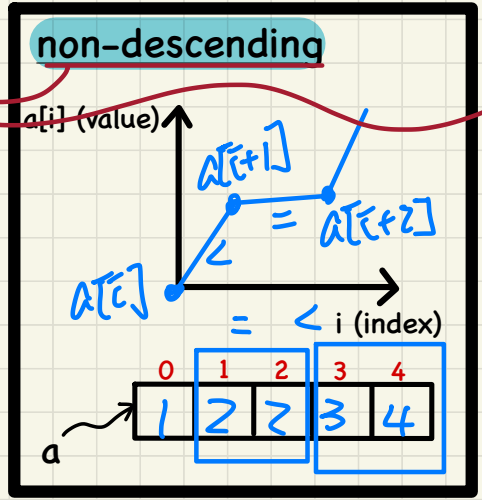
$\triangleq \neg(\text{descending})$

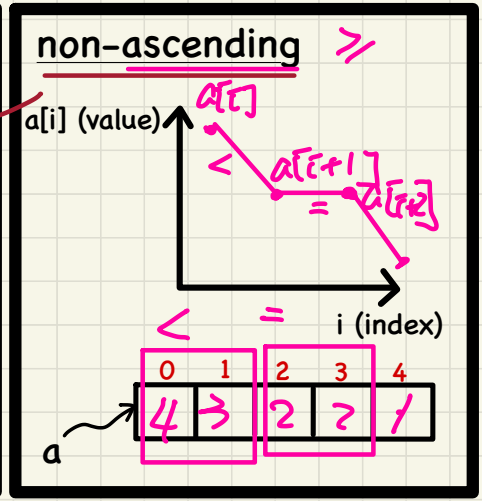
$\equiv \neg(a[i] > a[i+1])$

$\equiv a[i] \leq a[i+1]$









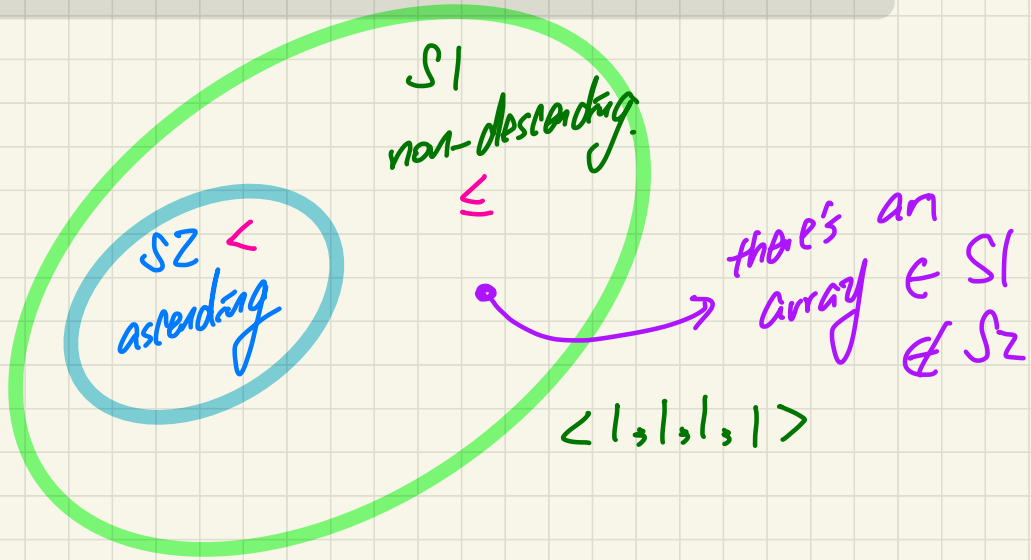
Exercise: Relating Sets of Sorted Arrays

Q. Consider the following two sets:

- S1: all arrays sorted in a **non-descending** order
- S2: all arrays sorted in an **ascending** order.

Formulate the relation between these two sets.

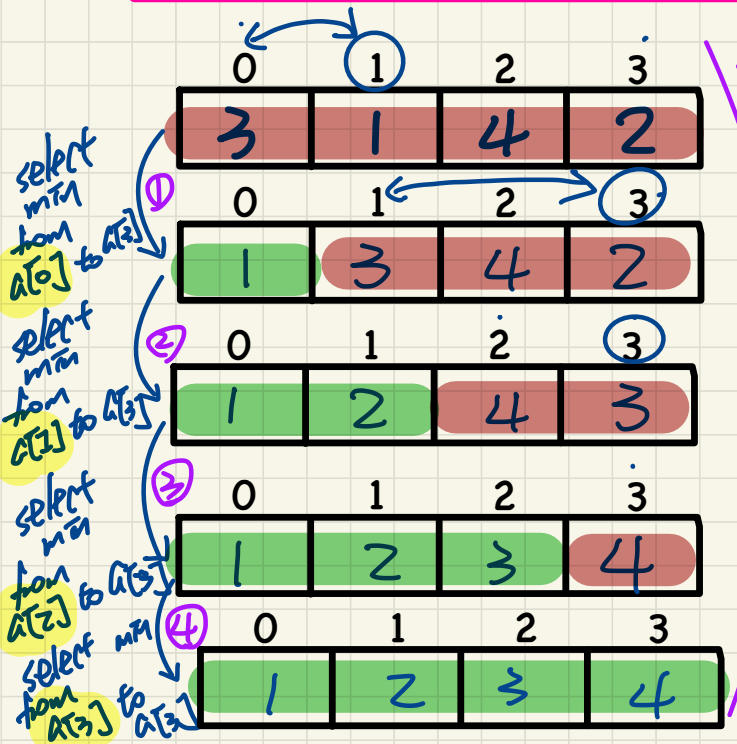
$S_2 \subset S_1$



Selection Sort

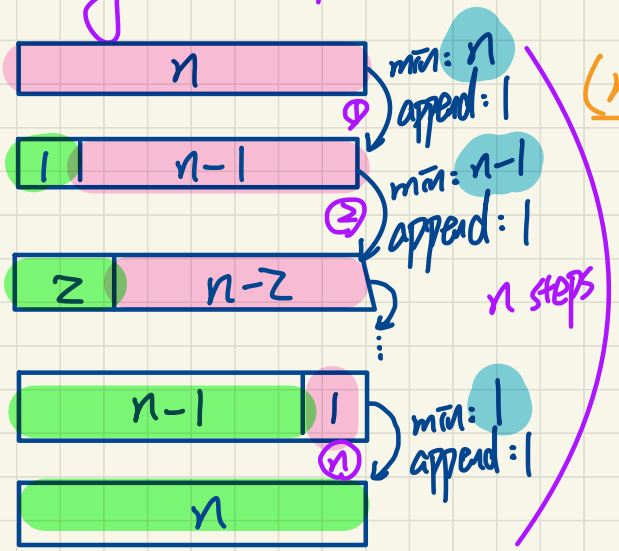
$O(\underbrace{n}_{\#\text{ steps}} \cdot \underbrace{1}_{\text{swap}} + [\underbrace{n}_{\text{1st sel.}} + (n-1) + \cdots + \underbrace{1}_{\text{last sel.}}]) = O(n + n^2) = O(n^2)$

Keep **selecting** minimum from the **unsorted** portion and appending it to the end of **sorted** portion.



steps? size of the ultimate sorted portion n

Decreasing Costs of select



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

Insertion Sort

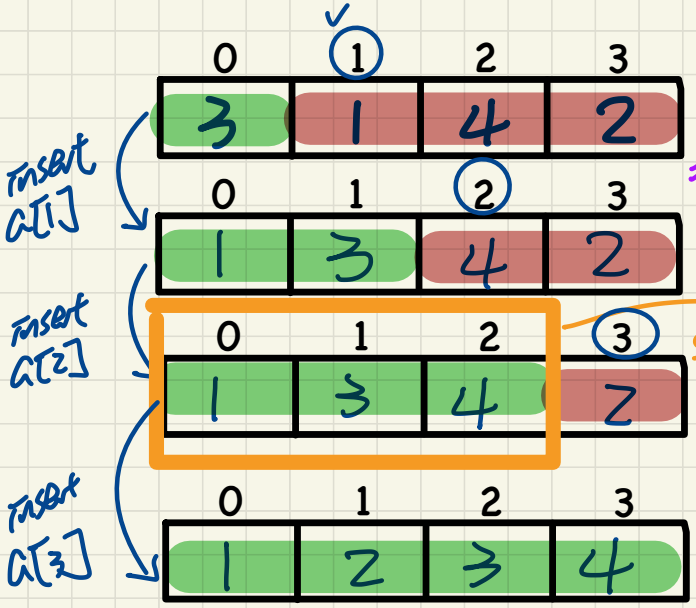
$O(1)$

$O((n-1) \cdot 1 + [1+2+\cdots+(n-1)]) =$

(n-1) steps · 1 get left-most of unsorted · $\frac{(1+(n-1))\cdot(n-1)}{2}$

$O(n-1+n^2) = O(n^2)$

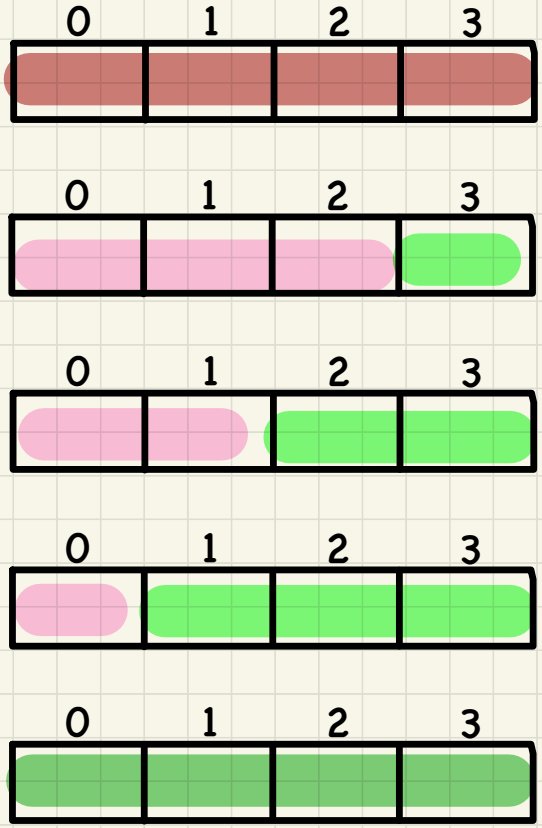
Keep getting 1st element from the **unsorted** portion and **inserting** it to the **sorted** portion.



size of sorted portion is n-1

n-1 steps

step	get	size of sorted portion	max cost for insert
1	a[1]	1	1
2	a[2]	2	2
3	a[3]	3	3
⋮		⋮	
n-1	a[n-1]	n-1	n-1



(1) Sorted portion is maintained on the right

(2) order: non-ascending.

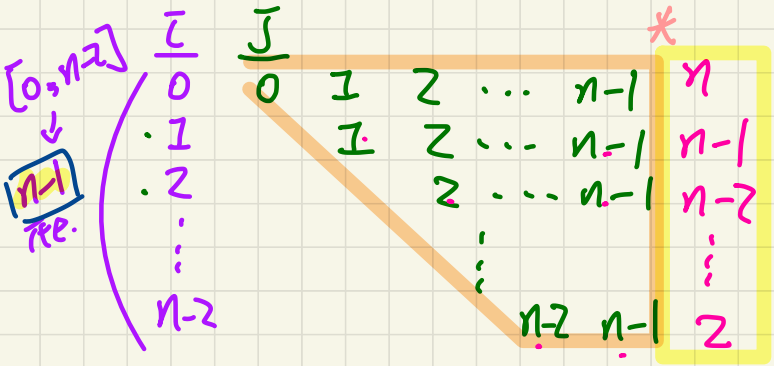
Selection Sort: Deriving Asymptotic Upper Bound

```
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;
```

a.length

exec. according to values of i

exec. according to values of (i, j)



$$O\left((n-1)\cdot 1 + \left[n + (n-1) + (n-2) + \cdots + 2\right]\cdot 1\right)$$

$\frac{(n+2)\cdot(n-1)}{2}$ → $O(n^2)$

(n-1): # it. outer loop

1: L3, L6~L8

[ ]·1: L5

$$= O(n - 1 + n^2) = O(n^2)$$

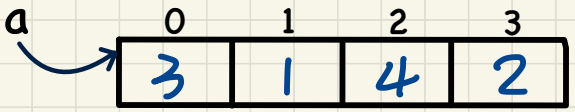
Insertion Sort: Deriving Asymptotic Upper Bound

```
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;
```

(Exercise)

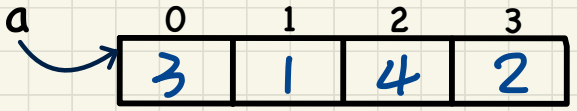
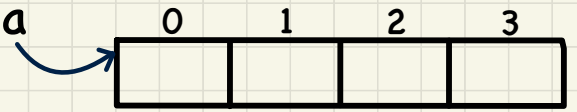
Selection Sort in Java

```
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;
```



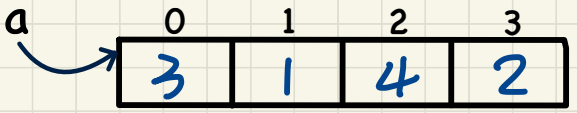
Inner Loop: select the next min from a[i] to a[n - 1] and put it to the end of the sorted region.

Outer Loop:
At the end of each iteration of the for-loop,
a is sorted from a[0] to a[i].

i	inner loop: j from ? to ?	midIndex at L6	after L6 - L8, a becomes?
			
			

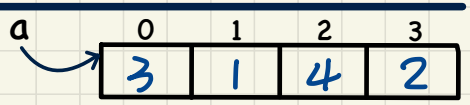
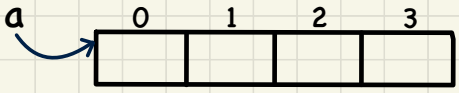
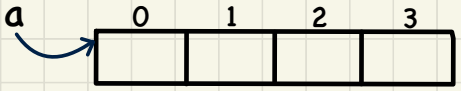
Insertion Sort in Java

```
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;
```



Inner Loop: find out where to insert current into a[0] to a[i] s.t. that part of a becomes sorted.

Outer Loop:
At the end of each iteration of the for-loop,
a is sorted from a[0] to a[i].

i	current after L3	j at L8	after L8, a becomes?
			
			
			

Lecture 9 - February 4

Arrays and Linked Lists

Q: Mixing Insertion & Selection Sorts
SLL: Visual Introduction & Operations
SLL in Java: Node vs. SinglyLinkedList

Announcements/Reminders

- **Assignment 2** (on **SLL**) to be released soon
- **Assignment 1** solution released
- ***splitArrayHarder***: an **extended** version released
- Lecture notes template available
- Office Hours: 3pm to 4pm, Mon/Tue/Wed/Thu
- Contact Information of TAs on common eClass site

* Precondition: $\forall i, j \cdot 0 \le i < \frac{n}{2} \wedge \frac{n}{2} \le j < n \Rightarrow input[i] \le input[j]$

Exercise: Mixing the "Best" from both Sorts?

Recall:

- In insertion sort, costs of insertions are increasing.
- In selection sort, costs of selections are decreasing.

Idea:

- Perform insertion sort until half of the input is sorted.
- Perform selection sort to finish sorting the remaining half.

Q: Will this "new" algorithm perform better than $O(n^2)$?

Not working

<2, 1, 7, 3, 4, 0>

↓ insertions (3)

<1, 2, 7, 3, 4, 0> ✓ ✓ ✓

↓ selections (3)

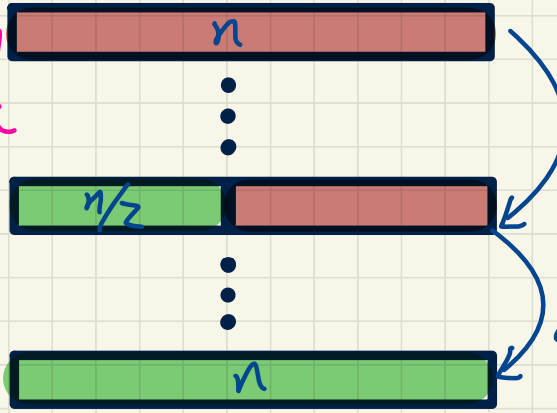
× <1, 2, 7, 0, 3, 4>

Working * <2, 0, 1, 4, 7, 3>

$O(n^2)$ No (a)

(b) justify

(c) assumption on input structure



$O(1 + 2 + \dots + \frac{n}{2}) = O(n^2)$ — $\frac{(1 + n/2) \cdot n/2}{2}$ — 1st ins. — last ins

$O(\frac{n}{2} + (\frac{n}{2} - 1) + \dots + 1) = O(n^2)$ — 1st sel. — last sel.

$O(n^2 + n^2) = O(n^2)$

Singly-Linked Lists (SLL): Visual Introduction

- A chain of connected nodes (via aliasing)
- Each node contains:
  - + reference to a data object
  - + reference to the next node
- Head vs. Tail
- Accessing a position in a linear collection:
  - + Array uses absolute indexing: O(1)
  - + SLL uses relative positioning: O(n)
- The chain may grow or shrink dynamically.

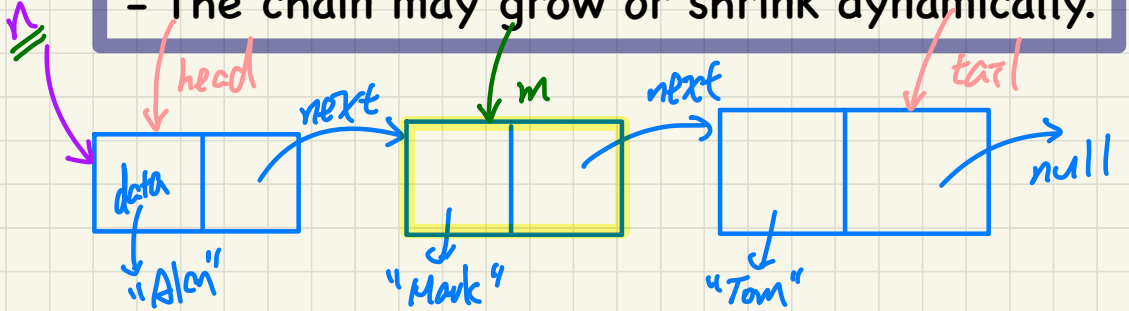
multiple expressions referring to same object (e.g. m == n.next)

(n: 1st node
n.data == "Alan"

(n.next != null
n.next.data == "Mark"

n.next.next.next.data
null

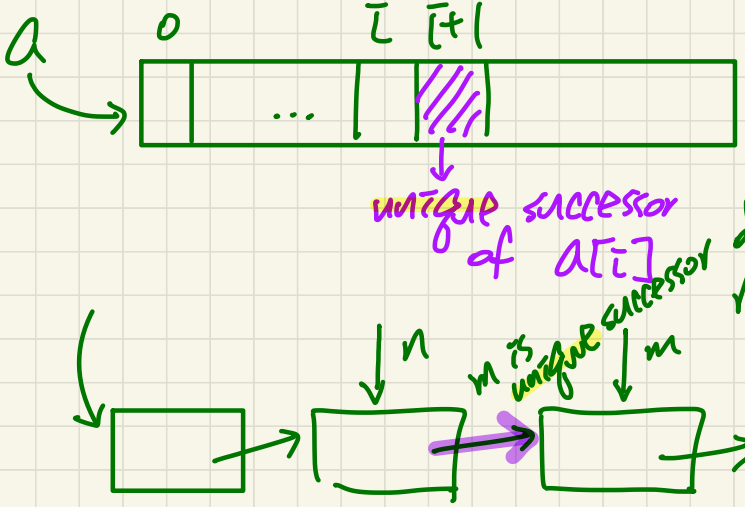
NullPointerError

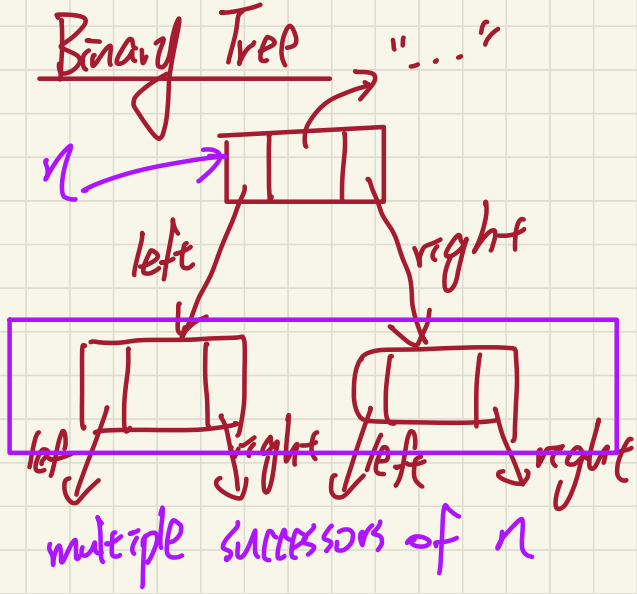


Linear vs. Non-Linear collections

Linear: each position in the collection has a unique successor.

Non-Linear: each position has multiple successors





Absolute Indexing of Arrays

O(1)

a[i]

starting address of the array

int. exp: calculating the address offset from a

Relative Positioning of LL

O(n)

l.get(i)

worst case: i gets close to the size of chain (n) ↘ ≈ # next ref. lookup

head

0 1 2 3

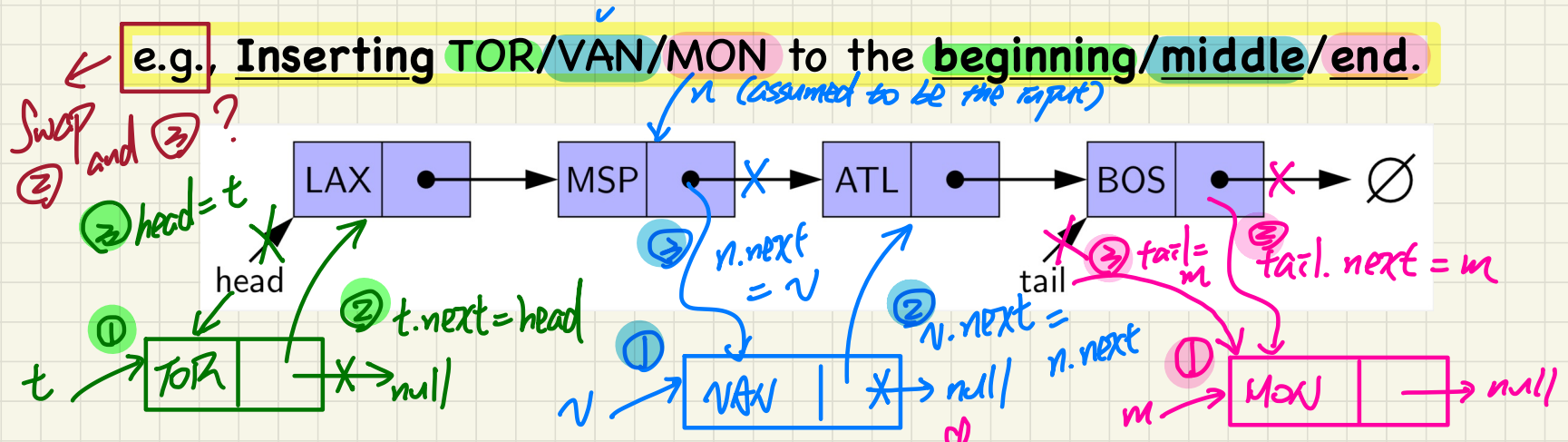
next next next next null

relative positioning (unique successor)

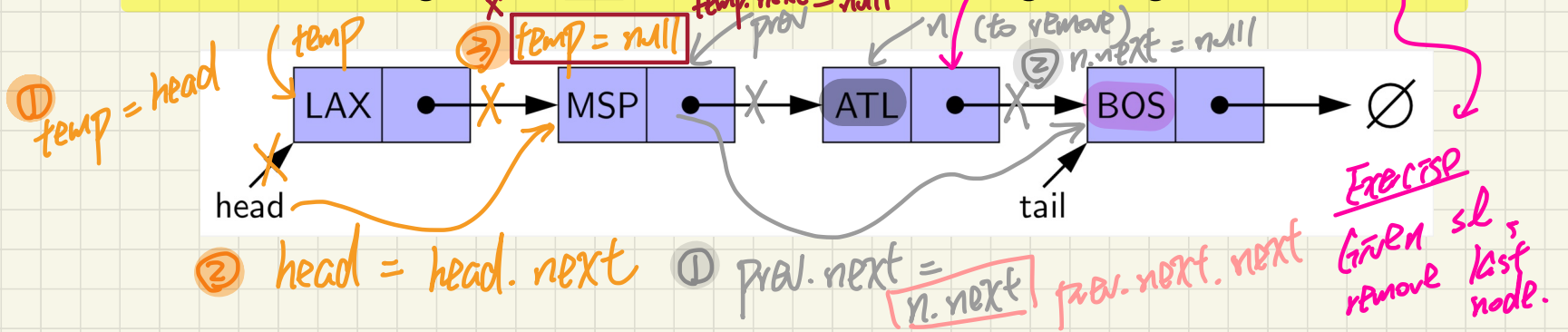
last node: next == null

A SLL Grows or Shrinks Dynamically

e.g., Inserting TOR/VAN/MON to the beginning/middle/end.



e.g., Removing LAX/ATL/BOS from the beginning/middle/end.



Implementing SLL in Java: SinglyLinkedList vs. Node

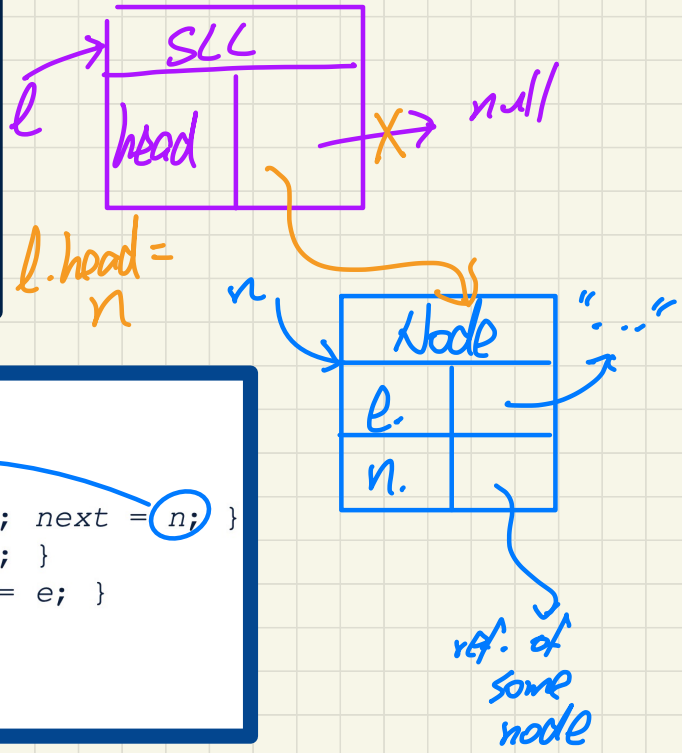
```
public class SinglyLinkedList {
   private Node head = null;
   public void setHead(Node n) { head = n; }
   public int getSize() { ... }
   public Node getTail() { ... }
   public void addFirst(String e) { ... }
   public Node getNodeAt(int i) { ... }
   public void addAt(int i, String e) { ...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

recursive type

```
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; }
}
```

next.

Runtime



Lecture 10 - February 6

Arrays and Linked Lists

SLL: List Constructions
SLL: getSize and getTail
Trading Space for Time: tail and size

Announcements/Reminders

- **Assignment 2** (on **SLL**) released
  + **Required** studies: Generics in Java (Slides 33 – 36)
  + **Recommended** studies: extra SLL problems
- **Assignment 1** solution released
- ***splitArrayHarder***: an **extended** version released
- Lecture notes template available
- Office Hours: 3pm to 4pm, Mon/Tue/Wed/Thu
- Contact Information of TAs on common eClass site

SLL: Constructing a Chain of Nodes

Alan → Mark → Tom

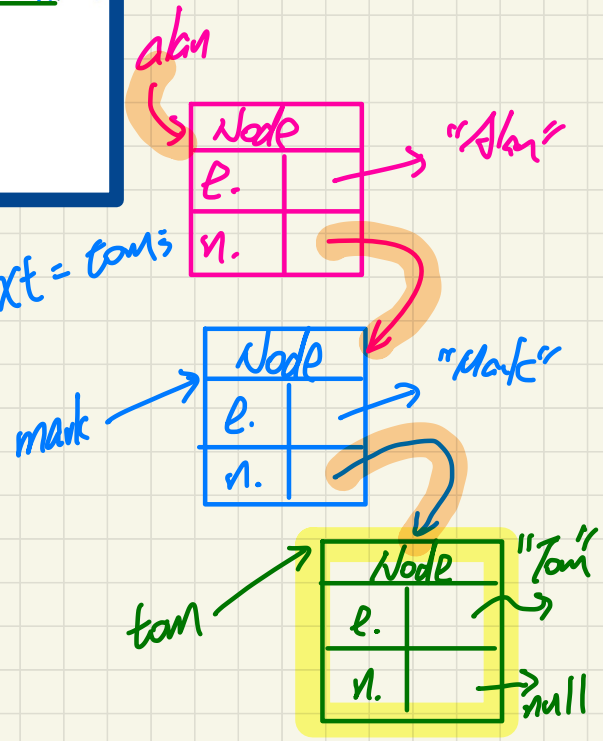
```
public class Node {
   private String element;
   private Node next;
   public Node(String e, Node n) { element = e; next = n; }
   public String getElement() { return element; }
   public void setElement(String e) { element = e; }
   public Node getNext() { return next; }
   public void setNext(Node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

Approach 1

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

mark.next = tom;

alan.next = mark

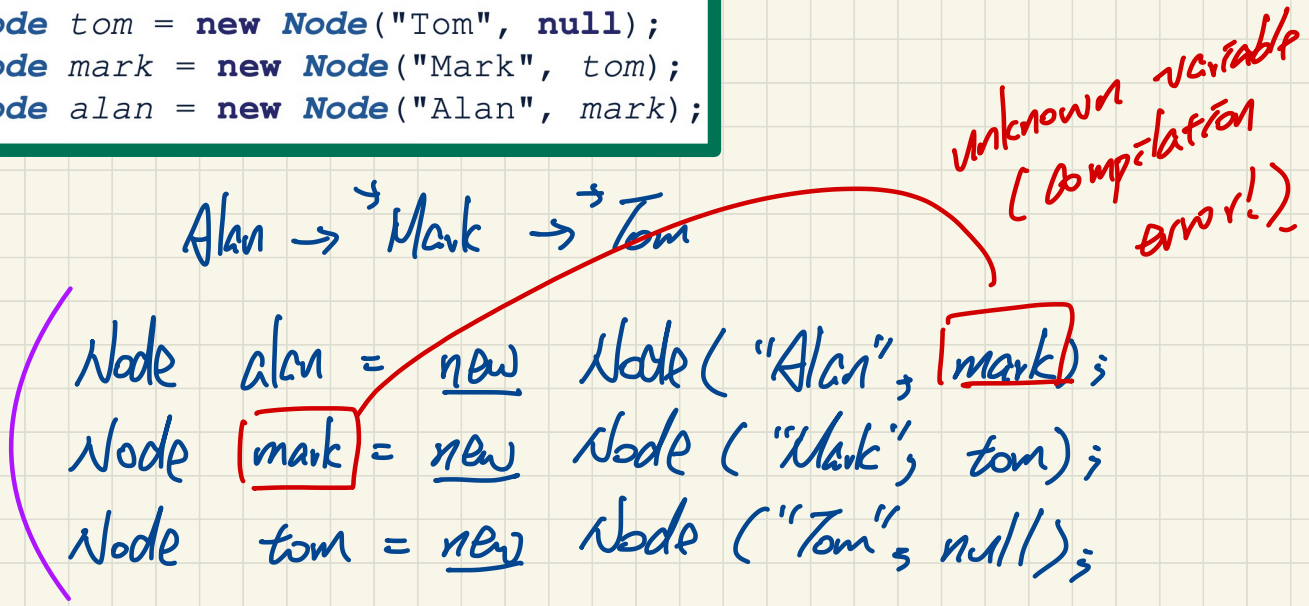


Aliasing

1. tom
2. mark.next → 3. alan.next.next

Approach 1

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



Node alan;

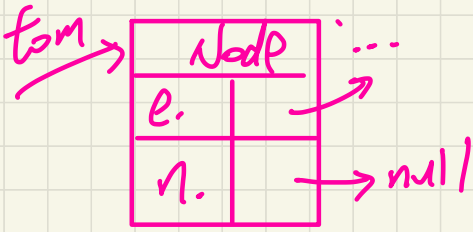
SLL: Constructing a Chain of Nodes

```
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; }
}
```

alan

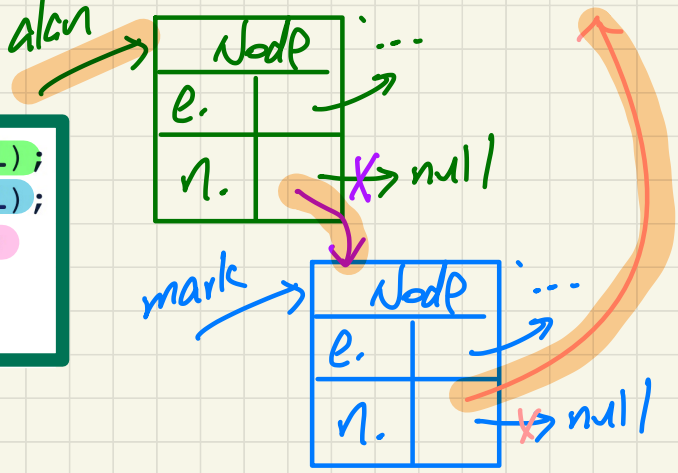
mark

tom



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);
```

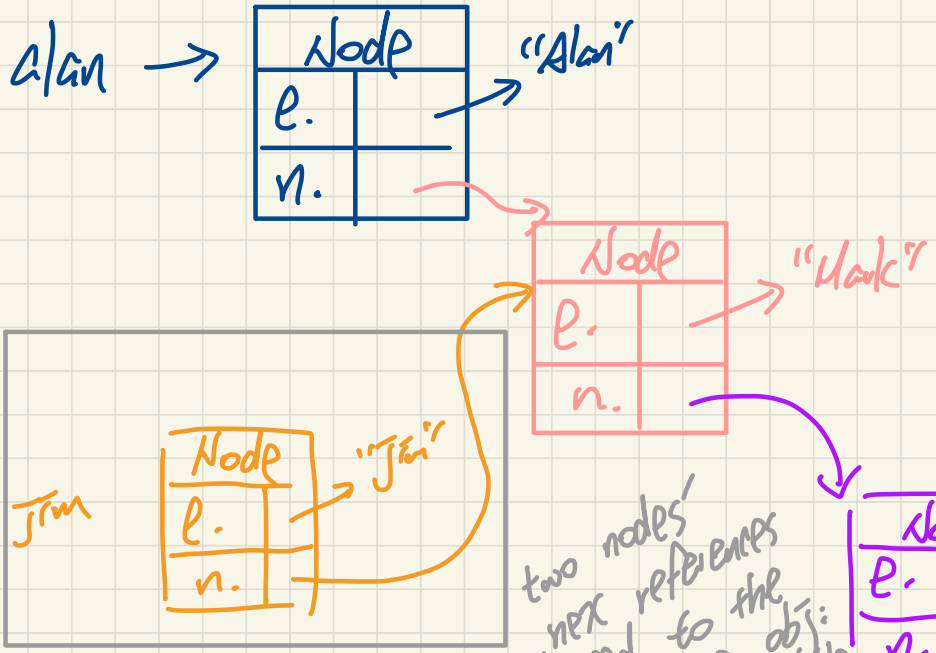


alan.next = mark;
mark.next = tom;

```
Node alan = new Node("Alan", );
```

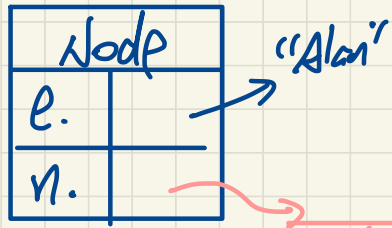
new Node("Mark",);

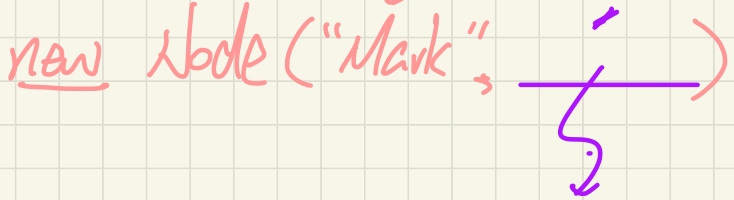
new Node("Tom", null)

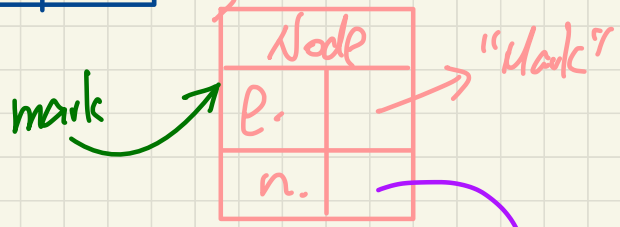


two nodes' next references aliased to the same obj: possible but not recommended!

