EECS2011 Fundamentals of Data Structures

Lecture Notes

Winter 2023

Jackie Wang

Lecture 1 - Monday, January 9

Lecture

Solving Problems via Data Structures

Searching

Assignments/ProgTests

starter project

- tests
  - ↳ Tests.java
- model
  - ↓ empty

→ when grading your PT submissions:

1) Compilation
2) passing starter tests
3) additional grading tests

give you some initial idea about how methods should be:

(1) declared

(2) implemented

only supply example scenarios (potentially incomplete)

↳ you are expected to:

1) do not hard-code your methods "just for" the given tests
2) write additional JUnit tests to cover missing scenarios.

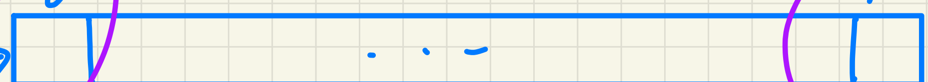
A Searching Problem

Efficient Solution

balanced binary search tree

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

0 $\qquad$ $10^7 - 1$



database

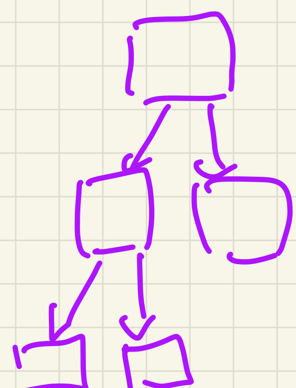
best case
1 iteration

worst case
10^7 iterations

\# residents

not efficient

n.



Lecture 2 - Wednesday, January 11

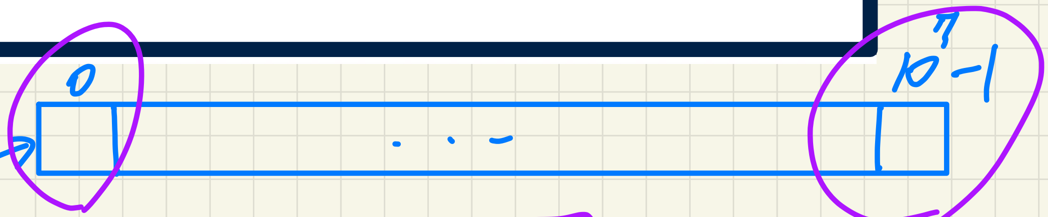
Lecture

Solving Problems via Data Structures

Routing & Compiler

A Searching Problem

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



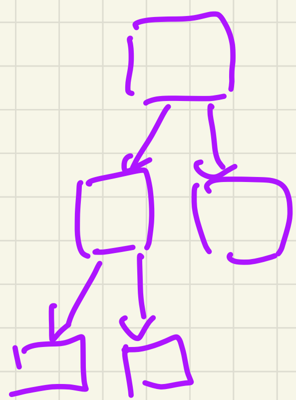
database

best case
1 iteration

worst case
10^7 iterations
\# residents
n
not efficient

Efficient Solution

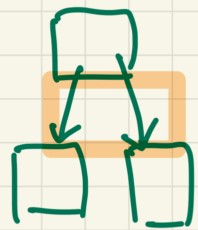
balanced binary search tree



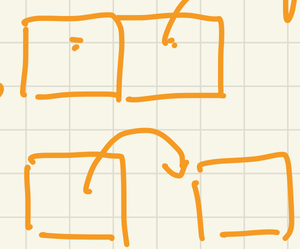
Balance Binary Search ✓ Tree vs. ~~Array~~

linear

guarantees height of tree : $\log_2 N$



multiple successors

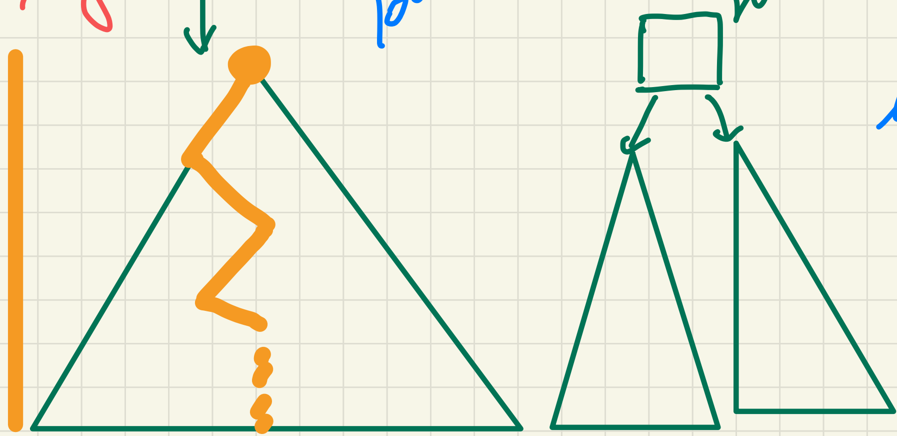


single → unique successor

$1000 \approx 2^{10}$

root

height



$\log_2 \boxed{10^7} = \log_2 (10^3)^{2.333}$

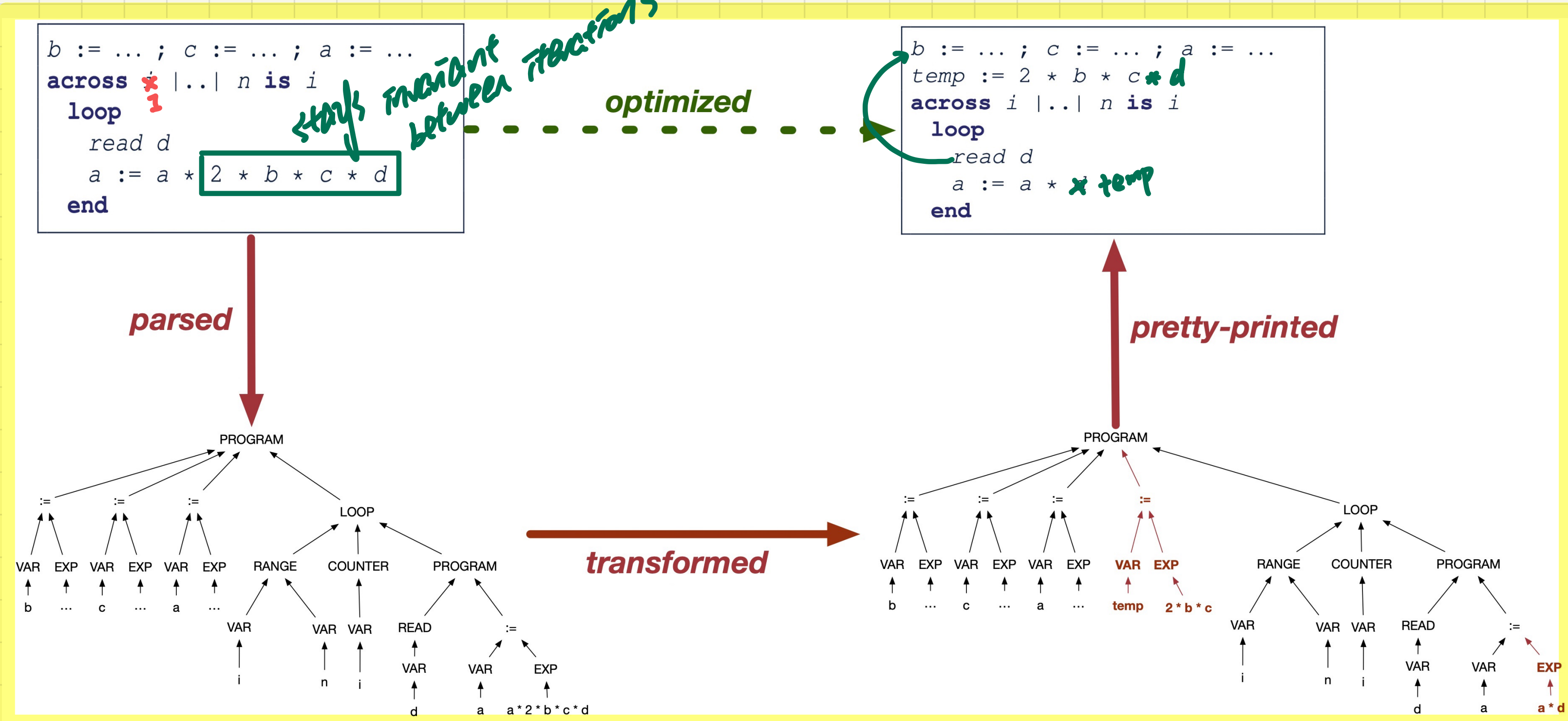
residents in city.

$\approx 2^{10}$

$\log_2 2^{23.3}$

Program Optimization Problem

EECS 4302 Compilers



Program Translation Problem

```
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

object-relational bridge

Lecture

Reviews on Recursion

Principle, Implementation, Tracing

Solving a Problem **Recursively**

each subproblem strictly smaller; otherwise infinite recursion

Given a **small** problem:

Solve it **directly**:

Given a **big** problem:

make a recursive call on each subproblem

Divide it into **smaller** problems:

Assume solutions to **smaller** problems:

Combine solutions to **smaller** problems:

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

Tracing **Recursion** via a **Stack**

- When a method is called, it is ***activated*** (and becomes *active*) and *pushed* onto the stack.
- When the body of a method makes a (helper) method call, that (helper) method is ***activated*** (and becomes *active*) and *pushed* onto the stack.

  ⇒ The stack contains activation records of all *active* methods.
  - *Top* of stack denotes the current point of execution.
  - Remaining parts of stack are (temporarily) ***suspended***.
- When entire body of a method is executed, stack is *popped*.

  ⇒ The current point of execution is returned to the new *top* of stack (which was ***suspended*** and just became ***active***).
- Execution terminates when the stack becomes *empty*.

method returns

method call

Runtime Stack

Recursive Solution: Fibonacci Numbers

0 1 2

$F = \boxed{1, 1}, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$

base cases

F_0
F_1

recursive cases

$\underline{\underline{F_4}} + \underline{\underline{F_5}} = \underline{\underline{F_6}}$

$F_n = F_{n-1} + F_{n-2}$

$n > 1$

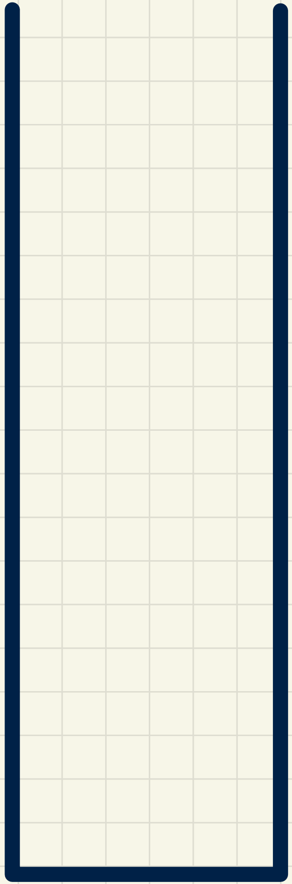
Recursive Solution in Java: Fibonacci Numbers

$$F_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ F_{n-1} + F_{n-2} & \text{if } n > 2 \end{cases}$$

```java
int fib (int n) {
  int result;
  if(n == 1) { /* base case */ result = 1; }
  else if(n == 2) { /* base case */ result = 1; }
  else { /* recursive case */
    result = fib (n - 1) + fib (n - 2);
  }
  return result;
}
```

Example: fib(4)

Exercise: Trace fib(4) via a call stack.



Runtime Stack

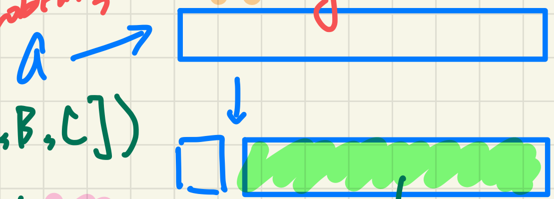
Recursion on an Array: Passing new Sub-Arrays