```
Node alan = new Node("Alan", );
```



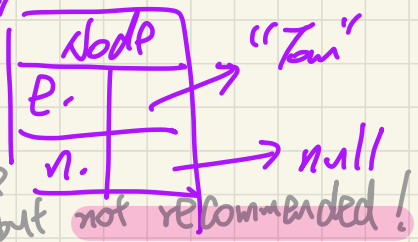





```
Node mark =
    alan.next;
```

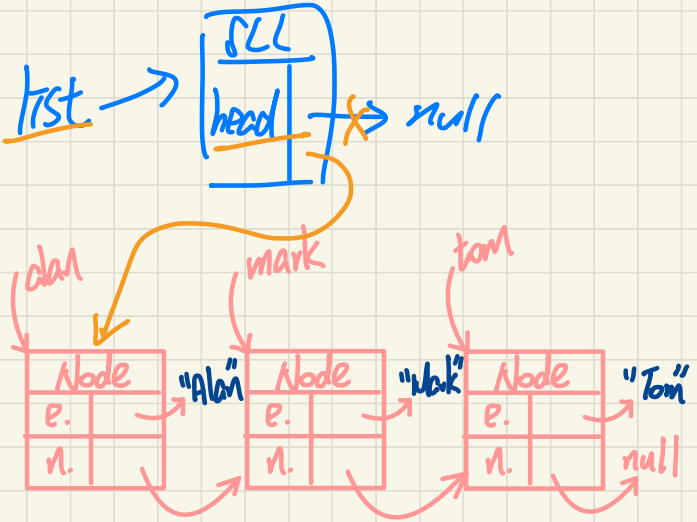
new Node("Tom", null)

two nodes' next references attached to the same obj: possible but not recommended!



SLL: Setting a List's Head to a Chain of Nodes

```
public class SinglyLinkedList {
   private Node head = null;
   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() { ... }
}
```



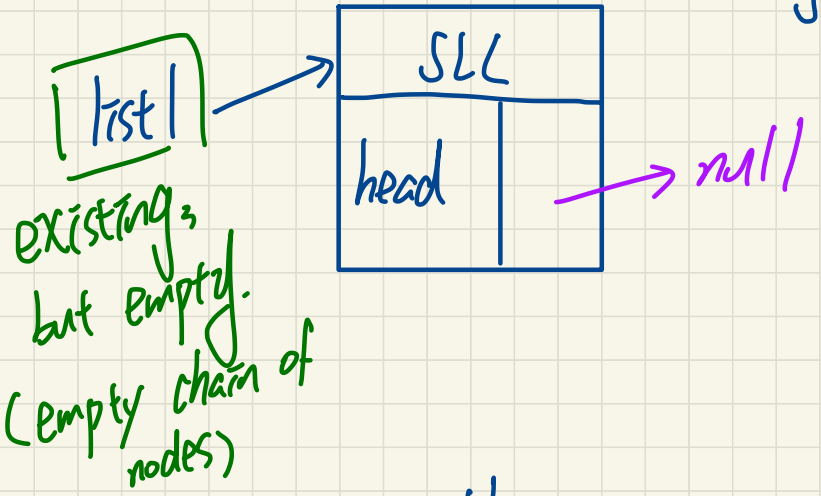
Approach 1

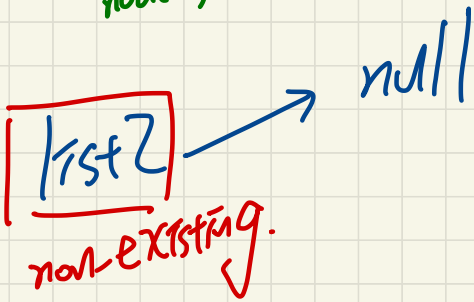
```
Node tom = new Node("Tom", null);
Node mark = new Node("Mark", tom);
Node alan = new Node("Alan", mark);
SinglyLinkedList list = new SinglyLinkedList();
list.setHead(alan);
```

Empty SLL

default const.

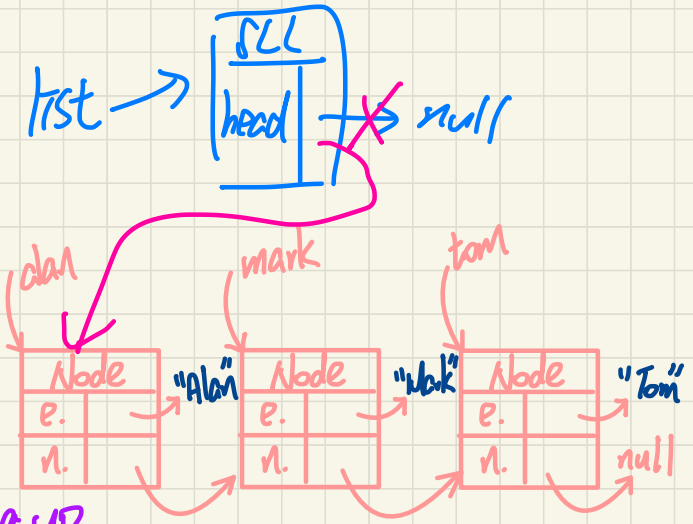
```
SLL list1 = new SLL();
SLL list2 = null;
```





SLL: Setting a List's Head to a Chain of Nodes

```
public class SinglyLinkedList {
   private Node head = null;
   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() { ... }
}
```



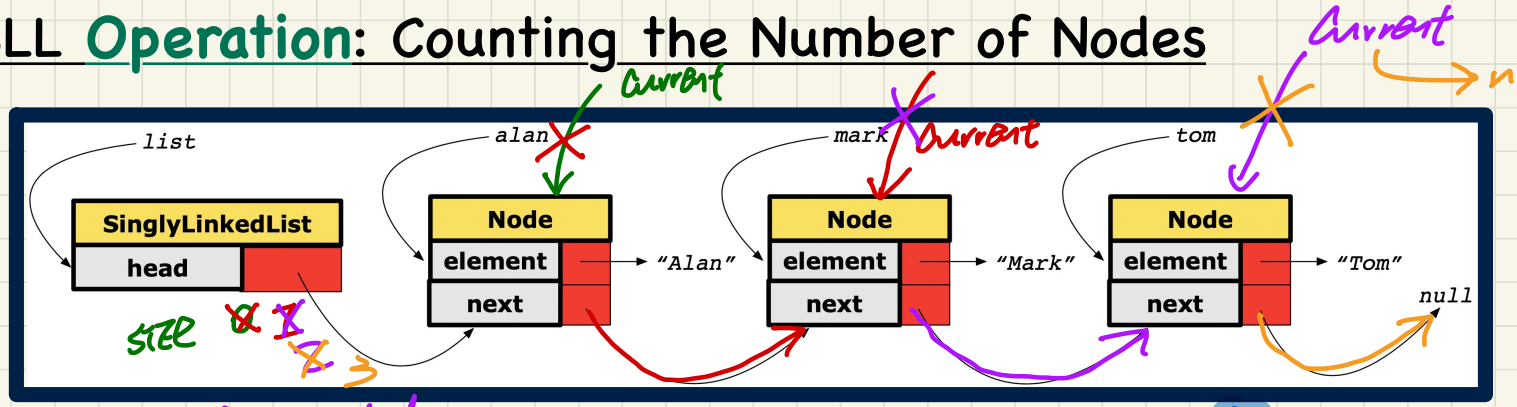
Approach 2

list.setHead(null) → clearing up the list

```
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);
```

list.setHead(tom)
↳ compiles and runs but potentially logical errors

SLL Operation: Counting the Number of Nodes



a SLL method

```
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  }
```

loop counter

exit when current == null

O(n · 1) = O(n)

ft. ↳ L5, L6

O(n) # iterations = # nodes

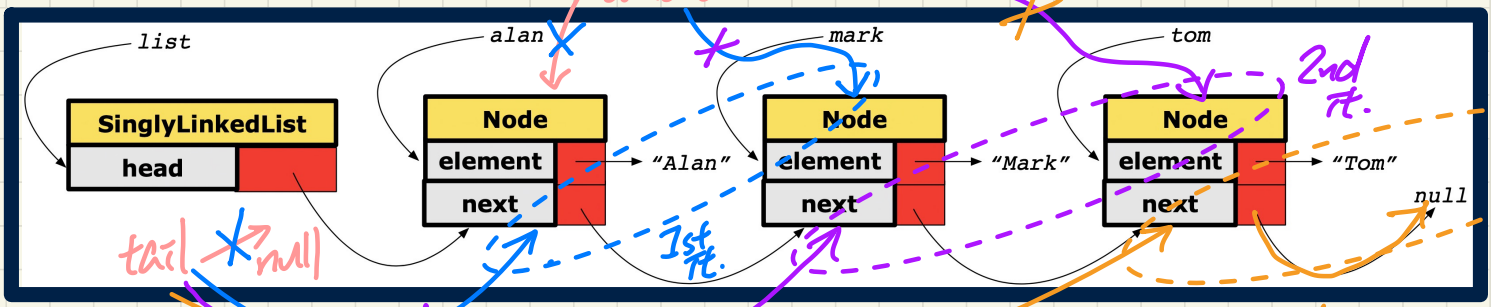
Trace: list.getSize()

current	current != null	End of Iteration	size
alan	alan != null	current == mark	1
mark	mark != null	current == tom	2
tom	tom != null	current == null	3
null	null != null		

(F)

Exercise: Use "current" only to implement getT. current is always one node ahead of tail.

SLL Operation: Finding the Tail of the List




```
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  }
```

Trace: list.getTail()

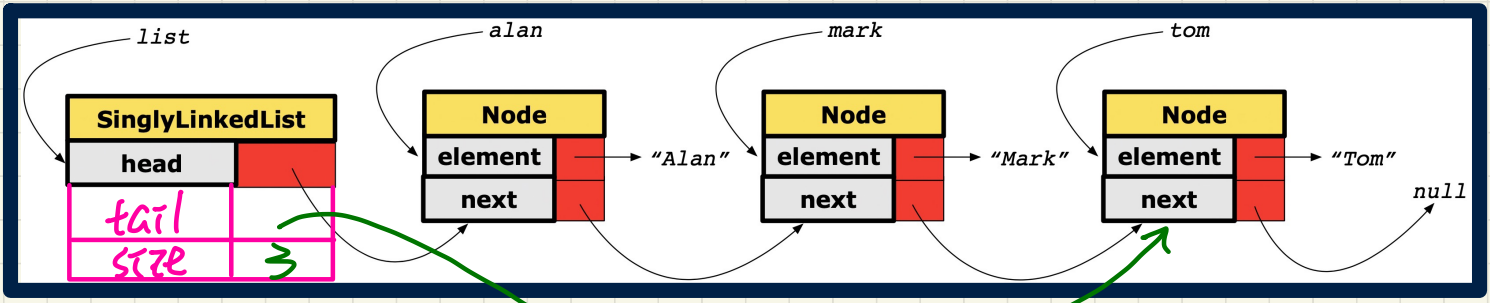
current	current != null	End of Iteration	tail

Iterations = size of list O(n)

O(n · 1) L5, L6 = O(n)

SLL: Trading Space for Time

waste more memory space to make the subsequent computations cheaper



SLL class

- head
- tail
- size

attributes ↓ more cost of space

```
SLL list = new SLLC();
    ⋮
```

attribute access ↓ less cost for running time

- list.tail
- list.size

O(1)

↳ turning loops to att. access

Catch for methods that might impact head, tail, or size of the SLL, bodies of imp. must update them properly.

Tutorial

Recursion Problem: splitArrayHarder

Coding in Java
Tracing
Exercises

Problem on Recursion

https://www.eecs.yorku.ca/~wangcw/teaching/lectures/2025/W/EECS2101/exercises/EECS2101-W25-Problem-Recursion-splitArray-Spec.pdf

A useful extension to the original `splitArray` problem:

+ Return an ArrayList of size 2:
+ If a split of equal sums (assumed to be unique) is possible:
  * index 0 of the returned list stores ArrayList of integers representing group 1.
  * index 1 of the returned list stores ArrayList of integers representing group 2.
+ If a split is not possible, both indices store empty lists.

e.g., **splitArrayHarder**({2, 2}) → <<2>, <2>>
e.g., **splitArrayHarder**({2, 3}) → <<>, <>>
e.g., **splitArrayHarder**({5, 2, 3}) → <<5>, <2, 3>>
e.g., **splitArrayHarder**({5, 2, 2}) → <<>, <>>

splitArrayHarder: Java Implementation

call by value

only modified in base case

```java
public ArrayList<ArrayList<Integer>> splitArrayHarder(int[] ns) {
    ArrayList<ArrayList<Integer>> output = new ArrayList<>();
    splitArrayHarderHelper(ns, 0, 0, 0, new ArrayList<Integer>(), new ArrayList<Integer>(), output);
    if(output.isEmpty()) {
        output.add(new ArrayList<Integer>());
        output.add(new ArrayList<Integer>());
    }
    return output;
}
```

g1 g2

either of them modified in every recursive call (non-base case)

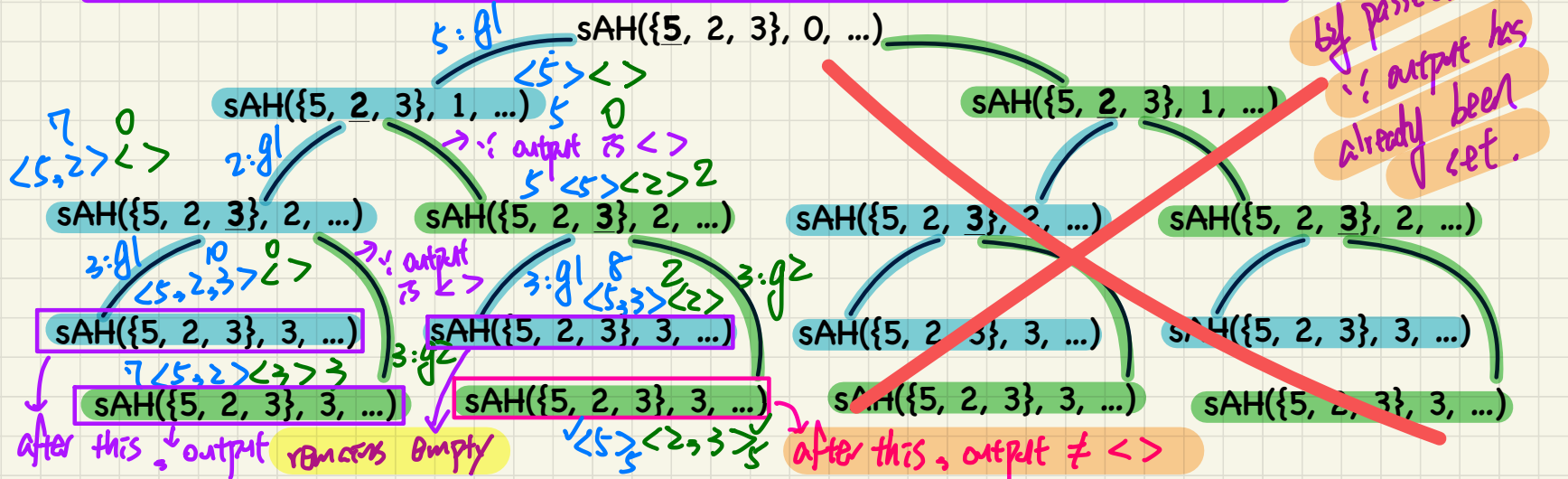
```java
private void splitArrayHarderHelper(int[] ns, int i, int sumOfGroup1, int sumOfGroup2,
        ArrayList<Integer> group1, ArrayList<Integer> group2, ArrayList<ArrayList<Integer>> output) {
    if(i == ns.length) {
        if(sumOfGroup1 == sumOfGroup2) {
            output.add(group1);
            output.add(group2);
        }
    }
    else {
        ArrayList<Integer> group1Extended = new ArrayList<>(group1);
        group1Extended.add(ns[i]);
        splitArrayHarderHelper(ns, i + 1, sumOfGroup1 + ns[i], sumOfGroup2, group1Extended, group2, output);

        if(output.isEmpty()) {
            ArrayList<Integer> group2Extended = new ArrayList<>(group2);
            group2Extended.add(ns[i]);
            splitArrayHarderHelper(ns, i + 1, sumOfGroup1, sumOfGroup2 + ns[i], group1, group2Extended, output);
        }
    }
}
```

splitArrayHarder: Tracing

Exercise: Trace in Eclipse

```
@Test
public void testSplitArrayHarder_03() {
    RecursiveMethods rm = new RecursiveMethods();
    int[] input = {5, 2, 3};
    ArrayList<ArrayList<Integer>> output = rm.splitArrayHarder(input);
    ArrayList<Integer> expectedGroup1 = new ArrayList<>(Arrays.asList(5));
    ArrayList<Integer> expectedGroup2 = new ArrayList<>(Arrays.asList(2, 3));
    assertTrue(output.size() == 2);
    assertEquals(expectedGroup1, output.get(0));
    assertEquals(expectedGroup2, output.get(1));
}
```

output : <<5>, <2,3>>



Lecture 11 - February 11

Arrays and Linked Lists

SLL: removeFirst, addLast
SLL: getNodeAt, insertAt

Announcements/Reminders

- ProgTest1 guide & example questions to be released
- *splitArrayHarder*: solution and tutorial video released
- Assignment 2 (on SLL) released
  + Required studies: Generics in Java (Slides 33 – 36)
  + Recommended studies: extra SLL problems
- Assignment 1 solution released
- Lecture notes template, Office Hours, TA Contact

SLL Operation: Inserting to the Front of the List

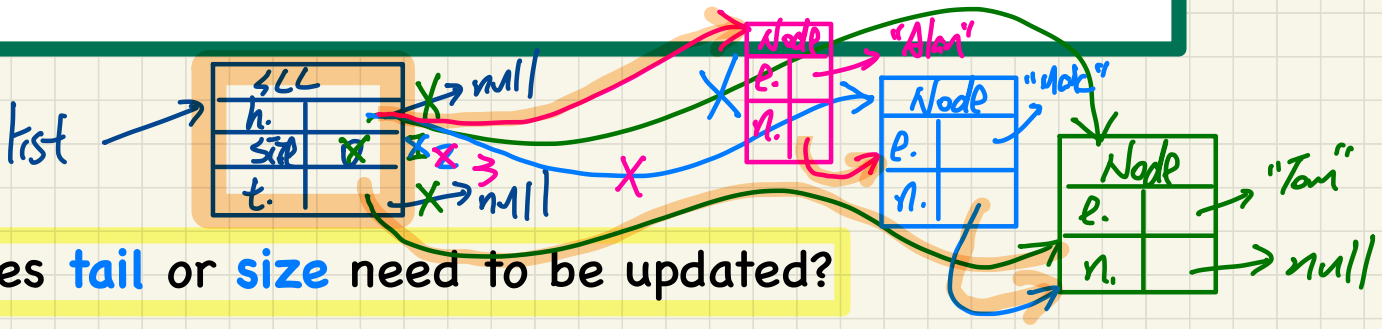
O(1)

```java
@Test
public void testSLL_02() {
    SinglyLinkedList list = new SinglyLinkedList();
    assertTrue(list.getSize() == 0);
    assertTrue(list.getFirst() == null);

    list.addFirst("Tom");
    list.addFirst("Mark");
    list.addFirst("Alan");
    assertTrue(list.getSize() == 3);
    assertEquals("Alan", list.getFirst().getElement());
    assertEquals("Mark", list.getFirst().getNext().getElement());
    assertEquals("Tom", list.getFirst().getNext().getNext().getElement());
}
```

first.size==0

what if interleaving addFirst and addLast

```
1  void addFirst (String e) {
2    head = new Node(e, head);
3    if (size == 0) {
4      tail = head;
5    }
6    size ++;
7  }
```



Q. Does **tail** or **size** need to be updated?

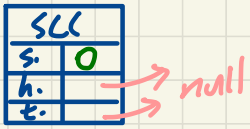
SLL Operation (sketch): Removing the First Node

void removeFirst()

Boundary Cases?

↳ size == 0 → Exception

↳ size == 1 → result: empty list



General Cases?

↳ size ≥ 2

O(1)

∵ none of the operations depends on the # nodes in the chain

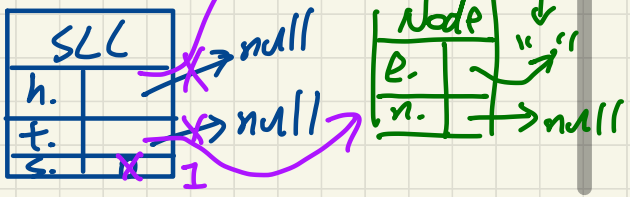
① , ② , ③ , ④

SLL Operation (sketch): Adding a Last Node

```
void addLast(String e)
```

Boundary Cases?

↳ size == 0



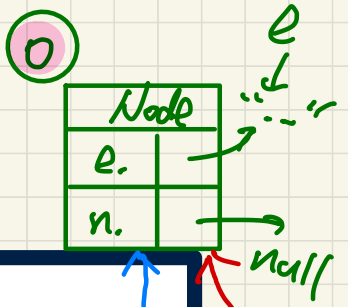
General Cases?

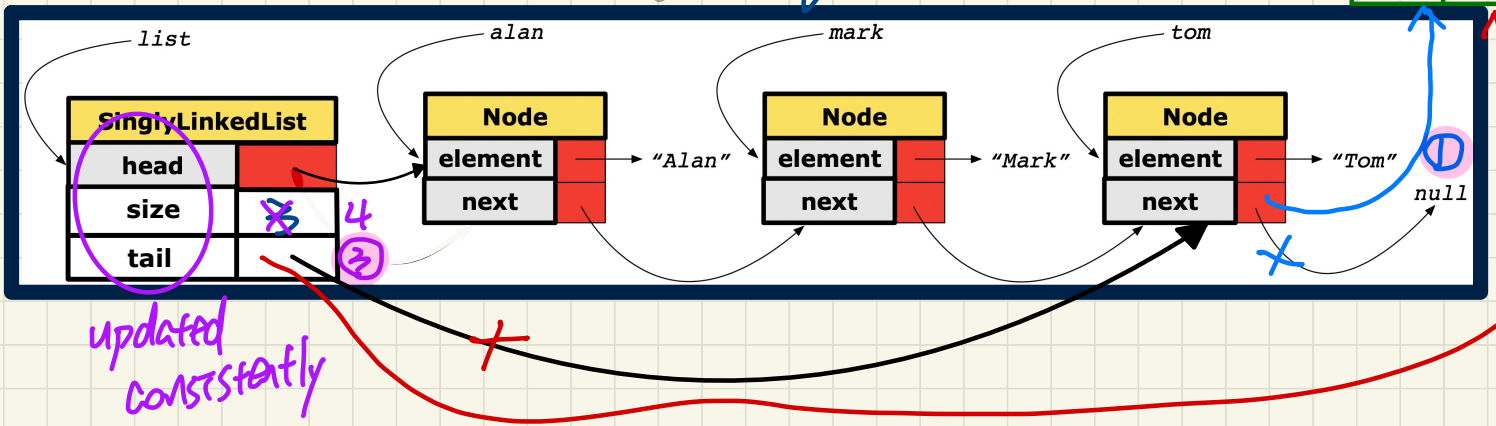
↳ size >= 1

O(1)

0, 1, 2, 3

all are variable assignments

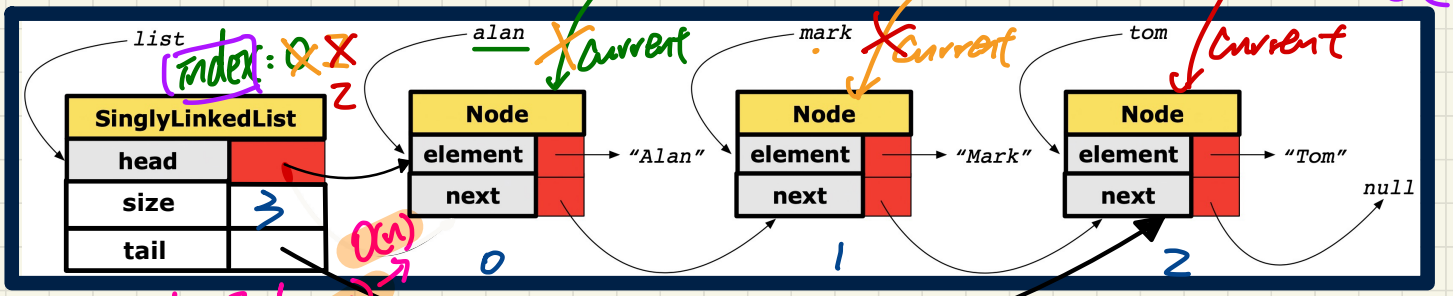




* current always references the node stored at index "index"

SLL Operation: Accessing the Middle of the List

O(i) = O(n-1) = O(n)



iterations = i

* to access the 2nd last node: getNodeAt (n-2)

$0 \le i \le lst.size - 1$

```
1  Node getNodeAt (int i) {
2    if (i < 0 || i >= size) { /* error
3    else {
4      int index = 0;
5      Node current = head;
6    ->while (index < i) { /* exit when
7    -> ->index ++;
8    -> ->current = current.getNext();
9      }
10     return current;
11   }
12 }
```

exit when: index == i

i == 2.

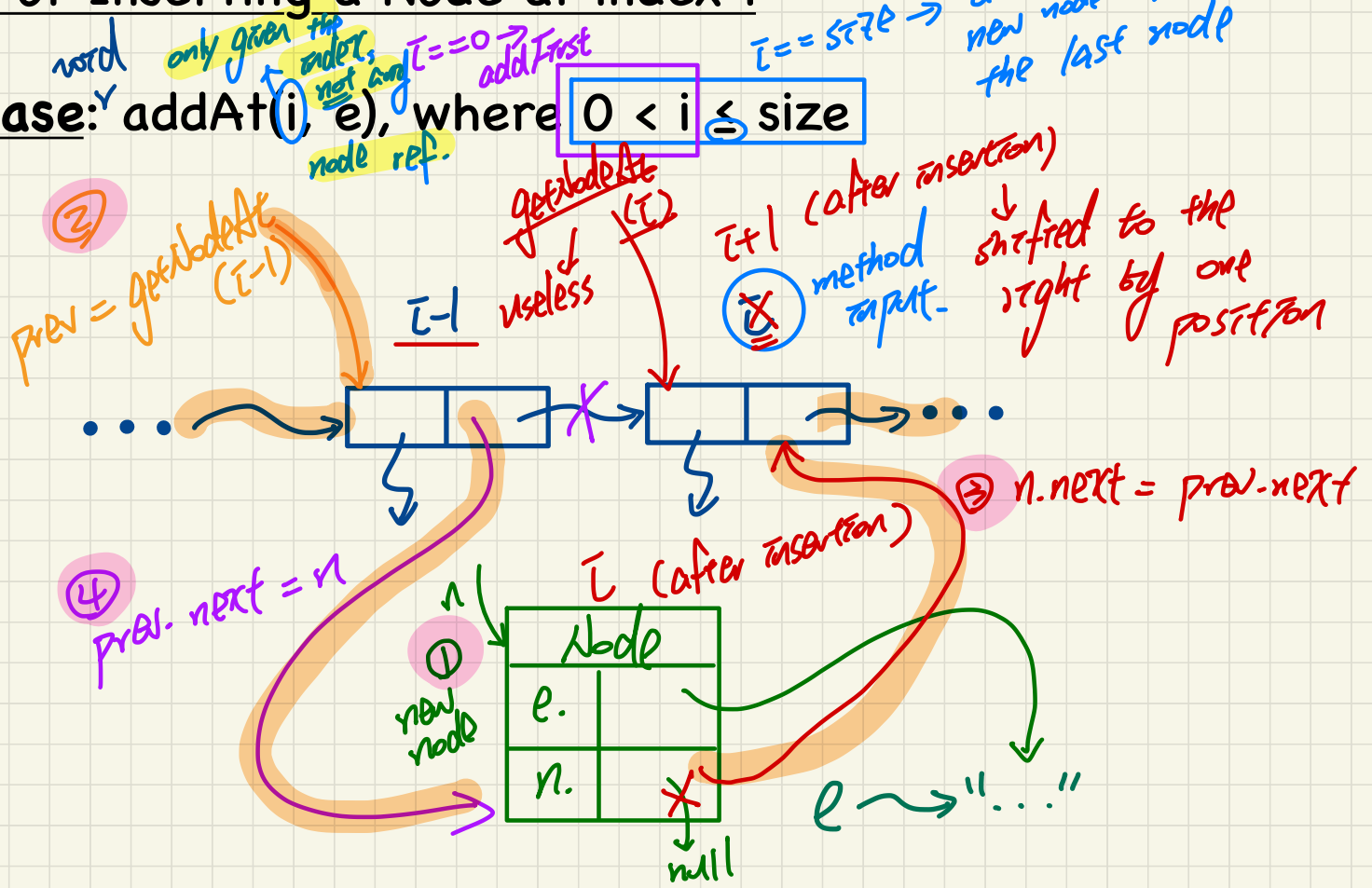
Trace: list.getNodeAt(2)

current	index	index < 2	Start of Iteration
alan	0	0 < 2	1st: index 0 → 1, current alan → mark
mark	1	1 < 2	2nd: index 1 → 2, current mark → tom
tom	2	2 < 2	
		F: exit	

Q. Does **tail** or **size** need to be updated? No!

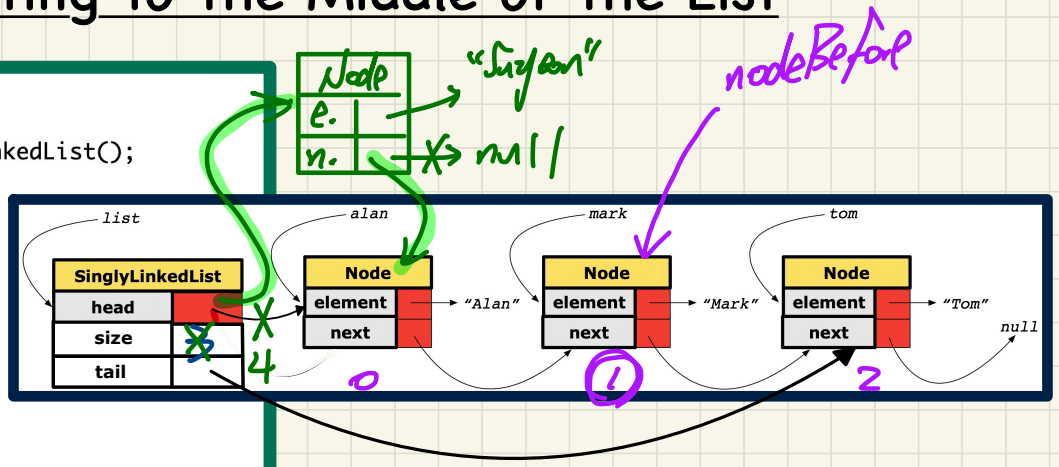
Idea of Inserting a Node at index i

Case: addAt(i, e), where $0 < i \leq size$



SLL Operation: Inserting to the Middle of the List

```java
@Test
public void testSLL_addAt() {
    SinglyLinkedList list = new SinglyLinkedList();
    assertTrue(list.getSize() == 0);
    assertTrue(list.getFirst() == null);

    list.addFirst("Tom");
    list.addFirst("Mark");
    list.addFirst("Alan");
    assertTrue(list.getSize() == 3);

    list.addAt(0, "Suyeon");
    list.addAt(2, "Yuna");
    assertTrue(list.getSize() == 5);
    list.addAt(list.getSize(), "Heeyeon");
    assertTrue(list.getSize() == 6);
    assertEquals("Suyeon", list.getNodeAt(0).getElement());
    assertEquals("Alan", list.getNodeAt(1).getElement());
    assertEquals("Yuna", list.getNodeAt(2).getElement());
    assertEquals("Mark", list.getNodeAt(3).getElement());
    assertEquals("Tom", list.getNodeAt(4).getElement());
    assertEquals("Heeyeon", list.getNodeAt(5).getElement());
}
```



Q. Does **tail** or **size** need to be updated?

```java
1  void addAt (int i, String e) {
2    if (i < 0 || i > size) {
3      throw new IllegalArgumentException("Invalid Index.");
4    }
5    else {
6      if (i == 0) {
7        addFirst(e);
8      }
9      else {
10       Node nodeBefore = getNodeAt(i - 1);
11       Node newNode = new Node(e, nodeBefore.getNext());
12       nodeBefore.setNext(newNode);
13       size ++;
14     }
15   }
16 }
```

O(n)

Lecture 12 - February 13

Arrays and Linked Lists

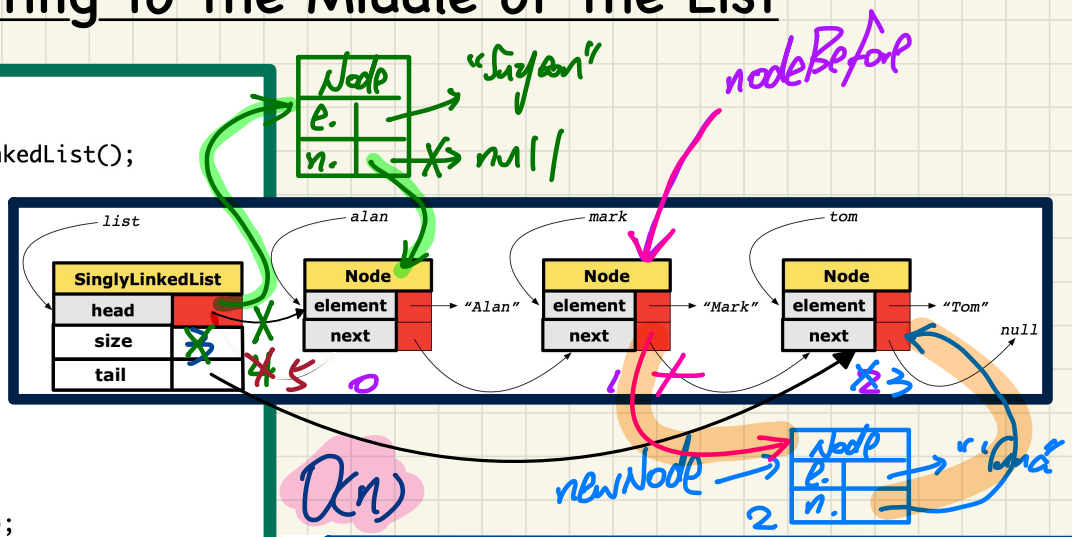
DLL: Introduction
DLL in Java: Node vs. Doubly-Linked Lists
DLL in Java: addBetween, remove

Announcements/Reminders

- **ProgTest1** guide & example questions released
- **In-person** Office Hours during RW to be announced
- ***splitArrayHarder***: solution and tutorial video released
- **Assignment 2** (on **SLL**) released
  + **Required** studies: Generics in Java (Slides 33 – 36)
  + **Recommended** studies: extra SLL problems
- **Assignment 1** solution released
- Lecture notes template, TA Contact

SLL Operation: Inserting to the Middle of the List

```java
@Test
public void testSLL_addAt() {
    SinglyLinkedList list = new SinglyLinkedList();
    assertTrue(list.getSize() == 0);
    assertTrue(list.getFirst() == null);

    list.addFirst("Tom");
    list.addFirst("Mark");
    list.addFirst("Alan");
    assertTrue(list.getSize() == 3);

    list.addAt(0, "Suyeon");
    list.addAt(2, "Yuna");
    assertTrue(list.getSize() == 5);
    list.addAt(list.getSize(), "Heeyeon");
    assertTrue(list.getSize() == 6);
    assertEquals("Suyeon", list.getNodeAt(0).getElement());
    assertEquals("Alan", list.getNodeAt(1).getElement());
    assertEquals("Yuna", list.getNodeAt(2).getElement());
    assertEquals("Mark", list.getNodeAt(3).getElement());
    assertEquals("Tom", list.getNodeAt(4).getElement());
    assertEquals("Heeyeon", list.getNodeAt(5).getElement());
}
```

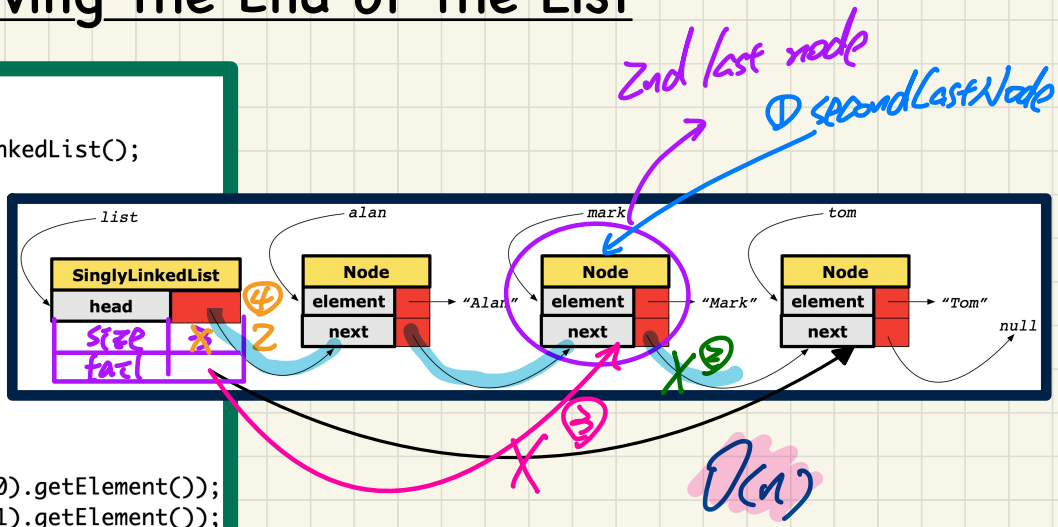



```java
1  void addAt (int i, String e) {
2    if (i < 0 || i > size) {
3      throw new IllegalArgumentException("Invalid Index.");
4    }
5    else {
6      if (i == 0) {
7        addFirst(e);
8      }
9      else {
10       Node nodeBefore = getNodeAt(i - 1);
11       Node newNode = new Node(e, nodeBefore.getNext());
12       nodeBefore.setNext(newNode);
13       size ++;
14     }
15   }
16 }
```

Q. Does **tail** or **size** need to be updated?

SLL Operation: Removing the End of the List

```java
@Test
public void testSLL_removeLast() {
    SinglyLinkedList list = new SinglyLinkedList();
    assertTrue(list.getSize() == 0);
    assertTrue(list.getFirst() == null);

    list.addFirst("Tom");
    list.addFirst("Mark");
    list.addFirst("Alan");
    assertTrue(list.getSize() == 3);

    list.removeLast();
    assertTrue(list.getSize() == 2);
    assertEquals("Alan", list.getNodeAt(0).getElement());
    assertEquals("Mark", list.getNodeAt(1).getElement());

    list.removeLast();
    assertTrue(list.getSize() == 1);
    assertEquals("Alan", list.getNodeAt(0).getElement());

    list.removeLast();
    assertTrue(list.getSize() == 0);
    assertNull(list.getFirst());
}
```



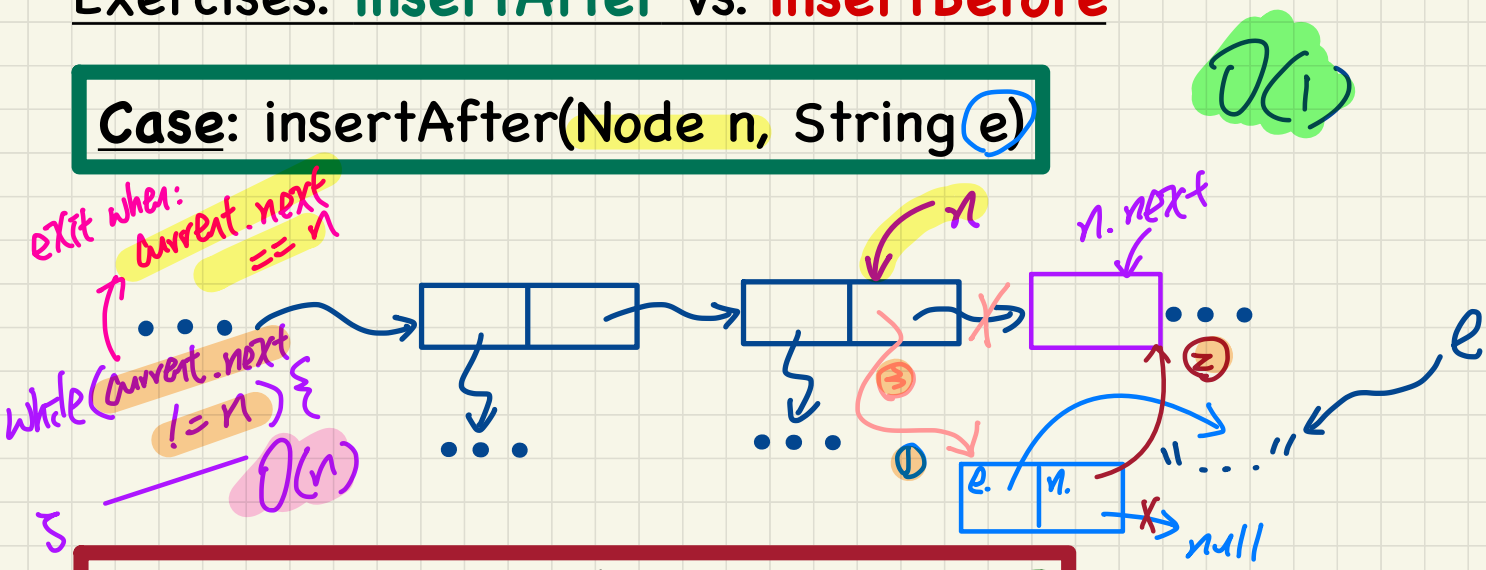

```java
1  void removeLast () {
2    if (size == 0) {
3      throw new IllegalArgumentException("Empty List.");
4    }
5    else if (size == 1) {
6      removeFirst();
7    }
8    else {
9      Node secondLastNode = getNodeAt(size - 2);
10     secondLastNode.setNext(null);
11     tail = secondLastNode;
12     size --;
13   }
14 }
```

O(n)

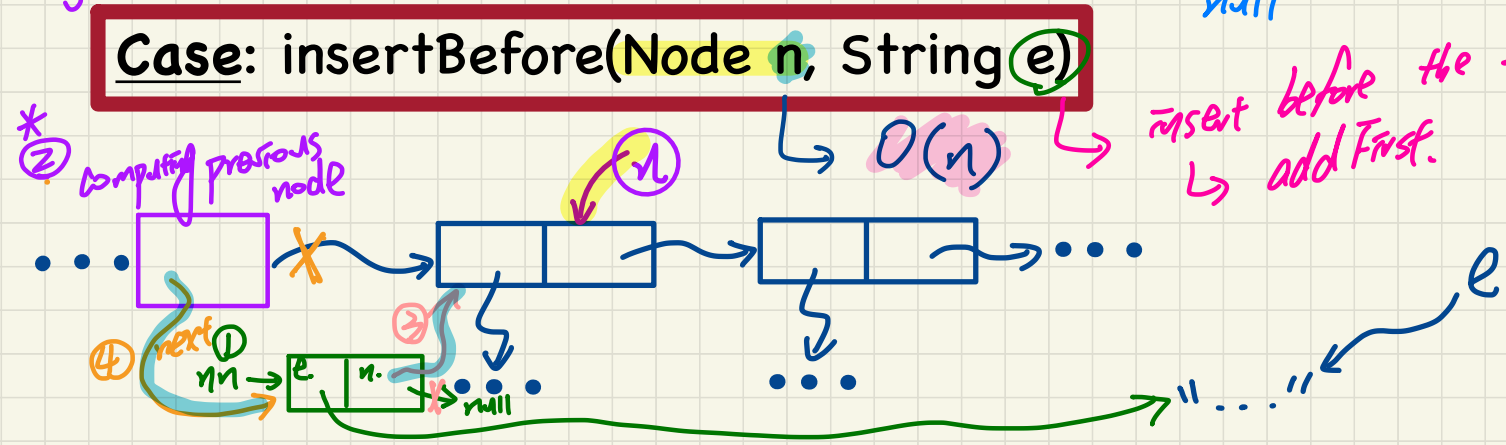
Q. Does tail or size need to be updated?

Exercises: insertAfter vs. insertBefore

Case: insertAfter(Node n, String e)



Case: insertBefore(Node n, String e)

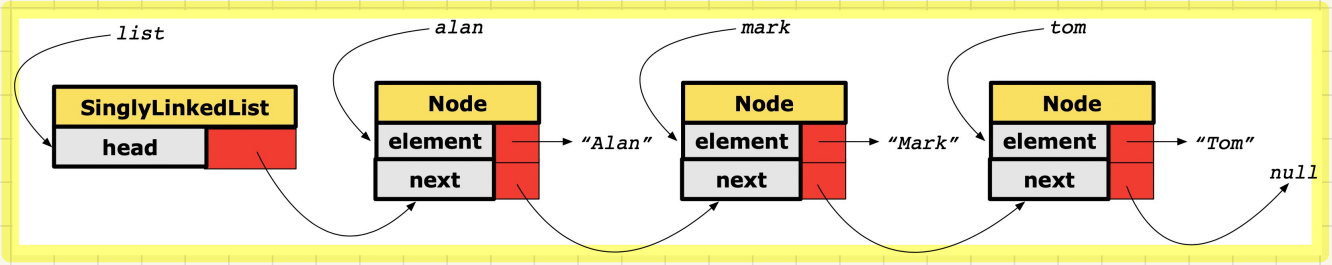


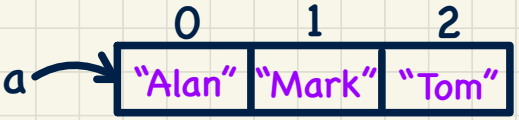
Running Time: Arrays vs. Singly-Linked Lists

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

array: a[a.length-1] = null

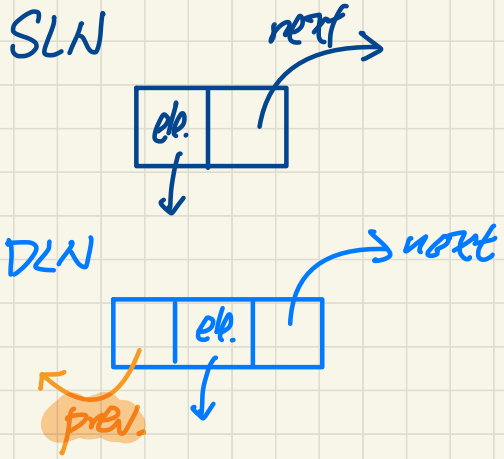
no need to traverse the list





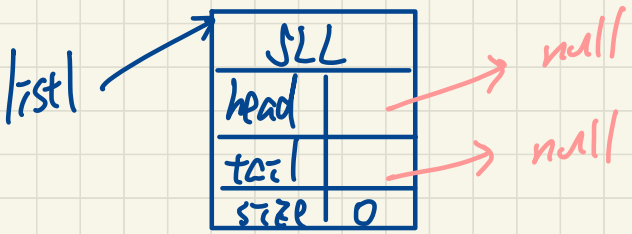
Doubly-Linked Lists (DLL): Visual Introduction

- A chain of bi-directionally connected nodes
- Each node contains:
  - + reference to a data object
  - + reference to the next node
  - + reference to the previous node
- A DLL is also a SLL:
  - + many methods implemented the same way
  - + some method implemented more efficiently
- Each DLL stores dedicated Header & Trailer Nodes (no head refeference and no tail reference)
- The chain may grow or shrink dynamically.
- Accessing a node in a DLL (via next or prev):
  - + Relative positioning: O(n)

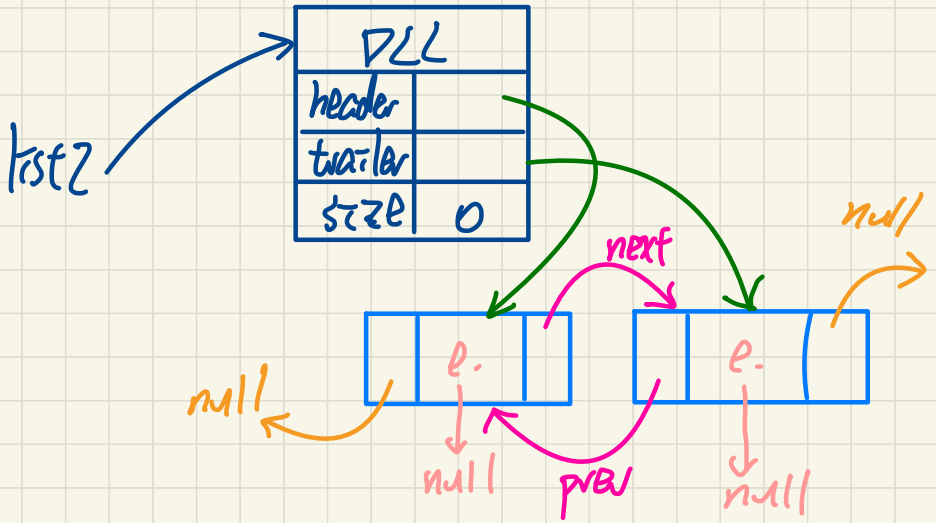


Empty Lists: SLLs vs. DLLs

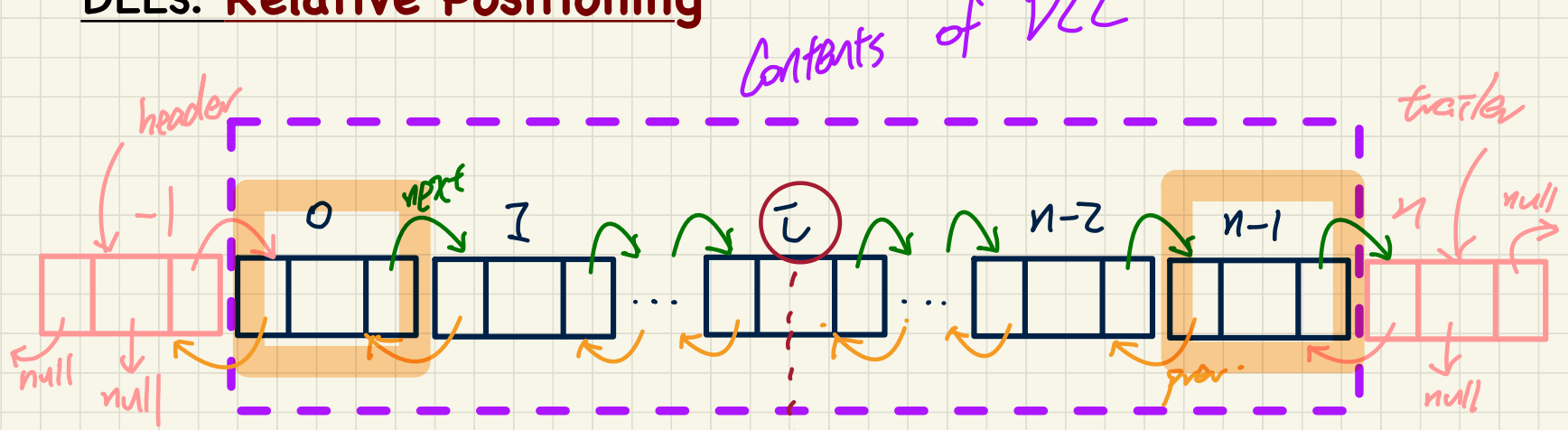
Empty SLL



Empty DLL



DLLs: Relative Positioning



O(1)
header.next : first node

O(1)
trailer.prev : last node

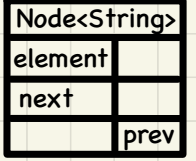
Worst case:
Around the middle
$O(\frac{n}{2}) = O(n)$

Why DLL (list) / DLN (node)?

1. performance (e.g. removeLast)
   - ↳ prev ref. for DLN

2. Code structure
   - don't need to handle edge cases explicitly.
   - ↳ header, trailer for DLL

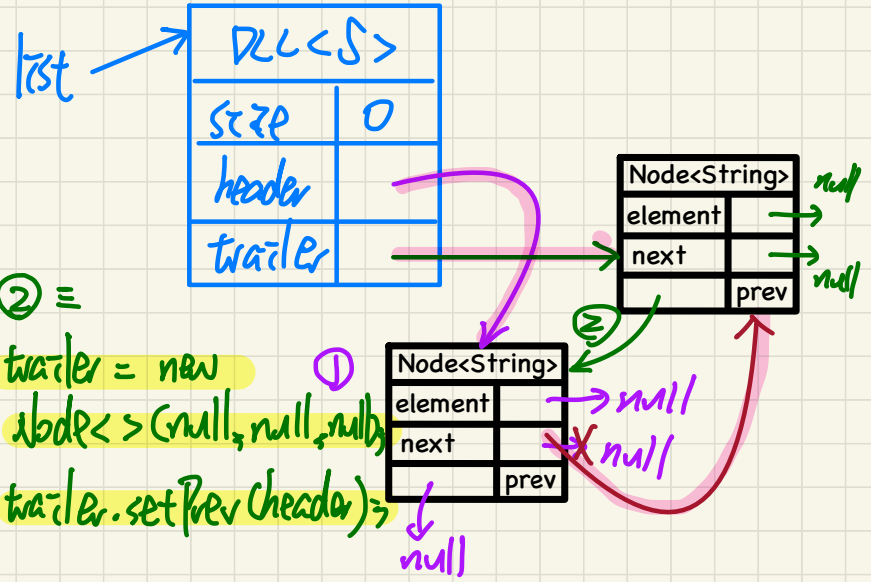
Generic DLL in Java: DoublyLinkedList vs. Node




```java
public class DoublyLinkedList<E> {
   private int size = 0;
   public void addFirst(E e) { ... }
   public void removeLast() { ... }
   public void addAt(int i, E e) { ... }
   private Node<E> header;
   private Node<E> trailer;
   public DoublyLinkedList() {
   ① header = new Node<>(null, null, null);
   ② trailer = new Node<>(null, header, null);
   ③ header.setNext(trailer);
   }
}
```

```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;
   }
}
```

```java
@Test
public void test_String_DLL_Empty_List() {
    DoublyLinkedList<String> list = new DoublyLinkedList<>();
    assertTrue(list.getSize() == 0);
    assertTrue(list.getFirst() == null);
    assertTrue(list.getLast() == null);
}
```



Generic DLL in Java: Inserting between Nodes

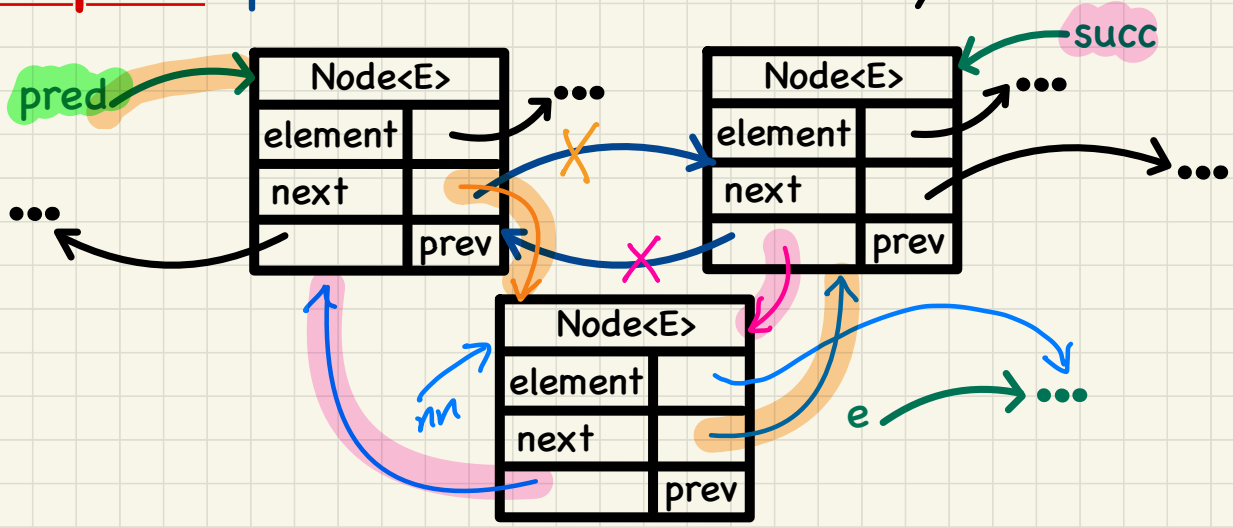
assumptions: pred.next == succ && succ.prev == pred

Node<E>	
element	
next	
	prev

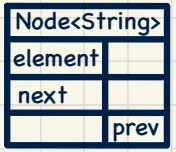
```
1  void addBetween(E e, Node<E> pred, Node<E> succ) {
2     Node<E> newNode = new Node<>(e, pred, succ);
3     pred.setNext(newNode);
4     succ.setPrev(newNode);
5     size ++;
6  }
```

ele. prev. next

Assumption: pred and succ are directly connected.



Generic DLL in Java: Inserting to the Front/End




```java
@Test
public void test_String_DLL_Insert_Front_End() {
    DoublyLinkedList<String> list = new DoublyLinkedList<>();
    list.addFirst("Mark");
    list.addFirst("Alan");

    assertTrue(list.getSize() == 2);
    assertEquals("Alan", list.getFirst().getElement());
    assertEquals("Mark", list.getFirst().getNext().getElement());

    list = new DoublyLinkedList<>();
    list.addLast("Mark");
    list.addLast("Alan");

    assertTrue(list.getSize() == 2);
    assertEquals("Alan", list.getLast().getElement());
    assertEquals("Mark", list.getLast().getPrev().getElement());
}
```

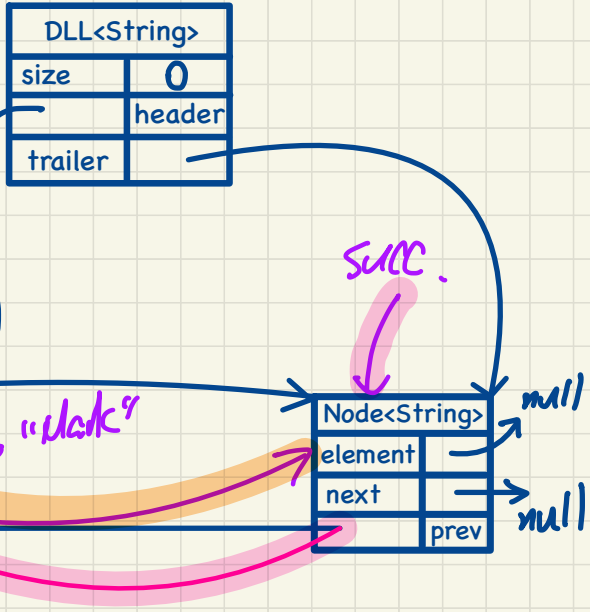
Execute

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

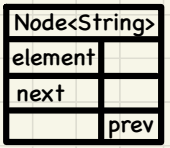
pred. succ.

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

pred. succ.



Generic DLL in Java: Inserting to the Middle




```java
@Test
public void test_String_DLL_addAt() {
    DoublyLinkedList<String> list = new DoublyLinkedList<>();
    list.addAt(0, "Alan");
    list.addAt(1, "Tom");
    list.addAt(1, "Mark");
    assertTrue(list.getSize() == 3);
    assertEquals("Alan", list.getFirst().getElement());
    assertEquals("Mark", list.getFirst().getNext().getElement());
    assertEquals("Tom", list.getFirst().getNext().getNext().getElement());
}
```

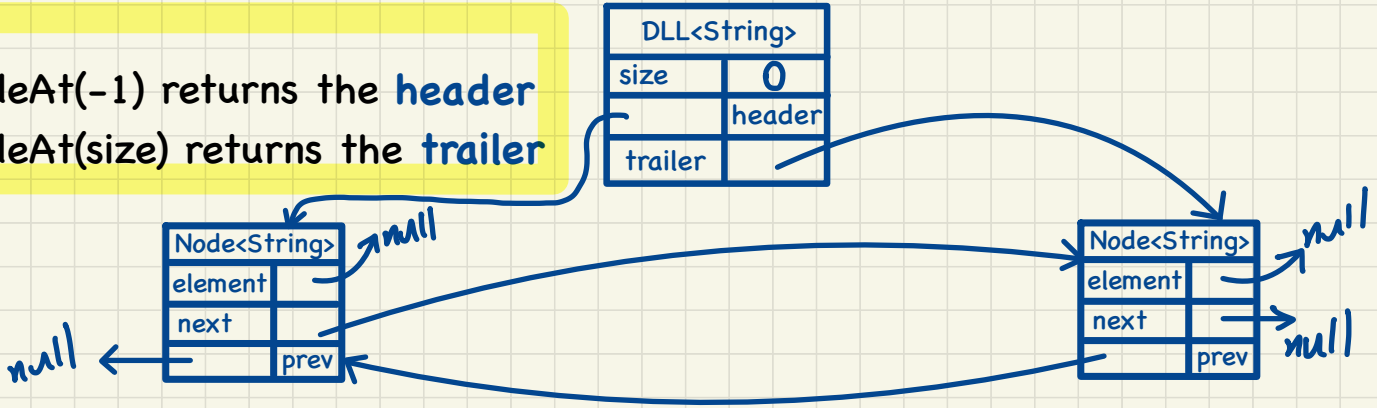
exercise.

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

O(n)

Notes.

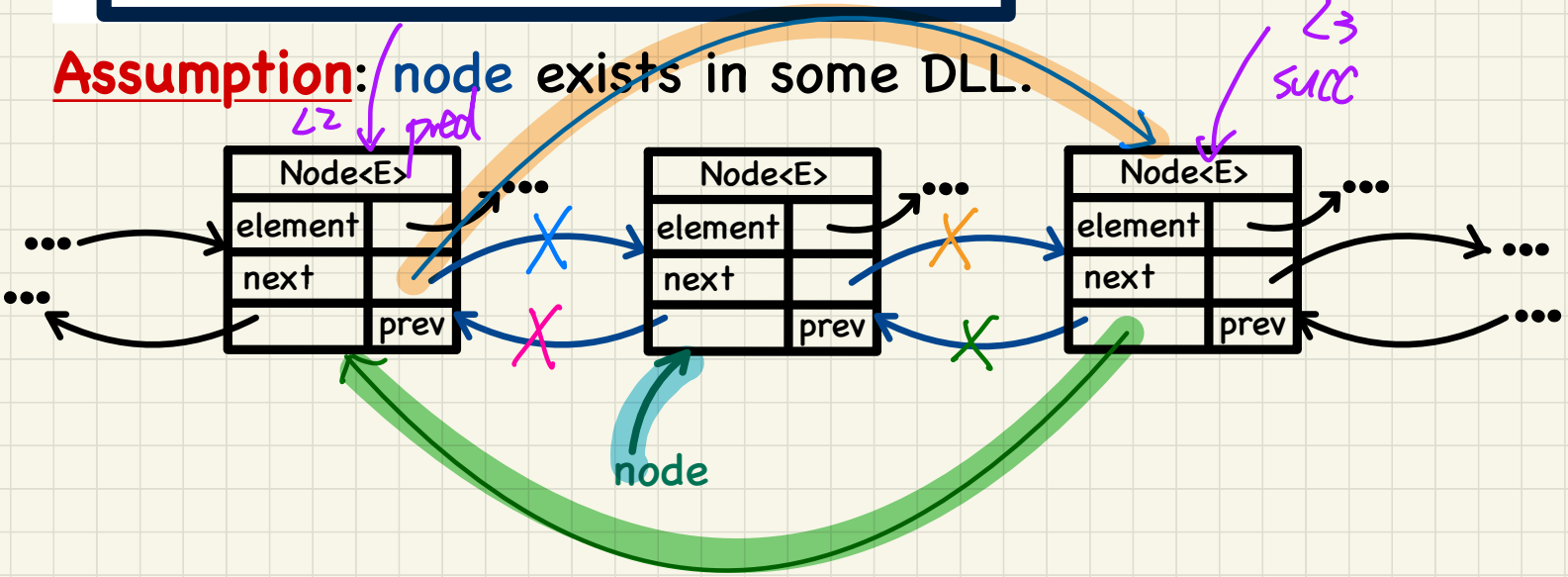
+ getNodeAt(-1) returns the **header**
+ getNodeAt(size) returns the **trailer**



Generic DLL in Java: Removing a Node

```
1  void remove (Node<E> node) {
2    Node<E> pred = node.getPrev();
3    Node<E> succ = node.getNext();
4    pred.setNext(succ);
5    succ.setPrev(pred);
6    node.setNext(null);
7    node.setPrev(null);
8    size --;
9  }
```

Assumption: node exists in some DLL.



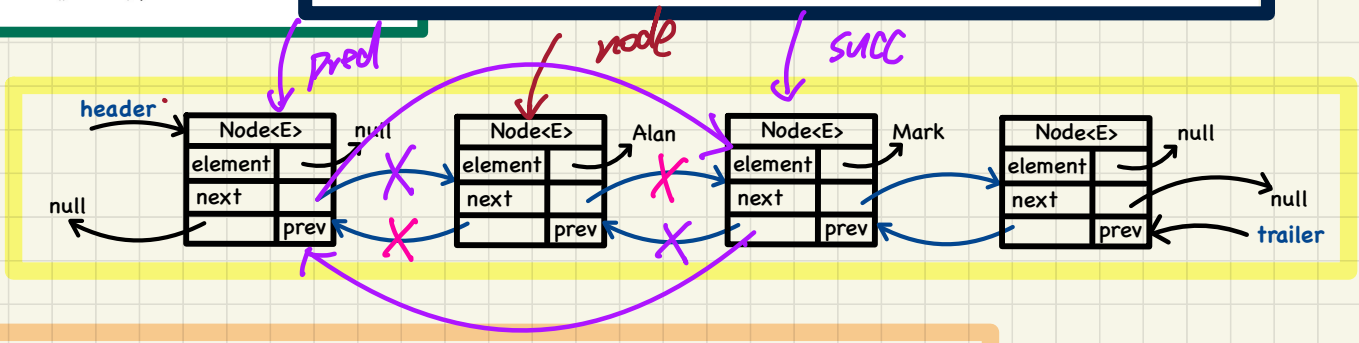
Generic DLL in Java: Removing from the Front/End

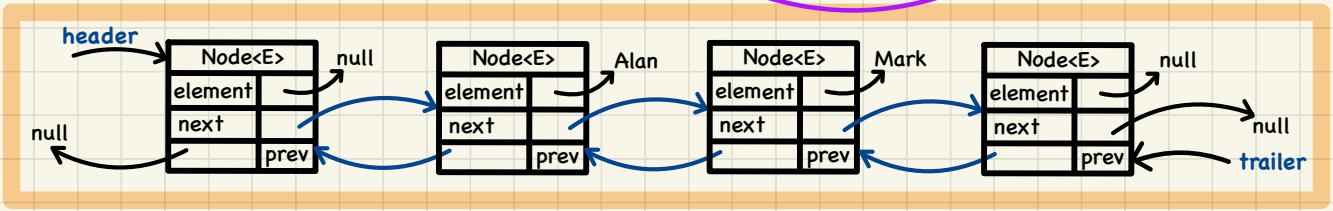
```java
@Test
public void test_String_DLL_Remove_Front_End() {
    DoublyLinkedList<String> list = new DoublyLinkedList<>();
    list.addFirst("Mark");
    list.addFirst("Alan");
    list.removeFirst();
    list.removeFirst();
    assertTrue(list.getSize() == 0);

    list = new DoublyLinkedList<>();
    list.addFirst("Mark");
    list.addFirst("Alan");
    list.removeLast();
    list.removeLast();
    assertTrue(list.getSize() == 0);
}
```

EXERCISE

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

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



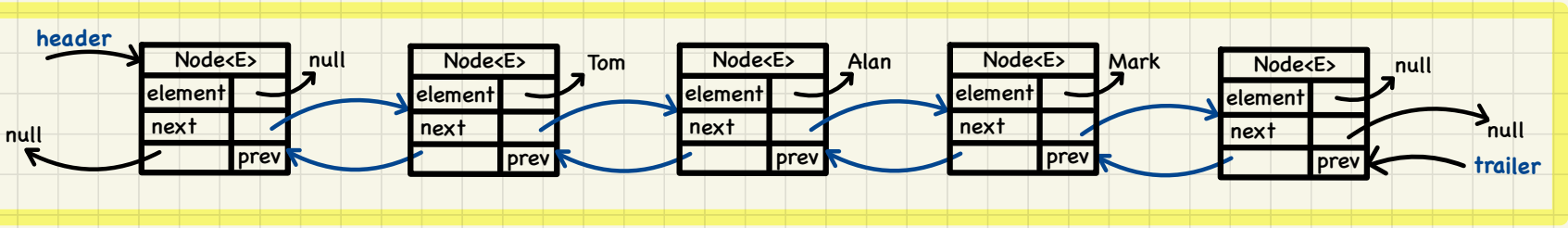


Generic DLL in Java: Removing from the Middle

```java
@Test
public void test_String_DLL_removeAt() {
    DoublyLinkedList<String> list = new DoublyLinkedList<>();
    list.addFirst("Mark");
    list.addFirst("Alan");
    list.addFirst("Tom");
    assertTrue(list.getSize() == 3);
    list.removeAt(1);
    assertTrue(list.getSize() == 2);
    list.removeAt(0);
    assertTrue(list.getSize() == 1);
    list.removeAt(0);
    assertTrue(list.getSize() == 0);
}
```

Examples