```java
void m(int[] a) {
  if(a.length == 0) { /* base case */ }
  else if(a.length == 1) { /* base case */ }
  else {
    int[] sub = new int[a.length - 1];
    for(int i = 1; i < a.length; i ++) { sub[i - 1] = a[i]; }
    m(sub) } }
```

efficiency problem, use call by value

Say a1 = {}, consider m(a1)

to resolve space



m([A, B, C])

Say a2 = {A, B, C}, consider m(a2)

m([B, C])

m([C])

sub problem

↳ subject to RCs.

parameter

m(int i)

call by value

argument

int n = 5;

m(n)

5

5

n

5

i

call by value: the parameter i stores a copy of the primitive input the value of n.

Call by value : Reference Type

Person jim = new Person(..);

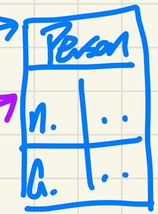
m(Person p)

m(jim)

call by value:
p = jim;

copy the address value of jim to p.

jim



p (alias of the obj).

Recursion on an Array: Passing Same Array Reference

ref type (call by value)

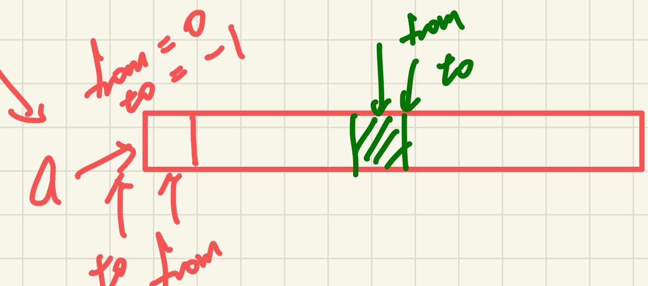
indicating the range of input array that's meant to be examined in the current recursive call.

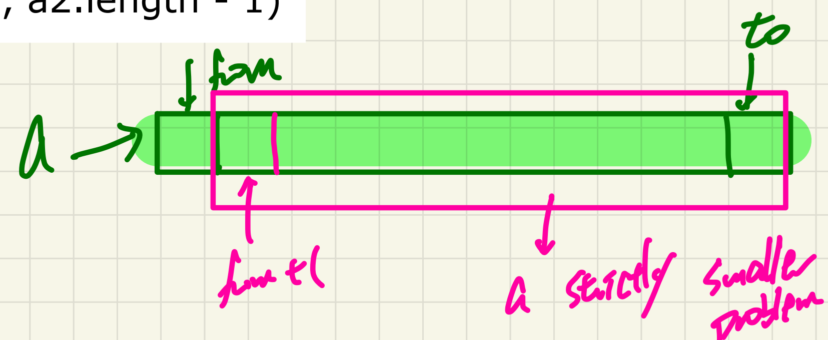
```
void m(int[] a, int from, int to) {
  if (from > to) { /* base case */ }
  else if (from == to) { /* base case */ }
  else { m(a, from + 1, to) } }
```

only a ref.

Say a1 = {}, consider m(a1, 0, a1.length - 1)

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





Problem: Are All Numbers Positive?

universal property

$(\forall x : \text{False} \bullet P(x)) \equiv \text{True}$

```
boolean allPositive(int[] a) {
  return allPositiveHelper (a, 0, a.length - 1);
}

boolean allPositiveHelper (int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return a[from] > 0;
  }
  else { /* recursive case */
    return a[from] > 0 && allPositiveHelper (a, from + 1, to);
  }
}
```

c.b.v.

recursive helper method

✓

B.C.

Is there

(existential prop?) a positive #?

↳ false

from, to

a

from+1

Empty array: all elements are positive (True)

∵ no way to find a witness to show otherwise

Lecture 3 - Monday, January 16

Announcements

- **Assignment 1** to be released next Monday
  + Background Study: **Basic Recursion**
  + Background Study: **Call by Value**
  + Look ahead: **WrittenTest1**

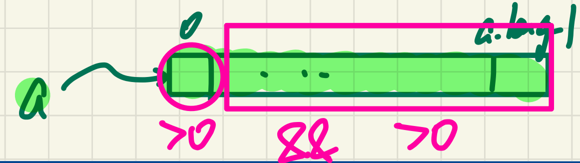
- WT/PT → only allowed to develop items from scratch (no Java library).
- API
- copies of subarray → 1. Java API (Arrays.copy)
  2. from, to (call by value)

Tracing Recursion

1. Stack (e.g. factorial, fib)

2. tree-like drawing

Tracing Recursion: allPositive



Say a = {}

allPositive(a)

allPH(a,0,-1)

a.length == 0

0

public

private

recursive helper method

```
boolean allPositive(int[] a) {
  return allPositiveHelper(a, 0, a.length - 1);
}

boolean allPositiveHelper(int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return a[from] > 0;
  }
  else { /* recursive case */
    return a[from] > 0 && allPositiveHelper(a, from + 1, to);
  }
}
```

0 > -1

current ele. being positive

conjunction

the rest of the array contains all positive.

a

a.length == 0

Tracing Recursion: allPositive

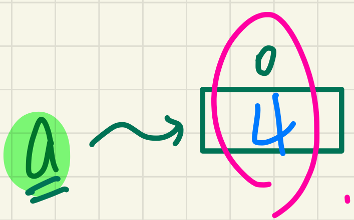
Say a = {4}

allPositive(a)

allPH(a,0,0)

a.length − 1

a[0] > 0




```
boolean allPositive(int[] a) {
  return allPositiveHelper(a, 0, a.length - 1);
}

boolean allPositiveHelper(int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return a[from] > 0;
  }
  else { /* recursive case */
    return a[from] > 0 && allPositiveHelper(a, from + 1, to);
  }
}
```

Tracing Recursion: allPositive

Bring an example Q. on tracing via stack.

Say a = {4,7,3,9}

```
boolean allPositive(int[] a) {
  return allPositiveHelper(a, 0, a.length - 1);
}

boolean allPositiveHelper(int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return a[from] > 0;
  }
  else { /* recursive case */
    return a[from] > 0 && allPositiveHelper(a, from + 1, to);
  }
}
```

4 - 1

0 > 3 F

0 == 3 F

0 + 1, 3

1 + 1, 3

allPositive(a)

allPH(a,0,3)

a[0] > 0 && allPH(a,1,3)

4 > 0 T

a[1] > 0 && allPH(a,2,3)

7 > 0 T

a[2] > 0   allPH(a,3,3)

3 > 0 T

T

base case

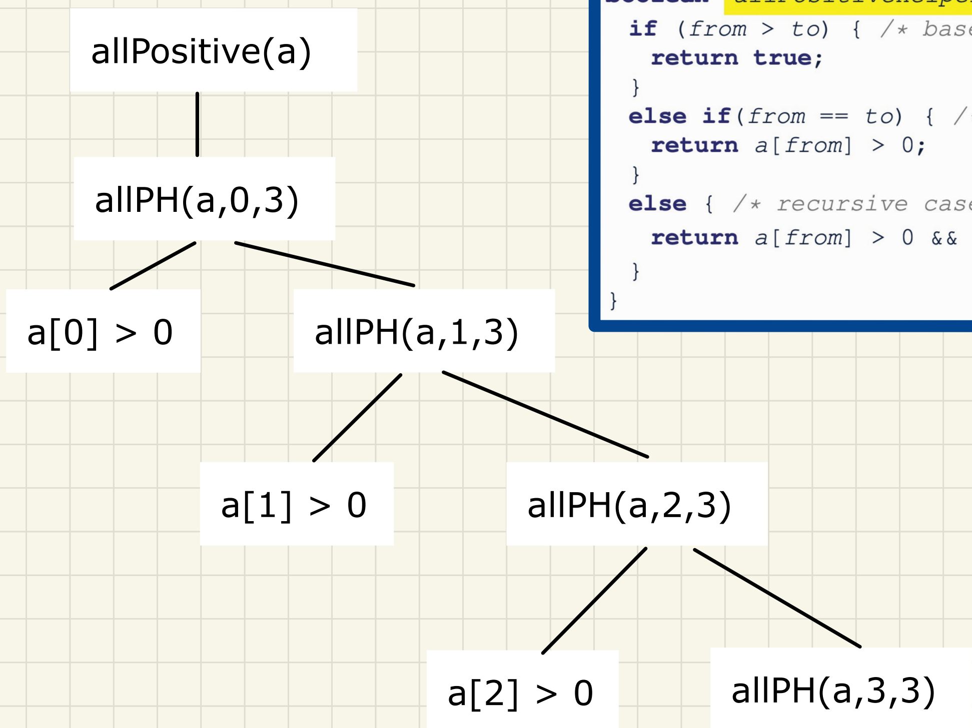
a → 0: 4 | 1: 7 | 2: 3 | 3: 9

from, to

Tracing Recursion: allPositive

Exercise

Say a = {5,3,-2,9}




```java
boolean allPositive(int[] a) {
  return allPositiveHelper (a, 0, a.length - 1);
}

boolean allPositiveHelper (int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return a[from] > 0;
  }
  else { /* recursive case */
    return a[from] > 0 && allPositiveHelper (a, from + 1, to);
  }
}
```

Lecture

Asymptotic Analysis of Algorithms

Measuring Running Time via Experiments

- arrays vs. linked lists

3	1	4	-23

✓

→ your own data struct. →

-23	1	4	3

Sorting.
- Concurrent algorithms
- distributed alg.

your own data struct. → arrays, linked lists

linked lists → SLL, DLL

Sorting

1. insertion sort
2. selection sort
3. merge sort
4. quick sort
5. heap sort → DS: ~~heap~~.

Example Experiment

space inefficiency ⇒ degraded RT

- *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; }
```

answer = answer + c;

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(); }
```

Lecture 4 - Wednesday, January 18

Announcements

- Assignment 1 to be released next Monday
  + Background Study: Basic Recursion
  + Background Study: Call by Value
  + Look ahead: WrittenTest1

Lecture

Asymptotic Analysis of Algorithms

Counting Primitive Operations

Accessing an object's attribute

In practice, the # of "dots" used to inquire some att. value does not depend on the input size.

Person p = new Person(...);

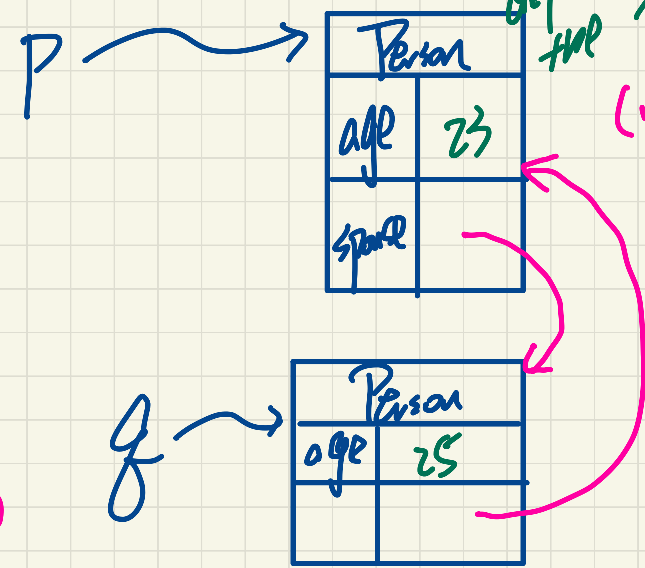
p.age

look up the address stored in p ⇒ constant (PO)

p.spouse.age

multiple lookups ⇒ PO

(no looping).



Method Call

obj.m(· · ·)

Case 1 : m contains POs only

```
m( ){
  ______
  ______
  ______
  ______
}
```

each line corresponds to a PO

can be:
(1) a method call
(2) a loop

may still be PO

Case 2 : m contains some non-PO

```
m( ){
  ______
  ______
  for(int i=0; i<n; i++){
    ...
  }
}  ______
```

size of some input array

a method call containing non-PO

e.g. int a[] = {2, 3, 4, 5} → 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; }
```

Line 2: 2
Line 3: n+1
Line 4: $(n-1)\cdot 2$
Line 5: $(n-1)\cdot 2$
Line 6: $(n-1)\cdot 2$
Line 7: 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?

i<n T: $(n-1) + 1$ → i<n F

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

$n-1$ → i<n T

$2 + (n+1) + (n-1)\cdot 6 + 1$

$= ?$

stay condition   progress

for (I ; S.C. ; P) {

}

executed the 1st time, if any, at the end of 1st iteration

e.g. for (; ______ ;) {
⋮
}

Example 2: Counting Number of Primitive Operations

```
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  }
```

(Exercise)

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)?

From Absolute RT to Relative RT

t → exact time taken to execute a PO

e.g. Mac M1 2ms

e.g. Mac'14 4ms

findMax contains $7n-2$ POs.

n → size of input array.

alg.