```java
removeAt (int i) {
  if (i < 0 || i >= size) {
    throw new IllegalArgumentExce
  else {
    Node<E> node = getNodeAt(i);
    remove (node);
  }
}
```



Lecture 13 - February 25

General Trees

Linear vs. Non-Linear Structures
General Trees: Terminology
Generic TreeNode in Java

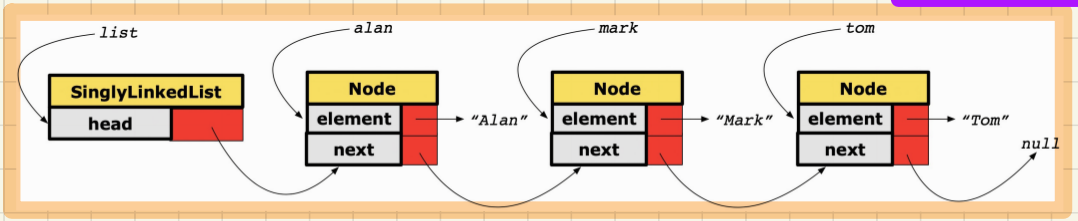
Announcements/Reminders

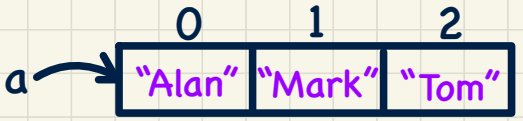
- Survey on **Makeup Lecture** for ProgTest1
- **Assignment 3** (on **linked** **Trees**) to be released
- **WrittenTest** guide to be released
- This week's office hour: 3pm, Wed

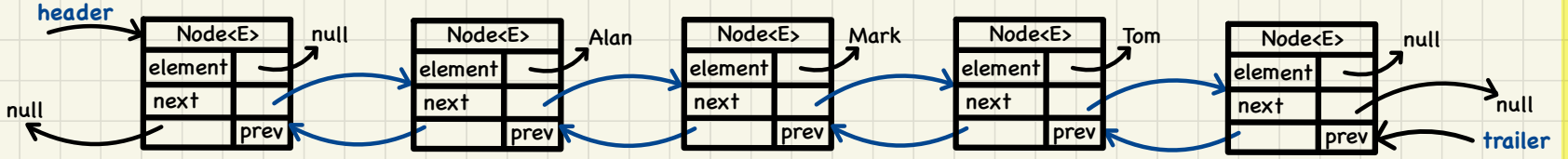
Running Time: Arrays vs. SLL vs. DLL

see earlier discussion

Operation \ Data Structure		Array	Singly-Linked List	Doubly-Linked List
size				O(1)
first/last element				O(n) → e.g. [illegible]
element at index i		O(1)	O(n)	
remove last element				
add/remove first element, add last element				O(1)
add/remove i^{th} element	given reference to $(i-1)^{th}$ element	O(n)	O(1)	
	not given			O(n)







Trees

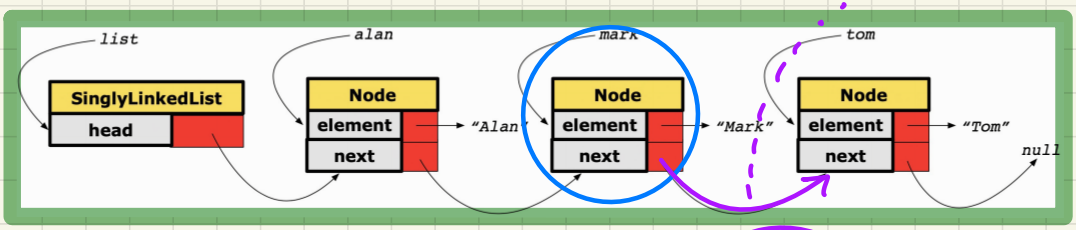
1. General Trees
2. Binary Trees (BTs)
3. Binary Search Trees (BSTs)
4. Balanced BSTs
5. Priority Queues
6. Heap
7. Heap Sort

a: 1, 2

b: 2, 3

c: 3, 4, 5, 6, 7

Linear vs. Non-Linear Structures

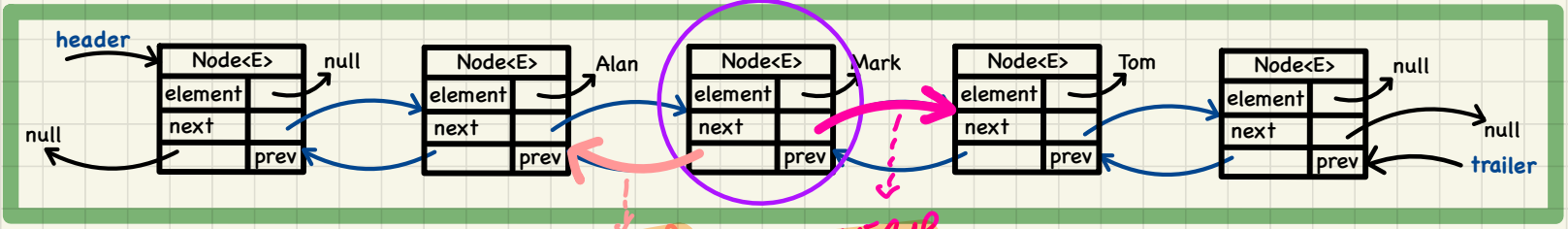


using next to access the unique successor

index of unique pred. $i-1$

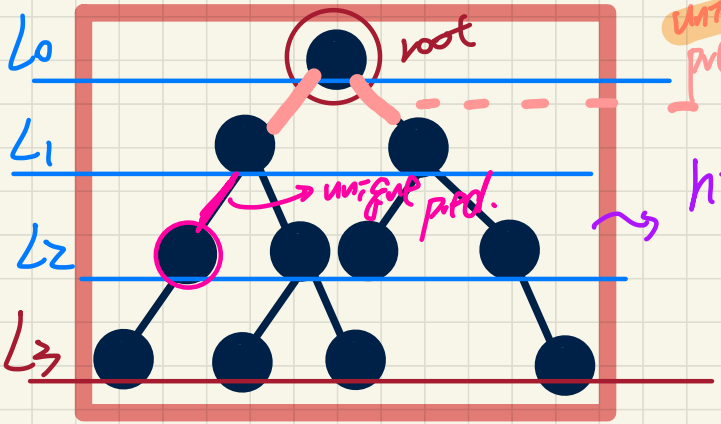
i

index of unique succ. $i+1$



unique predecessor

unique successor



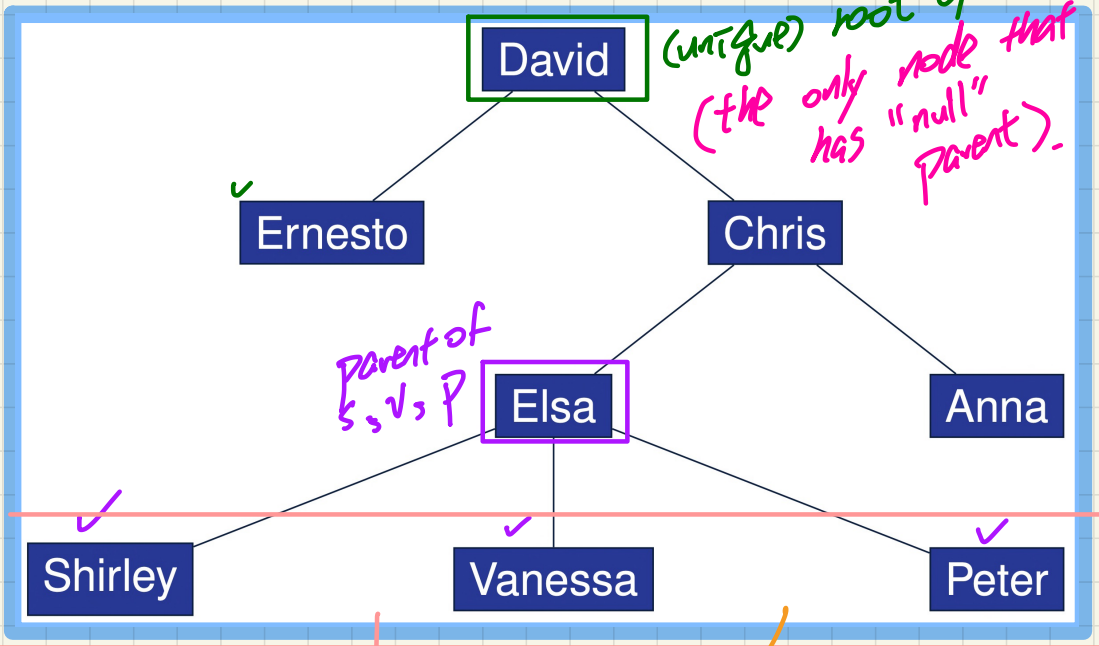
hierarchical structure (levels)

multiple "successors" of the root

↓

in programming they are stored in a tree DS (e.g. S2C)

General Trees: Terminology (1)



(unique) root of the tree (the only node that has "null" parent).

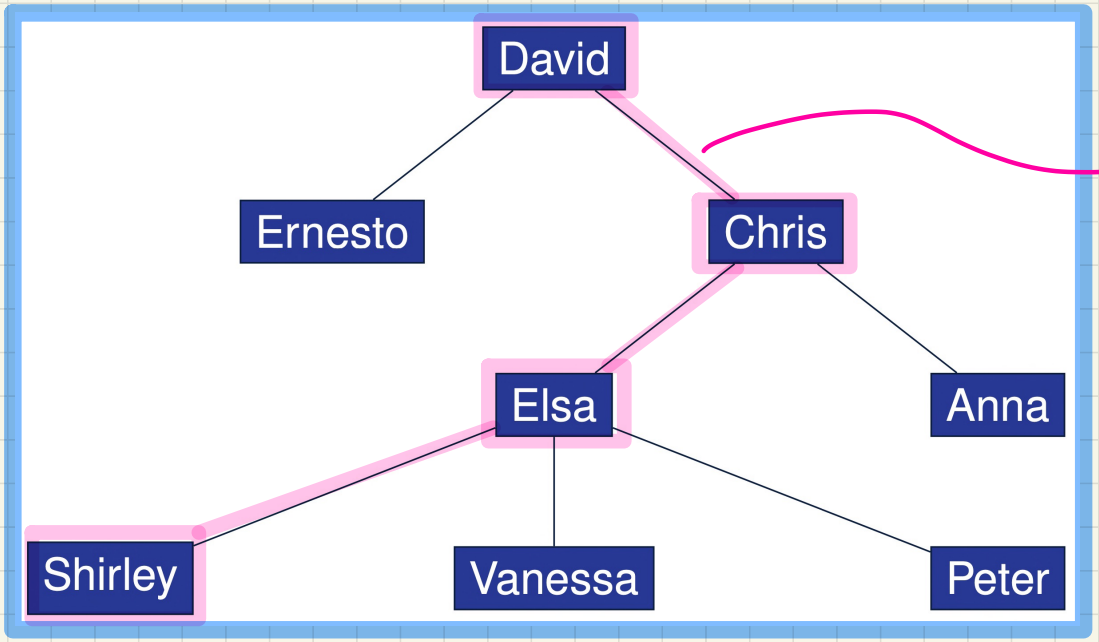
- root
- parent
- children
- ancestors
- descendants
- siblings

parent of S, V, P

children of Elsa

nodes without children

these 3 nodes are siblings. (at the same level).



unique ← parent is unique

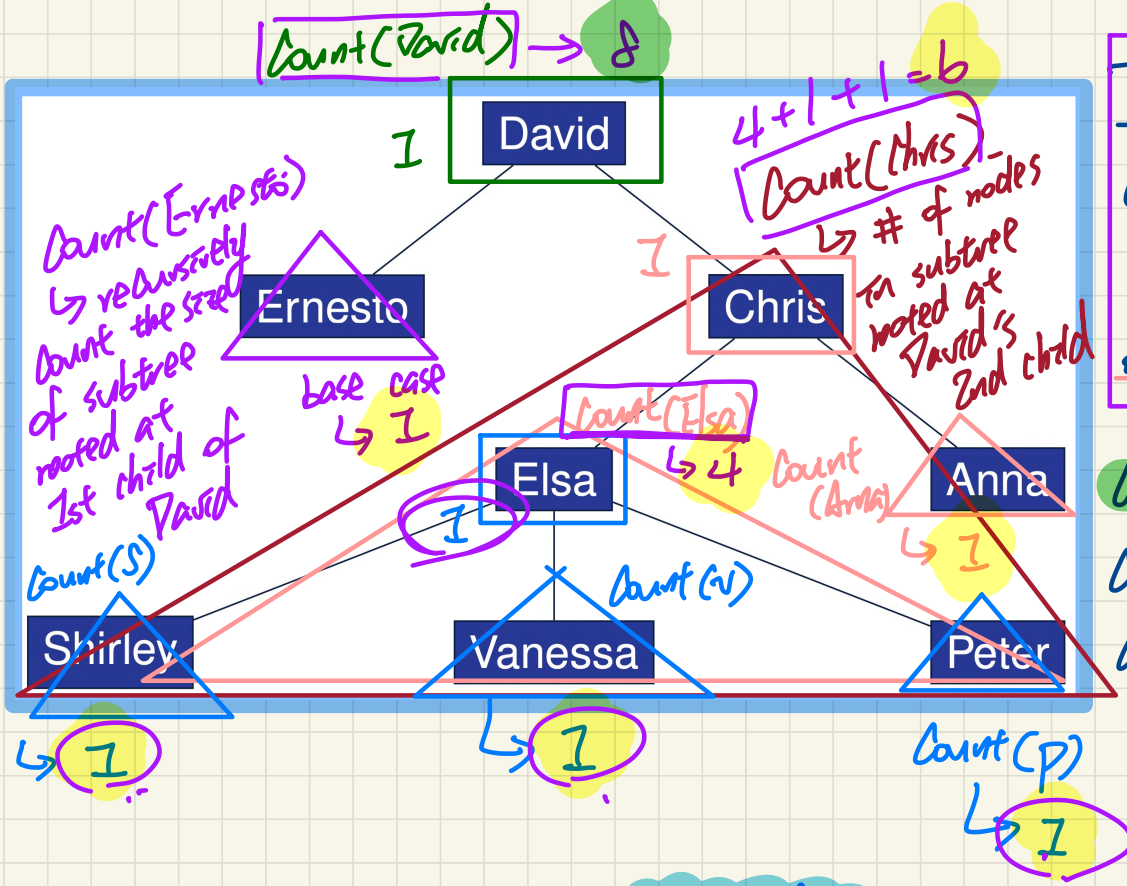
ancestor path of shirley

(shirley, Elsa, Chris, David)

node	Descendants
Vanessa	Vanessa
Elsa	Elsa, S, V, P
David	all nodes in the tree.

A node is both its ancestor and descendant.

General Trees: Terminology (2)



Problem:

Given a node n, return the size of subtree rooted at n.

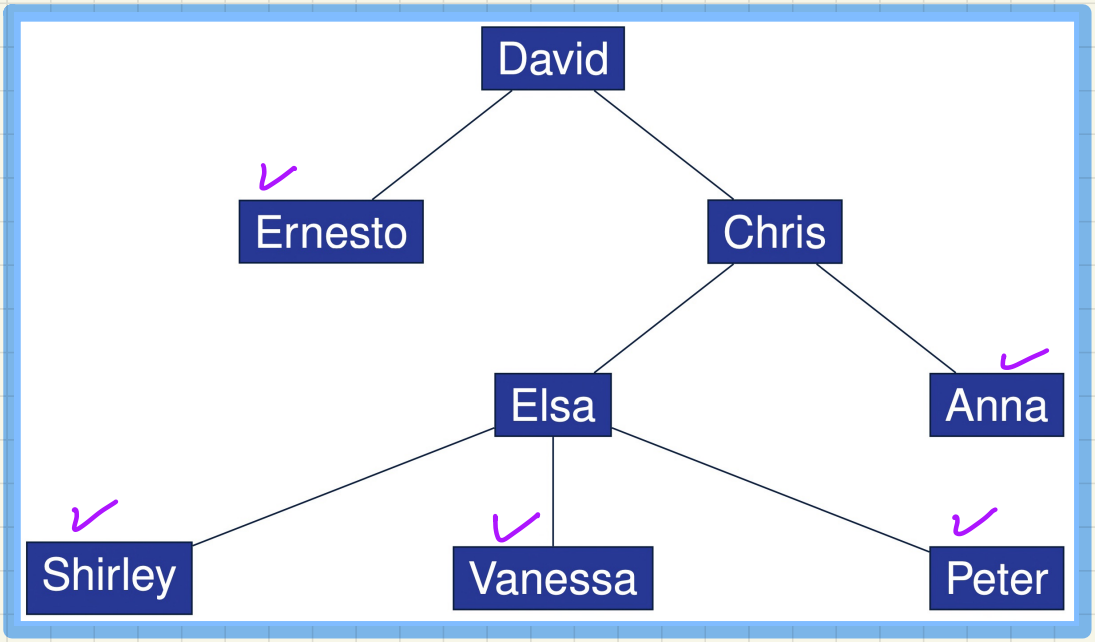
(# descendants)

count(David) = 8

count(Chris) = 6

count(Shirley) = 1

- subtrees are the subproblems for you to make recursive calls on.

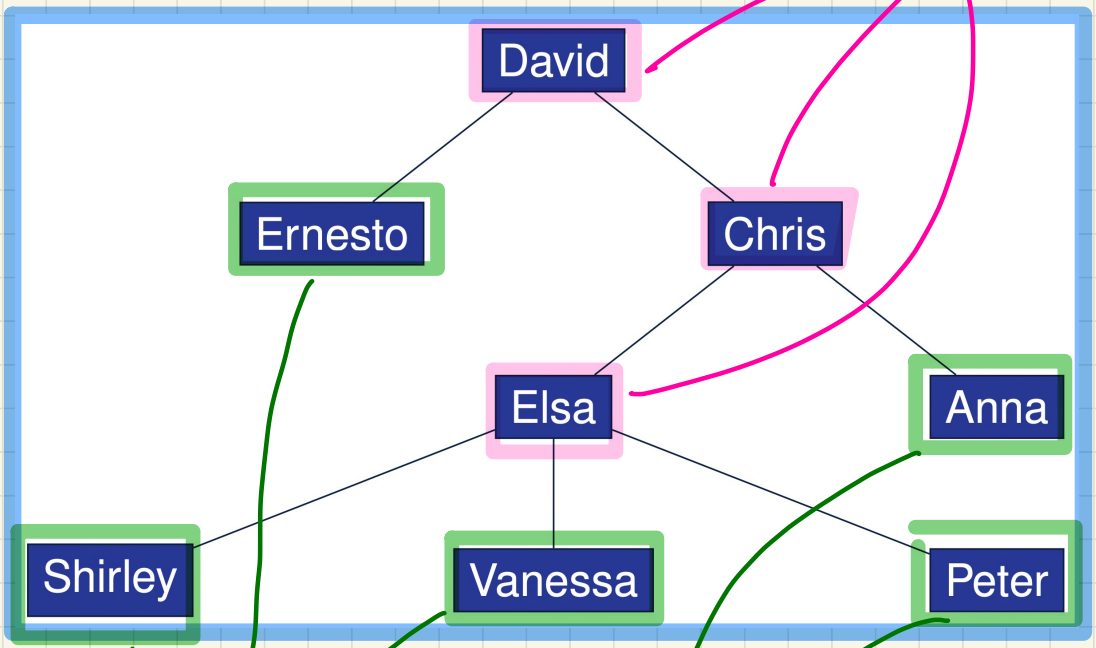


How many subtrees in total?

n = # nodes in the tree.

size	# subtrees
1	5
⋮	⋮
n	1 rooted at David.

General Trees: Terminology (3)

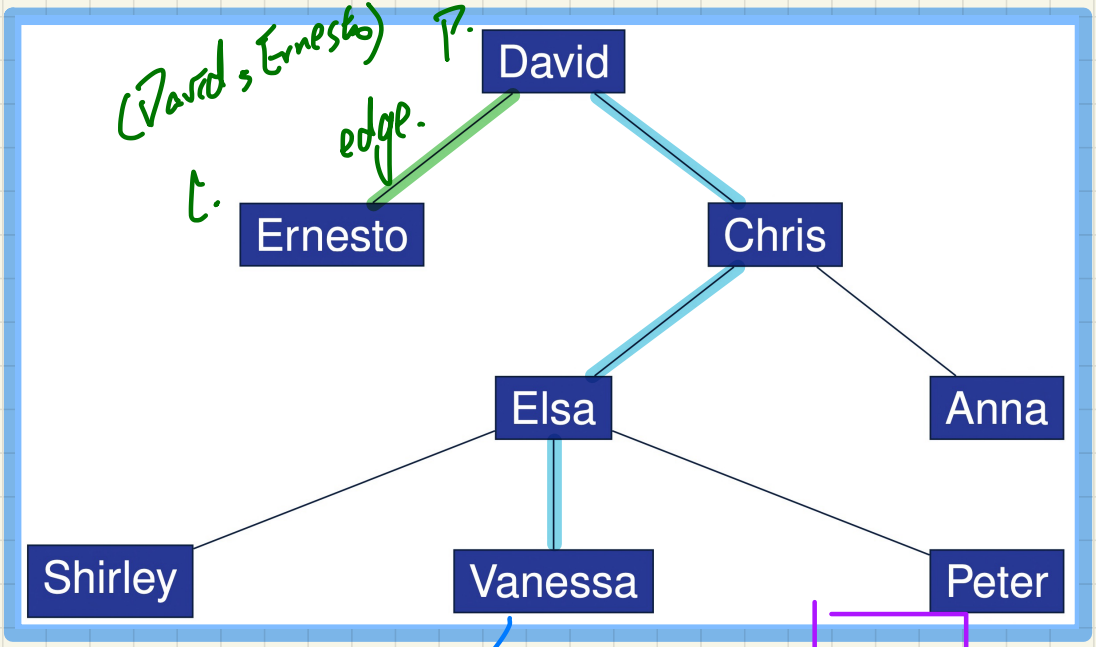


internal nodes → recursive cases

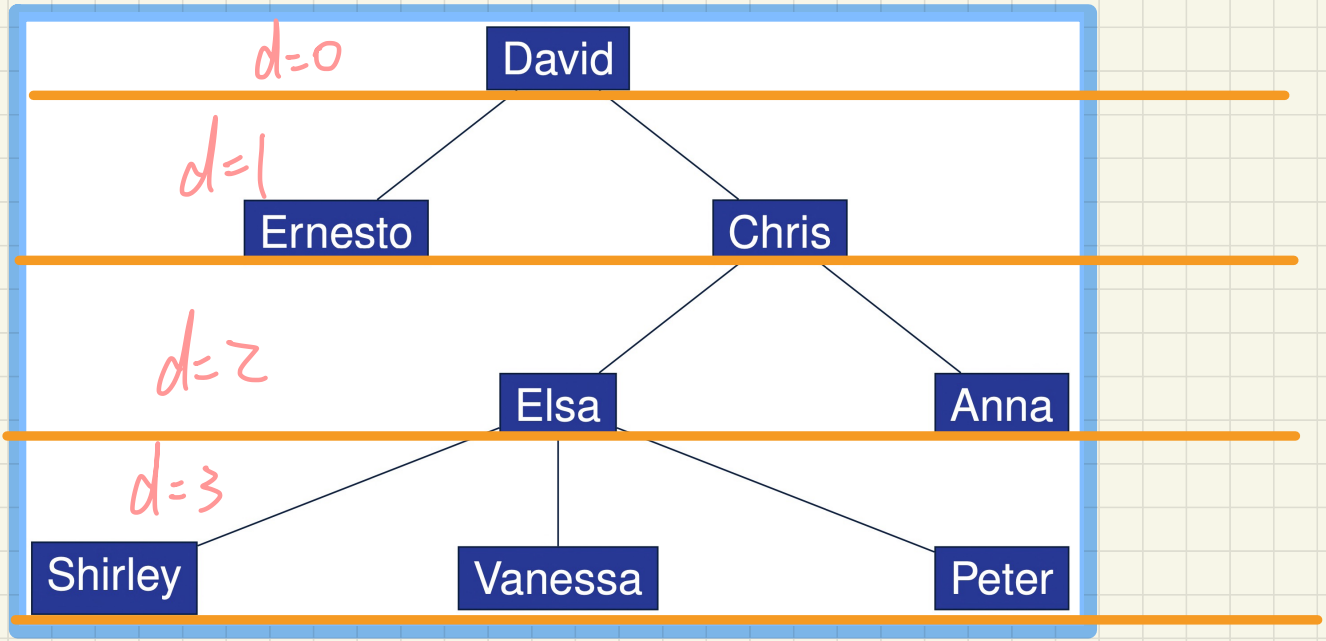
- external nodes
- internal nodes

external nodes (leaves) → base cases

General Trees: Terminology (4)

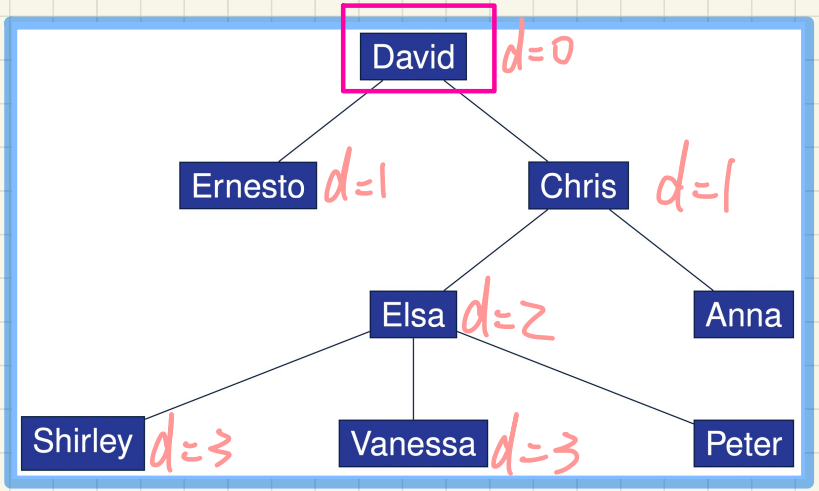


- edge
- path
- depth
- height



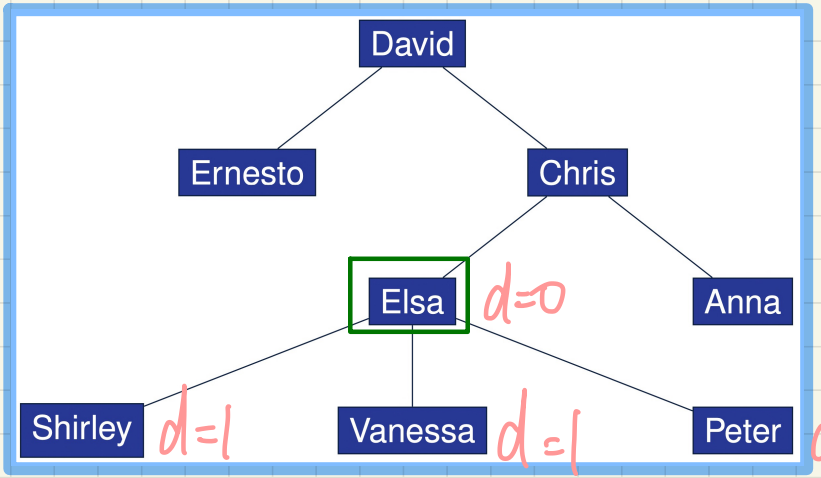
depth of a node n: # edges (parent links) from n to root.

height of a tree: max depth of descendants of its root.



Height of tree rooted at David:

(3)



Height of tree rooted at Elsa:

(1).

Lecture 14 - March 4

General Trees, Binary Trees

Initializing a Generic Array
Recursive Definitions of (Binary) Trees
Trees in Java: Construction, Depth

Announcements/Reminders

- ProgTest1 results to be released by Friday, Mar 14
- Makeup Lecture (on ADTs, Stacks) posted
- Assignment 3 (on linked Trees) released
- WrittenTest guide and example questions to be release
- Lecture notes template, Office Hours, TA Contact

General Trees: Recursive Definition

- root
- size

Case 0: Empty Tree

ϕ

root → null

size == 0

Case 1: A singleton tree

root → [: | children] → null

implementation detail

ArrayList<TreeNode> empty (.size() == 0)

Case 2: ≥ 2 nodes

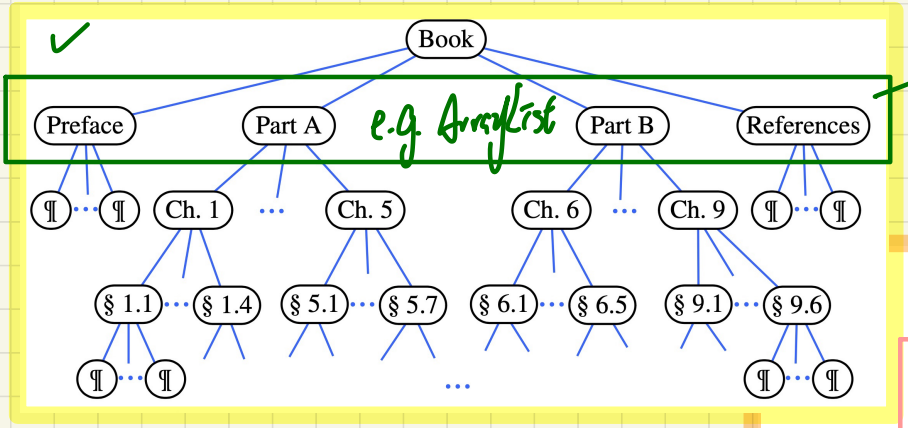
root → [: | c.] → [0 | 1 | 2 | ... | MAX-1]

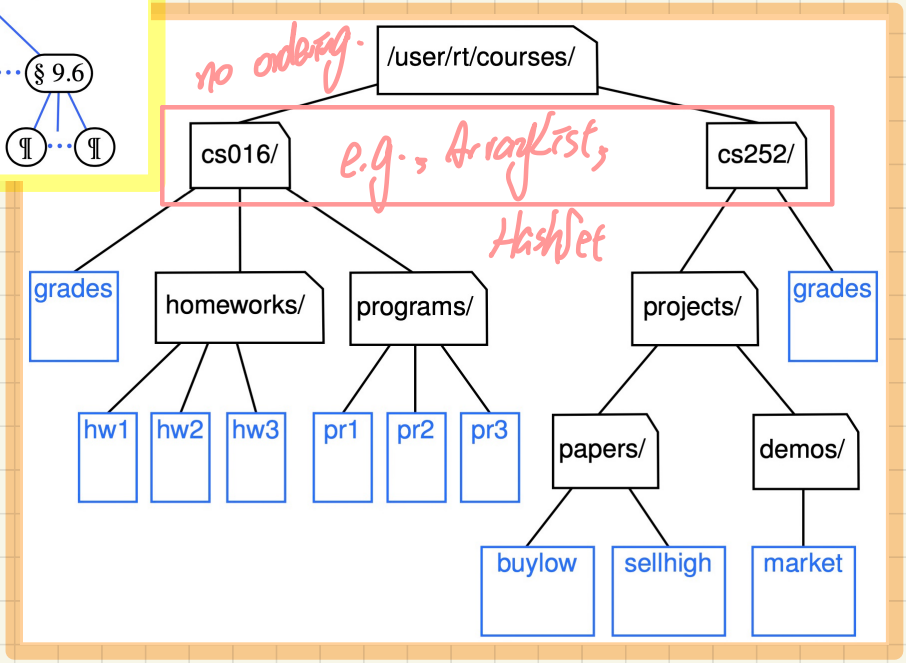
TreeNode[]

[c.] → subtree rooted at 1st child

[c.] → subtree rooted at 2nd child

General Trees: Ordered vs. Unordered Trees





red
↳ tree :/

Generic, General Tree Nodes

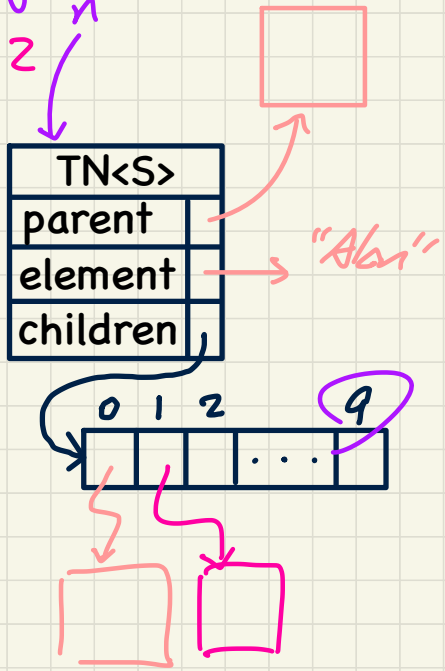
n.noc ≤ n.children.length

n.children.length == 10

n.noc == 0

x 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 TreeNode(E element) {
    this.element = element;
    this.parent = null;
    this.children = (TreeNode<E>[])
      Array.newInstance(this.getClass(), MAX_NUM_CHILDREN);
    this.noc = 0;
  }
  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) { ... }
}
```



Compare:

+ prev ref.
+ next ref.

in a DLN.

Instantiating Generic Structures in Java

```
class ArrayStack<E> {
  private E[] data;
  ...
  public ArrayStack<E>() {
    data = new E[10];
  }
}
```

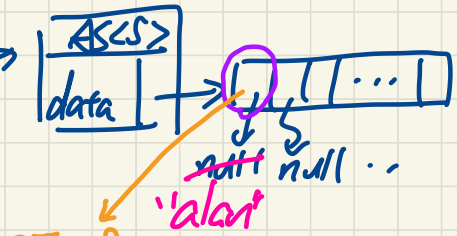
```
class TreeNode<E> {
  private TreeNode<E>[] children;
  ...
  public TreeNode<E>() {
    children = (TreeNode<E>[]) Array.newInstance(this.getClass(), 10);
  }
}
```

data = (E[]) new Object[10];

AS<String> s = new AS<>();

s.push(23); ✗

s.push("alan");



E is wrapped within the TreeNode class

RT: Object[]

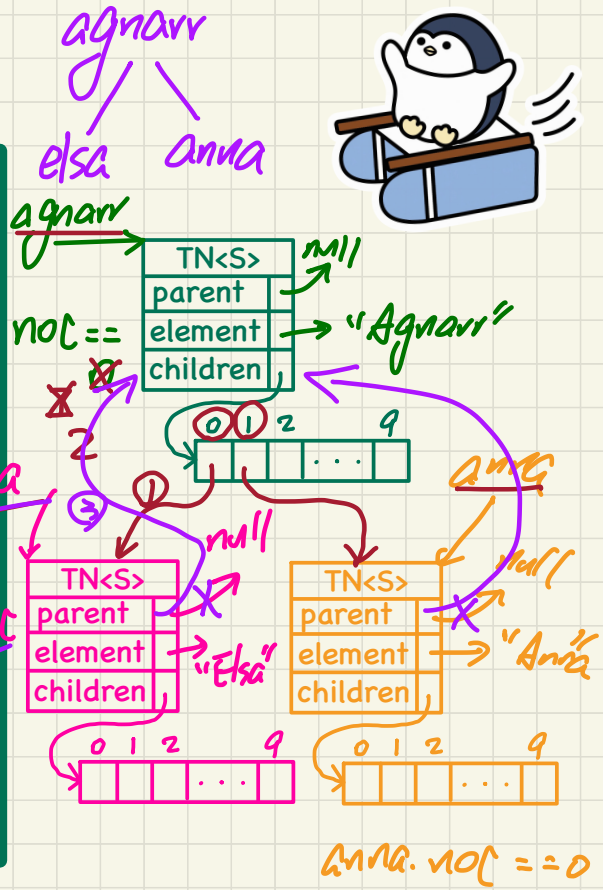
children = (TreeNode<E>[]) new Object[10];

ST: String

DT: String

Tracing: Constructing a Tree

```java
@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);
}
```

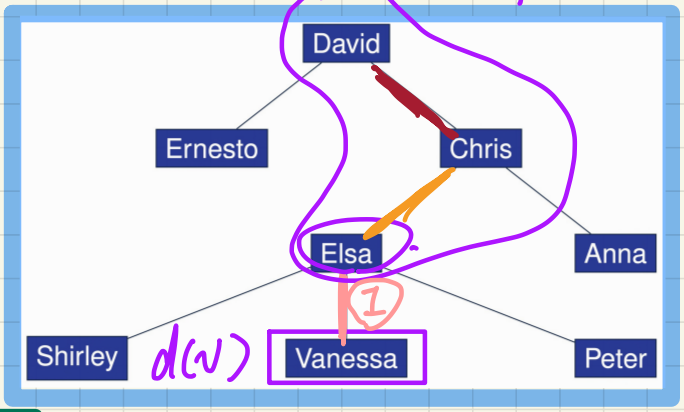


Tracing: Computing a Node's Depth

root of some subtree

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

depth of root is 0.




```
@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));
}
```

depth(vanessa)

↳ .getP != null

= 1 + depth(Elsa)

↳ .getP != null

= 1 + 1 + depth(Chris)

= 1 + 1 + 1 + depth(David)

0

= 3

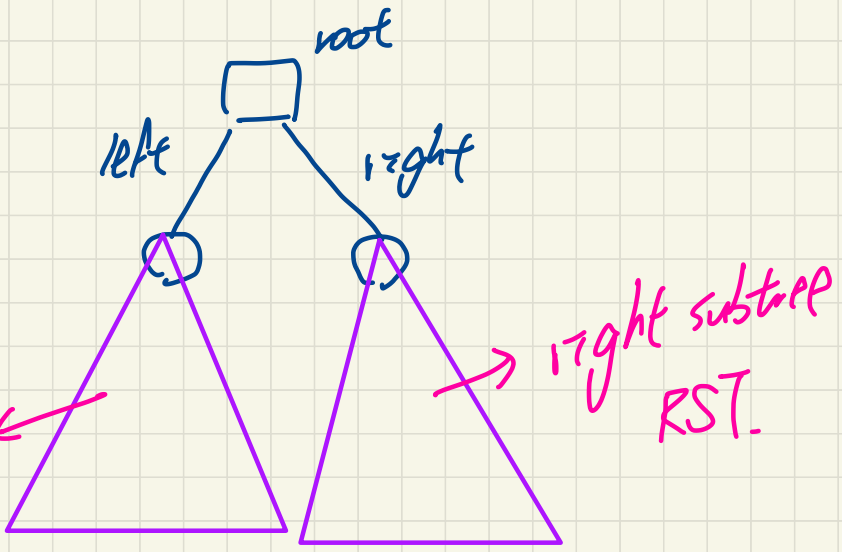
Binary Trees: Recursive Definition

- root
- size

Case 0: Empty Tree

Case 1: Singleton Tree

Case 2: ≥ 2 nodes



Deriving the Sum of a Geometric Sequence

Initial Term: I
Common Factor: r
Number of Terms: k

2^0 2^{10}

$1 + 2 + 4 + 8 + 16 + \cdots + 1024$

$*2$ $*2$

$k = 11$

$$S_k = \underbrace{I}_{I \cdot r^0} + I \cdot r + I \cdot r^2 + I \cdot r^3 + \cdots + I \cdot r^{k-1}$$

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

$$r \cdot S_k - S_k = (r-1) \cdot S_k = I \cdot r^k - I = I \cdot (r^k - 1)$$

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

For BT: $r = 2$, $I = 1$ (root)

$$S_k = \frac{1 \cdot (2^k - 1)}{2 - 1} = 2^k - 1$$

(k depth of "bottom" node)

Lecture 15 - March 6

Binary Trees

Binary Trees: Math Properties
Tree Traversals

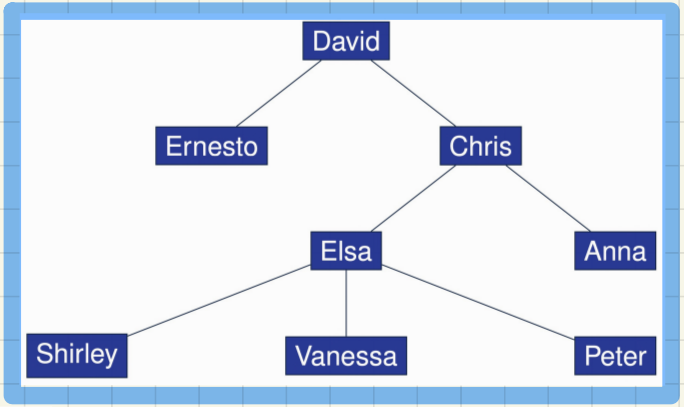
Announcements/Reminders

- Assignment 3 (on linked Trees) released
- WrittenTest
  + guide released
  + example questions to be release
- Makeup Lecture (on ADTs, Stacks) posted
- Lecture notes template, Office Hours, TA Contact

Tracing: Computing a Tree's Height

```
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;
  }
}
```

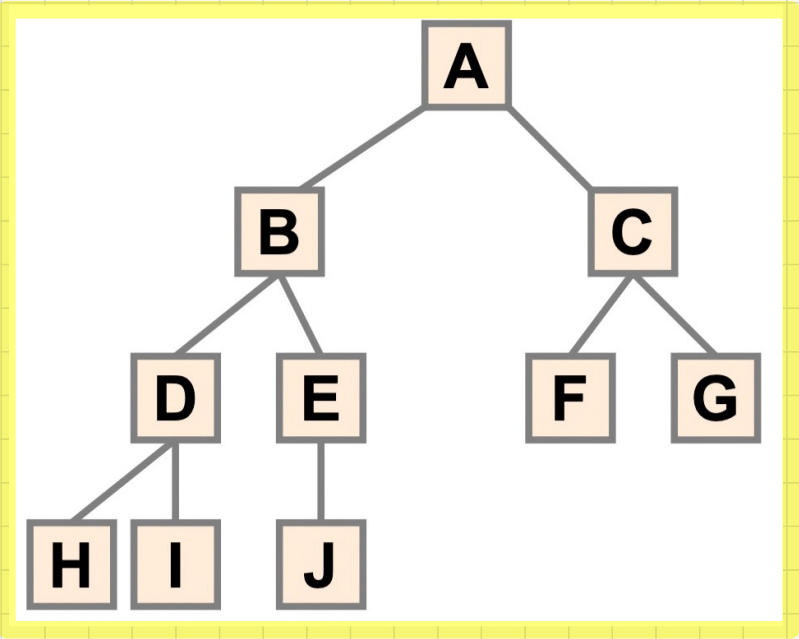
noC



height(chris)

```
@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));
}
```

BT Terminology: LST vs. RST



Strategy of Recursion on BT:

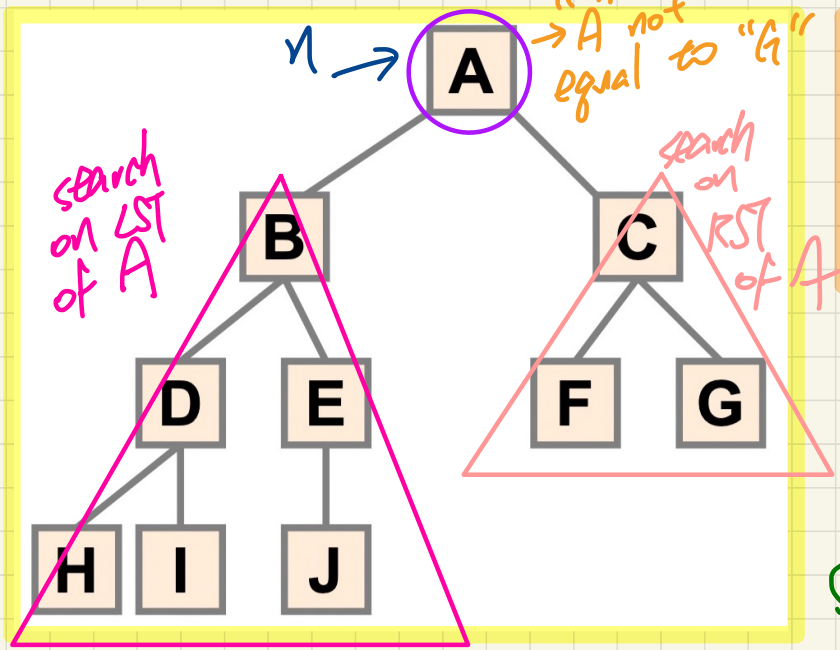
+ Do something on root
+ Recur on LST
+ Recur on RST

e.g.,

+ counting size

BT Terminology: LST vs. RST

search(A, "G") → T
search(A, "Z") → F



Strategy of Recursion on BT:

+ Do something on root
+ Recur on LST
+ Recur on RST

e.g.,

+ searching item

search(n, e) n: tree node, e: data ele.

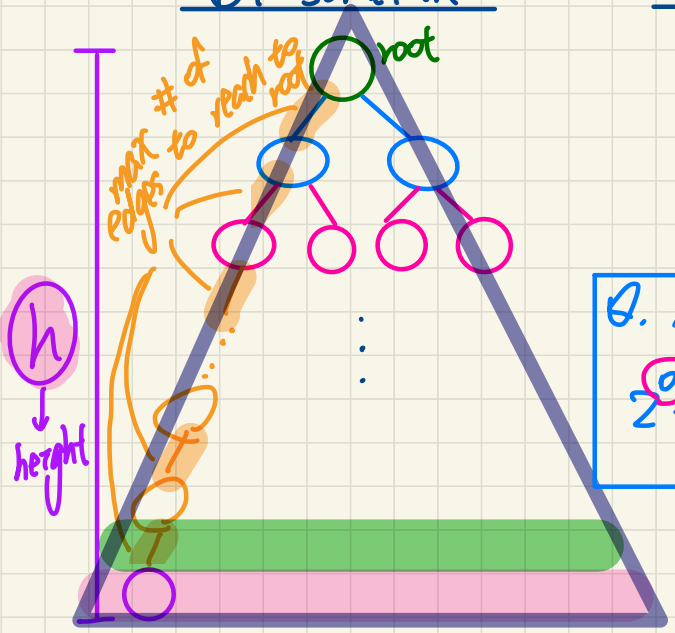
```
① if (n.element.equals(e)){
      return true;
  }
② if(n.left != null){
      return search(n.left, e);
  }
③ if(n.right != null){
      return search(n.right, e);
  }
④ return false; // if n is external, n.element ≠ e
```

BT Terminology: Depths, Levels, Max # of Nodes

height.

$S_k = \frac{I \cdot (r^k - 1)}{r - 1}$ BT: $\frac{1 \cdot (2^k - 1)}{2 - 1} = 2^k - 1$ *

BT structure



Level ??

d

0

1

2

Exercise
Max # of nodes from Level 0 to Level h-1?

Max # nodes at Level ??

$1 = 2^0$

$2 = 2^1$

$4 = 2^2$

Q. Max # of nodes in a BT with height h?

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

$h - 1$

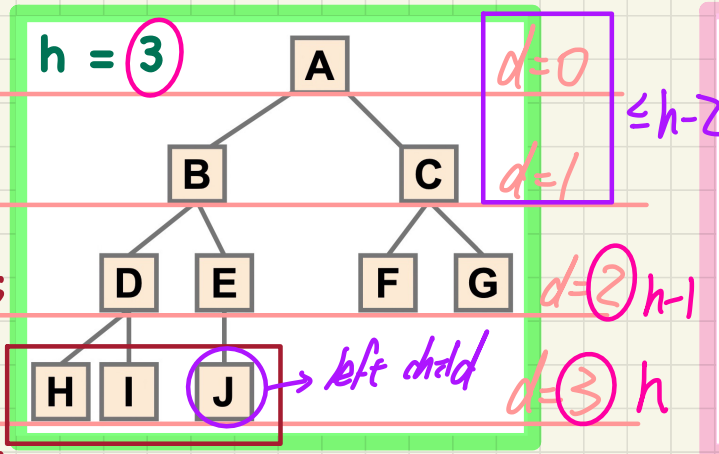
h

2^{h-1}

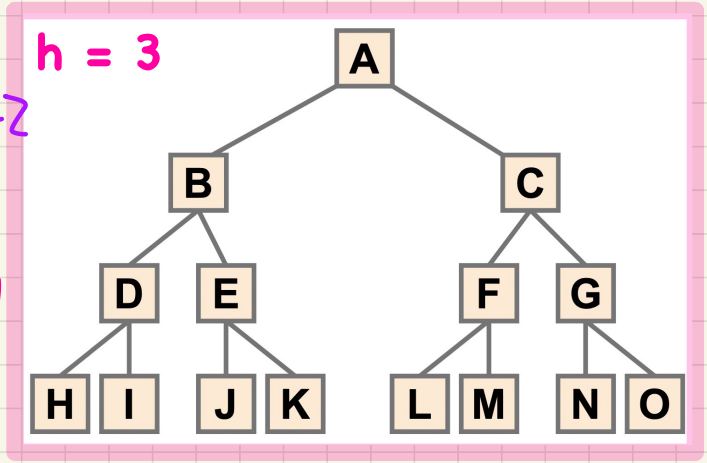
2^h

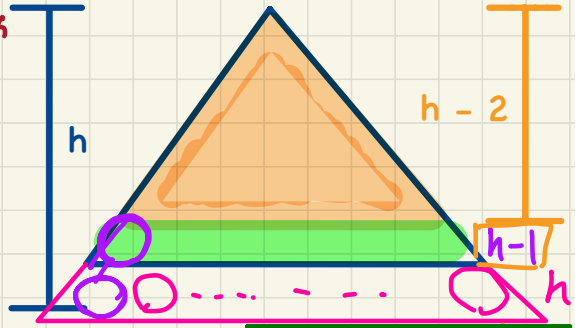
BT Terminology: Complete vs. Full BTs

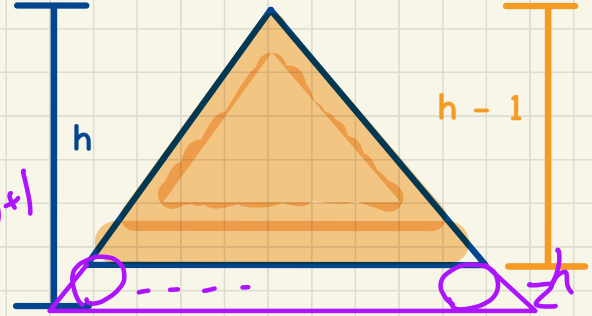
$S_k = 2^k - 1$



- nodes with d = h
- children of nodes at level h-1







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

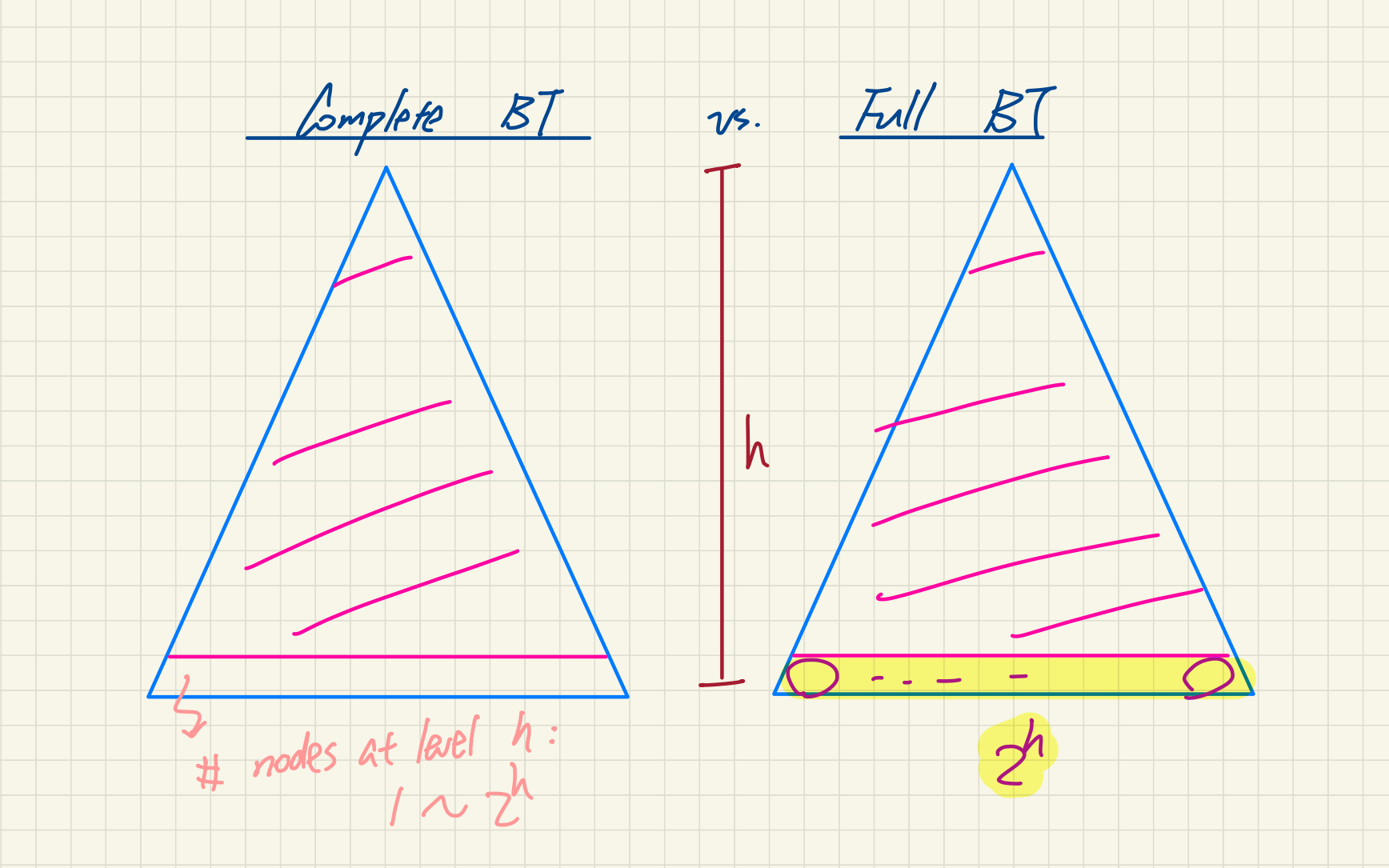
Max # nodes? $(2^0 + 2^1 + \cdots + 2^{h-1}) + 2^h$

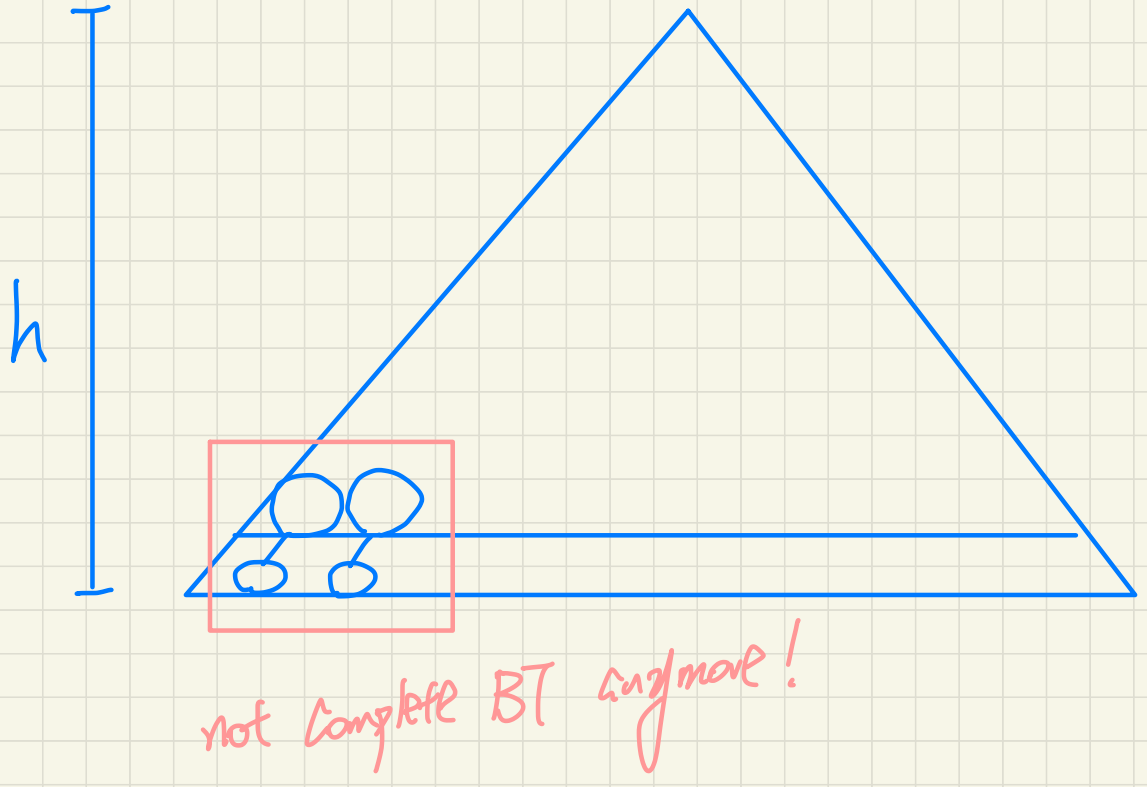
$(2^h - 1) + 1 = 2^h$

$2^{h+1} - 1$

Min # nodes?

Max # nodes? $2^0 + 2^1 + \cdots + 2^h$


Complete BT
vs.
Full BT
h
nodes at level h:
$1 \sim 2^h$
2^h



not Complete BT anymore!

BT Properties: Bounding # of Nodes

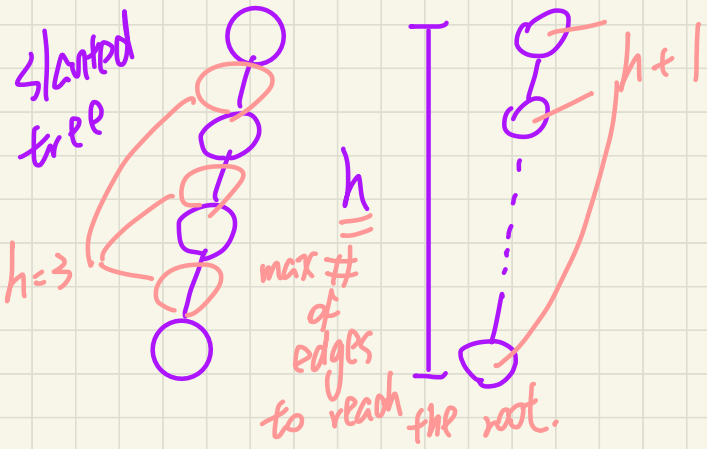
Given a *binary tree* with ***height*** *h*, the ***number of nodes*** *n* is bounded as:

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

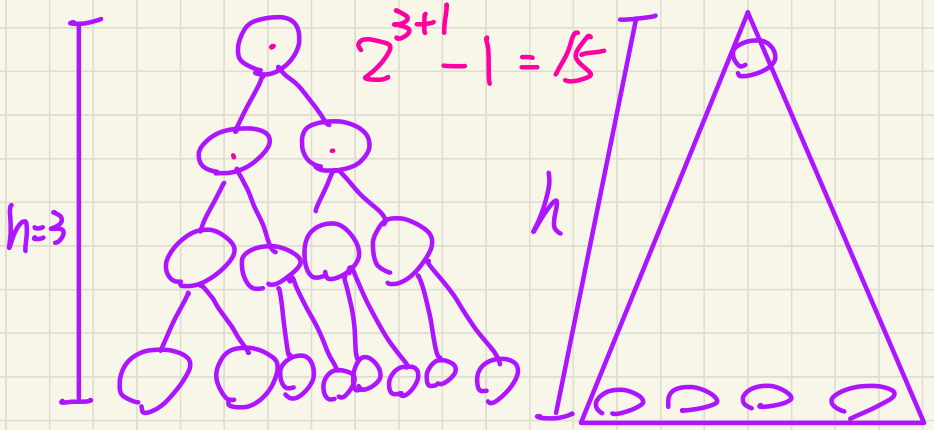
For example, say h = 3 → max depth is 3

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

Minimum # of nodes



Maximum # of nodes



BT Properties: Bounding Height of Tree

$\log z^x = x$

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

$n = 7$

max: $\log(7+1) - 1 = 2$

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

For example, say n = 7

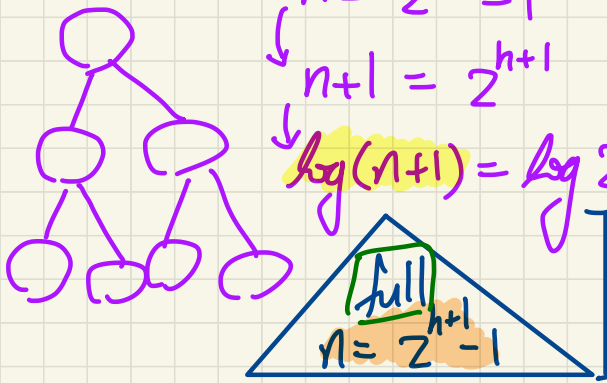
Minimum height → # nodes is max for that min height.

$n = 2^{h+1} - 1$

$n+1 = 2^{h+1}$

$\log(n+1) = \log 2^{h+1} = h+1$

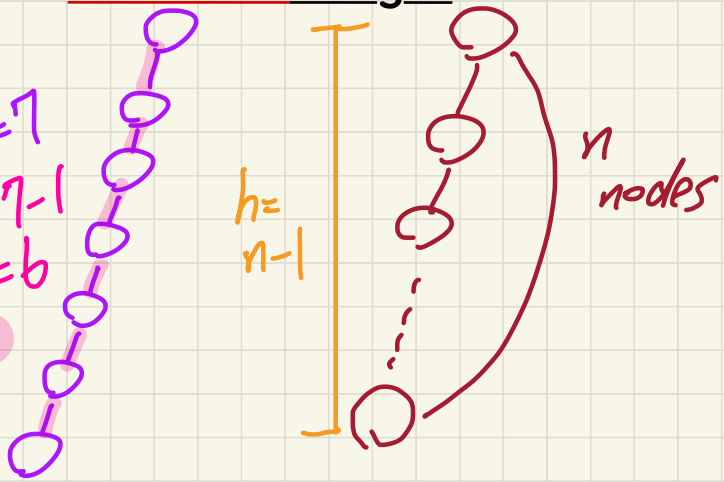
$h = \log(n+1) - 1$ min



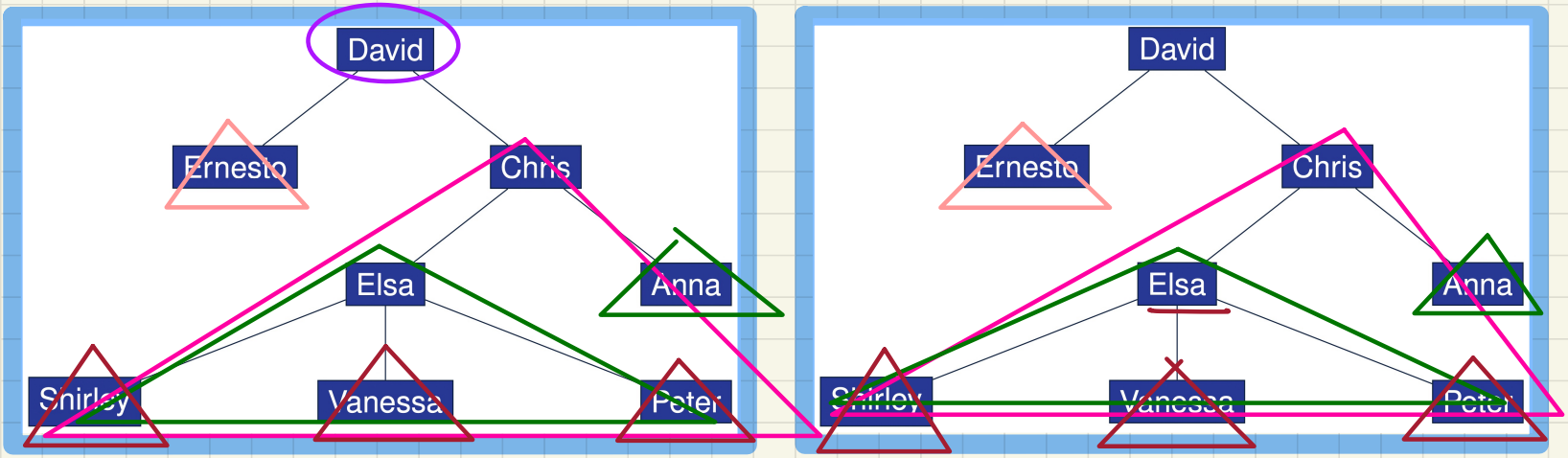
Maximum height

$n = 7$

$h = 7 - 1 = 6$



General Tree Traversals: Pre-Order vs. Post-Order



Pre-Order Traversal from the Root

David Ernesto Chris Elsa S V P Anna

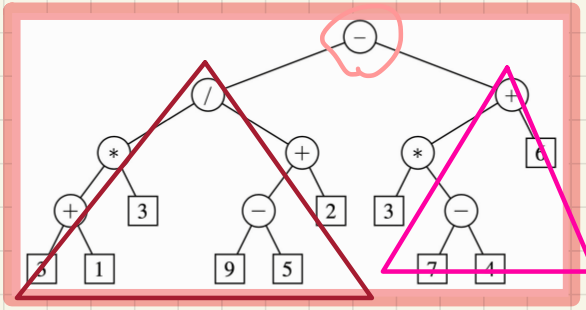
po(E) po(Chris) po(Anna)

Post-Order Traversal from the Root

Ernesto S V P Elsa Anna Chris David

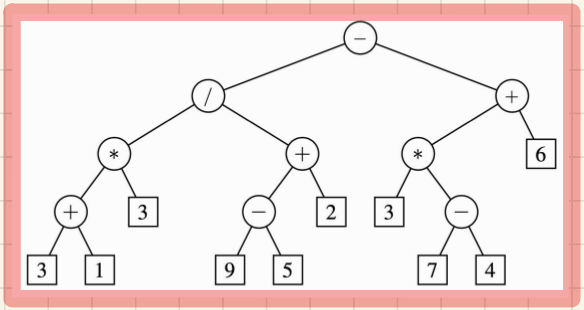
po(E) po(Elsa) po(Anna) po(Chris)

Binary Tree Traversals

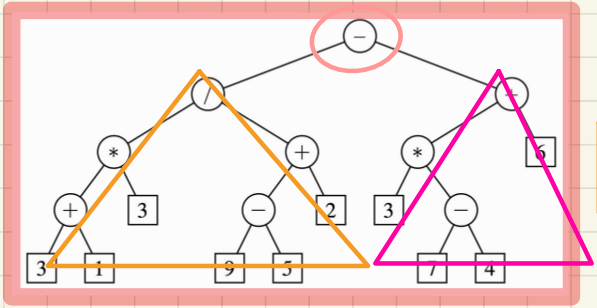


Pre-Order Traversal

− / * + 3 1 3 + − 9 5 2 + * 3 − 7 4 6



In-Order Traversal



Post-Order Traversal

3 1 + 3 * 9 5 − 2 + / 3 7 4 − * 6 + −

Review Q & A - Mar. 10

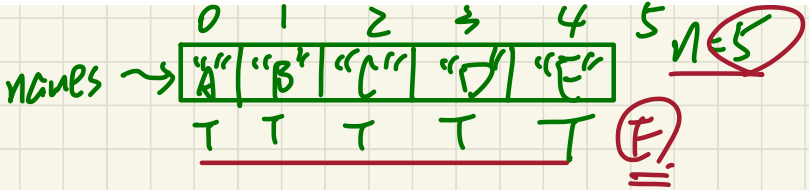
Written Test

Asymptotic Analysis
Instantiating Generics

```
① boolean foundEmptyString = false;
② int i = 0;
③ while (!foundEmptyString && i < names.length) {
④   if (names[i].length() == 0) {
⑤     /* set flag for early exit */
⑤     foundEmptyString = true;
⑤   }
⑥   i = i + 1;
  }
```

worst case is when the "" is never found

→ # iterations = names.length = n



n = 5

① 1

② 1

③ (n+1) · 4

④, ⑤, ⑥ 6 · n

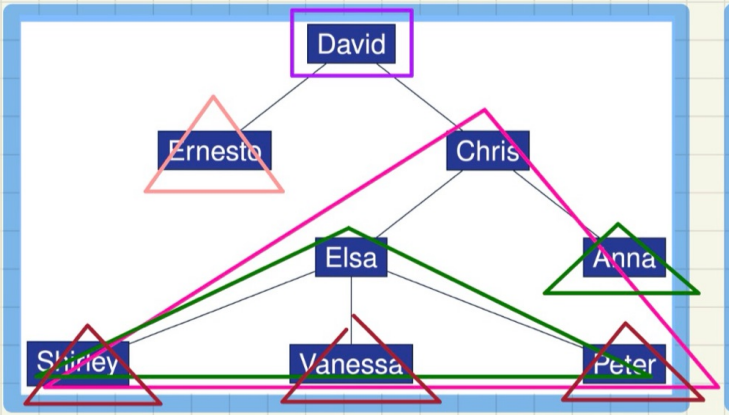
iterations : n

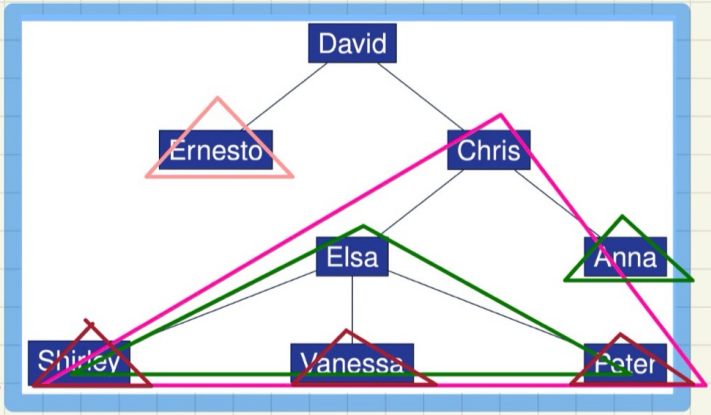
times while cond. is evaluated : n+1 (F)

1 + 1 + (4n+4) + 6n

= 10n + 6

General Tree Traversals: Pre-Order vs. Post-Order





Pre-Order Traversal from the Root

David Ernesto po(E) Chris Elsa S V P po(Elsa) po(Chris) Anna po(Anna)

Post-Order Traversal from the Root

Ernesto po(E) S V P Elsa po(Elsa) Anna po(A) Chris po(C) David

a1.length == n a2.length == m

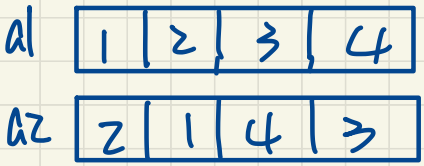
```
boolean contentsMatch(int[] a1, int[] a2){
    for(int i=0; i< a1.length; i++){
        for(int j=0; j< a2.length; j++){
            // check to see if a2[j] == a1[i]   O(1)
        }
    }
    for(int i=0; i< a2.length; i++){
        for(int j=0; j< a1.length; j++){
            // check to see if a1[j] == a2[i]   O(1)
        }
    }
}
```

n · m

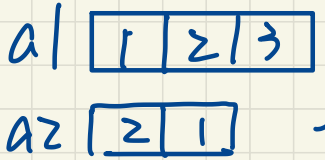
$O(n \cdot m \cdot 1 + m \cdot n \cdot 1)$

$= O(2 \cdot n \cdot m)$

$= O(n \cdot m)$ m · n



true



false

$S1 = S2 \iff S_1 \subseteq S_2 \wedge S_2 \subseteq S_1$

highest power

$5n^2 + 3n \cdot \log n + 2n + 5$ is $O(n^2)$ $[c = 15, n_0 = 1]$

$f(n)$ $g(n)$

Proof

choose $C = |5| + |3| + |2| + |5| = 15$

$n_0 = 1$

Verify: $f(1) \leq C \cdot g(1)$

$5 \cdot 1^2 + 3 \cdot 1 \cdot \log 1 + 2 \cdot 1 + 5$

$= 12$

$\leq 15 \cdot 1^2 = 15$

```
public class MyClass<G, H, I> {
    private G[] a;
    public G m1 (I p1, H p2) {
        /* details of implementation omitted */
        return null;
    }
    public void m2 (H p1,  G p2) {
        /* details of implementation omitted */
    }
}
```

```
public class MyClass<G, H, I> {
    private G[] a;
    public G m1 (I p1, H p2) {
        /* details of implementation omitted */
        return null;
    }
    public void m2 (H p1,  G p2) {
        /* details of implementation omitted */
    }
}
```

Now consider the following declarations from another class (which intends to use the above generic class):

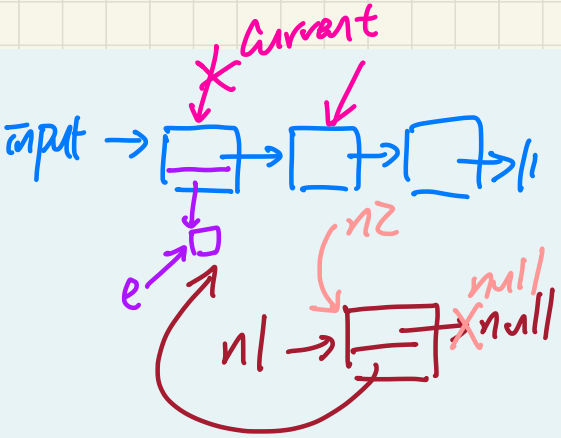
```
Boolean b;
String s;
Integer i;
MyClass<Integer, String, Boolean> obj1;
MyClass<Boolean, String, Integer> obj2;
```

s = obj1.m2 ("alan", 5);	Choose...
b = obj2.m2 ("alan", **false**);	Choose...
obj1.m1 ("alan", 5);	Choose...
obj2.m2 ("alan", **false**);	Choose...
obj1.m2 (5, "alan");	Choose...
obj2.m2 (**false**, "alan");	Choose...
i = obj1.m1 (**true**, "mark");	Choose...
b = obj2.m1 (3, "mark");	Choose...
b = obj1.m1 (**true**, "mark");	Choose...
i = obj2.m1 (3, "mark");	Choose...

Consider the following method which intends to reverse the input chain of nodes:

```
public Node<String> reverseOf(Node<String> input) {
    Node<String> n2 = null;
    Node<String> current = input;
    while(current != null) {
        String e = current.getElement();
        Node<String> n1 = new Node<>(e, null);

        /* missing some line(s) of code */

        current = current.getNext();
    }
    return n2;
}
```

n1.setNext(n2);
n2 = n1;



In the above method, some line or lines of code are missing (from where the comment is) in order for the implementation to be correct. Choose **the** line or **all** lines that are needed.

- [ ] n1.setNext(n2);
- [ ] n2 = n1;
- [ ] n2.setNext(n1);
- [ ] n1.setNext(current);
- [ ] n1 = n2;
- [ ] n2 = current;

Your answer is incorrect.

The correct answers are:

n1.setNext(n2);,

n2 = n1;

Lecture 16 - March 11

Binary Trees, Binary Search

Bounding Internal vs. External Nodes
Proper Binary Trees
Binary Search: Ideas, Java

Announcements/Reminders

- Assignment 3 (on linked Trees) released
- WrittenTest guide & example questions released
- WrittenTest review session materials posted
- Makeup Lecture (on ADTs, Stacks) posted
- Lecture notes template, Office Hours, TA Contact

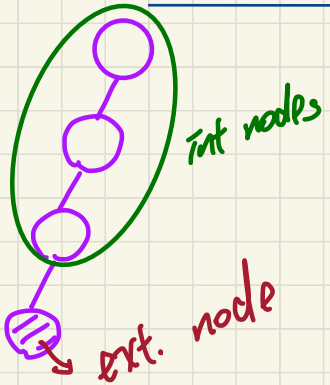
BT Properties: Bounding # of External Nodes

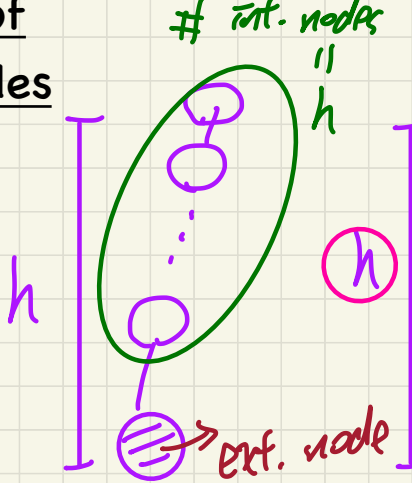
Given a *binary tree* with ***height*** h, the ***number of external nodes*** n_E is bounded as:

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

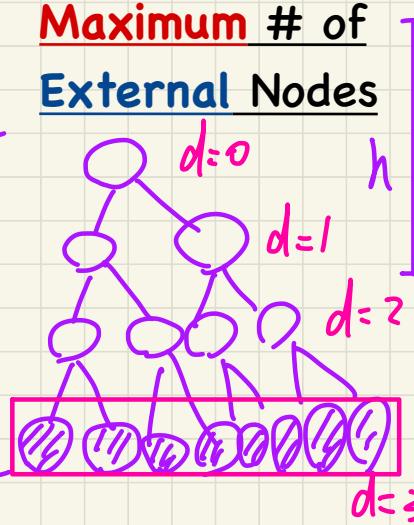
For example, say h = 3 → max # of edges from bottom node to root.

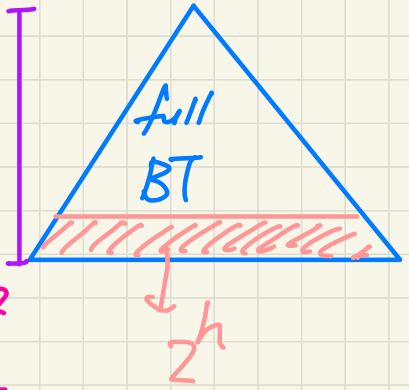
Minimum # of External Nodes





Maximum # of External Nodes



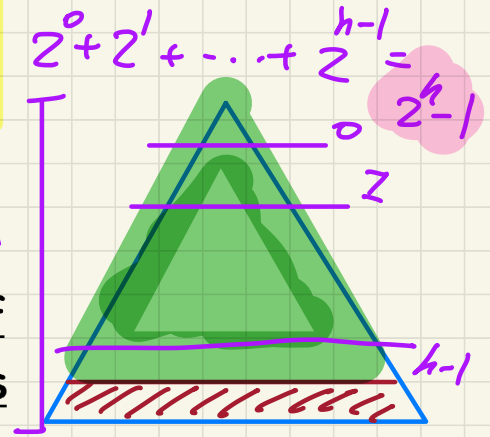


BT Properties: Bounding # of Internal Nodes $S_k = 2^k - 1$

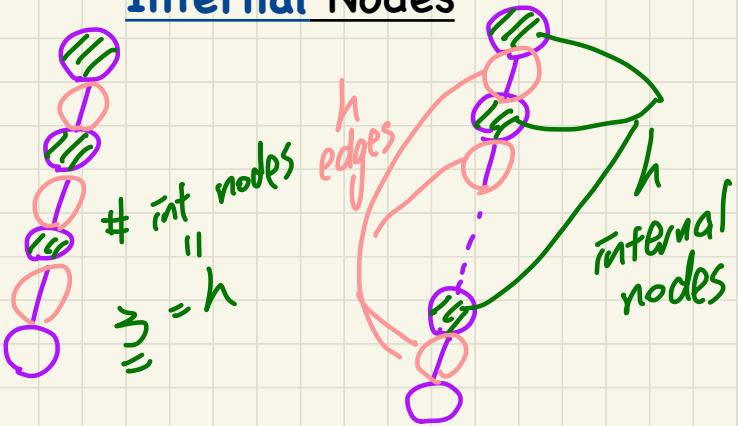
Given a *binary tree* with ***height*** h, the ***number of internal nodes*** n_I is bounded as:

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

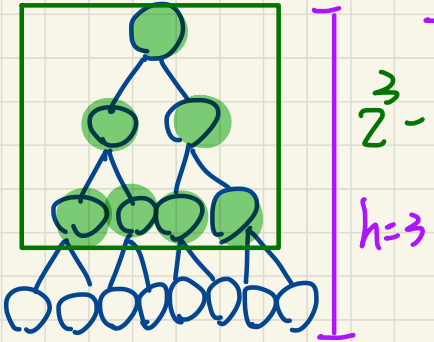
For example, say h = 3



Minimum # of Internal Nodes



Maximum # of Internal Nodes



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

before the ext ← n_I, n_E | n_I', n_E' after ext

Given a *binary tree* that is:

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

We can then expect that: $n_E = n_I + 1$

Induction on Size of Proper BT

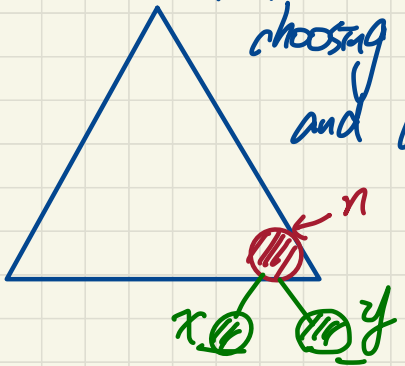
① Base Case: Smallest Proper BT REVIEW

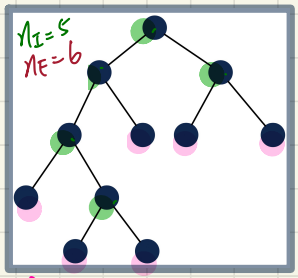
$n_E = 1$, $n_I = 0$) property holds

② Inductive Hypothesis (I.H.):

For a proper BT of size > 1: $n_E = n_I + 1$

③ Given a proper BT, extend it by choosing the right-most ext. node and converting it to an int. node with child nodes (x) and (y).





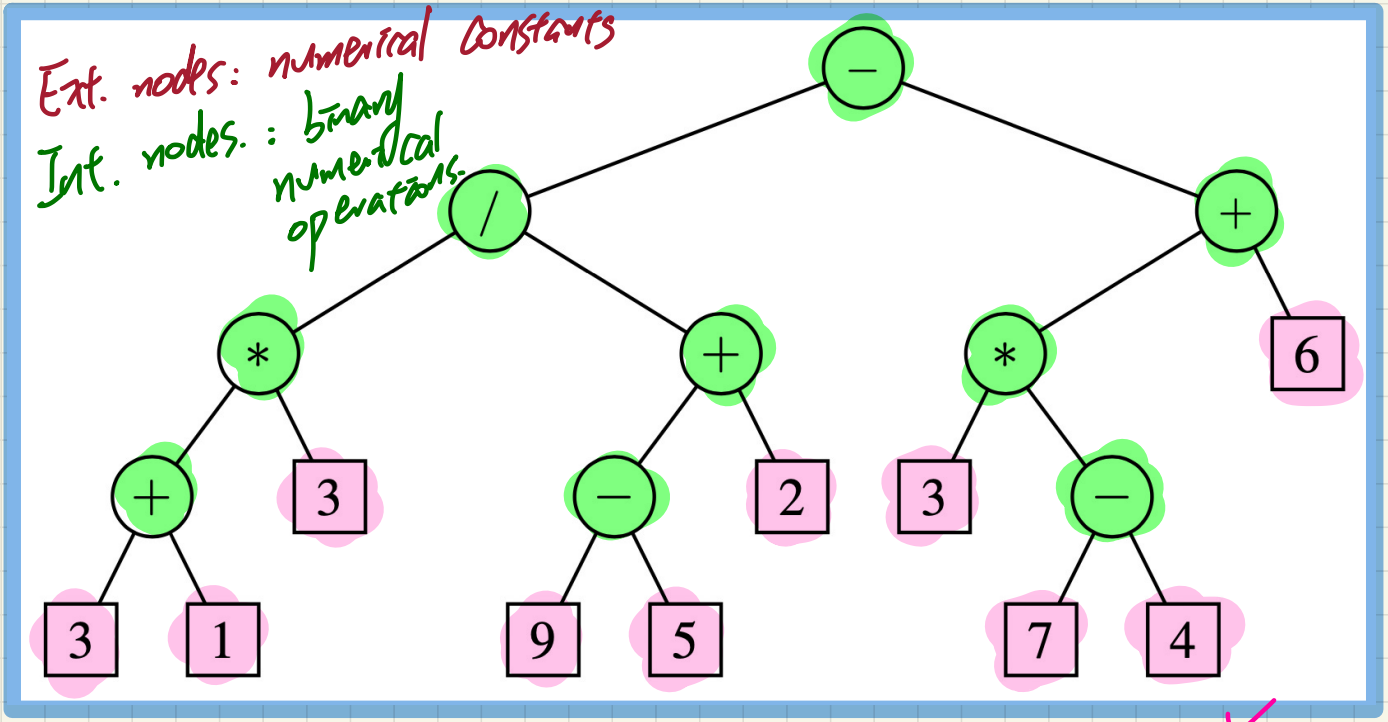
④ $n_E' \overset{(1)}{=} n_E + 1 \overset{I.H.}{=} (n_I + 1) + 1 \overset{(2)}{=} n_I' + 1$

(1) $n_E' = n_E \underbrace{-1}_{n} \underbrace{+2}_{x,y} = n_E + 1$

(2) $n_I' = n_I \underbrace{+1}_{n}$

show: $n_E' = n_I' + 1$

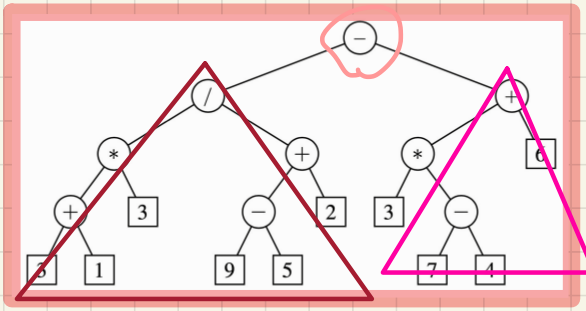
Applications of Binary Trees: Infix Notation



BT's not proper.

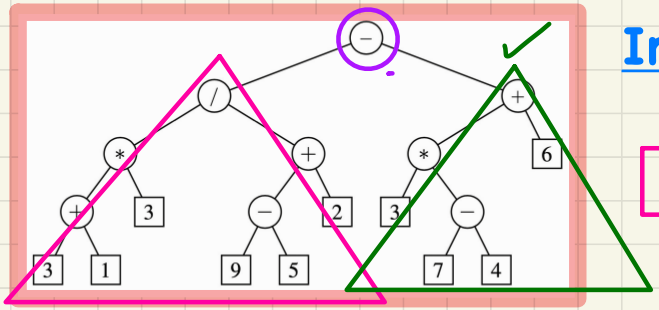
Q. Is the binary tree necessarily proper?

Binary Tree Traversals



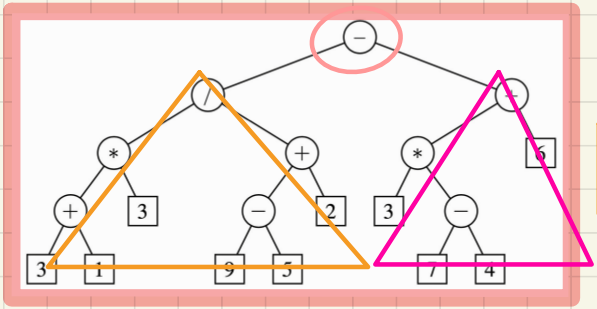
Pre-Order Traversal

− / * + 3 1 3 + − 9 5 2 + * 3 − 7 4 6



In-Order Traversal

3 + 1 * 3 / 9 − 5 + 2 − 3 * 7 − 4 + 6

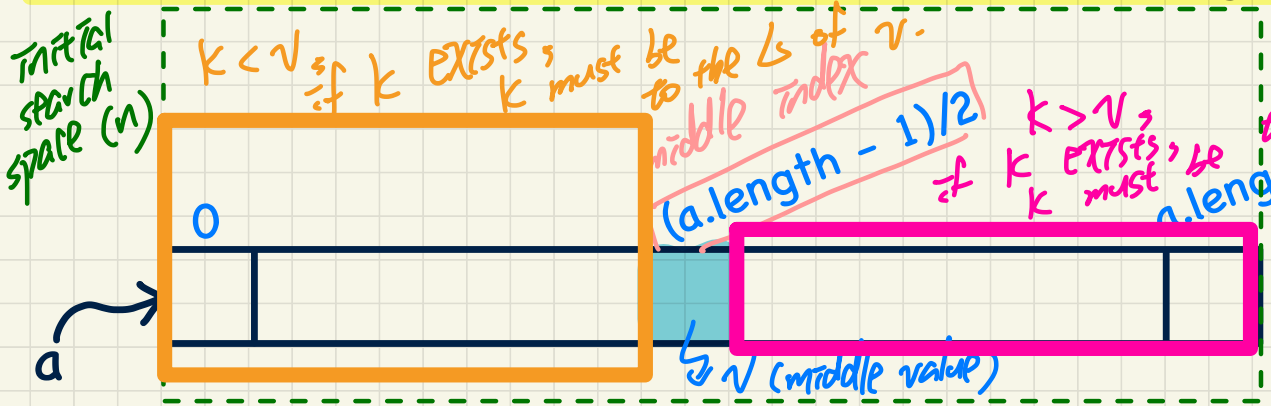


Post-Order Traversal

3 1 + 3 * 9 5 − 2 + / 3 7 4 − * 6 + −

Binary Search: Ideas

Precondition: Array **sorted** in non-descending order



Size of Search Space

n, n/2, n/4, ..., 1

comparisons

log n (worst case)

Search: Does key **k** exist in array **a**?

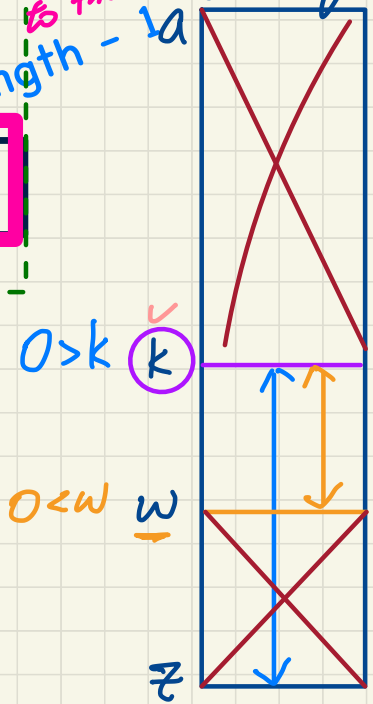
① Access the middle of search space

② Compare k against v:

k == v → found

k < v → Recur on the left

k > v → Recur on the right.



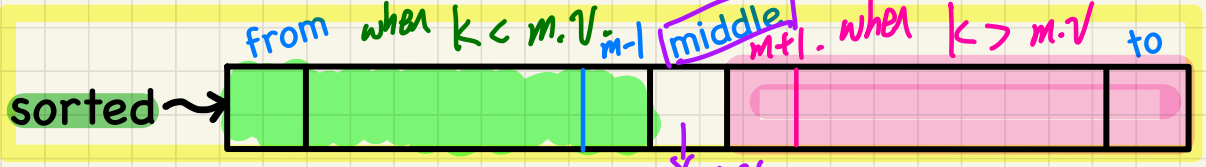
Binary Search in Java

```
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; }
  }
}
```

base

recursive cases

k == m.v.



Lecture 17 - March 18

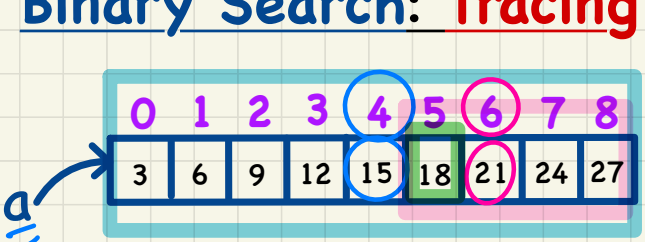
Binary Search, Merge Sort

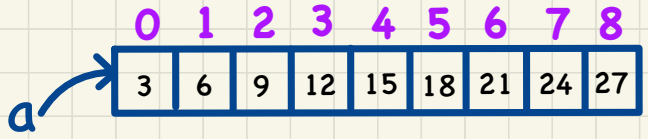
Binary Search: Tracing, Running Time
MergeSort: Ideas, Java, Tracing

Announcements/Reminders

- Assignment 3 (on linked Trees) solution released
- WrittenTest and ProgTest1 results & feedback released
- ProgTest2 guide & example questions to be released
- Makeup Lecture (on Queues) posted
- Lecture notes template, Office Hours, TA Contact

Binary Search: Tracing





Exercise

search(a,7)

binarySearchH(a,0,8,7)

binarySearchH(a,0,3,7)

binarySearchH(a,2,3,7)

binarySearchH(a,2,1,7)

search(a,18) key

binarySearchH(a,0,8,18)

$m = \frac{0+8}{2} = 4$

m.v. a[4] == 15

18 > 15

↳ go R

(4+1, 8)

binarySearchH(a,5,8,18)

$m = \frac{5+8}{2} = 6$

m.v. a[6] == 21

18 < 21

↳ go L.

(5, 6-1)

binarySearchH(a,5,5,18)

a[5] == 18

temp ↳ found !!

Binary Search: Running Time

```
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; }
  }
}
```

n T(n) = ?

T(0) = 1

T(1) = 1

O(1)

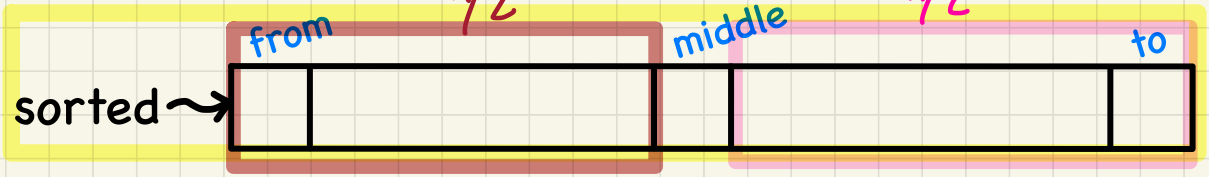
n/2

assume: $T(\frac{n}{2})$

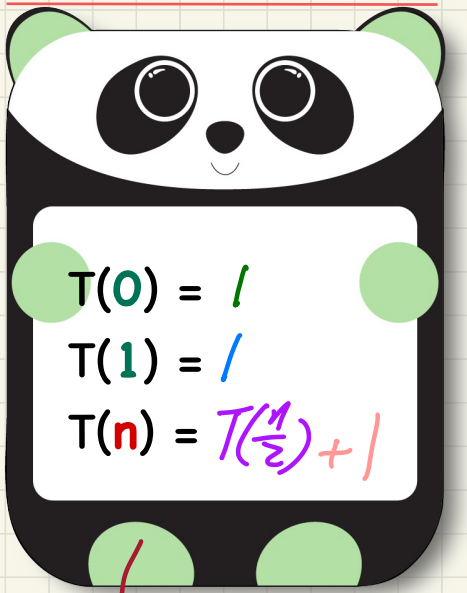
n/2

assume: $T(\frac{n}{2})$

T(n)



Running Time as a Recurrence Relation



wrong: (∵ if-else if)

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

Running Time: Unfolding Recurrence Relation

$T(0) = 1$

$T(1) = 1$

✓ $T(n) = T(n/2) + 1$

Assume without loss of generality: $n = 2^x$ for $x \geq 0$

Warm-up: $T(\frac{n}{2}) = T(\frac{n/2}{2}) + 1 = T(\frac{n}{4}) + 1$

$T(n) = T(\frac{n}{2}) + 1$ (2¹)

$= (T(\frac{n}{4}) + 1) + 1$ (2²)

$= ((T(\frac{n}{8}) + 1) + 1) + 1$ (2³)

$\vdots$

$= T(1) + 1 + 1 + \cdots + 1$ (how many? $\log n$.)