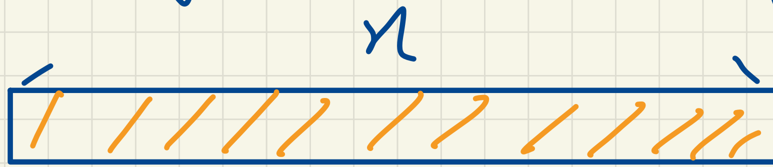
e.g. # PO ≤ 2 → constant despite input size

\# PO $2n$ → linear on the input size

Algorithm 1 $\boxed{(\underbrace{7n-2}_{\#\ POs})\cdot t}$ abs. time

Algorithm 2 $\boxed{(\underbrace{10n+3}_{\#\ POs})\cdot t}$

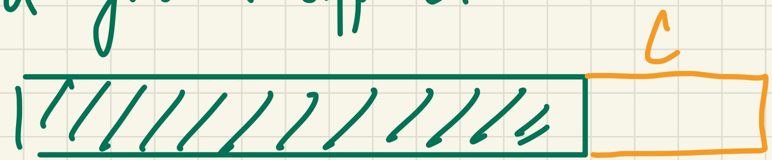
Keep adding elements to array



a

amortized analysis

① fixed growth approach



② doubling

Lecture 5 - Monday, January 23

Announcements

- **Assignment 1** to be released tonight

  ↳ 6pm ~ 9pm

Lecture

Asymptotic Analysis of Algorithms

Asymptotic Upper Bound

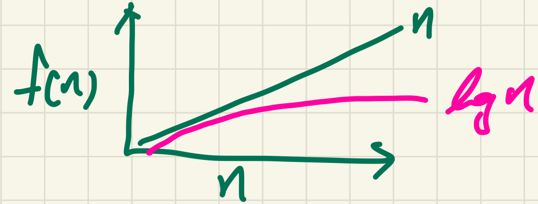
n vs. $7n$

↳ asymptotically, they're just the family of "n"

$O(n)$

$$7n + 2n \cdot \log n + 3n^2$$

input size



approximate the above running time function

lowest term

$n^{1+<1} = n^{1.\ldots}$

lower power than 1.

$$7 \cdot n^1 + 2 \cdot n^1 \cdot \log n + 3 \cdot n^2$$

this is what matters, disregarding ALL lower terms

multiplicative constants

lower term

higest power

findMax

↳ $7n - 2$

↳ (1) # Pos

e.g. $n = 1 \rightarrow 5$ Pos

$n = 10 \rightarrow 68$ Pos

(2) relative RT

RT

$$\boxed{f(n) = 5}$$

$f(0) = 5$

$f(10) = 5$

$f(1M) = 5$

RT is independant of the input size

Polynomial: n^d

$d \geq 2$

$f(n)$ — $O(g(n))$

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

- $f(n)$: RT function
  - ↳ inputsize → relative RT
  - e.g. findmax has relative RT: $7n - 2$

- $g(n)$: reference function
  (further manipulation on $g(n)$ expection)
  - $O(n)$ ✓
  - $O(\underline{\underline{2}} \cdot n)$ ✗
  - $O(\underline{\underline{2}}n + \underline{\underline{\tfrac{1}{2}}})$ ✗

<u>Goal</u> Prove $f(n)$ is $O(\boxed{g(n)})$

↳ not including
(1) lower terms
(2) multiplicative constants

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: u.b.e.

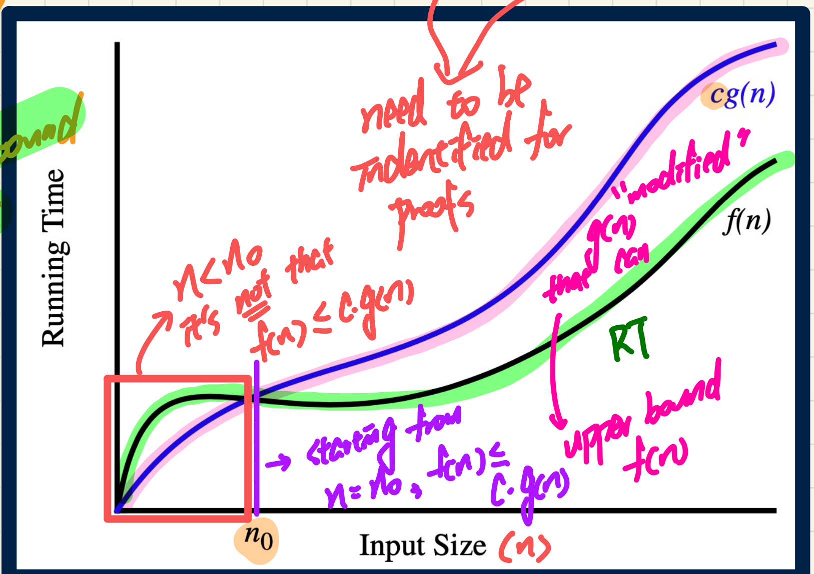
$$f(n) \leq c \cdot g(n) \quad \text{for } n \geq n_0$$

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

O(g(n))

f(n)

starting point of upper bound effect



Example:

f(n) = 8n + 5

g(n) = n ref. function

Prove:

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

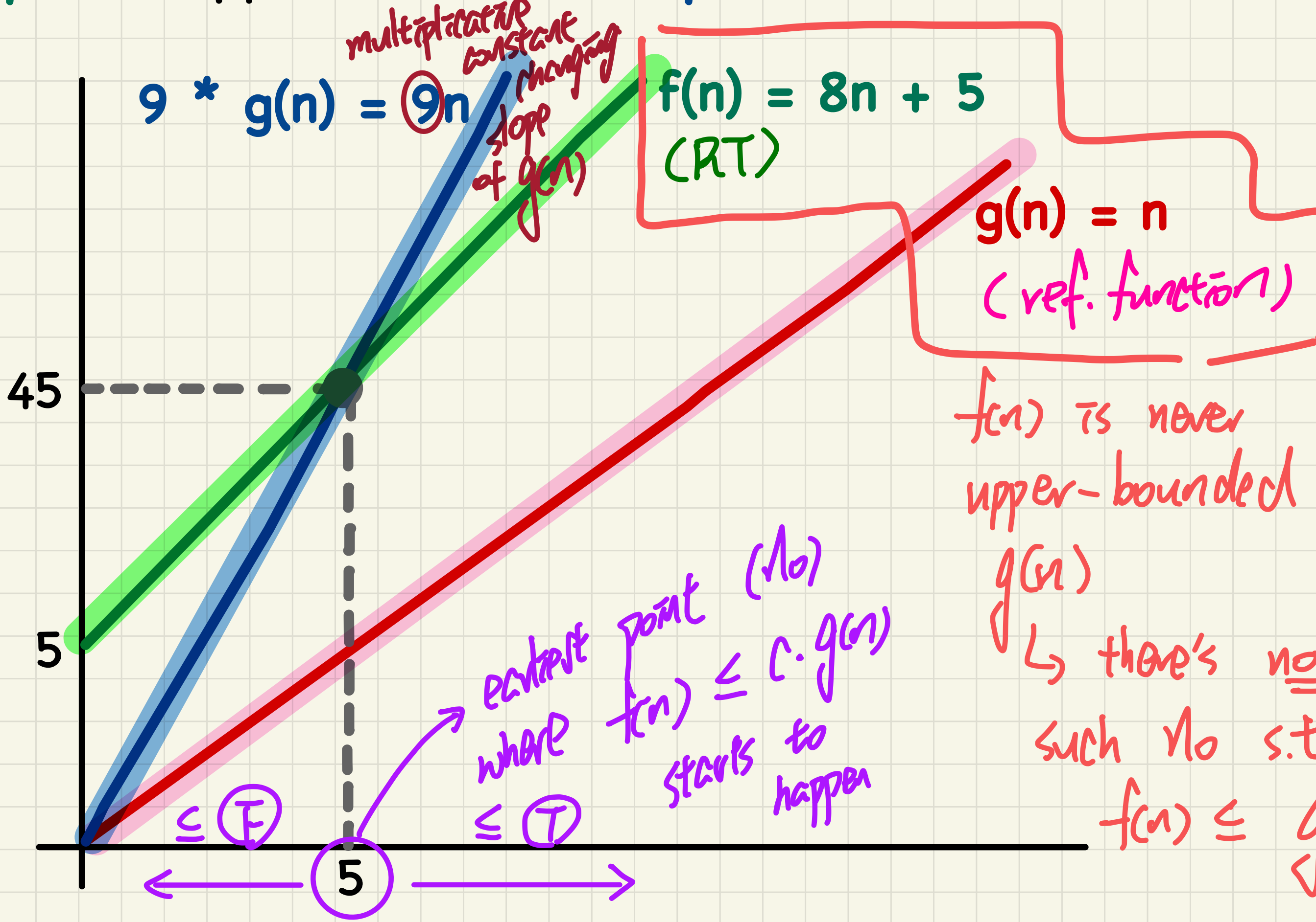
Choose:

c = 9

What about n0?

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

Asymptotic Upper Bound: Example



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

↳ there's not such n_0 s.t. $f(n) \le g(n)$

higest power

$$f(n) \overset{RT}{=\!=} 5n^{4} + 3n^{3} + 2n^{2} + 4n + 1 \cdot n^{0}$$

(1) Guess: $f(n)$ is $O(n^4)$

(2) Prove:

Choose c: $|5| + |3| + |2| + |4| + |1| = 15$

Choose n_0: 1.

Lecture

Asymptotic Analysis of Algorithms

Asymptotic Upper Bounds of Math Functions

Asymptotic Upper Bounds: Example (1)

$$\boxed{\log 1 = 0}$$

$5n^2 + 3n \cdot \log n + 2n + 5$ is $O($[redacted]$)$

Problem (1) State and (2) prove the asymptotic upper bound of the above function.

(1) $O(n^2)$

(2) Prove by choosing:

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

$n_0 = 1$

Verify: when $n = 1$

show: $f(n) \le \boxed{15 \cdot n^2}$

$5 \cdot 1^2 + 3 \cdot 1 \cdot \overset{0}{\log 1} + 2 \cdot 1 + 5$

$= 12 \le \boxed{15 \cdot 1^2}$

Lecture 6 - Wednesday, January 25

Announcements

- Assignment 1 released:
  + Tracing Recursion:
    - Paper: Call Stack vs. Tree
    - Debugger in Eclipse
  + Help: Scheduled Office Hours & TAs
  + Look ahead: WrittenTest1

$* \; 1^0 = 1^1 = \cdots = 1^d = 1$

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

We prove by choosing

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

(1) $f(1) \le c \cdot \cancel{n^d}\; 1^d$

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

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 $O(n^d)$. → $g(n)$

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

$** \; a_0 \le |a_0|, \; a_1 \le |a_1|, \; \ldots, \; a_d \le |a_d|$

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

$$\overset{*}{=} (a_0 + a_1 + \cdots + a_d) \cdot 1^d \overset{**}{\le} (|a_0| + |a_1| + \cdots + |a_d|) \cdot 1^d$$

Upper-bound effect holds? $[f(n) \le (|a_0| + |a_1| + \cdots + |a_d|) \cdot n^d]$ $\boxed{n > 1}$

$*** \; n^0 \le n^d, \; n^1 \le n^d, \; \ldots, \; n^d \le n^d$

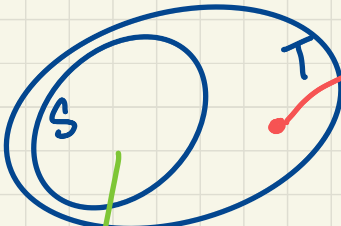
$$f(n) = a_0 \cdot n^0 + a_1 \cdot n^1 + \cdots + a_d \cdot n^d$$

$$\overset{***}{\le} (a_0 + a_1 + \cdots + a_d) \cdot n^d \overset{**}{\le} (|a_0| + |a_1| + \cdots + |a_d|) \cdot n^d$$

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

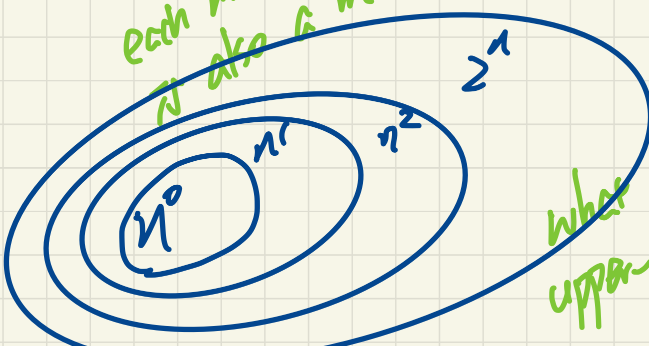
Proper Subset

$S \subset T$

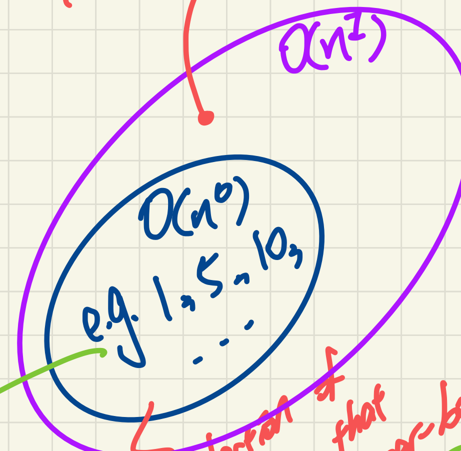


there's at least one member of T that's not a member of S

each member in S is also a member in T



there's at least one function that can be u.b. by n^1 but not u.b. by n^0



what can be upper-bounded by n^0 can also be upper-bounded by n^1

the collection of functions that can be upper-bounded by n^0

" 1

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

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

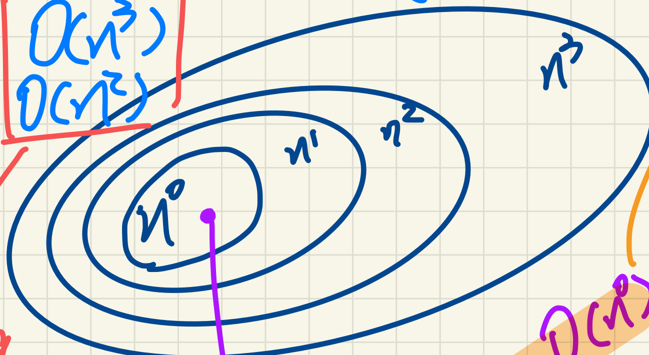
} two possible solutions to the same problem

What if inaccurate u.b. is given?