$= 1 + \log n \cdot 1 = \log n$

$1 = \frac{n}{n} = \frac{n}{2^{\log n}}$

$2^{\log n} = n$

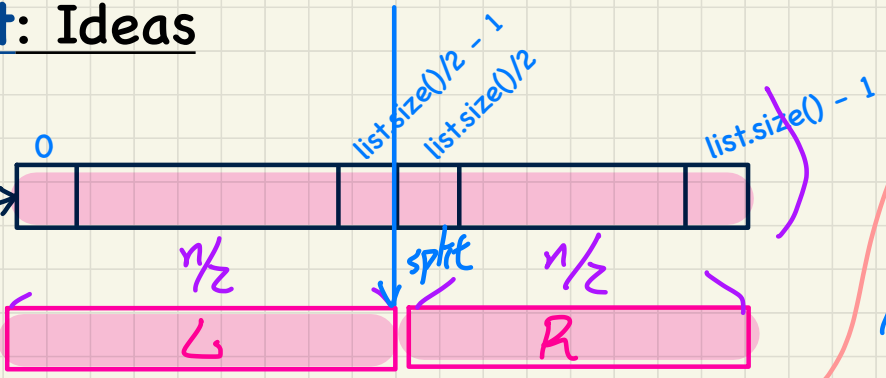
$n = 8$

$\frac{8}{2^{\log 8}} = \frac{n}{n} = 1$

Merge Sort: Ideas

- split
- merge

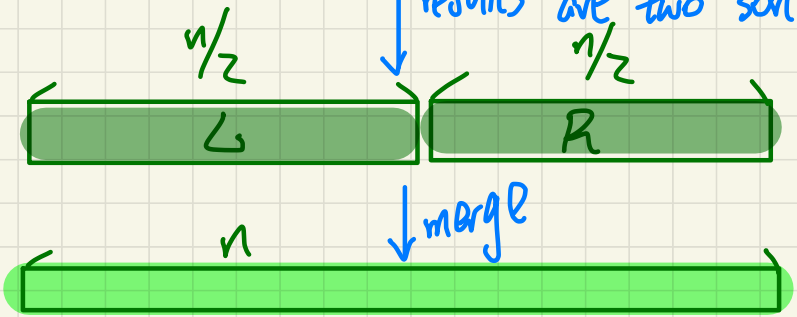
sort (list)



sort recursively

sort (L) sort (R)

results are two sorted lists



Each recursive call:

1. split ($n \to \frac{n}{2}, \frac{n}{2}$) O(1)

$\frac{n}{2} \to \frac{n}{4}, \frac{n}{4}$

$2 \to 1, 1$ (sorted)

2. merge O(n)

$1, 1 \to 2$,

$2, 2 \to 4$,

$\frac{n}{4}, \frac{n}{4} \to \frac{n}{2}$

$\frac{n}{2}, \frac{n}{2} \to n$

Merge Sort in Java

Total # iterations for merge: (L.size + 0) + (R.size − 0)

RT for merge: O(1 · (L.size + R.size))

* Exhaust either L or R = L.size + R.size

↓ O(1)

e.g. L is exhausted

≈ i + j iterations so far. (i = L.size)

** Append remaining elements from R

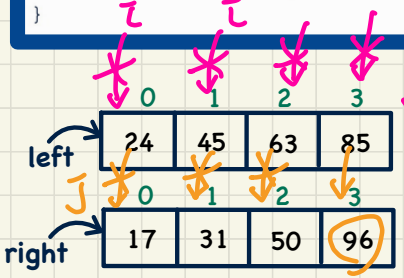
≈ R.size − j iterations.

```
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;
}
```

base cases

O(1)

O(L.size + R.size) L is exhausted



Precondition: L and R are sorted

```
/* 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;
}
```

exit: i >= L.size || j >= R.size

O(1)

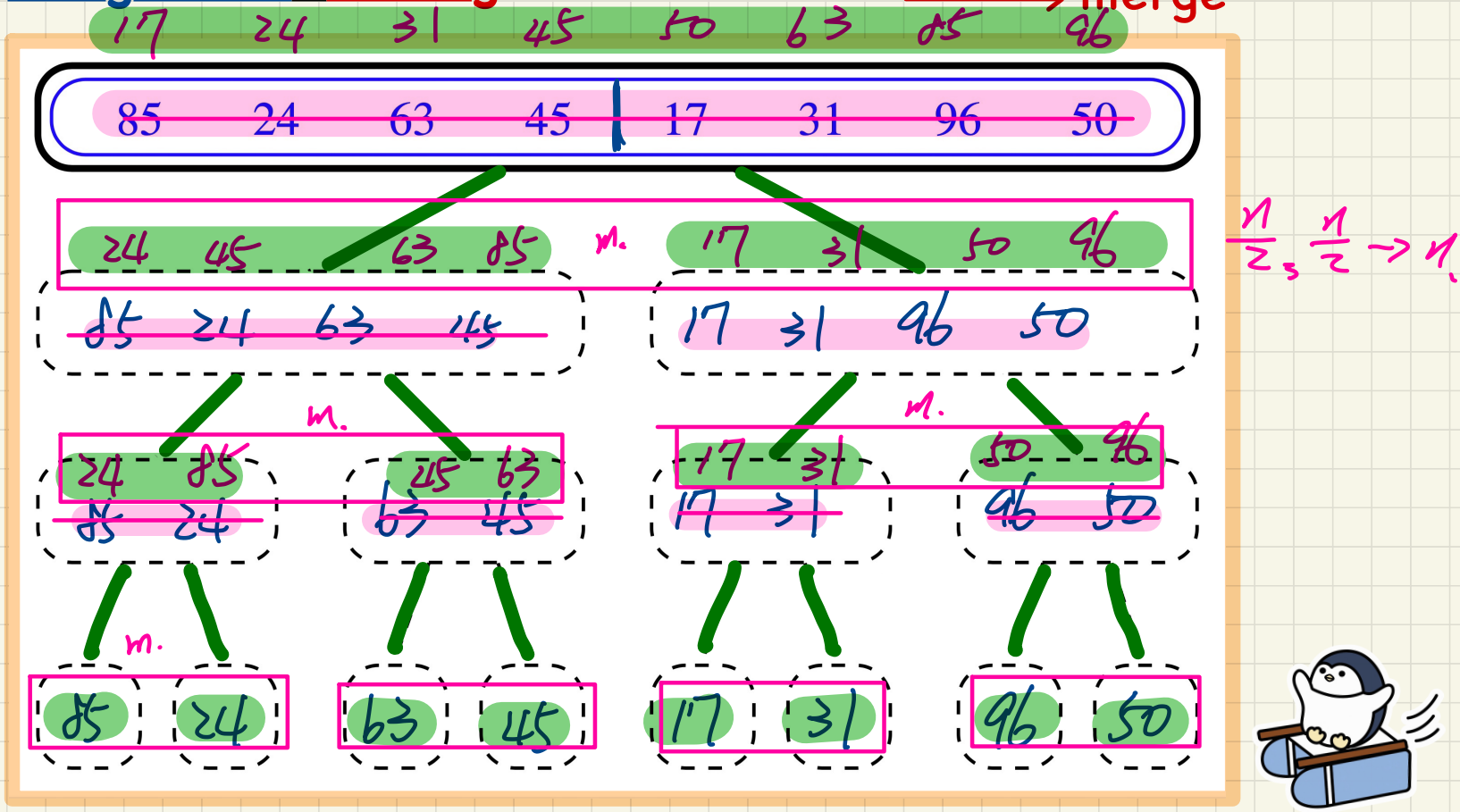
O(1)

append the remaining elements from the unexhausted list.

Merge Sort: Tracing

split

merge



Lecture 18 - March 20

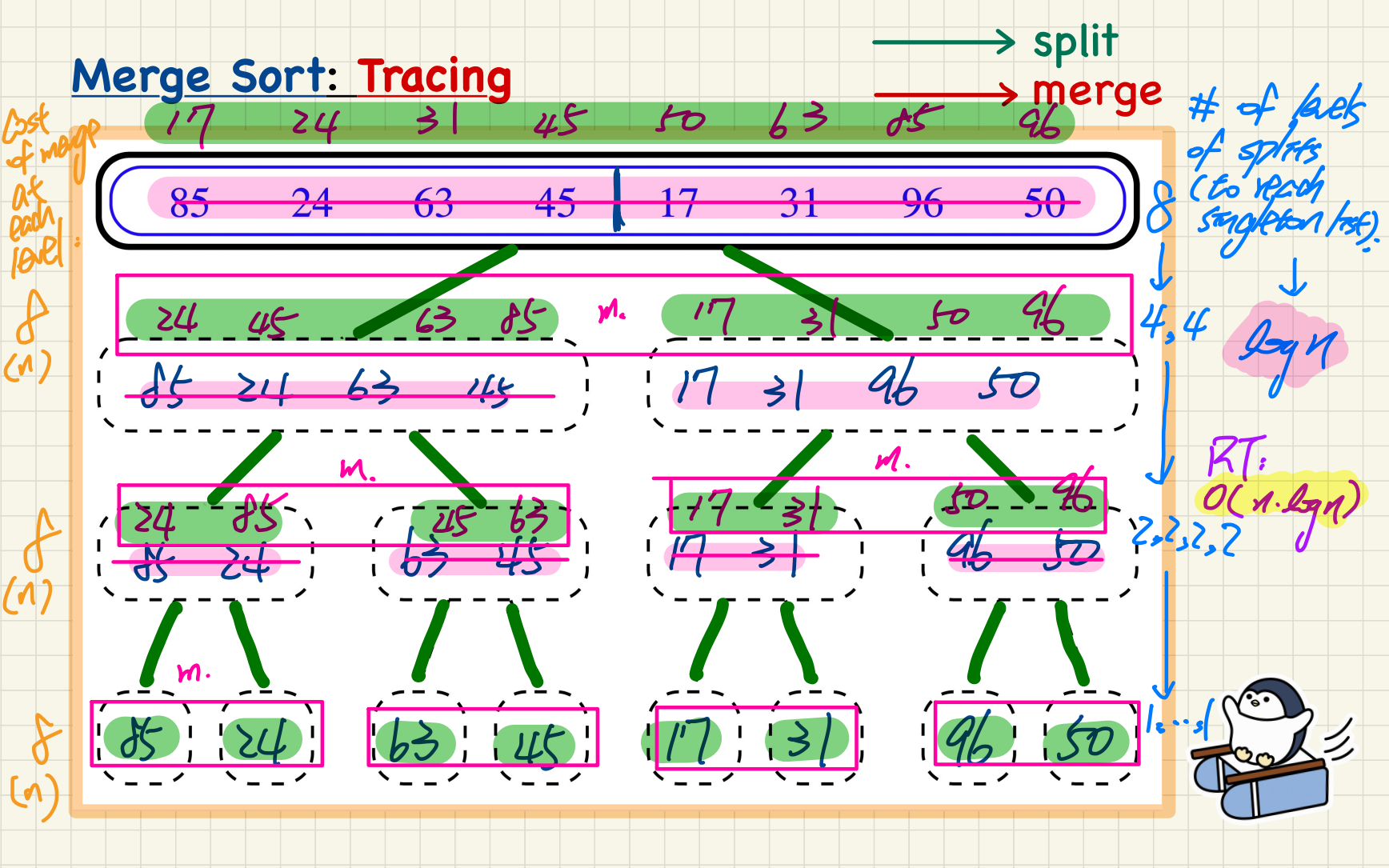
Merge Sort, Quick Sort, BST

MergeSort: Recurrence Relation
QuickSort: Ideas, Java, RT

Announcements/Reminders

- ProgTest2 info & example questions released
- Assignment 3 (on linked Trees) solution released
- WrittenTest and ProgTest1 results & feedback released
- Makeup Lecture (on Queues) posted
- Lecture notes template, Office Hours, TA Contact

Merge Sort: Tracing



Merge Sort: Running Time

```java
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;
}
```

$T(0) = 1$

$T(1) = 1$

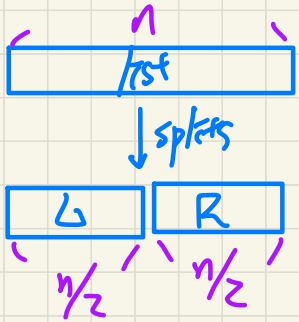
$O(1)$

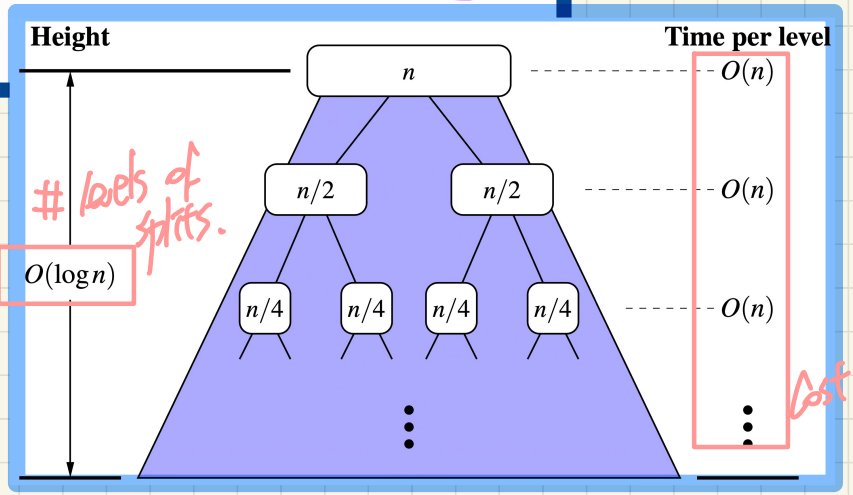
$T(n)$

assump: $T(\frac{n}{2})$

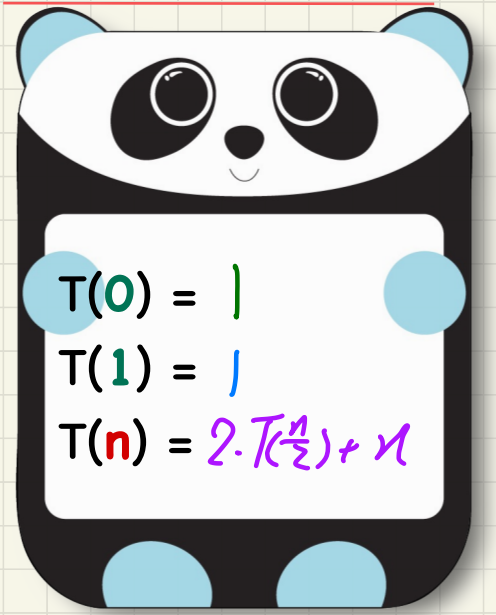
assump: $T(\frac{n}{2})$

$O(n)$





Running Time as a Recurrence Relation



Running Time: Unfolding Recurrence Relation

$T(0) = 1$

$T(1) = 1$

$T(n) = 2 \cdot T(n/2) + n$

Warm-up: $T(\frac{n}{2}) = 2 \cdot T(\frac{n/2}{2}) + \frac{n}{2}$

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

$= 2 \cdot (2 \cdot T(\frac{n}{4}) + \frac{n}{2}) + n \quad [4 \cdot T(\frac{n}{4}) + 2n]$

$= 2 \cdot (2 \cdot (2 \cdot T(\frac{n}{8}) + \frac{n}{4}) + \frac{n}{2}) + n \quad [8 \cdot T(\frac{n}{8}) + 3n]$

$\vdots$

$= 2^{?} \cdot T(1) + ?n = 2^{\log n} \cdot 1 + \log n \cdot n =$

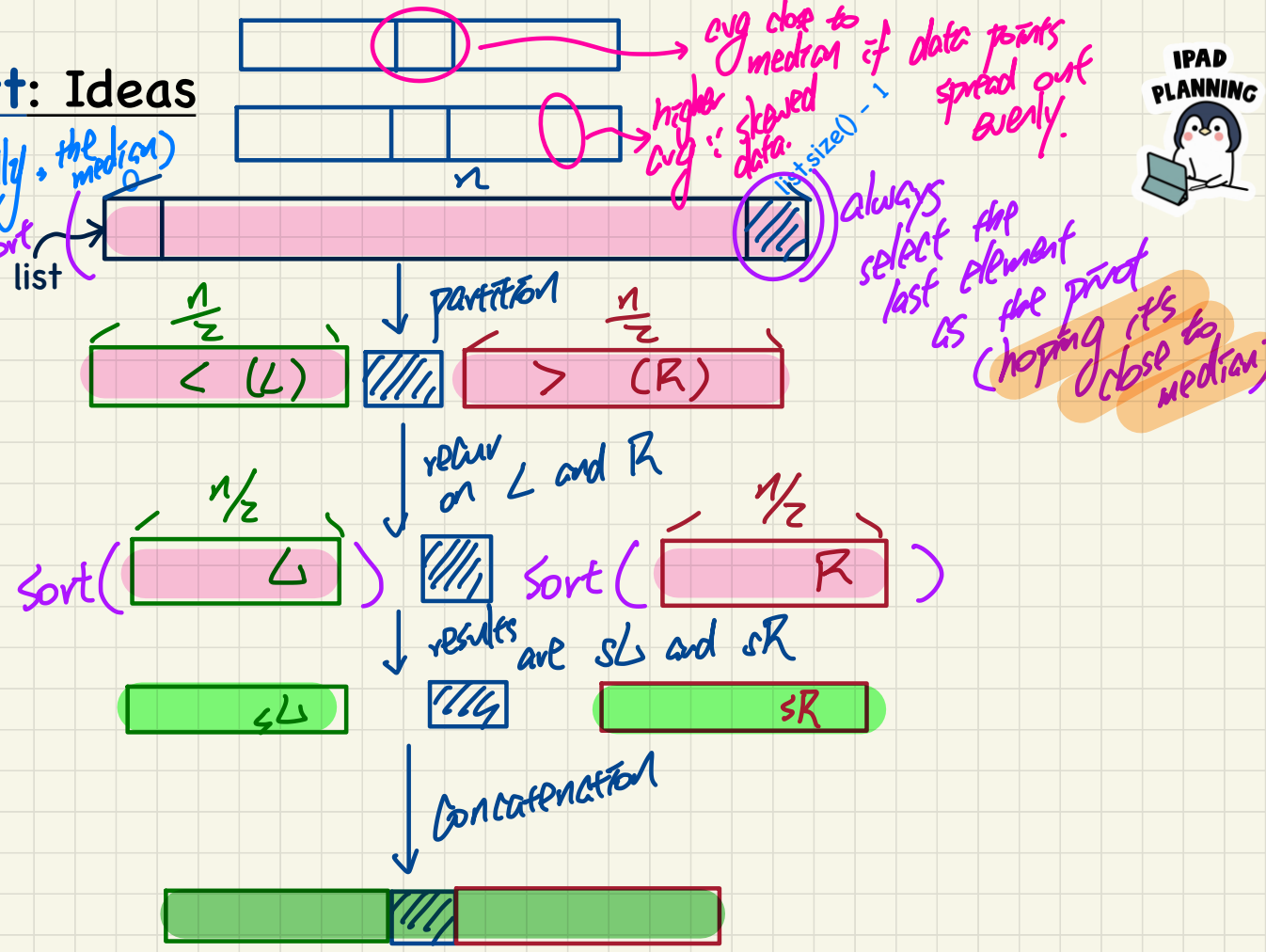
$I = \frac{n}{n} = \frac{n}{2^{\log n}}$

$n + \log n \cdot n$ is $O(n \cdot \log n)$

Quick Sort: Ideas

- pivot selection (ideally, the median)
- partition
- concatenation

median of medians algorithm → find median in O(n)



IPAD PLANNING

is merging sL and sR necessary? No!

Quick Sort in Java

Median of medians algo: O(n)
↳ as opposed to sort and select middle

```
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;
}
```

T(n)

O(1)

→ linear scan of the input list is necessary: O(n)

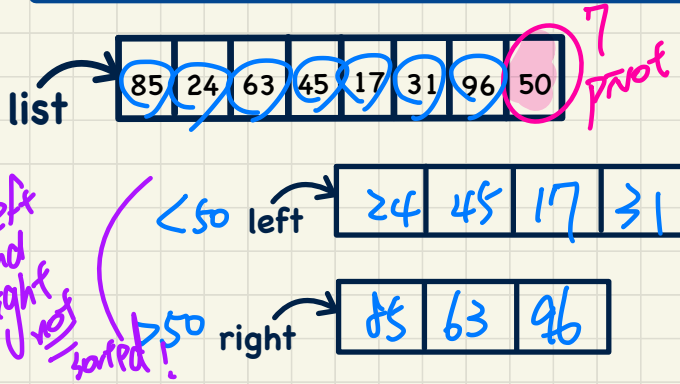
T(n/2) → only if pivot ≈ median

T(n/2)

Concat. O(1)

partition

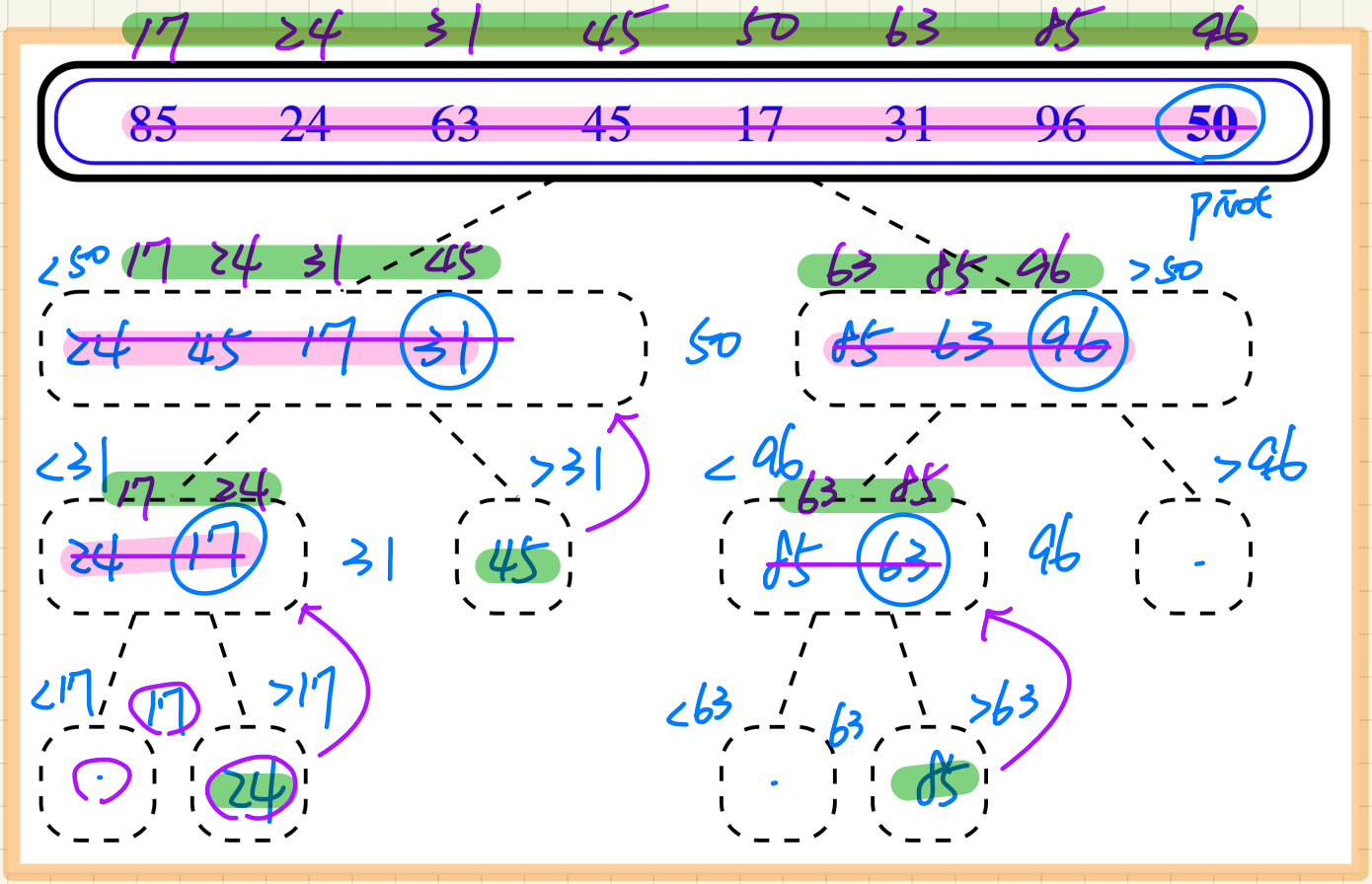
```
List<Integer> allLessThanOrEqualTo(int pivotIndex, List<Integer> list)
  List<Integer> sublist = new ArrayList<>();
  int pivotValue = list.get(pivotIndex);
  for(int i = 0; i < list.size(); i ++) {
    int v = list.get(i);
    if(i != pivotIndex && v <= pivotValue) { sublist.add(v); }
  }
  return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



Quick Sort: Tracing

→ split partition

→ concatenate



Quick Sort: Worst-Case Running Time

```
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;
}
```

O(1)

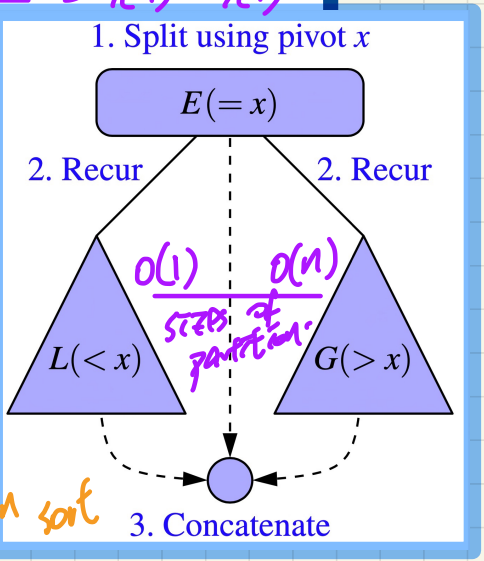
O(n)

O(1)

bad pivot >> median
<< median

|left| << |right|

T(n) T(1)



4 partitions

9 7 5 3 1

n-1

9 7 5 3

9 7 5

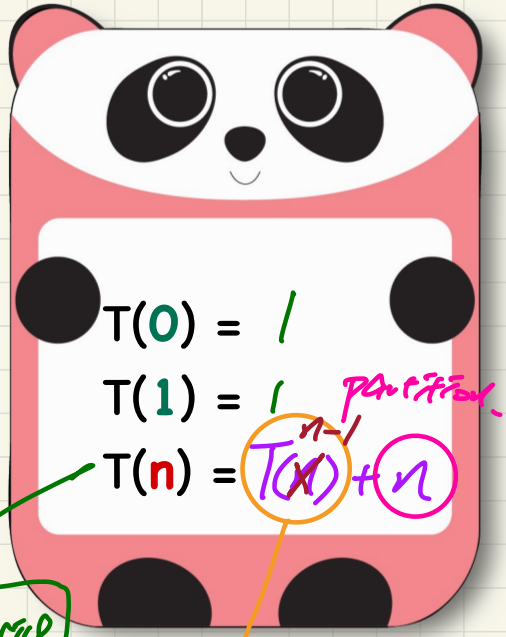
O(n)

9 7

9

overall RT is $O(n^2)$ ≈ selection sort, insertion sort

Running Time as a Recurrence Relation



Exercise

a half that's much larger than the other

Quick Sort: Best-Case Running Time

```
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;
}
```

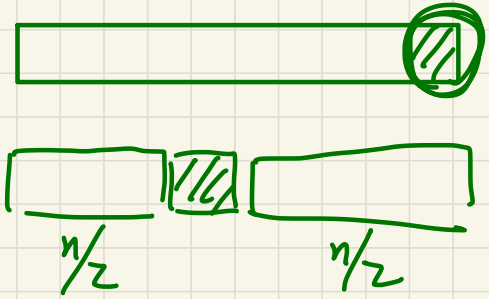
pivot ≈ median

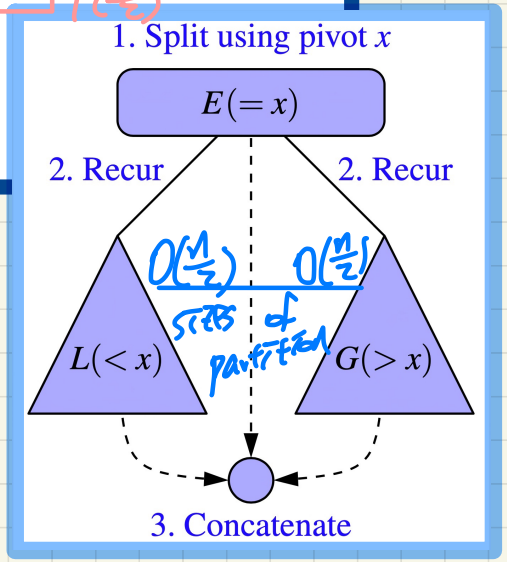
O(n)

O(1)

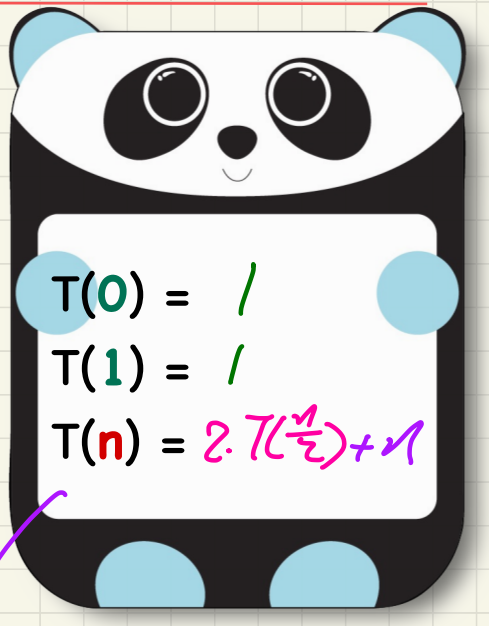
T(n/2)

T(n/2)





Running Time as a Recurrence Relation



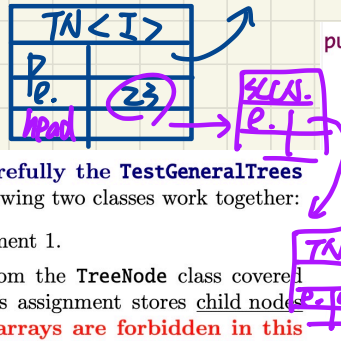
Exercise

Review Q & A - Mar. 21

Programming Test 2

Assignment 3 Solution

Generic Classes



This assignment requires the more intermediate use of generics. **Study carefully the TestGeneralTrees class given to you (in the tests package)**, which contains how the following two classes work together:

- SLLNode<E>: this class is essentially the Node class you used in Assignment 1.
- TreeNode<E>: this class is similar to but should be distinguished from the TreeNode class covered in Lecture W8. The version of TreeNode class you are given for this assignment stores child nodes as **a chain of singly-linked nodes**. **Remember that primitive arrays are forbidden in this assignment.**
- In the TestGeneralTrees class, pay attention to the following type declarations:
  - `TreeNode<String> n;` declares a tree node storing some string value: we write `n.getElement()` to retrieve the string value. In this case, the generic parameter E declared in the TreeNode class (i.e., TreeNode<E>) is instantiated by String.
  - `TreeNode<Integer> n;` declares a tree node storing some integer value: we write `n.getElement()` to retrieve the integer value. In this case, the generic parameter E declared in the TreeNode class (i.e., TreeNode<E>) is instantiated by Integer.
  - `SLLNode<TreeNode<String>> tn;` declares a singly-linked node storing the reference of some tree node (which in turn stores some string value): we write `tn.getElement()` to retrieve the tree node (of type TreeNode<String>) and write `tn.getElement().getElement()` to retrieve the stored string value. In this case, the generic parameter E declared in the SLLNode class (i.e., SLLNode<E>) is instantiated by TreeNode<String> (which in tern instantiates the generic parameter E in the TreeNode class by String).
  - `SLLNode<TreeNode<Integer>> tn;` declares a singly-linked node storing the reference of some tree node (which in turn stores some integer value): we write `tn.getElement()` to retrieve the tree node (of type TreeNode<Integer>) and write `tn.getElement().getElement()` to retrieve the stored integer value. In this case, the generic parameter E declared in the SLLNode class (i.e., SLLNode<E>) is instantiated by TreeNode<Integer> (which in tern instantiates the generic parameter E in the TreeNode class by Integer).

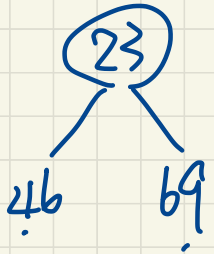
```
public class TreeNode<E> {
    private E element; /* data object */
    private TreeNode<E> parent; /* unique parent
    private SLLNode<TreeNode<E>> headOfChildList
    private SLLNode<TreeNode<E>> tailOfChildList

    public TreeNode(E element) {

    public E getElement() {

    public void setElement(E element) {

    public TreeNode<E> getParent() {

    public void setParent(TreeNode<E> parent) {

    public SLLNode<TreeNode<E>> getChildren() {

    public void addChild(TreeNode<E> child) {
}
```

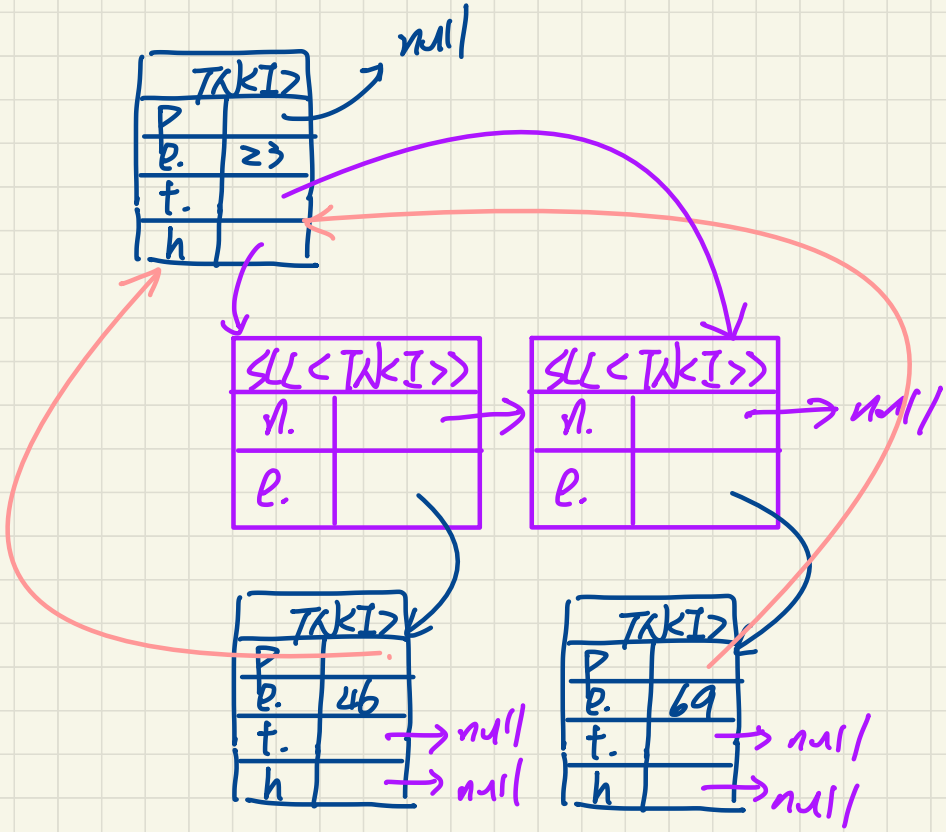
head.getE. → TN<E>

head.getE().getE()

TN<E>

```
public class SLLNode<E> {
    private E element;
    private SLLNode<E> next;