$f_1(n)$ is $O(n^3)$

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

misleading for decision making



$f(n) = 1$

system.out.println("Hello World");

$f(n)$ is $O(n^0)$ ✓

the most accurate upper bound that's correct.

$f(n)$ is $O(n^1)$ ✓

$f(n)$ is $O(n^2)$ ✓

⋮

$f(n)$ is $O(2^n)$ ✓

Asymptotic Upper Bounds: Example (2)

$20n^3 + 10n \cdot logn + 5$ is $O($[redacted]$)$

Derive[1] and Prove[2] the most accurate asymptotic u.b. of the above function.

(1) $O(n^3)$

(2) Prove by choosing: $c = |20| + |10| + |5| = 35$

$n_0 = 1$

$c \cdot g(1) = 35 \cdot 1^3 = 35$

Verify: $\boxed{f(1) \le c \cdot g(1)}$ $f(1) = 20 \cdot 1^3 + 10 \cdot 1 \cdot \log 1 + 5 = 25$

Asymptotic Upper Bounds: Example (3)

$\boxed{3 \cdot \log n + 2 \text{ is } O(\text{[redacted]})}$ $\equiv 3 \cdot \log n + 2 \cdot n^0$

(1) $O(\log n)$

(2) Prove by choosing: $c = |3| + |2| = 5$

$n_0 = 1$

Verify $\boxed{f(1) \stackrel{\text{failed}}{\leq} c \cdot g(1)}$ failed

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

$c \cdot g(1) = 5 \cdot \log 1 = 0$

$c = |3| + |2| = 5$

$n_0 = 2$

(exercise!)

Asymptotic Upper Bounds: Example (4)

2^{n+2} is $O($[redacted]$)$

$$2^{n+2} = 2^n \cdot 2^2$$

$$= \cancel{4} \cdot 2^n$$

$O(2^{n+2})$ X

$\underline{\underline{O(2^n)}}$.

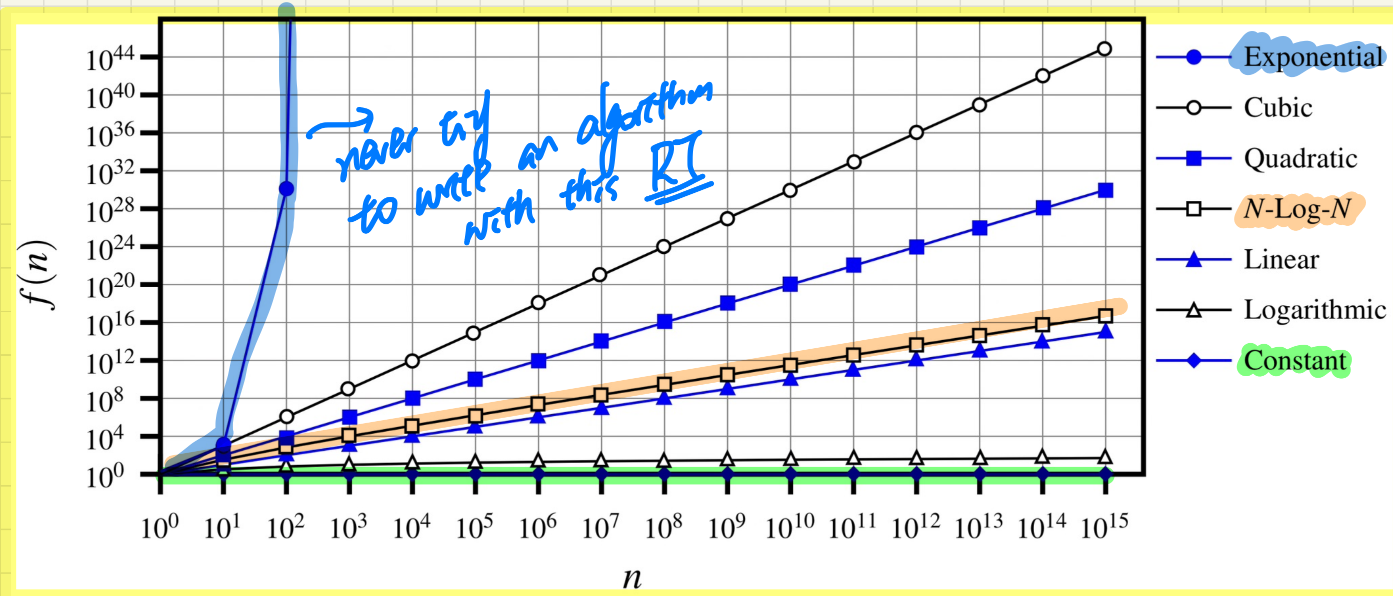
Asymptotic Upper Bounds: Example (5)

$2n + 100 \cdot logn$ is $O($[redacted]$)$

Exercise

$n! = n \cdot (n-1) \cdots$

Running Time vs. Input Size: Common Rates of Growth



2^n vs. 3^n $(2^?)^n$

$O(2^n)$

Lecture

Asymptotic Analysis of Algorithms

Asymptotic Upper Bounds of Implemented Algorithms

Determining the Asymptotic Upper Bound (1)

```
int maxOf (int x, int y) {
  int max = x;  1.
  if (y > x) {  1.
    max = y;  1.
  }
  return max;  1.
}
```

$$O(1+1+1+1) = O(4) = \boxed{O(1)}.$$

$$4 \cdot n^0$$

Determining the Asymptotic Upper Bound (2)

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

body of loop

$$O(\underbrace{1}_{\text{L2}} + \underbrace{n}_{\text{header of loop}} + \underbrace{n}_{\text{\# iterations}} \cdot (\underbrace{1+1+1}_{\text{each iteration}}) + \underbrace{1}_{\text{return}}) = \boxed{O(n)}$$

Determining the Asymptotic Upper Bound (3)

$[a, b] = b - a + 1$

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

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

body of inner loop: 1

i ++; → 1

Pattern of loop Counters

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

outer loop runs for **n times**

inner loop runs for **n times**

(continued on Mon).

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

~~n~~ 10,000

$10{,}000 = n^0 \cdot 10{,}000$

$O(n)$

Constants

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

~~n~~ 1M

~~n~~ 1B

$O(1)$

Lecture 7 - Monday, January 30

Announcements

- Written Test 1 guide released
  + EECS account login (for WSC computers)
  + PPY account + Duo Mobile (for eClass)
- Assignment 1 due in a week:
  + Tracing Recursion:
    - Paper: Call Stack vs. Tree
    - Debugger in Eclipse
  + Help: Scheduled Office Hours & TAs

Determining the Asymptotic Upper Bound (3)

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

body of inner loop: 1

$[a, b] = b - a + 1$

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

* P_1 gets executed once for every value of j.
* P_2 gets executed once for every value of i.

P1

P2

Pattern of loop Counters

n^2 combinations of $[i, j]$

i	j					
0	0	1	2	...	$(n-1)$	n
1	0	1	2	...	$(n-1)$	n
2						
⋮	⋮					
$n-1$	0	1	2	...	$(n-1)$	n

outer loop runs for n times

inner loop runs for n times

$$O(\underbrace{n^2 \cdot 1}_{L4 \sim L6} + \underbrace{n \cdot 1}_{L7} + \underbrace{1}_{L8})$$

iterations of outer loop

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

$$= O(n^2)$$

Determining the Asymptotic Upper Bound (4)

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

1 $\quad n \quad$ 1 $\quad n^2 \quad$ 1

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

Determining the Asymptotic Upper Bound (5)

size of $[2, n-1]$ is $n-2$.

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

pattern of combining (i, j)?

Pattern of (i, j)

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

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

$i = 0 \quad j = 0, 1, \ldots, n-1 \quad n$

$i = 1 \quad j = 1, \ldots, n-1 \quad n-1$

$i = 2 \quad j = 2, \ldots, n-1 \quad n-2$

$\vdots$

$i = n-1 \quad j = n-1 \quad 1$

$$O(\underbrace{1}_{L2} + \underbrace{\boxed{n^2}}_{\text{\# of combinations of } i, j} \cdot \underbrace{1}_{L5} + \underbrace{1}_{L6})$$

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

Sum of Arithmetic Sequence

$1 + 2 + 3 + \cdots + 9 + 10$

$(1 + 10) \cdot \frac{\cancel{11}\,10}{2} = ?$

$I + 0 \cdot C$

$\underset{\text{first term}}{\underline{I}} \overset{+C}{+} (I + \underset{\text{constant}}{\boxed{C}}) \overset{+C}{+} (I + 2 \cdot C) + \cdots + (I + (n-1) \cdot C)$

$= \frac{[I + (I + (n-1) \cdot C)] \cdot n}{2}$

$n \to$ # of terms

e.g. $\underset{\text{first term}}{\underline{1}} \overset{+1}{+} 2 \overset{+1}{+} 3 + \cdots + \underline{n}$

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

↳ is $O(n^2)$

Lecture

Arrays vs. Linked Lists

Asymptotic Upper Bounds of Array Operations

$a[i] = a\boxed{[\underline{i+1}]};$

Pos: 4 $(= 2^0 \cdot 4)$

object creation: $O(1)$

Inserting into an Array

copy input[0] ⋮ input[i-1] to result

where to insert.

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

$j = 0..(i-1)$ $0..0$

$[i+1, n] = n-(i+1)+1 = n-i$

copy a[i-1] to result a[i]

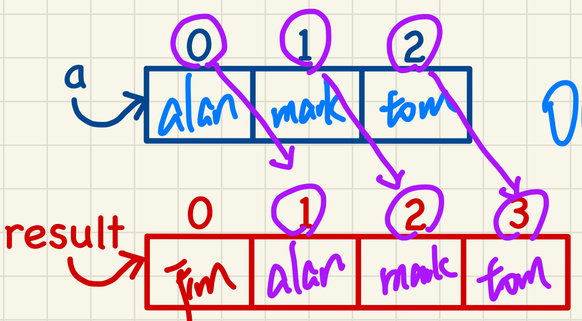
$O(i \cdot 1) = O(n)$ worst case: $i == n$

$O((n-i)\cdot 1) = O(n)$ worst case: $i == 0$

a.length

Example:

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

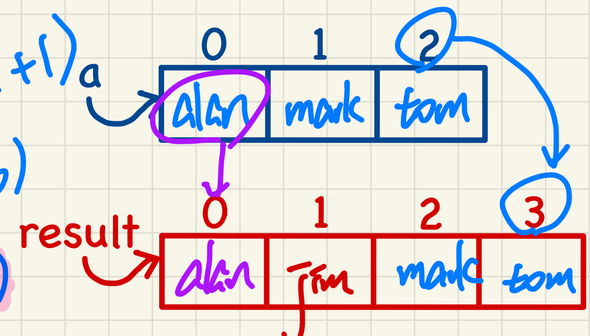


$O(1+n+1+n+1) = O(2n+3) = O(n)$

Example:

result[0] = a[0];
result[1] = jim;
result[2] = a[1]
result[3] = a[2]

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



result[j] = a[j-1]

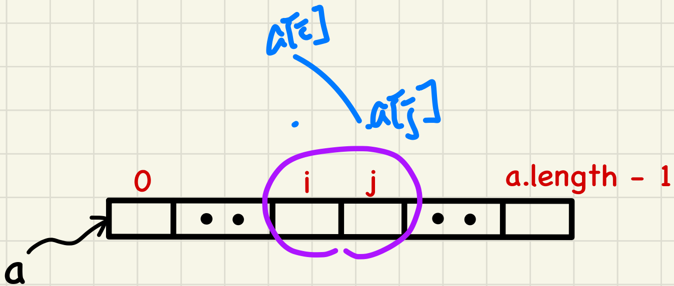
Exercise: insertAt({alan, mark, tom}, 3, jim, 3)

Lecture

Arrays vs. Linked Lists

Selection Sort vs. Insertion Sort

Sorting Orders of Arrays



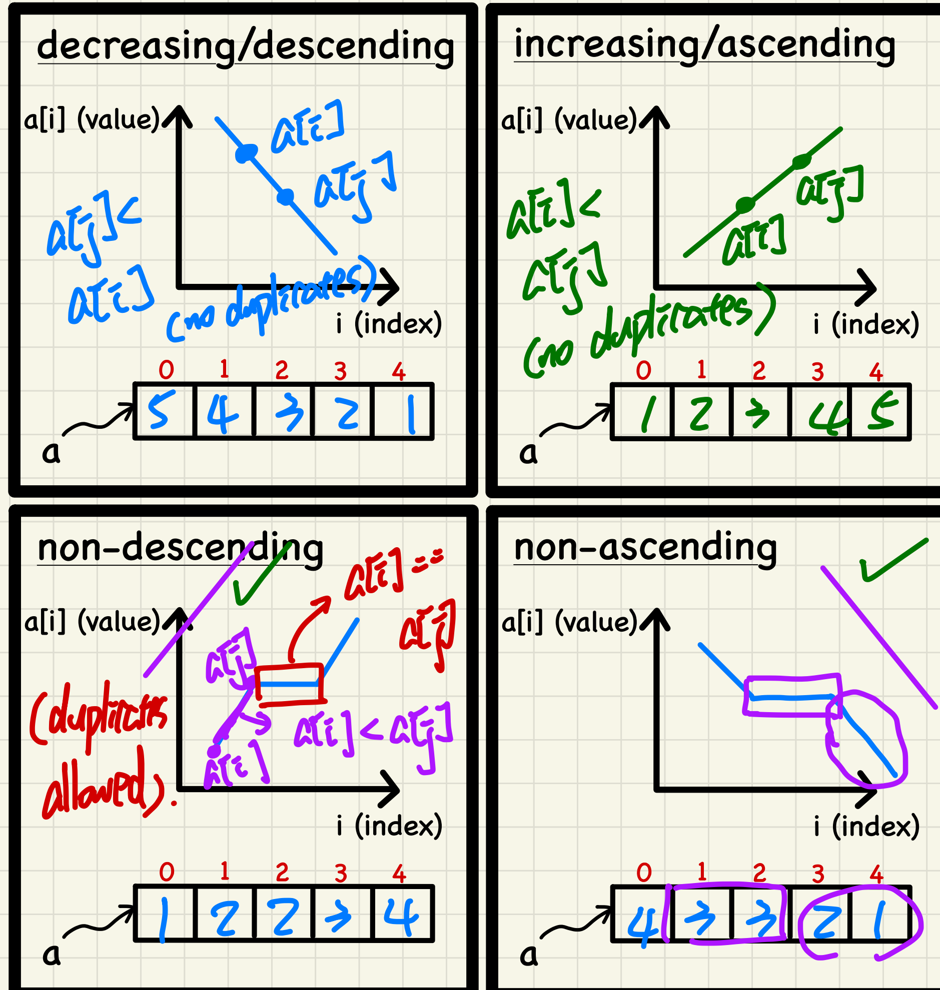
non-descending

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

$\downarrow$

$\neg(a[i] > a[j])$

$\equiv a[i] \leq a[j]$



Lecture 8 - Wednesday, February 1

Announcements

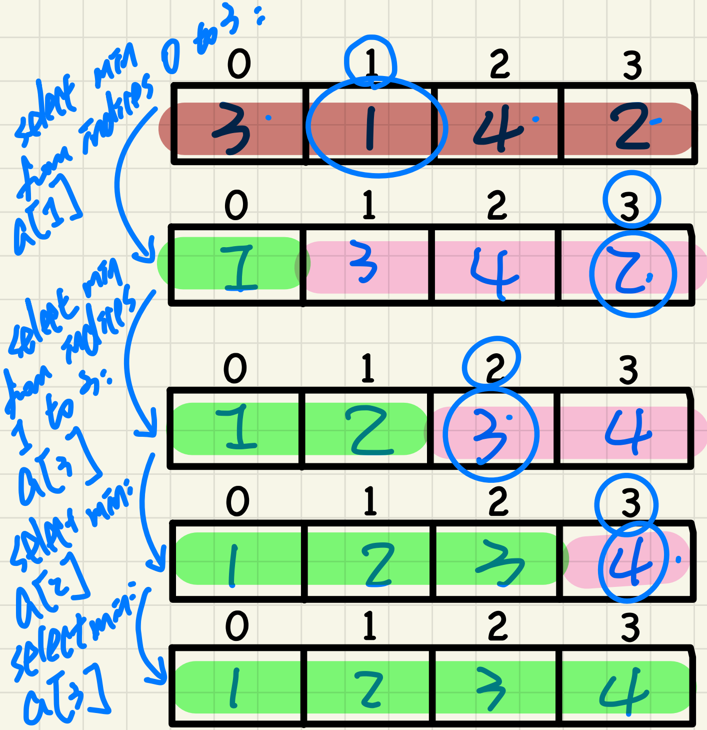
- Written Test 1 guide released
  + EECS account login (for WSC computers)
  + PPY account + Duo Mobile (for eClass)
  + Practice Questions & Review Session **Survey**
- Assignment 1 due soon!
  + Help: Scheduled Office Hours & TAs

Selection Sort

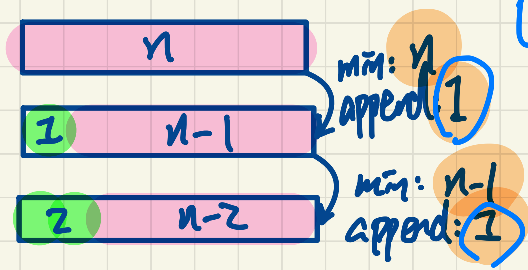
① $\underline{\underline{n}}$ iterations (need to choose min n times).

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

(when started, sorted portion is empty)



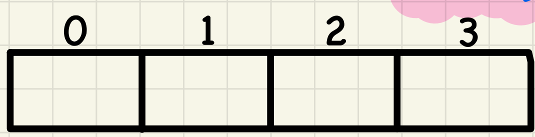
$\frac{(n+1) \cdot n}{2}$



$O(n \cdot 1 + \underline{(n+(n-1)+\dots} \underline{+1})$

\# of appendings $= O(?)$

$= O(n^2)$



In-place sorting

↳ sorting procedure operates directly on the original input array.

Insertion Sort

iterations (for choosing): $\underline{n}$

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

insert: a[1]

0	1	2	3
3	1	4	2

Cost
choose: 1
insert: size 1

in general, it's a subarray of size ? for insertion

unsorted

sorted

0	1	2	3
1	3	4	2

insert: a[2]

0	1	2	3
1	3	4	2

size n-1 for the last iteration

Cost
choose: 1
insert: size 2

insert: a[3]

0	1	2	3
1	2	3	4

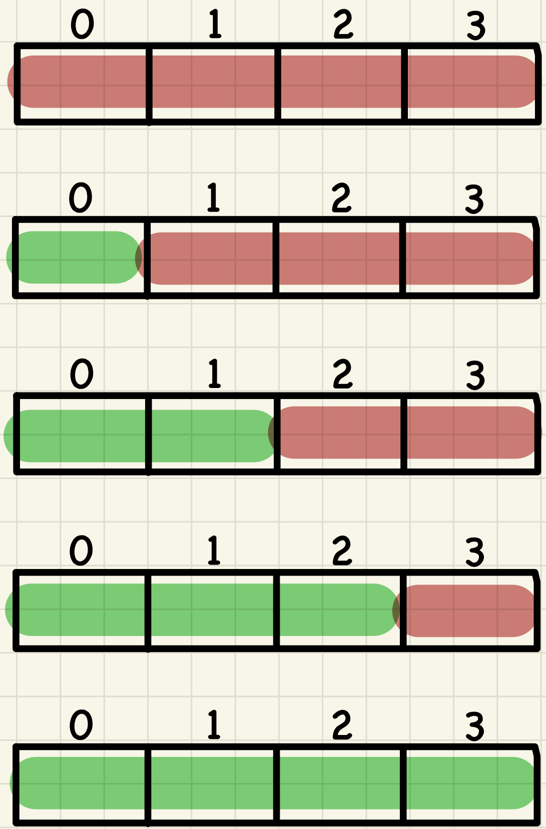
Cost
choose: 1
insert: size 3
(n-1)

$$O\big(n \cdot \underbrace{1}_{\text{choosing 1st unsorted element}} + \underbrace{(1+2+\dots(n-1))}_{\text{insertions}=O(?)}\big)$$

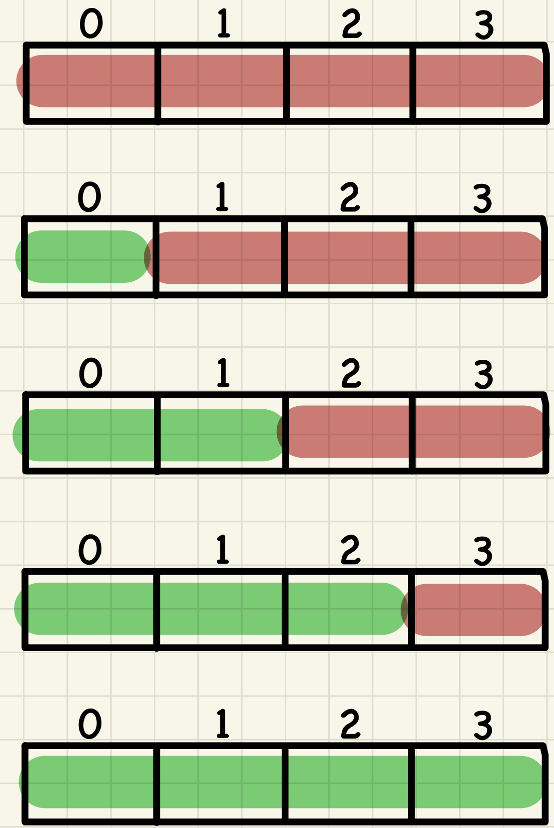
$= O(n^2)$

0	1	2	3

Selection Sort



Insertion Sort



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;
```

n → a.length

L5: 1 ✓

L6–8: 1

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

i	j	
0	0 1 ... n-1	n
1	1 ... n-1	n-1
2	2 ... n-1	n-2
3	⋮	⋮
⋮		
n-2	n-1	1

$[0, n-2]$ $= (n-2)-0+1$ $= n-1$

$$O\Big(\underbrace{(n-1)}_{\text{\# of iterations of outer loop}}\cdot\underbrace{1}_{L3,\,L6\text{-}8} + \underbrace{(n+(n-1)+\dots+1)}_{\text{\# combinations of } i,\,j}\cdot\underbrace{1}_{L5}\Big)$$

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

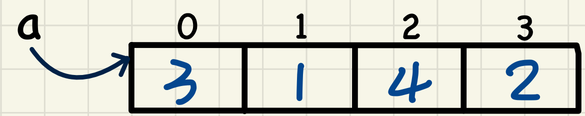
Insertion Sort: Deriving Asymptotic Upper Bound

```
void insertionSort(int[] a, int n)
  for (int i = 1; i < n; i ++)
    int current = a[i];
    int j = i;
    while (j > 0 && a[j - 1] > current)
      a[j] = a[j - 1];
      j --;
    a[j] = current;
```

Exercise

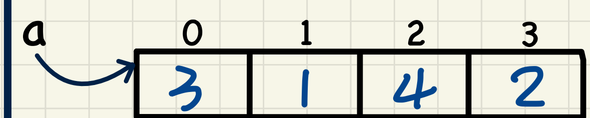
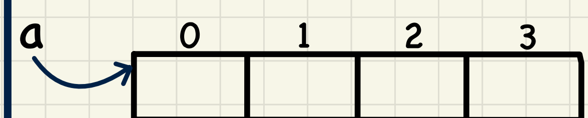
Selection Sort in Java