    public SLLNode(E e, SLLNode<E> n) {

    public E getElement() {

    public SLLNode<E> getNext() {

    public void setNext(SLLNode<E> n) {

    public void setElement(E e) {
}
```





A3 starter

↳ tests

↳ TestsTrees.java

Task 1 Rank

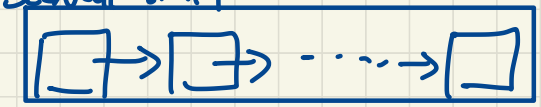
task1(TN n, int i, int j)

(1) n

(2) n

(a) pre-order or post-order traversal.

SLNode chain



(b) Sort the traversal result

↳ e.g. insertion sort or selection sort

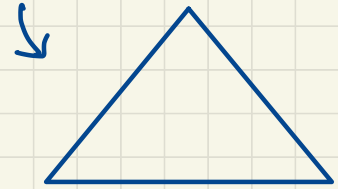
(c) extract nodes from indices i to j.

(a) traverse, and when each node is added, insert it s.t. the result chain of nodes remains sorted.

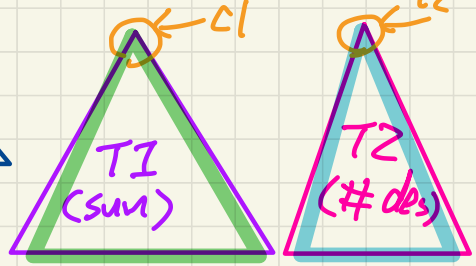
Task 2: Stats (sum of values, # des.)

(1) n

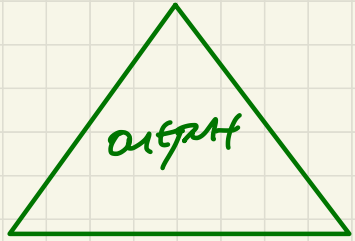
input →



(2) n



output



Lecture 19 - March 27

Binary Search Trees

BST: Search Property
BST: Sorting Property
BST: Constructing BST Nodes

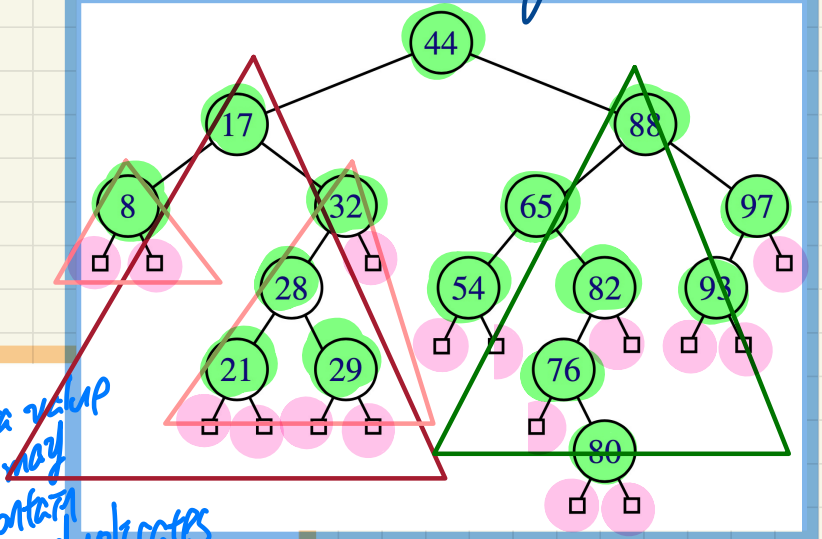
Announcements/Reminders

- Assignment 4 (on linked Trees) released
- Makeup Lecture to be posted
- Lecture notes template, Office Hours, TA Contact

Binary Search Trees: Recursive Definition

Search Property : should hold recursively on the root of every subtree

- external node
- internal node
  - + LST
  - + RST



every

for searching ↳ no duplicates

Node p stores (key(p), value(p)) — P → data value ↳ may contain duplicates

LST

Each node n of LST is such that key(n) < key(p)

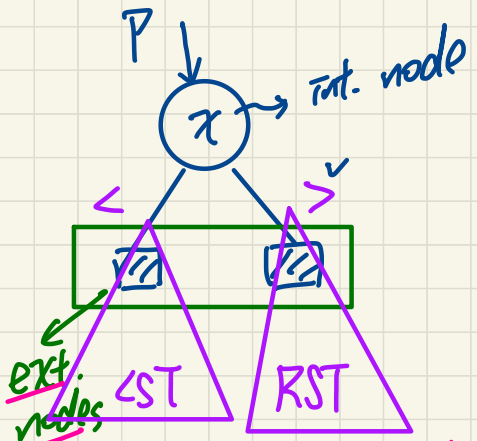
$\forall n \cdot n \in LST(p) \Rightarrow key(n) < key(p)$

RST

Each node n of RST is such that key(n) > key(p)

$\forall n \cdot n \in RST(p) \Rightarrow key(n) > key(p)$

Is a Singleton BT a BST?



(≈ header/trailer in DLN
↳ code structure
↳ does not improve performance).

$\forall x \cdot \boxed{false \Rightarrow P(x)}$

zero of $\Rightarrow$: True

Search Property

$\forall n : \boxed{n \in LST(p)} \Rightarrow$ True

false

$key(n) < key(p)$

$\wedge$

$\forall n : \boxed{n \in RST(p)} \Rightarrow$ True

false

$key(n) > key(p)$

Binary Search Trees: Sorting Property

- BST: Non-Linear Structure
- In-Order Traversal

≈ Concat. in quicksort.

sorted seq. of all keys in RST

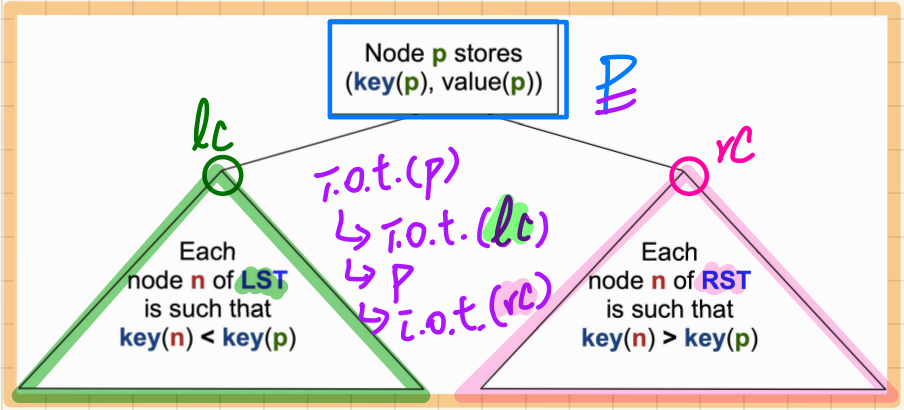
T.O.t.(LST) p T.O.t(RST)

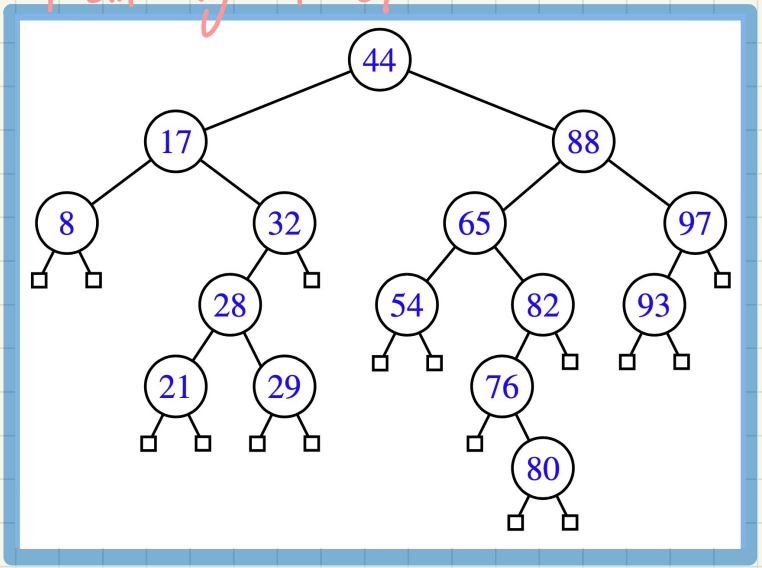
sorted seq. of all keys in LST

Given a BT that's a BST:

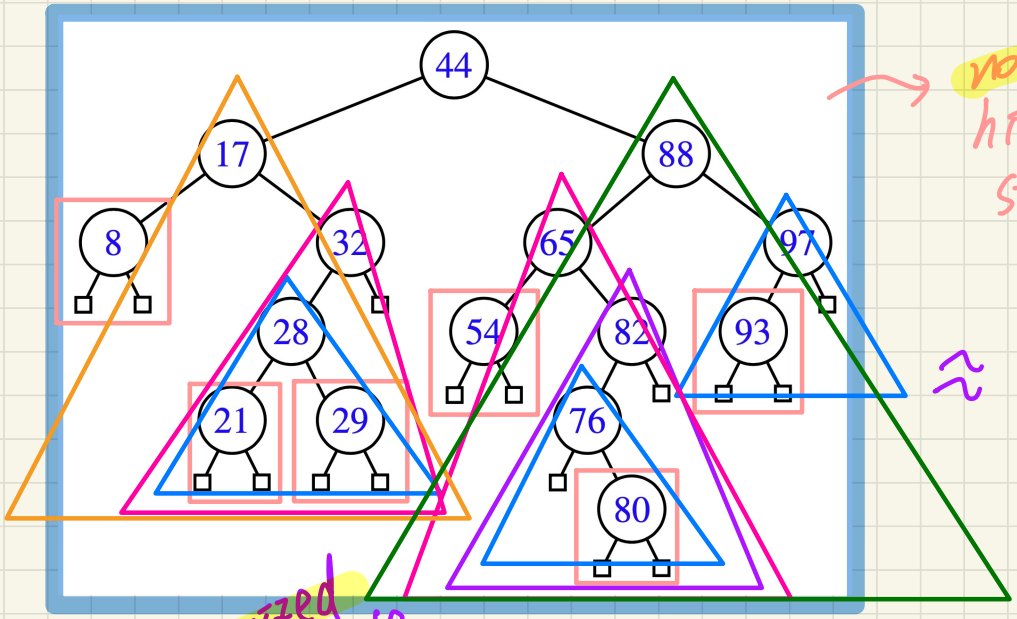
(1) search property sat. (recursively)

(2) in-order traversal (T.O.t.):





Building Sorted Seq. from In-Order Traversal on BST



non-linear, hierarchical structure

≈ concat. in quick sort!

I.O.T.

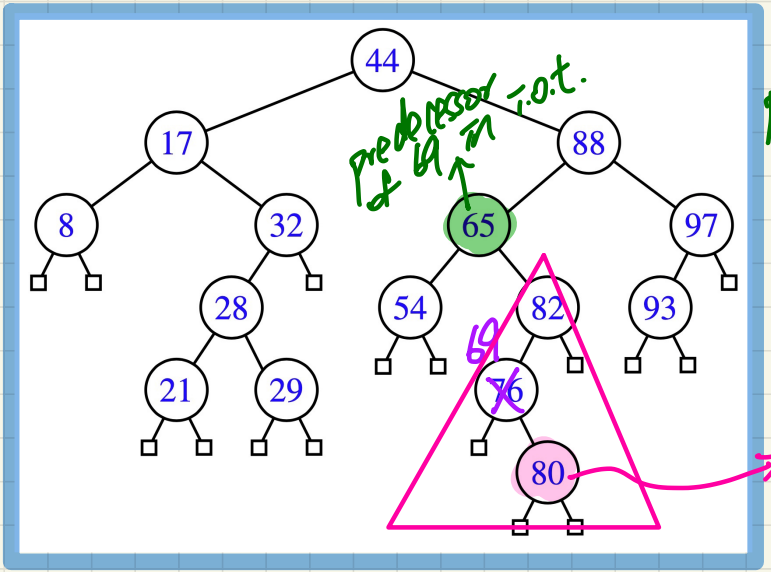
linearized version of the tree

8 17 21 28 29 32 44 54 65 76 80 82 88 93 97

Exercise: Checking the Search Property (1)

Remember: For a BT to be a BST, the Search Property should hold recursively on the root of each subtree.

In-Order: <8, 17, 21, 28, 29, 32, 44, 54, 65, 76, 80, 82, 88, 93, 97>



predecessor

successor

69

ok :)

65 < 69 < 80

predecessor of 69 in i.o.t.

successor of 69 in i.o.t.

Exercise: Checking the Search Property (2)

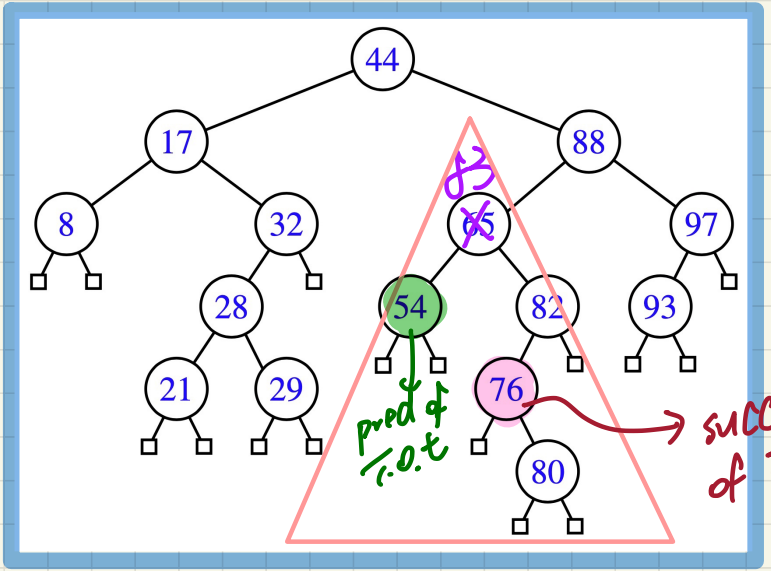
Remember: For a BT to be a BST, the Search Property should hold recursively on the root of each subtree.

In-Order: <8, 17, 21, 28, 29, 32, 44, 54, 65, 76, 80, 82, 88, 93, 97>

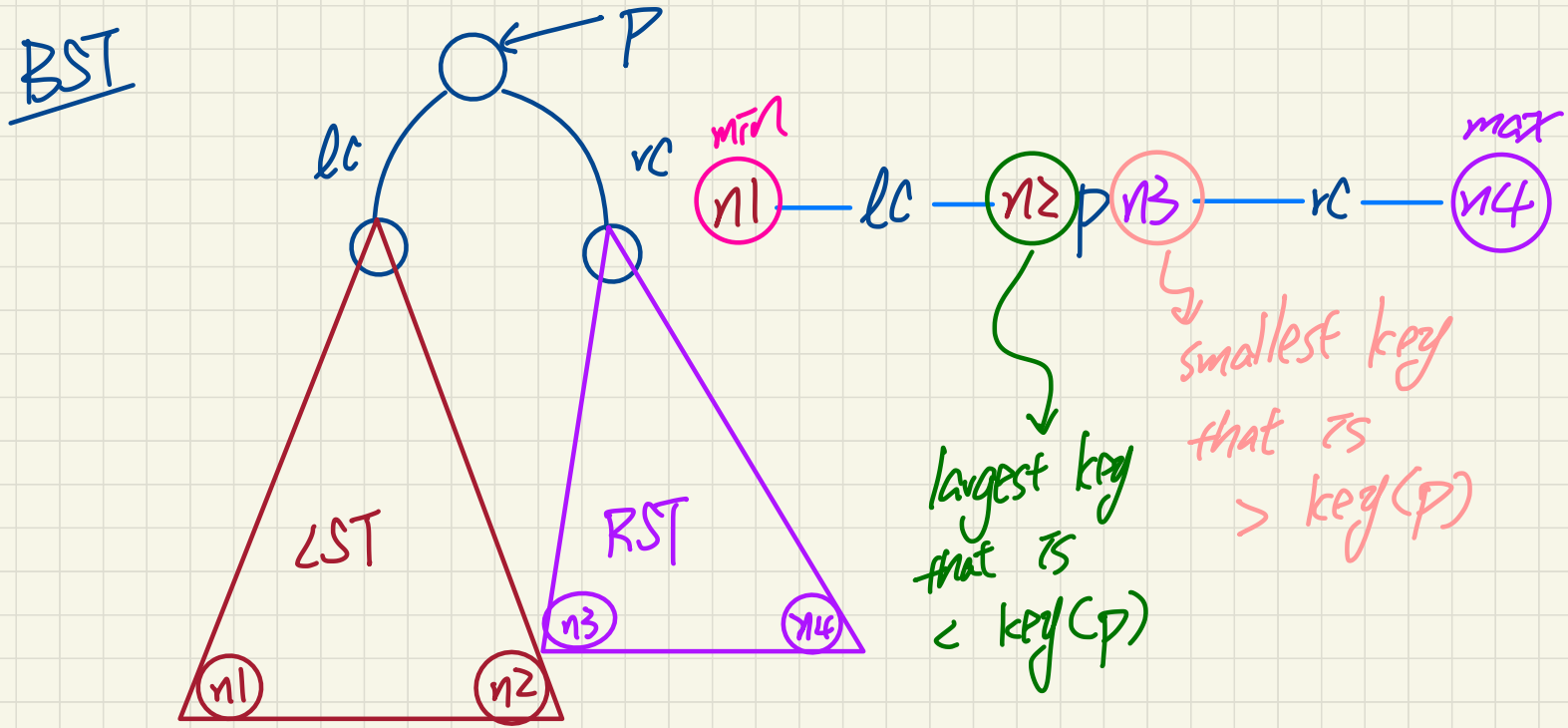
pred. 65 succ.

54 < 65

65 ≮ 76 false



Visual Summary: In-Order Traversal on BST



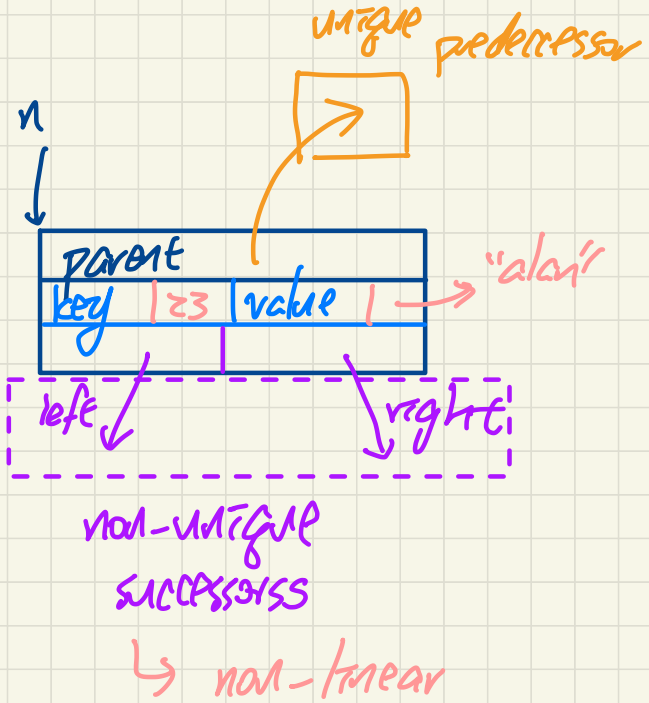
Generic, Binary Tree Nodes

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

Entry

ext. node

int. node.



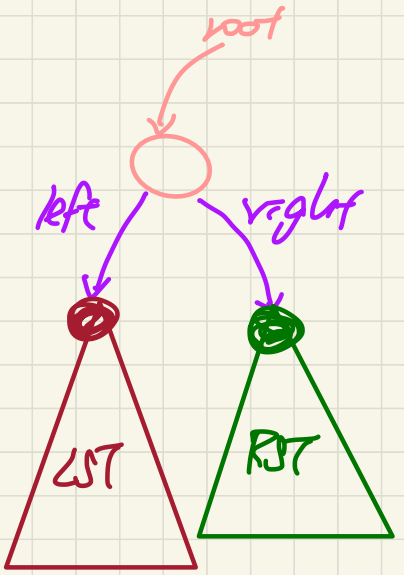
Compare:

+ prev ref.
+ next ref.

in a DLN.

Generic, Binary Tree Nodes - Traversal

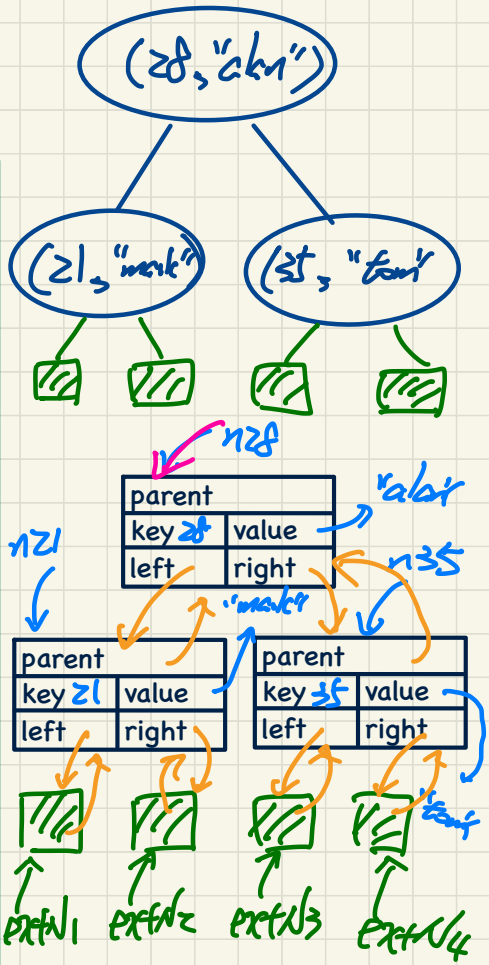
```java
import java.util.ArrayList;
public class BSTUtilities<E> {
  public ArrayList<BSTNode<E>> inOrderTraversal(BSTNode<E> root) {
    ArrayList<BSTNode<E>> result = null;
    if(root.isInternal()) {
      result = new ArrayList<>();

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

      result.add(root);

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



Tracing: Constructing and Traversing a BST

```java
@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());
}
```

sorted in-order seq.



Lecture 20 - April 1

Binary Search Trees, Balanced BSTs

BST: Searching, Insertion
Hight Balance Property
Priority Queue: Introduction

Announcements/Reminders

- Assignment 4 (on linked Trees) released
- Makeup Lecture (for ProgTest2) to be posted
- Bonus opportunity: Final Course Evaluation
- Office hours 3pm Tue/Wed/Thu this week
- Lecture notes template, Office Hours, TA Contact

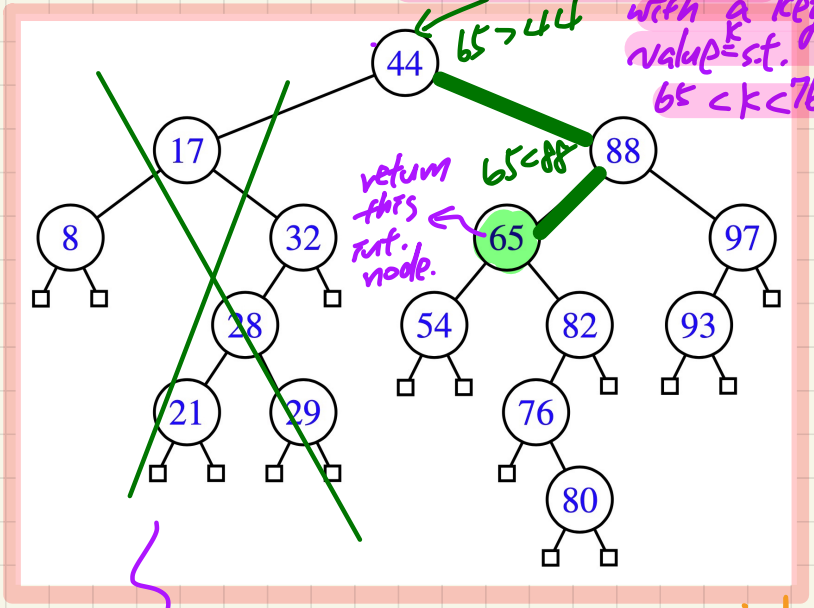
* i.o.t. : 54 65 ext 76 80 82 ...

BST Operation: Searching a Key

① successful
↳ r.n. int. node storing the matching key value
② unsuccessful
↳ rn. ext node

** for a subsequent insertion, this searching ext node can be set with a key value = k s.t. 65 < k < 76

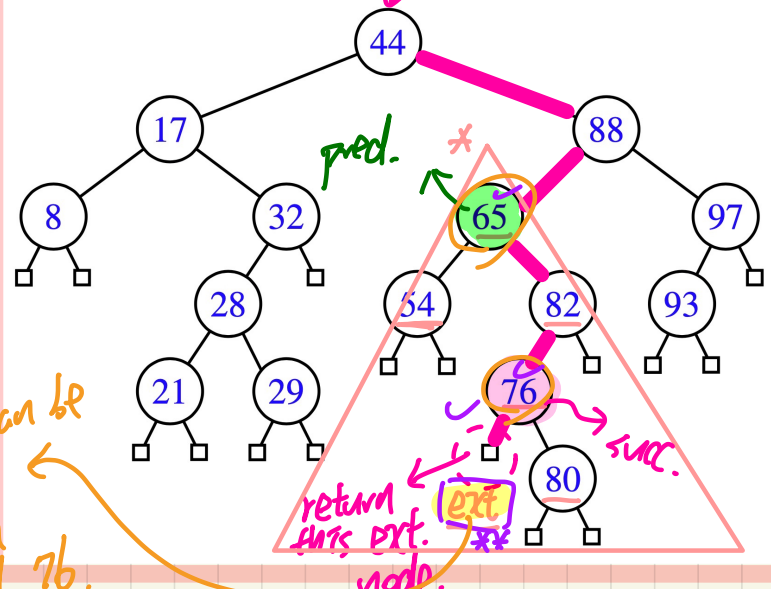
Search key 65 (key)



eliminate this subtree (∵ all keys < 44)
↳ (how large? 1/2 or O(1))

Search key 68

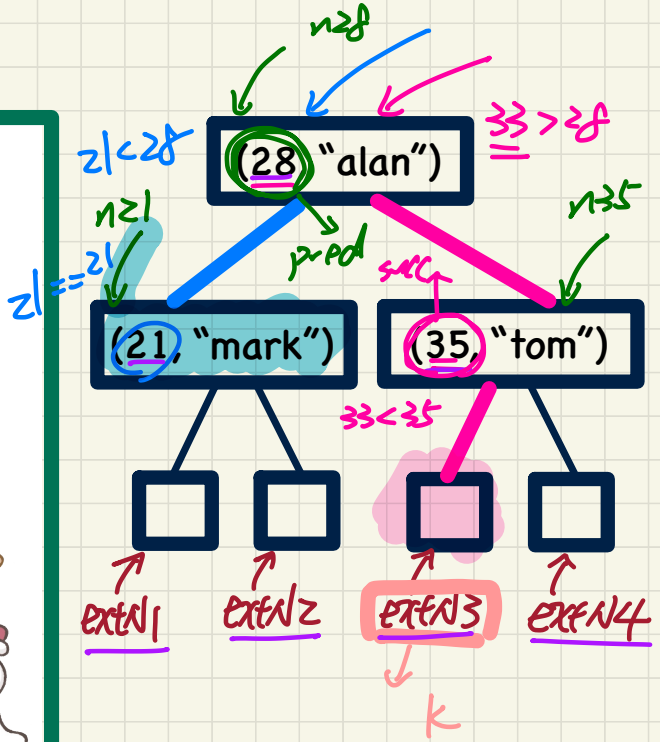
65 < k(ext) < 76



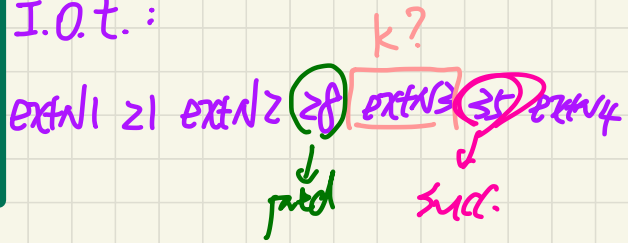
this key can be set between 65 and 76.

Tracing: Searching through a BST

```java
@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* */
}
```



I.O.t.:



* Recall: given N nodes : $\_\_\_ \le h \le \_\_\_$

Running Time: Search on a BST

ext. or int.

```
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);
  }
  else if(k > p.getKey()) {
    result = search(p.getRight(), k);
  }
  return result;
}
```

p: root

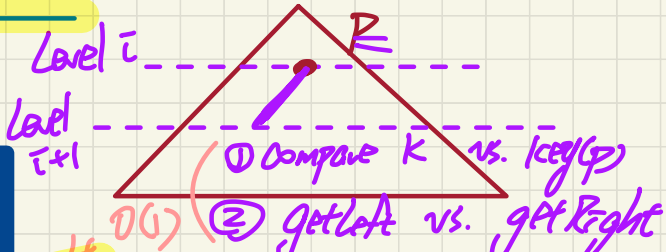
k: key to search.

ext. node

int. node

R1

R2

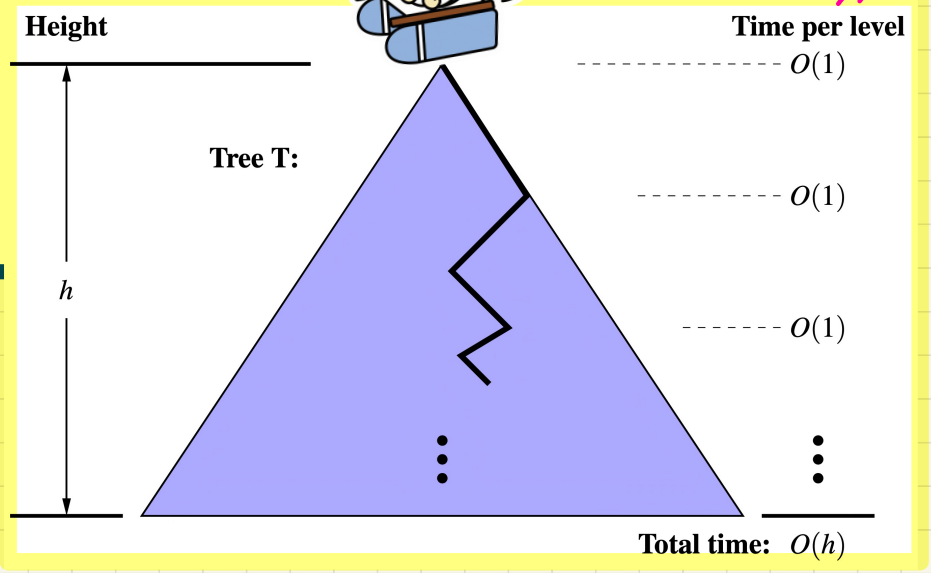


tight, accurate

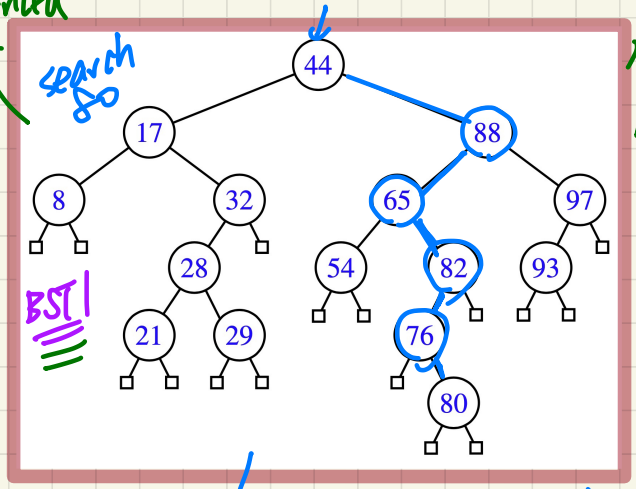
(A1) *O(h)

(A2) O(logN) → best case, lower bound

(A3) O(N) → worst case, upper bound



Binary Search: Non-Linear vs. Linear Structures



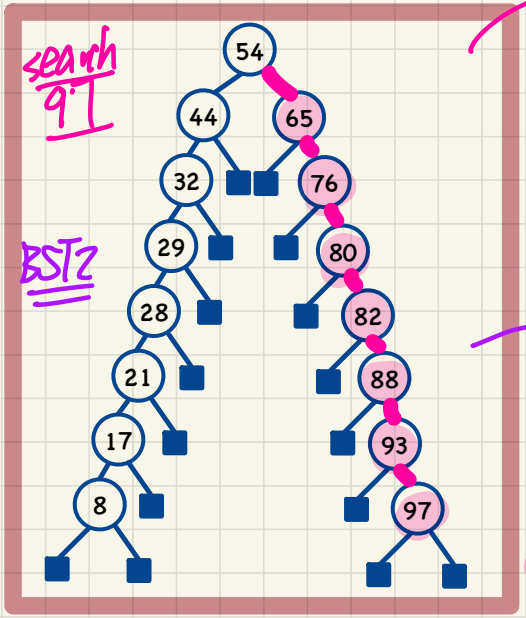
$N = 15$

$h = 5$

$\approx O(\log N)$

searching 80 on BST1 $\approx$ performing a binary search on a

T.o.t.(BST1) = T.o.t(BST2) = a



$N = 15$

$h = 7$

$\frac{N}{2}$ $O(N)$

searching 97 on BST2 $\approx$ performing a linear search on a

a

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
8	17	21	28	29	32	44	54	65	76	80	82	88	93	97

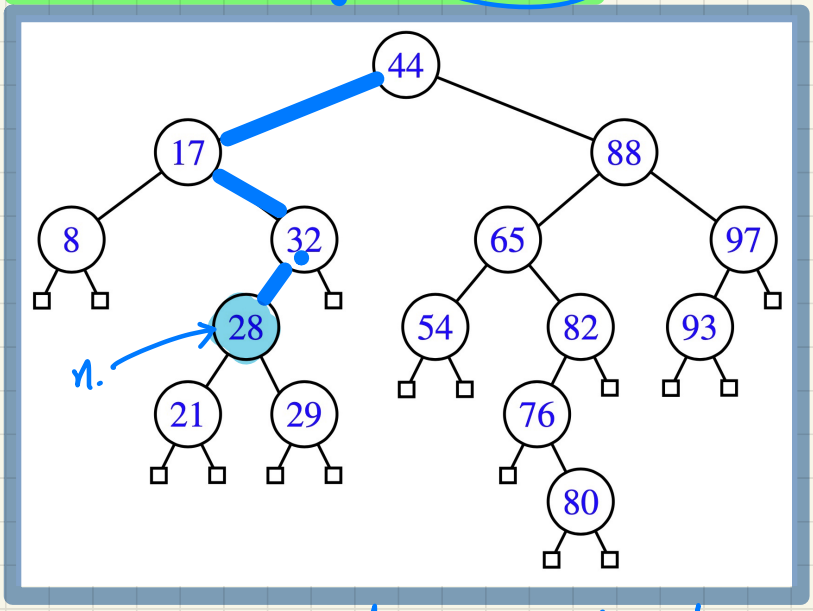
REVIEW ! a

Visualizing BST Operation: Insertion

RT: dominated by O(h) searching →

Insert Entry (28, "suyeon")

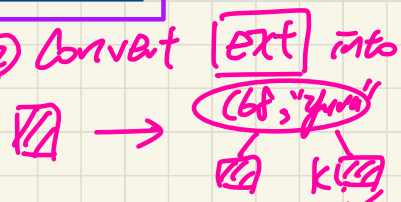
k. v.



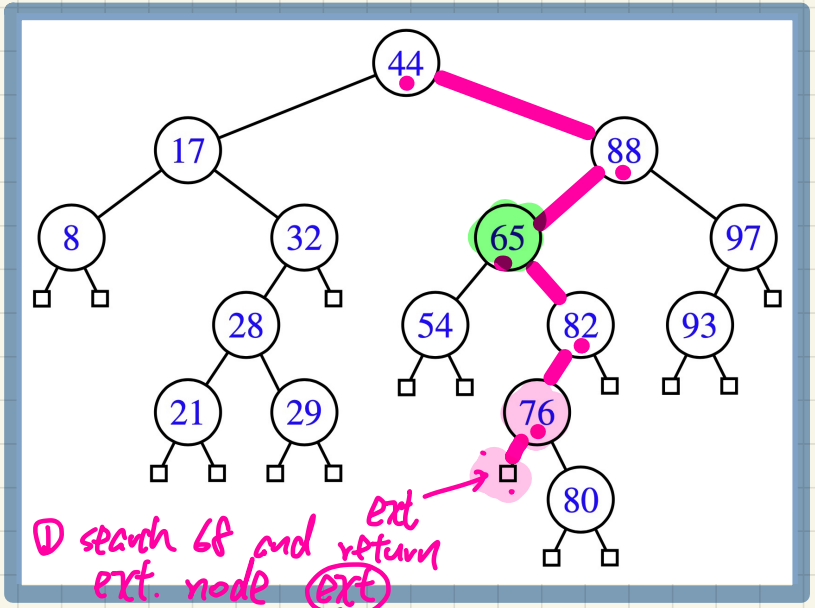
① search 28 and return int. node n

② n.setElement("suyeon");

② convert ext into an int. node.



Insert Entry (68, "yuna")

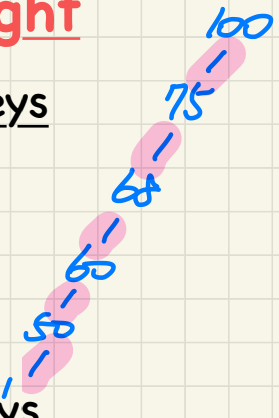


① search 68 and return ext. node ext

Worst-Case RT: BST with Linear Height

Example 1: Inserted Entries with Decreasing Keys

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



Example 2: Inserted Entries with Increasing Keys

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

BST with O(N) height
⇒ search/insert/delete can be O(N)

exercises (heights? N? log N?)

Example 3: Inserted Entries with In-Between Keys

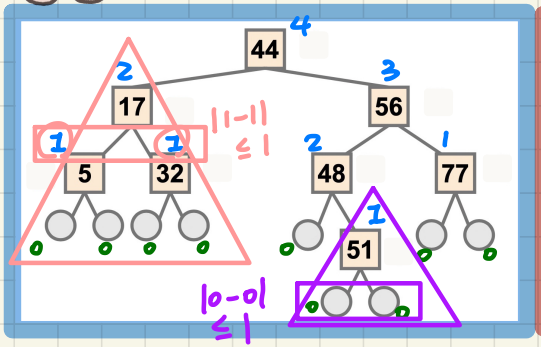
<1, 100, 50, 75, 60, 68>

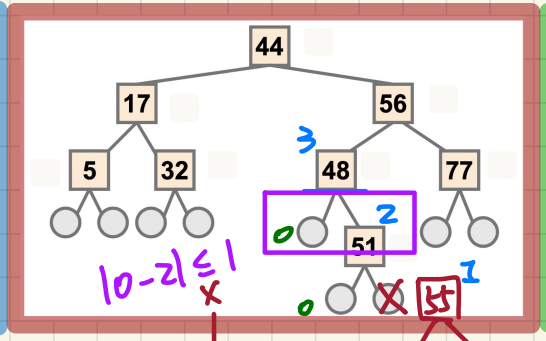
Balanced BST: Definition

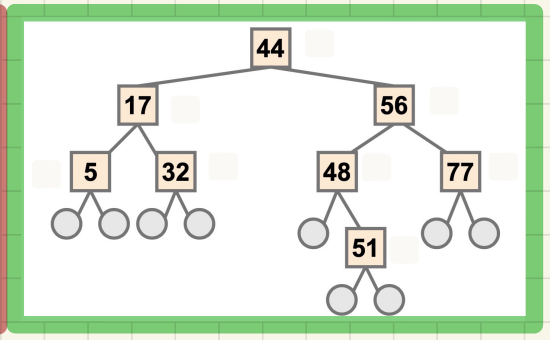
- internal node
- height
- height balance

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

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





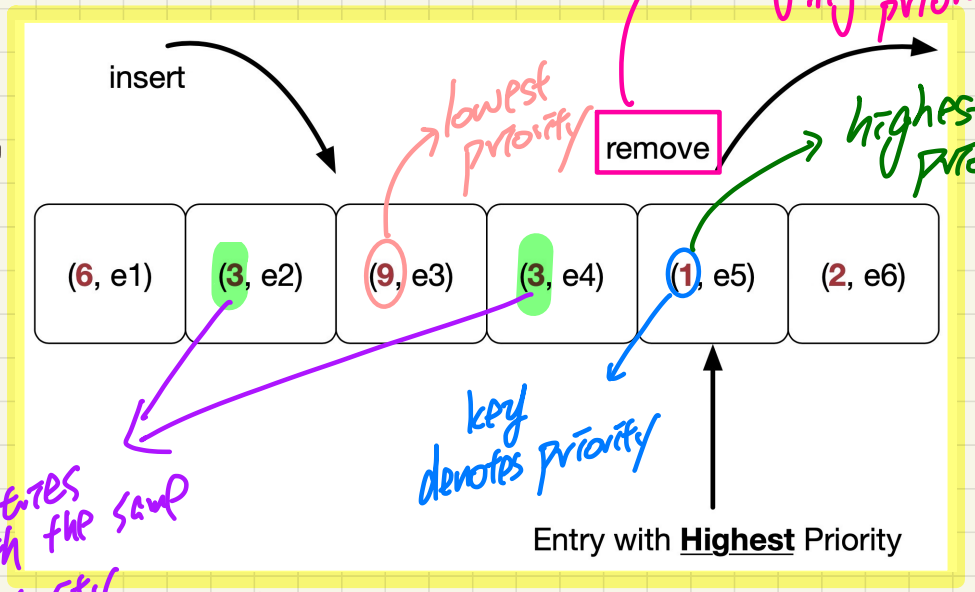


Q. Is the above tree a balanced BST?

Q. Still a balanced BST after inserting 55?

Q. Still a balanced BST after inserting 63?

What is a Priority Queue (PQ)



remove entry with highest priority

k ↓ priority ↑

lowest priority

highest priority

key denotes priority

Entries with the same priority (does not matter which entry is retrieved)

Compare PQ with FIFO queue

1. entries removed from PQ according to priorities.
2. entries removed from FIFO acc. to chronological order

Lecture 21 - April 3

Priority Queues, Heap, Heap Sort

PQ: List Implementations
Heap: Structure, Relational Properties
Heap: Insertion, Deletion
Heap Sort

Announcements/Reminders

- Assignment 4 (on linked Trees) released
- Makeup Lecture (for ProgTest2) posted
- Bonus opportunity: Final Course Evaluation
- Office hours 3pm Thu this week
- Office hours, review session, ex. questions to be releasd
- Lecture notes template, Office Hours, TA Contact

List-Based Implementations of Priority Queue (PQ)

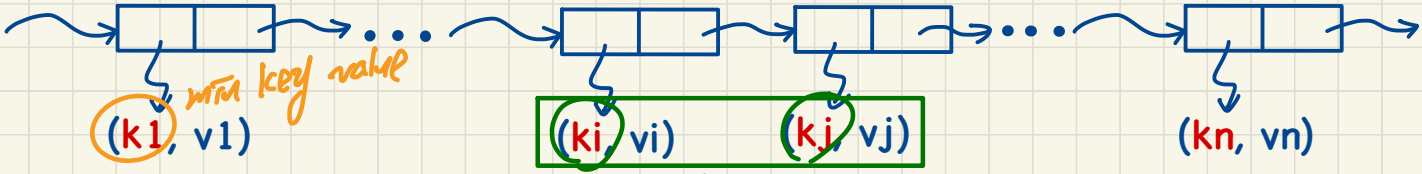
PQ Method	List Method	
	SORTED LIST	UNSORTED LIST
size	list.size O(1)	
isEmpty	list.isEmpty O(1)	
min	list.first O(1)	search min O(N)
insert	insert to "right" spot O(N)	insert to front O(1)
removeMin	list.removeFirst O(1)	search min and remove O(N)

existing space for true

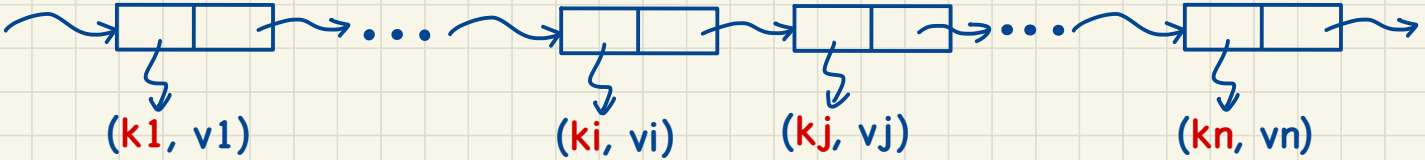
(2) infrequent expansion to P.Q.

Approach 1: Sorted List

more suitable: (1) frequent retrieval/removal of top-prior. entries



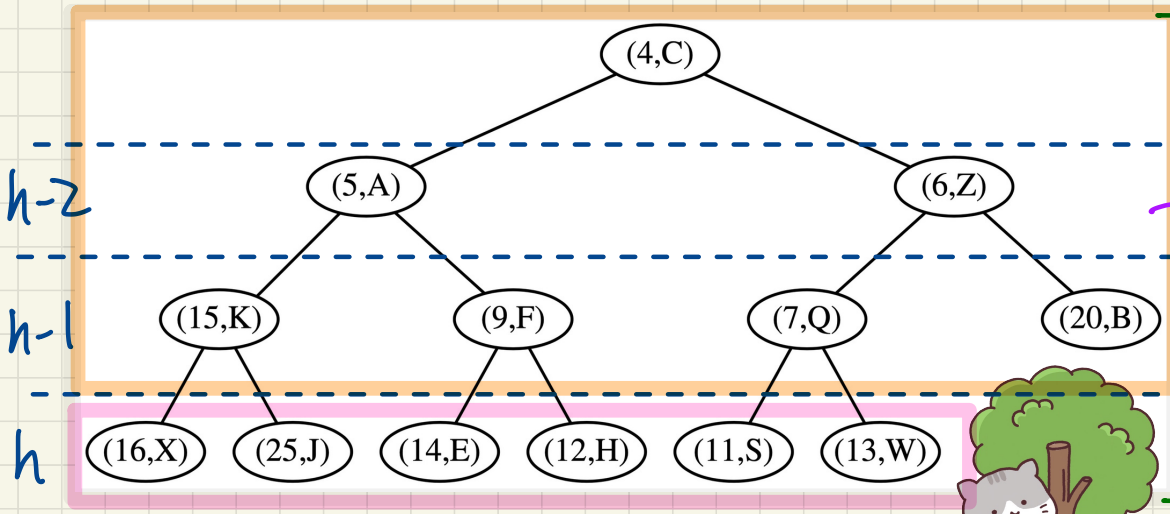
Approach 2: Unsorted List



Heaps: Structural Properties of Nodes

→ height: $\log N$.