```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;
```

a

0	1	2	3
3	1	4	2

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?
			a 0 1 2 3 3 1 4 2
			a 0 1 2 3 (empty)
			a 0 1 2 3 (empty)

Lecture

Arrays vs. Linked Lists

Singly-Linked Lists - Intuitive Introduction

Singly-Linked Lists (SLL): Visual Introduction

- A chain of connected nodes
- Each node contains:
  + reference to a data object
  + reference to the next node

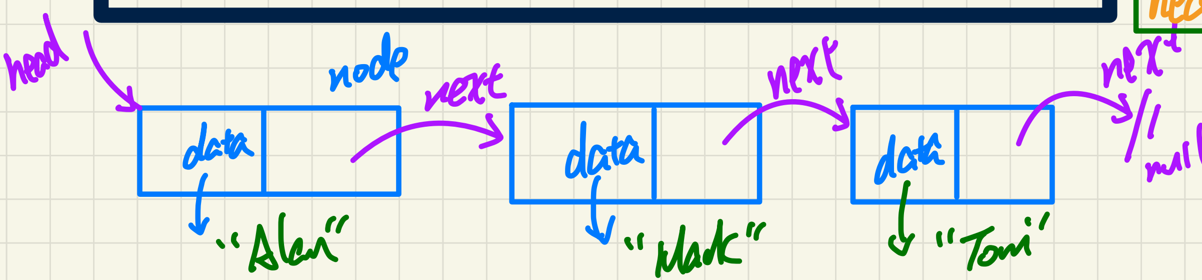
linear: each node has a unique successor

- Accessing a node in a list:
  + * Relative positioning: O(n)
  + Absolute indexing: O(1)
- The chain may grow or shrink dynamically.
- Head vs. Tail

head != null : 1st node
head.next != null : 2nd node
head.next.next != null : 3rd node

head.data : "Alan"
head.next.data : "Mark"
head.next.next.data : "Tom"
(1) head.next.next.next null
(2) head.next.next.next.data

null
NullPointerExcep.



Friday, February 3

Written Test 1 Review

* k<n only executed before outer loop exits (when j=n)

Count # of Primitive Operations

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

2   1   1

i = i+1   j = j+1 (2)   (2)

sum = sum + (5)

i=1	i<n
1	T
⋮	F
n−1	
n	

1. # times i<n evaluated? n
2. # times body of loop exec? n−1

$\underbrace{1}_{i=1} + \underbrace{n}_{i<n} + \underbrace{2}_{i++}\cdot(n-1) + \underbrace{4}_{\angle 4}\cdot(n-1) = \boxed{7n-5}$

for each value of j making j<n true, k<n is evaluated (n+1) times.

this k<n cond.

n

k<n → F

j	k					
0	0	1	2	...	n−1	n
1	0	1	2	...	n−1	n
2	0	1	2	...	n−1	n
⋮	...					⋮
n−1	0	1	2	...	n−1	n
n						

j<n → F

$\underbrace{1}_{j=0} + \underbrace{n}_{k=0} + \underbrace{(n+1)}_{j<n} + \underbrace{n\cdot(n+1)}_{k<n} +$

$\underbrace{2\cdot n}_{j++} + \underbrace{2\cdot n^2}_{k++} + \underbrace{5\cdot n^2}_{\angle 9} + \underbrace{1}_{\angle 10}$

$= ?$

Count # of Pos

$$24 + 21 + 18 + 15 + 12 = \frac{(24+12) * 5}{2}$$

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

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

how many times?

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

↳ when i is between 0 and n-1, j<n is evaluated between i and n.

i						
0	0	1	2	...	(n-1)	n
1		1	2	...	(n-1)	n
2			2	...	(n-1)	n
⋮		⋮				
n-1					(n-1)	n

$j<n \to$ F

$i<n$ n → F

$$\underbrace{1}_{i=0} + \underbrace{n}_{j=i} + \underbrace{n+1}_{i<n} + \underbrace{\frac{((n+1)+2)\cdot n}{2}}_{j<n}$$

$$+ \underbrace{n\cdot 2}_{i++} + \underbrace{\frac{(n+1)\cdot n}{2}\cdot 2}_{j++} + \underbrace{\frac{(n+1)\cdot n}{2}\cdot 3}_{L5}$$

$$+ 1 = ?$$

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

```
for ( int j = 0 ; j <= i-1 ; j ++ ) {
    for ( int k = i+1 ; k <= n ; k ++ ) {
    }
}
```

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

$n-(i+1)+1$

$= \boxed{n-i}$

$(n-i)\cdot i$

$= \underline{\underline{n}}\cdot\cancel{i} - i^2$ constant

$O(n)$

```
int count = 0;
for (int i=n/2; i<=n; i++)
    for (int j=1; j+n/2<=n; j = j++)
        for (int k=1; k<=n; k = k * 2)
            count++;
```

$O(n)$

$j + n/2 <= n$

$j <= n - \frac{n}{2} = \frac{n}{2}$

1

$\log n$

How many times j changes its value?

assume $n = 1000$

$k = 1 = 2^0$

$2 = 2^1$

$4 = 2^2$

$8 = 2^3$

$\vdots$

$512 = 2^9$

$10 = \lceil \log_2 1000 \rceil$

i	j				
$n/2$	1	2	3	...	$\frac{n}{2}$
$n/2+1$	1	2	3	...	$\frac{n}{2}$
$n/2+2$	1	2	3	...	$\frac{n}{2}$
$\vdots$					
n	1	2	3	...	$\frac{n}{2}$

$\approx \frac{n}{2}$

k: $1 \ \ldots \ \log n \ \ldots \ \frac{n}{2}$

$O\left(\frac{n}{2} \cdot \frac{n}{2} \cdot \log n\right) = O(n^2 \cdot \log n)$

Lecture 9 - Wednesday, February 8

Announcements

- Released soon:
  + **WrittenTest1** result (Friday or Monday the latest)
  + **Assignment1** solution
- **Assignment 2** to be released by the end of today or early tomorrow (Thursday)

by Thursday.

- To make up the test time on Monday, videos will be released

assumed by next week's class

Lecture

Arrays vs. Linked Lists

Singly-Linked Lists -
Java Implementation: String Lists
Initializing a List

Singly-Linked Lists (SLL): Visual Introduction

int[] a = new int[ ];

fixed length.

- A chain of connected nodes
- Each node contains: linear: each node has a unique successor
    + reference to a data object
    + reference to the next node
- Accessing a node in a list:
    * + Relative positioning: O(n)
    + Absolute indexing: O(1)
- The chain may grow or shrink dynamically.
- Head vs. Tail

head : 1st node != null

head.next : 2nd node != null

head.next.next : 3rd node != null

head.data : "Alan"

head.next.data : "Mark"

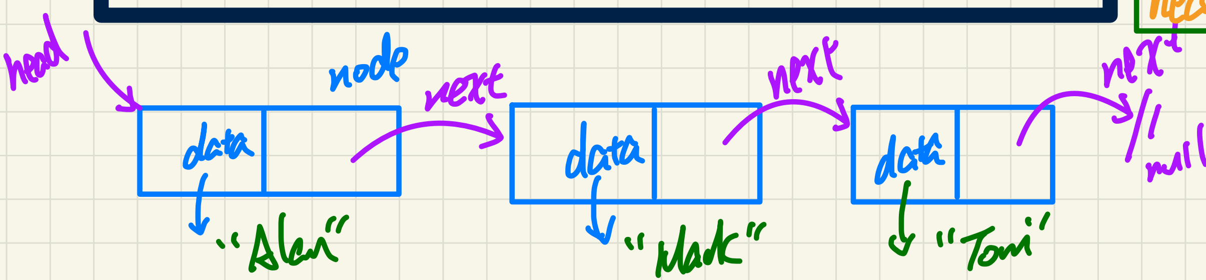
head.next.next.data : "Tom"

(1) head.next.next.next null

(2) head.next.next.next.data

null

NullPointerExcep.



ArrayList

library class

↳ resizable array

↳ doubling

Linked-Lists

↳ good for implementing specialized ops.

Absolute Indexing of Arrays

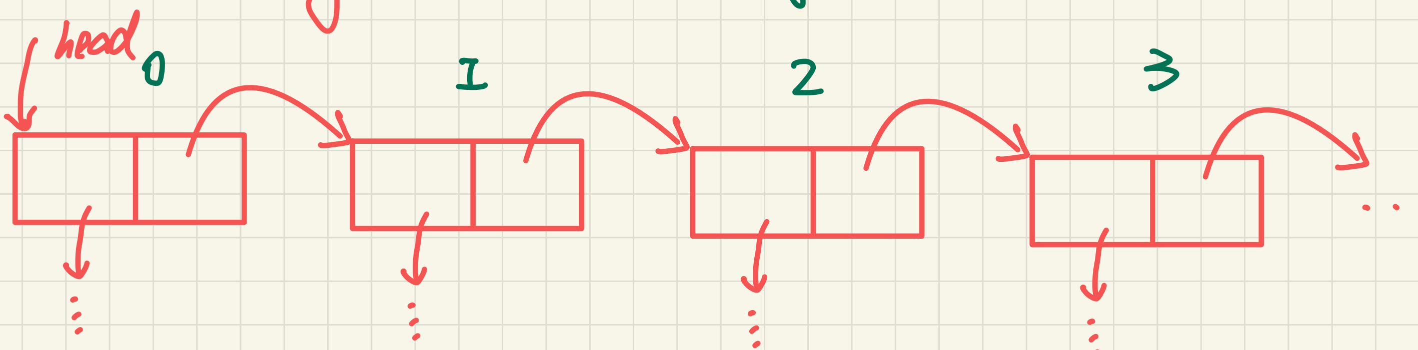
$a[i] \rightarrow O(1)$

int.

Relative Positioning of LL

$O(n)$ getNodeAt(i)

position in chain of nodes

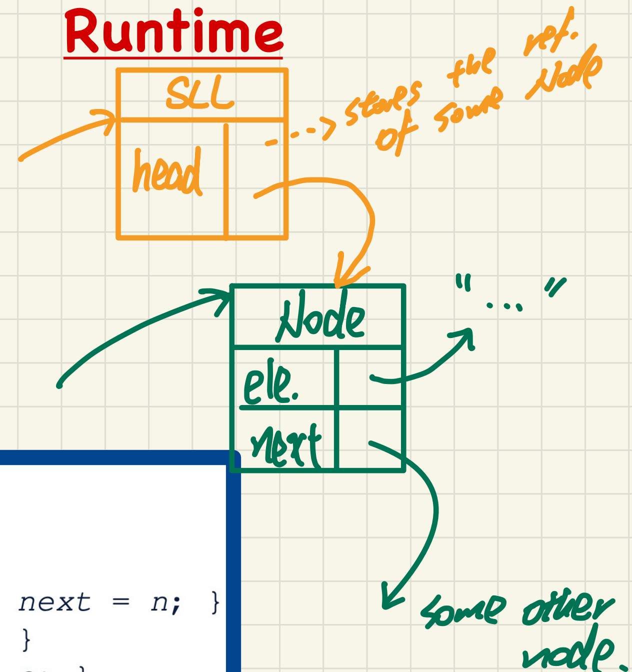


Implementing SLL in Java: SinglyLinkedList vs. Node

```java
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() { ... }
}
```

```java
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; }
}
```

Runtime



SLL: Constructing a Chain of Nodes

tom → mark → alan.

```java
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; }
}
```

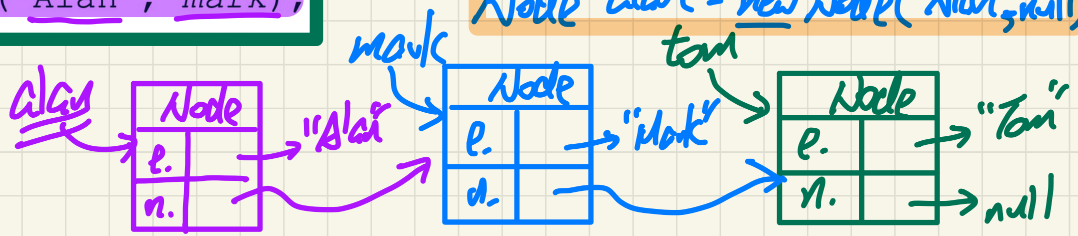
Approach 1

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

Exercise

not compiling!

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



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; }
}
```

mark this alan mark.

tom malc tom

Aliasing

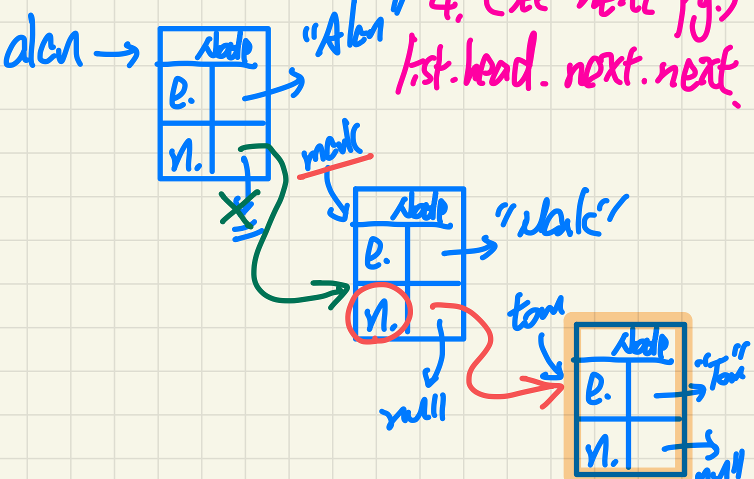
↳ an object's ref being stored in multiple variables.

1. tom
2. mark.next
3. alan.next.next
4. (see next pg.)

list.head.next.next

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);
```

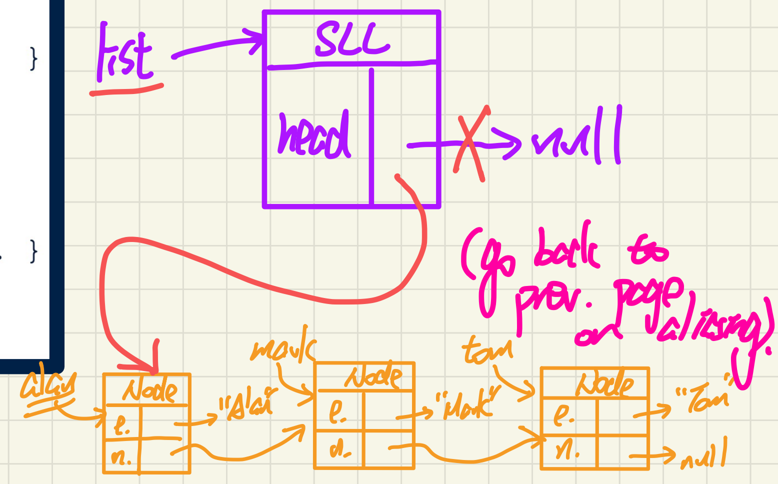


alan.next = mark ;

mark.next = tom ;

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

```java
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

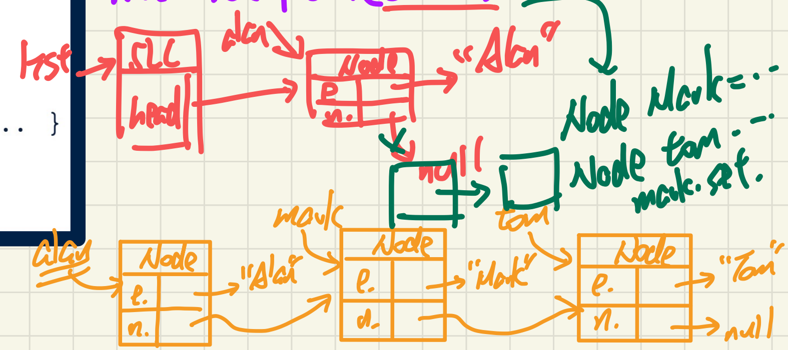
```java
Node tom = new Node("Tom", null);
Node mark = new Node("Mark", tom);
Node alan = new Node("Alan", mark);
SinglyLinkedList list = new SinglyLinkedList();
list.setHead(alan);
```

Initialize head to default null

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

```java
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() { ... }
}
```

Node alan = ... ;
SLL list = ... ;
list. setHead(alan);



Approach 2

```java
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);
```

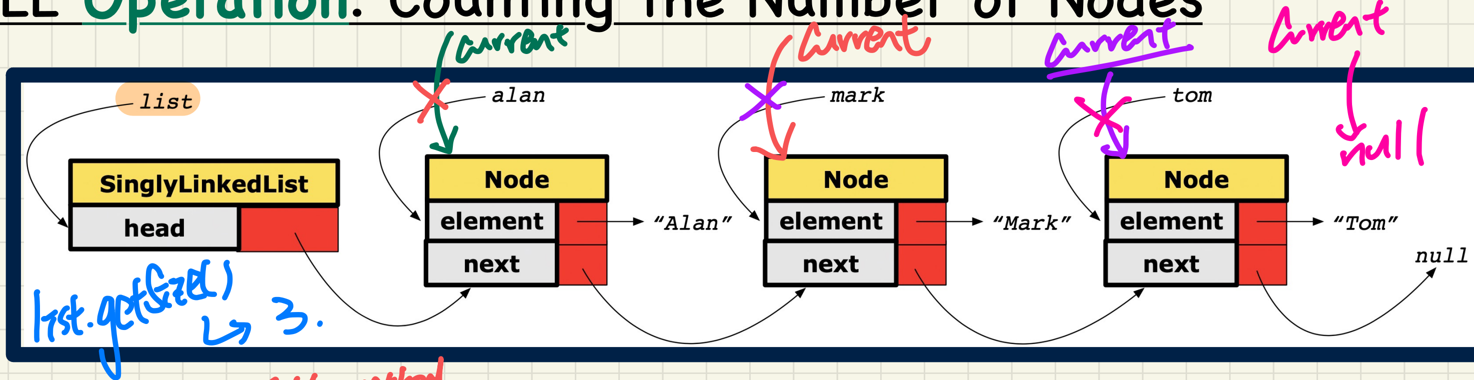
identical to Approach 1.

Lecture

Arrays vs. Linked Lists

Singly-Linked Lists -
Java Implementation: String Lists
Operations on a List

SLL Operation: Counting the Number of Nodes



SLL method

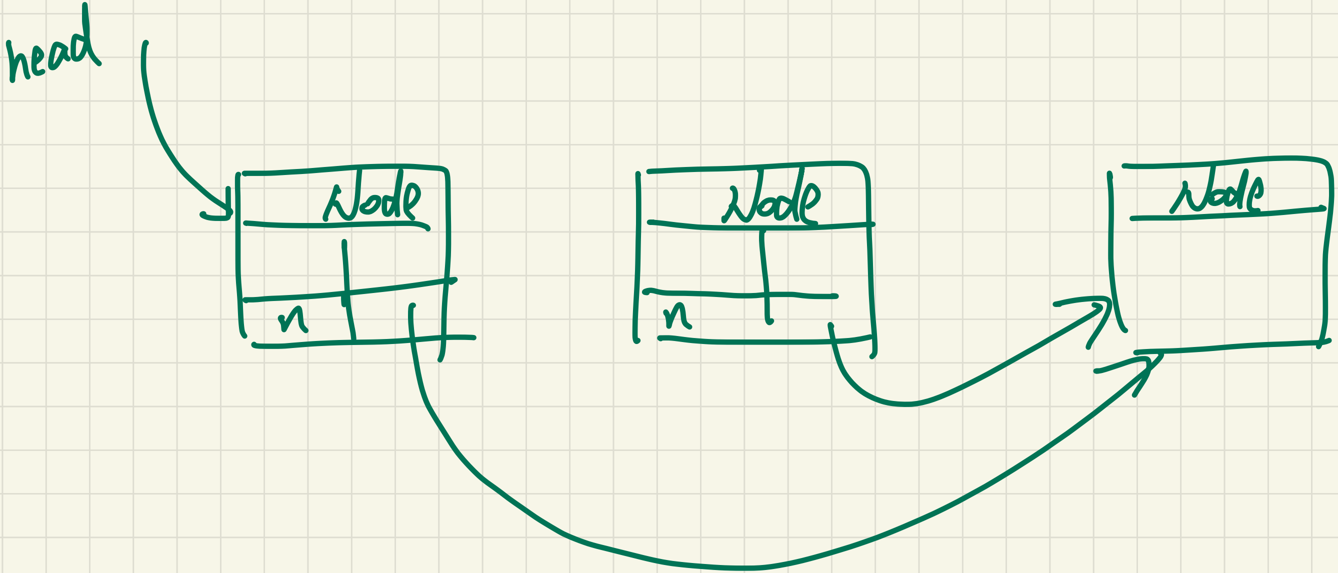
```
int getSize() {
  int size = 0;
  Node current = head;
  while (current != null) {
    current = current.getNext();
    size ++;
  }
  return size;
}
```

exit when current == null

Trace: list.getSize()

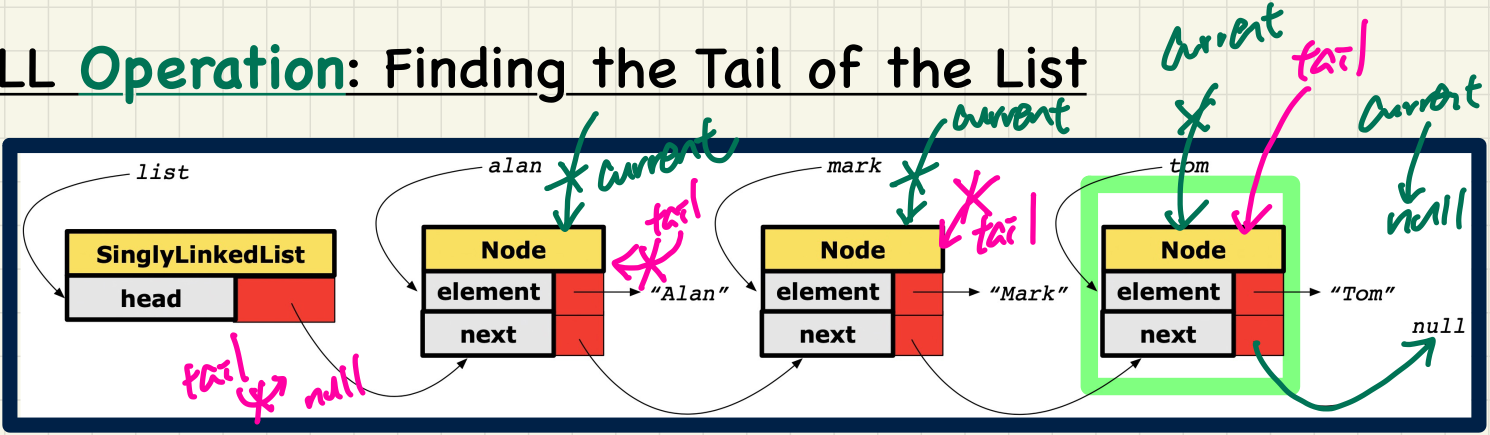
current	current != null	End of Iteration	size
alan	alan != null T	current == mark	1
mark	mark != null T	current == tom	2
tom	tom != null T	current == null	3
null	null != null F		

O(n)



2 .

SLL Operation: Finding the Tail of the List




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

O(n)

Trace: list.getTail()

current	current != null	End of Iteration	tail

SLL class

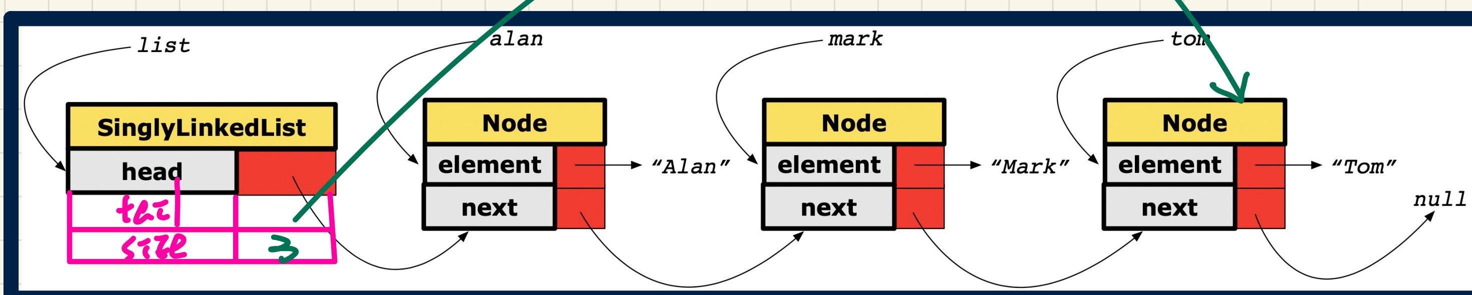
↳ head

↳ tail

↳ size

list.tail
list.size] O(1)

trading size for time.



Lecture 10 - Monday, February 13

Announcements

- Assignment 2 released
  + Required & Recommended Studies
  + Looking Ahead: Programming Test 1
- Assignment 1 solution released

Assume

SLL class : head, tail, size

attributes → $O(1)$

Catch: for methods that might impact the head, tail, or size of a SLL, the body of the method should update these attributes accordingly.