Property: The tree is a complete Binary Tree



$h = \lfloor \log N \rfloor$

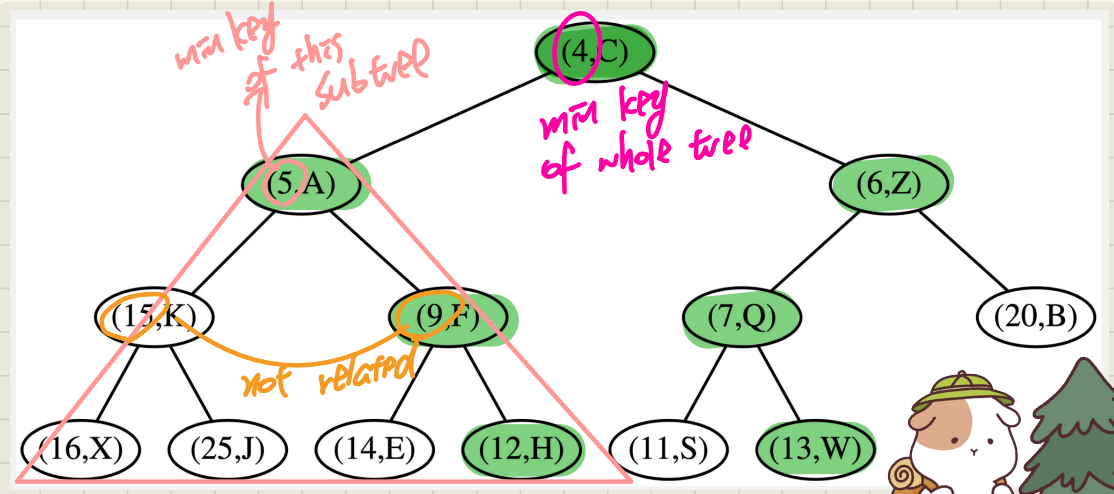
→ # nodes from levels 0 to $h-1$: $2^h - 1$

Min # Nodes: $(2^h - 1) + 1$ 2^h

Max # Nodes: $(2^h - 1) + 2^h$ $2^{h+1} - 1$

Heaps: Relational Properties of Keys

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



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

P2. min key exists in the root

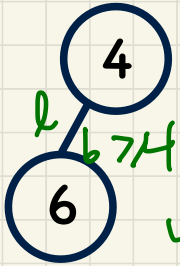
P3. keys between LST and RST are not related.

Example Heaps

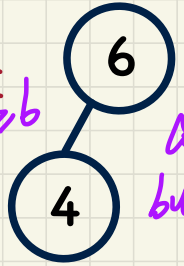
Example 1

smallest possible one-node heap

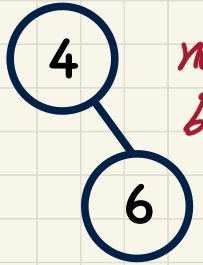
Example 2



Example 3

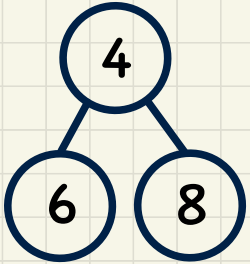


Example 4

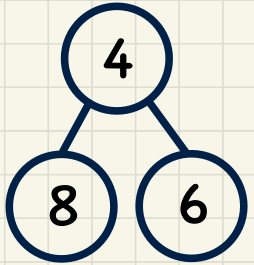


not complete but sat. HOP

Example 5



Example 6



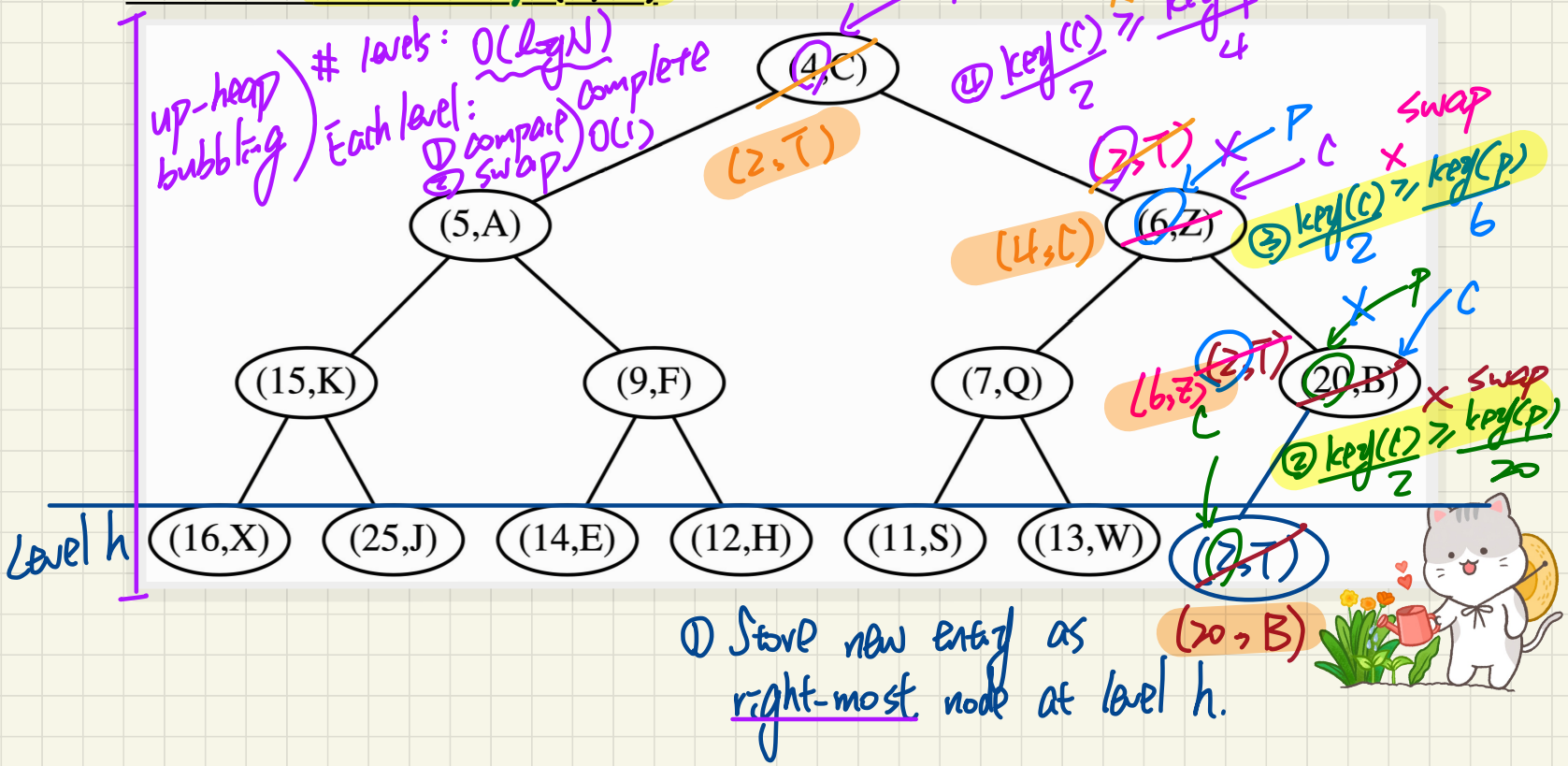
heaps!

full BT

⇒ complete BT

Heap Operations: Insertion

Insert a new entry (2, T)



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

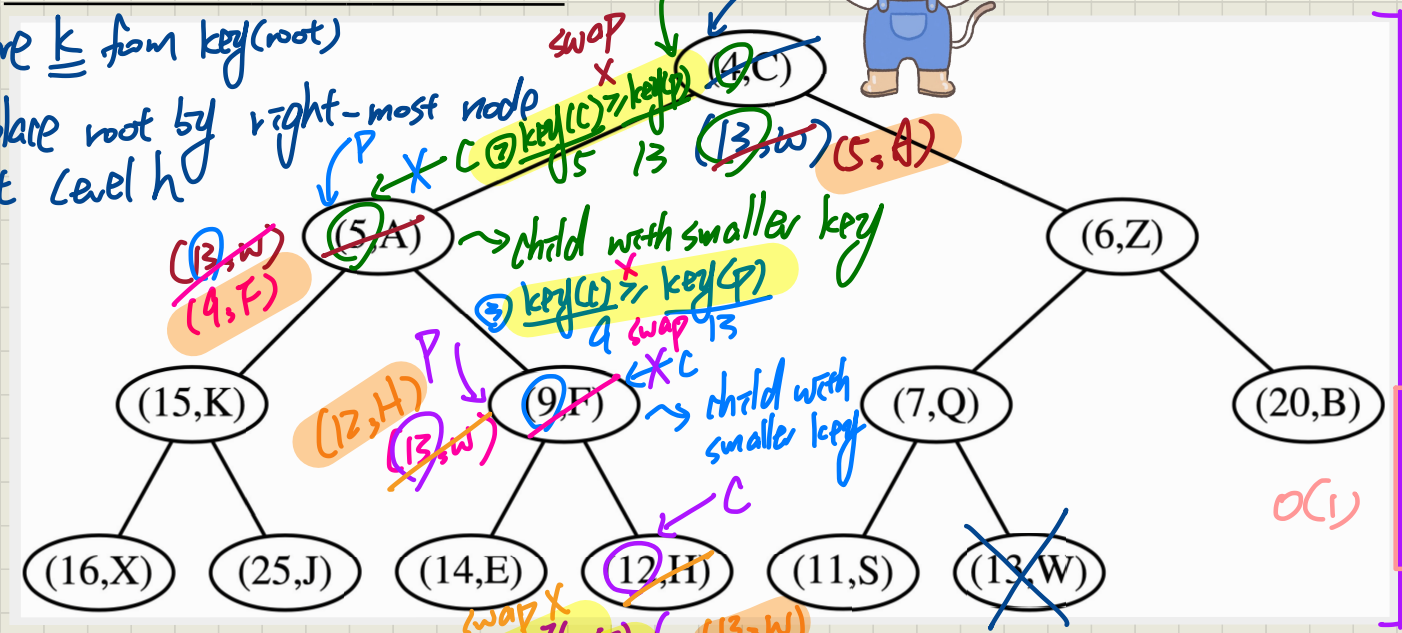
Heap Operations: Deletion

entry with min key (root)

Delete the root/minimum

① store k from key(root)

Replace root by right-most node at level h



down-heap bubbling.

levels: O(log N)

Each level:
① choose c
② compare
③ swap

O(1)

⑤ return k. O(1 · log N)

deletion ← O(log N)

Heap Sort: Ideas

$O(N \cdot \log N)$

construct a heap out of N entries

(A) Top-Down

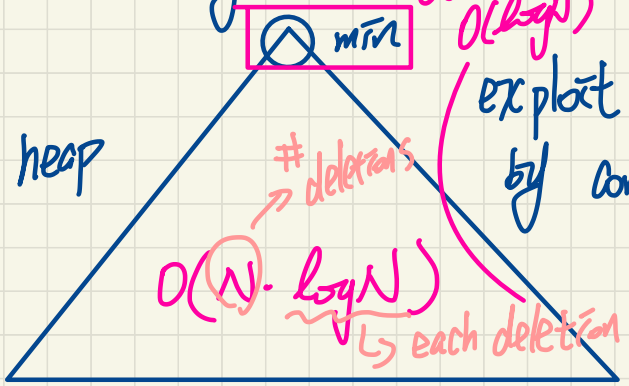
$O(N \cdot \log N)$

(B) Bottom-Up

$O(N)$

$\sim$ selection sort

↳ selection: $O(N)$



extract: $O(\log N)$

exploit the Hop (relational) by continuing to delete/extract min/root until trap is empty.

Exam

- ~ 1~2 review sessions
- ~ PDF guide
- ~ question booklet (answer booklets)
  - ↳ no calculator
  - ↳ little to none multiple choice Qs.
  - ↳ definitions
  - ↳ short answers (explanations, justifications)
  - ↳ coding/tracing
  - ↳ proofs (e.g., asymptotic U.B., trees)

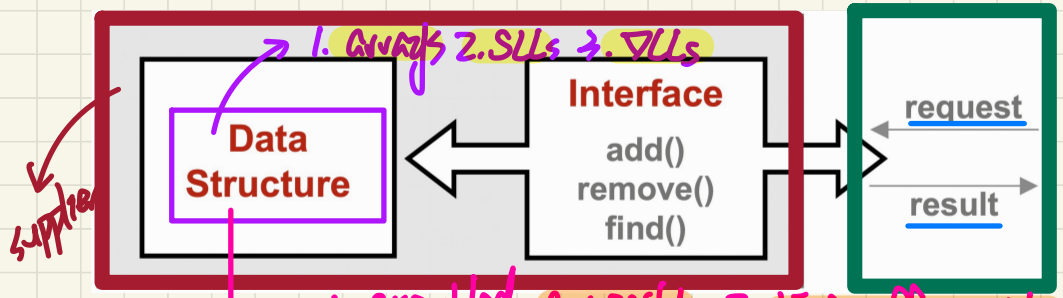
~ Jeff
Sunila
Jadoo

- ~ slides/iPad notes
- ~ example code
  - ↳ BST/Node
- ~ assignments

Makeup Lecture (ProgTest1)

ADTs, Stacks

Abstract Data Types (ADTs)



client relies on the public interface of ADT()
1. input types
2. output types
3. what to be expected on IO related.

```
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);
  } }
```

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

Java API ≈ Abstract Data Types

NT is subject to ambiguities & contradictions.

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())

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

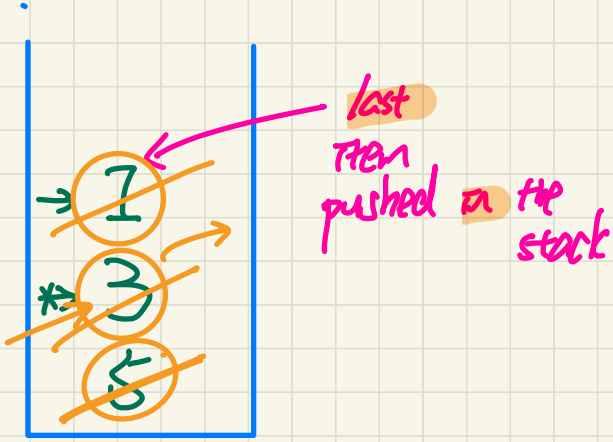
---

```
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.

Stack ADT: Illustration

	isEmpty	size	top
new stack	T	0	n.a.
push(5)	F	1	5
push(3)	F	2	3
push(1)	F	3	1
pop rtn. 1	F	2	3
pop rtn. 3	F	1	5
pop rtn. 5	T	0	n.a.



last pushed element

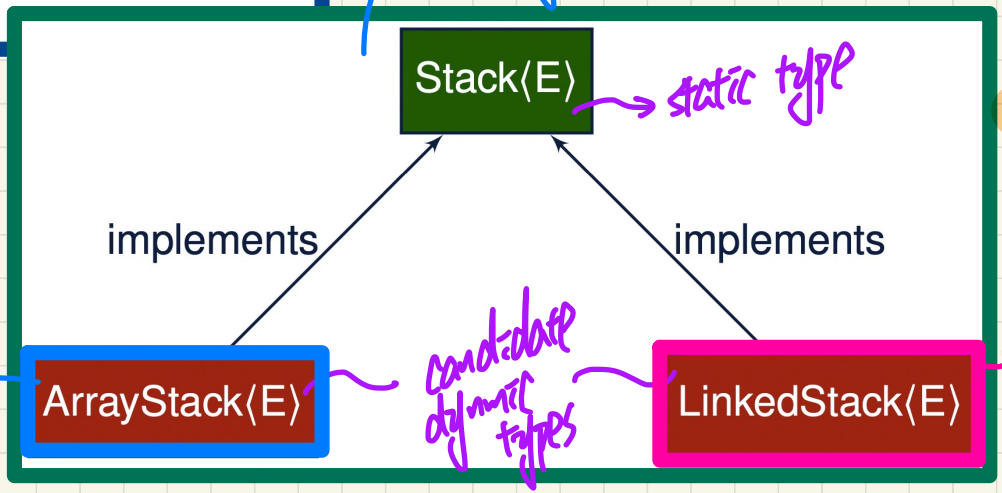
2nd last pushed element

(LIFO)

The order in which items are popped off the stack is the reverse of how these items were pushed.

Implementing the Stack ADT in Java: Architecture

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



Implementing the Stack ADT using an Array

```
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); }                    O(1)
  public boolean isEmpty() { return (t == -1); }           O(1)

  public E top() {                                         O(1)
    if (isEmpty()) { /* Precondition Violated */ }
    else { return data[t]; }
  }
  public void push(E e) {                                  O(1)
    if (size() == MAX_CAPACITY) { /* Precondition Violated */ }
    else { t ++; data[t] = e; }
  }
  public E pop() {                                         O(1)
    E result;
    if (isEmpty()) { /* Precondition Violated */ }
    else { result = data[t]; data[t] = null; t --; }
    return result;
  }
}
```

limitation: fixed size

ArrayStack<String>

↳ instantiates E for Stack:

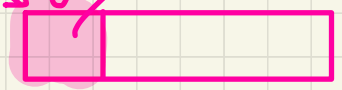
Stack<String>

(E[]) Object[____]

↳ what you have to write in Java.

efficient top. of stack

goal: treat this first item as the top.



Implementing the Stack ADT using a SLL

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

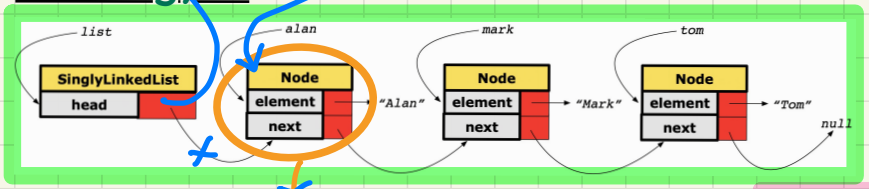
Stack Method	Singly-Linked List Method	
	Strategy 1	Strategy 2
size	list.size	
isEmpty	list.isEmpty	
top	list.first ✓	list.last
push	list.addFirst	list.addLast
pop	list.removeFirst	list.removeLast

Improved to O(1) if a DLL is used.

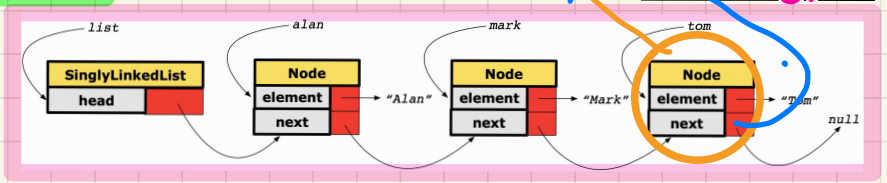
O(1)

O(1)

Strategy 1



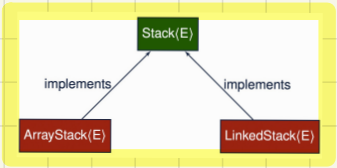
Strategy 2



Stack ADT: Testing Alternative Implementations

Stack<S> s = new Stack<>();

✗ interface can't be a DT.




```
public class ArrayStack<E> implements Stack<E> {
  private final int MAX_CAPACITY = 1000;
  private E[] data;
  private 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;
  }
}
```

static type

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

dynamic type

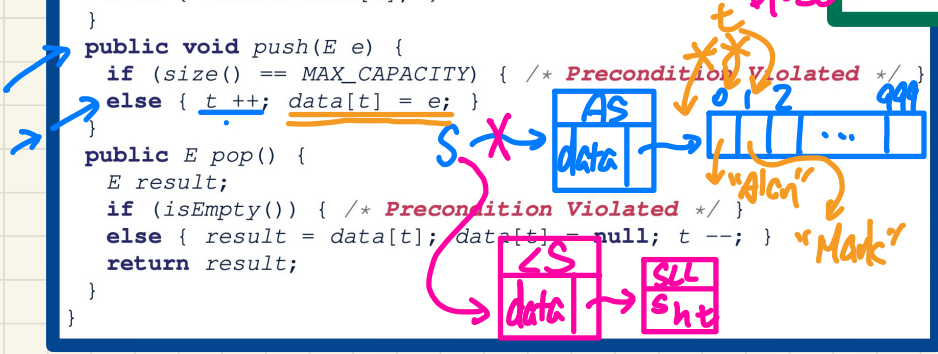
DT: AS.

version in AS.

DT changes

DT: LS

version in LS. class of Stack?



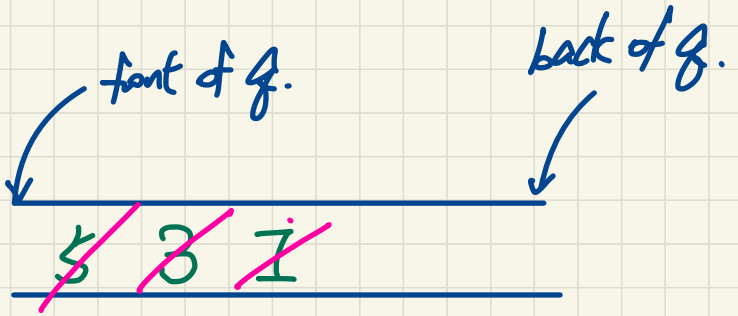
Is the DT of S a descendant of:		*	**
S	instanceof Stack	T	T
S	instanceof ArrayStack	T	F
S	instanceof LinkedStack	F	T

Makeup Lecture (WrittenTest)

Queues, Circular Arrays, Deque

Queue ADT: Illustration

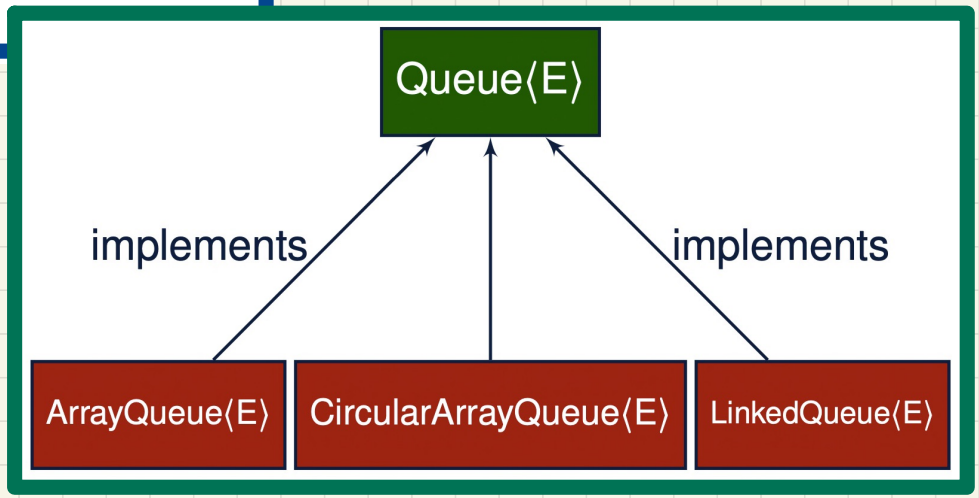
	isEmpty	size	first
new queue	T	0	n.a.
enqueue(5)	F	1	5
enqueue(3)	F	2	5
enqueue(1)	F	3	5
dequeue rtn. 5	F	2	3
dequeue rtn. 3	F	1	1
dequeue rtn. 1	T	0	n.a.



→ First-In First-Out (FIFO)

Implementing the Queue ADT in Java: Architecture

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



Implementing the Queue ADT using an Array

```
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;
    }
  }
}
```

O(1)

O(1)

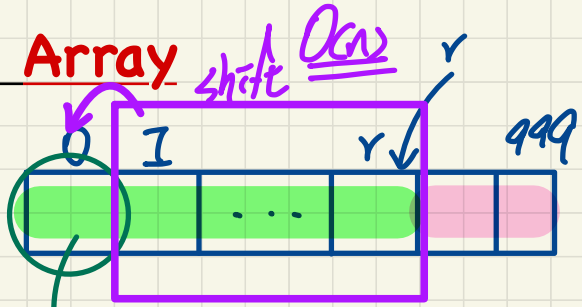
O(1)

front index

O(1)

O(n)

shifting "2nd item" and onwards to the left by one position



front of queue

limitation: no resizing.

to improve this, we need to be flexible about where the front index is ⇒ Circular array.

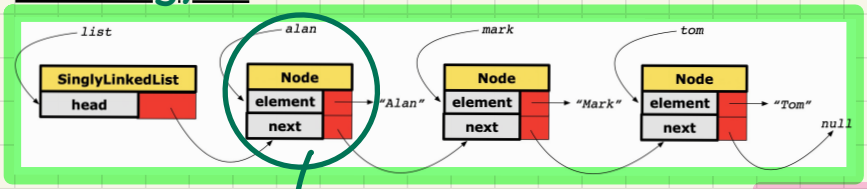
Implementing the Queue ADT using a SLL

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

O(n)
① use SI instead
② use DLL instead

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

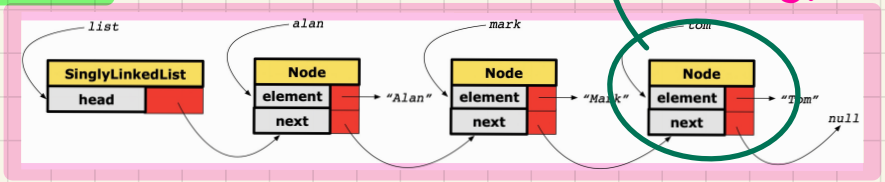
Strategy 1



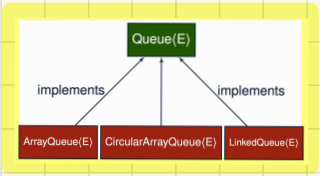
first of queue

first of queue.

Strategy 2



Stack ADT: Testing Alternative Implementations




```
public class ArrayQueue<E> implements Queue<E> {
  private final int MAX_CAPACITY = 1000;
  private E[] data;
  private int r = -1; /* 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;
    }
  }
}
```

dynamic binding.

```
@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

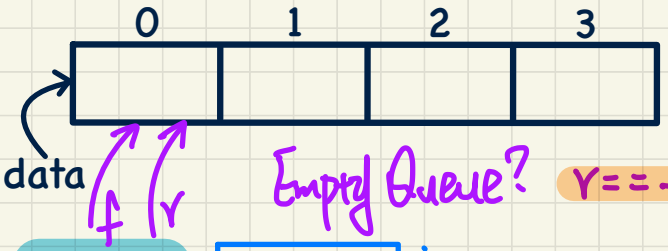
size of Queue vs. size of array | % modulo

Implementing the Queue ADT using a Circular Array

1. fix-sized (no resizing)
2. flexible for performing "dequeue"

Assume: A circular array of length 4.

Phase 0: Empty Queue q



f=0;
r=0;

Empty Queue? r==f

Phase 1: enqueue 3 elements

- data[r] = item;
- r++;

Queue Full? (r+1)%N == f
when r points to is the only empty slot.

(r+1)%4 = 3

(r+1)%4 = 4 = 0

[f, r-1] = (r-1)-f+1 = r-f

empty slots before f | # slots before this index

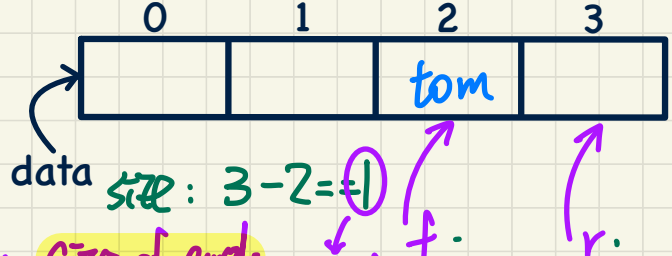
size: r>=f → r-f

3 = 1+(4-2)

size: r<f → r+(N-f)

Phase 2: dequeue 2 times

data[f]=null; f++;



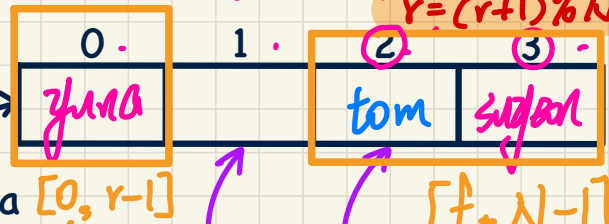
size: 3-2=1

size of array: 4 (fixed) → size of 4.

(3+1)%4 = 0

Phase 3: enqueue 2 elements

data[r] = item; ~~r++;~~ r=(r+1)%N



[0, r-1] → (r-1)-0+1 = r

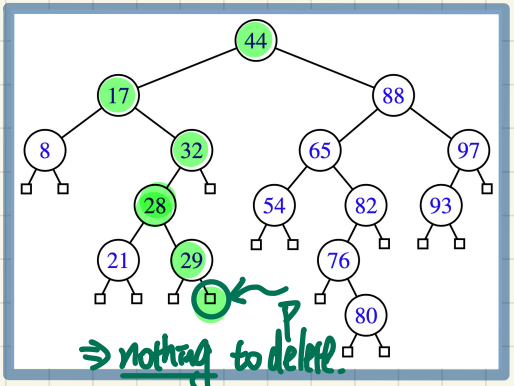
[f, N-1] → (N-1)-f+1 = N-f

Makeup Lecture (ProgTest2)

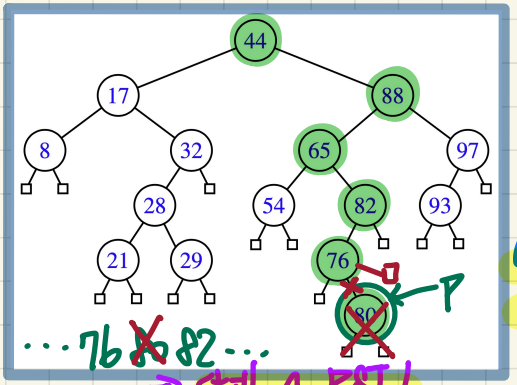
BST Deletions, Constructing Heap, Array-Based Implementations of BTs

Visualizing BST Operation: Deletion

Case 1: Delete Entry with Key 31

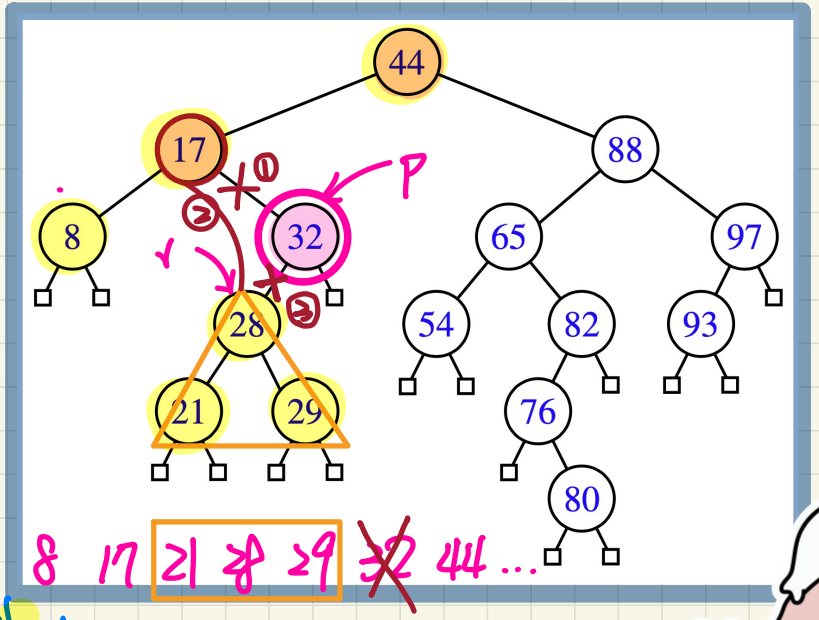


Case 2: Delete Entry with Key 80



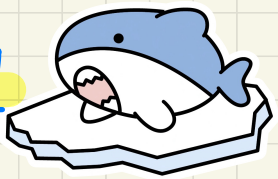
Orphan subtree: a subtree rooted at an internal child node

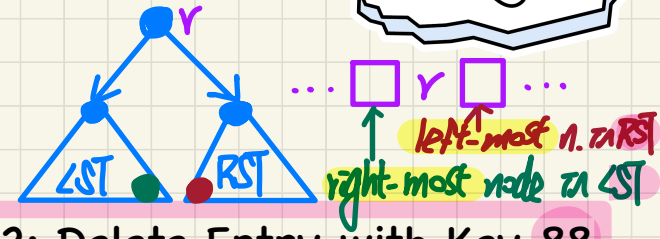
Case 3: Delete Entry with Key 32



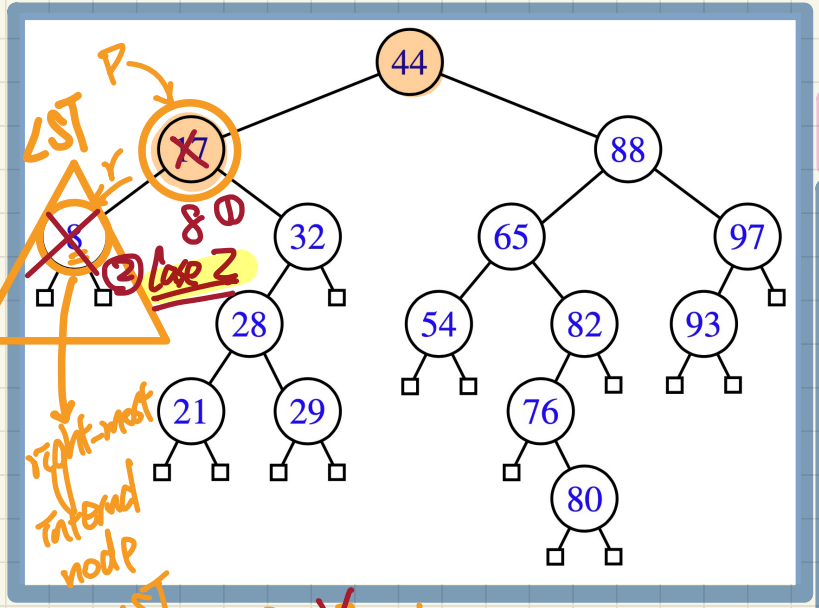
Visualizing BST Operation: Deletion

In-Order Traversal

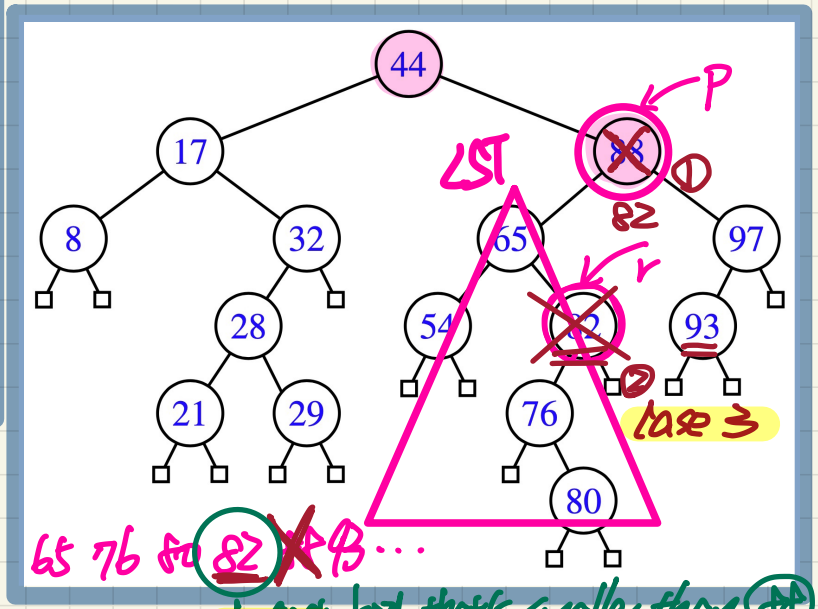




Case 4.1: Delete Entry with Key 17



Case 4.2: Delete Entry with Key 88



Top-Down Heap Construction

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

- Initialize an ***empty heap h***.
- As each new entry **e** = (*k*, *v*) is supplied, **insert** **e** into ***h***.

Exercise: Build a heap out of the following 15 keys:

<16, 15, 4, 12, 6, 7, 23, 20, 25, 9, 11, 17, 5, 8, 14>

first inserted to level 2

Assumption: Key values supplied **one at a time**.

first inserted to level 1

$* \le 1 + \log_2 n \cdot (2^1 + 2^2 + \dots + 2^h)$

$= 1 + \log_2 n \,(n-1)$

$O(n \cdot \log n)$

RI: # nodes at level i # up-heap bubbling steps

$1 + 2^1 \cdot 1 \le \log n$

$+ 2^2 \cdot 2 \le \log n$

$+ \dots$

$+ 2^h \cdot h \log n$

$2^0 = 1$ Level 0

$2^1 = 2$ Level 1

$2^2 = 4$ Level 2

2^h Level h

Exercise: Complete inserting the remaining keys to the heap.

Bottom-Up Heap Construction

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*** $h \geq 1$ $\quad [\, h = (\log(N + 1)) - 1 \,]$
- 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.

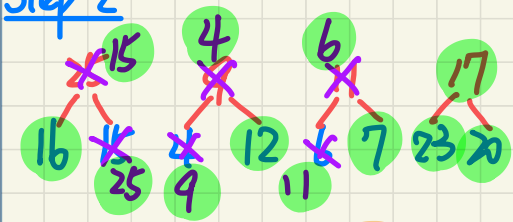
  $\therefore \frac{N+1}{2^{h+1}} = 1$ heap, each with height h and size $2^{h+1} - 1$ constructed.

Exercise: Build a heap out of the following 15 keys:
<16, 15, 4, 12, 6, 7, 23, 20, 25, 9, 11, 17, 5, 8, 14>

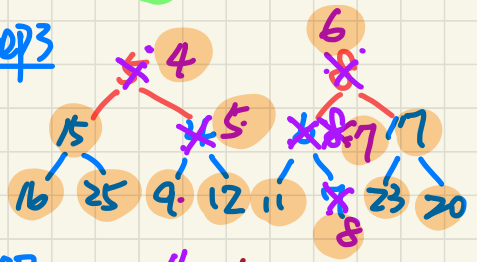
Assumption: Key values supplied **all at once**.

50% Step 1
8 heaps, size 1, height 0
16 15 4 12 6 7 23 20

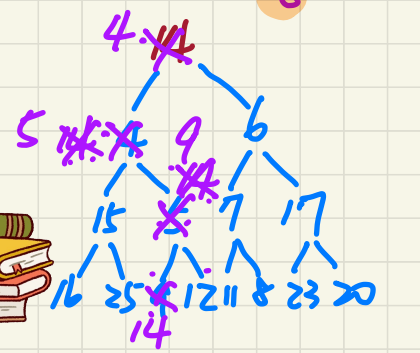
Step 2
25%



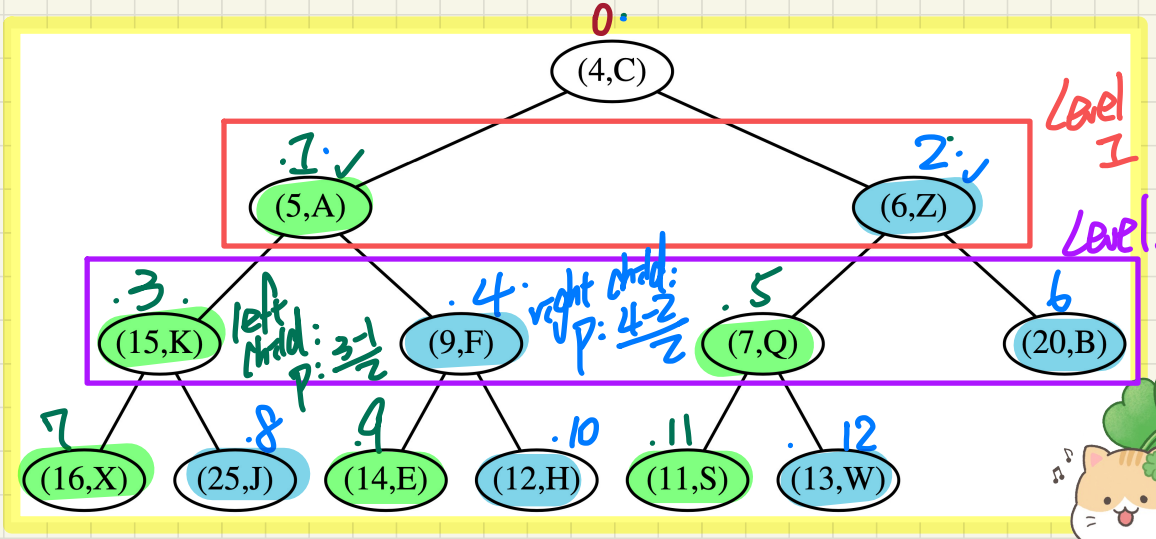
Step 3
12.5%



Step

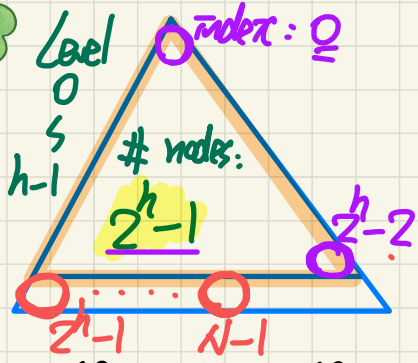


Array-Based Representation of a Complete BT



Exercise

What if the BT is not complete? (bad for space util.)



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

0	1	2	3	4	5	6	7	8	9	10	11	12
(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)

I hope you enjoyed learning with me 🤗

All the best to you !