SLL Operation: Inserting to the Front of the List

```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());
}
```

head

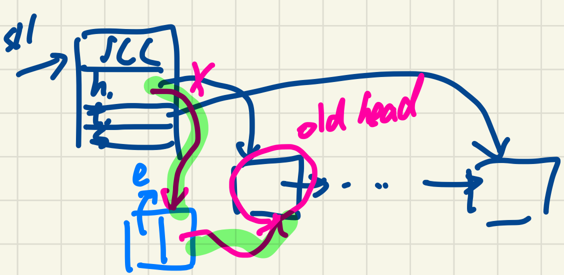
O(1)

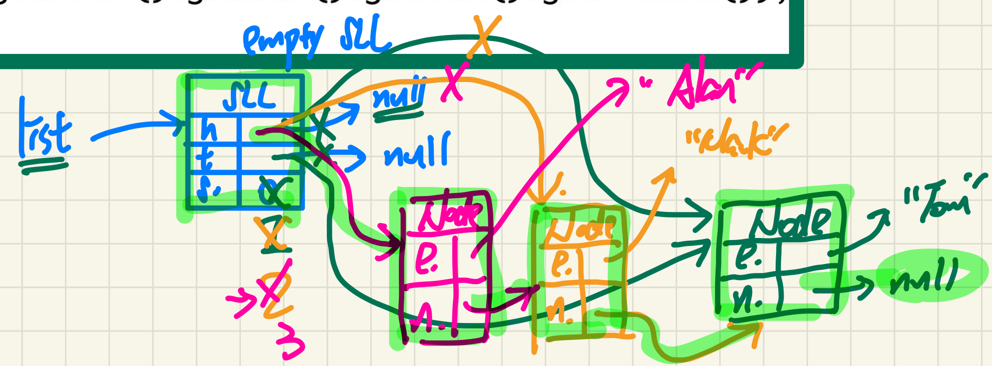
O(1)

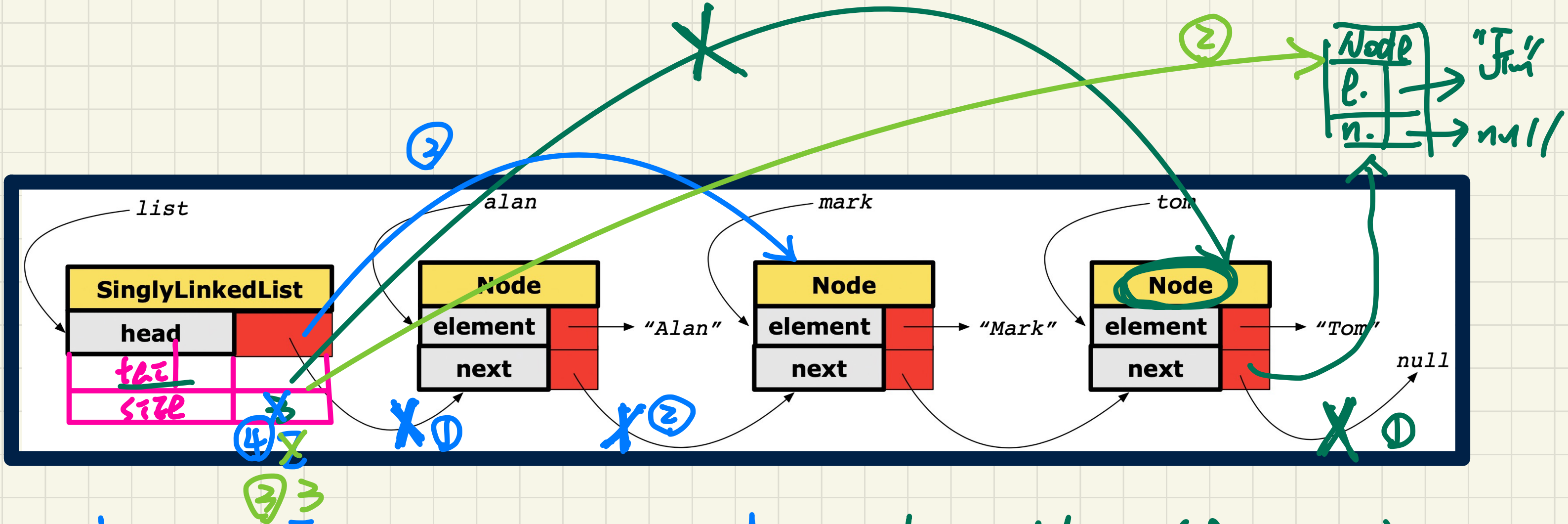
setting head, tail, next refs.

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

attributes updated if necessary.







SLL

void removeFirst( )

① ② ③ ④

If size == 1
↳ after removal, list becomes empty
↳ tail = null

If size == 0
↳ throw some exception.

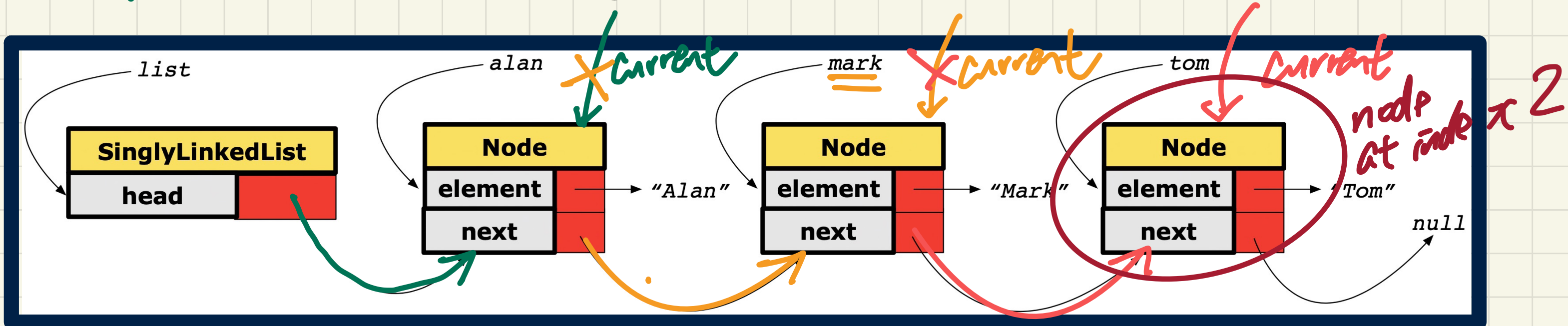
void addLast(String e)

list.addLast("Jim")

①, ②, ③

if size == 0
addFirst.

SLL Operation: Accessing the Middle of the List



SLL class

2

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

i valid

exit: index ≥ i

Trace: list.getNodeAt(2)

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

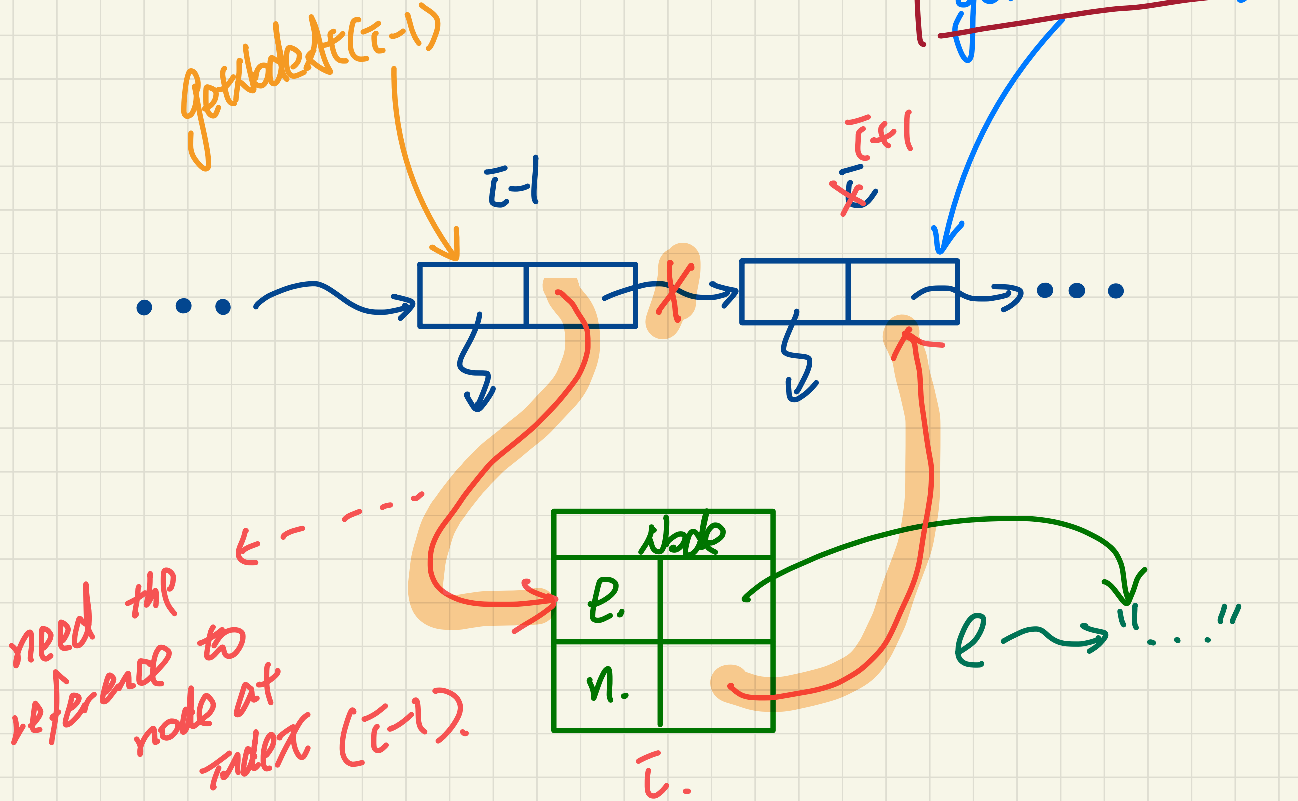
RT: worst is when i = list.size() - 1 $O(n)$

Idea of Inserting a Node at index i

"useless!"

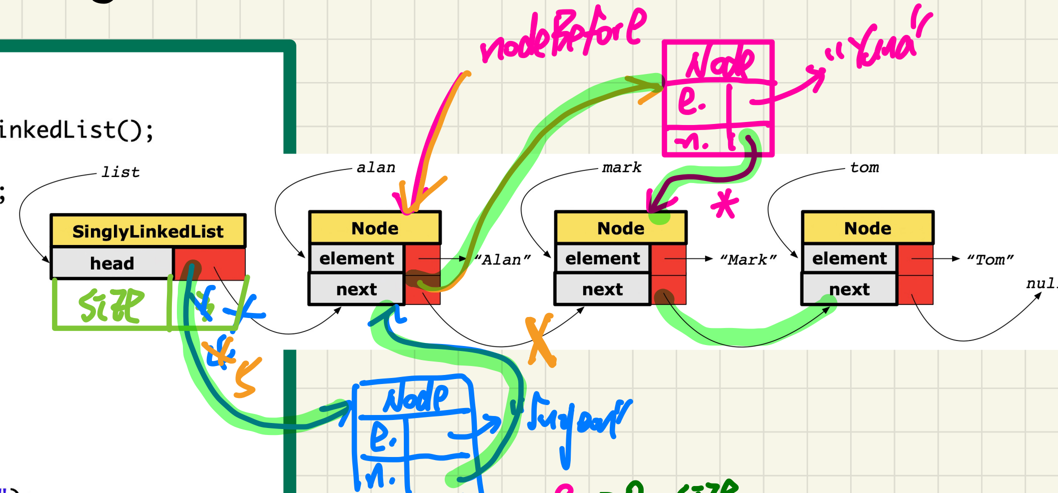
$\boxed{\text{getNodeAt}(i)}$

Case: addAt(i, e), where i > 0



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());
}
```



2 → 0..size

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

getNodeAt(1)

O(n)

O(n)
↳ dominated by finding node at index (i-1)

Lecture 11 - Wednesday, February 15

Announcements

- Assignment 2 released
  + Required & Recommended Studies
  + Looking Ahead: Programming Test 1
    - Monday, Feb. 27; during class time; WSC; 1 hour
    - Covers:
      * Assignment 1 (recursion)
      * Assignment 2 (generic SLL) → strings.
- Assignment 1 solution released

SLL Operation: Removing the End of the List

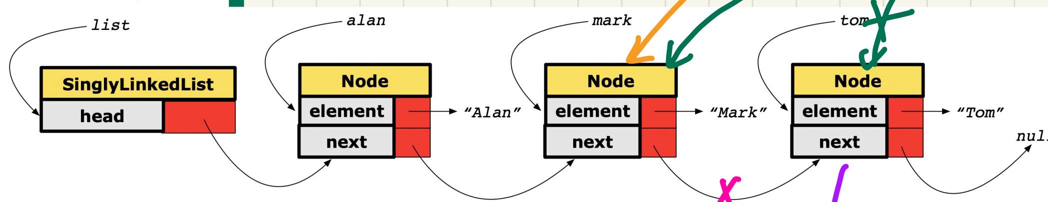
```java
@Test
public void testSLL_removeLast() {
    SinglyLinkedList list = new SinglyLinkedList();
    assertTrue(list.getSize() == 0);
    assertTrue(list.getFirst() == null);

    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());
}
```

O(n)

3 - 2 = 1

secondLastNode

tail



will be garbage-collected.

2

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

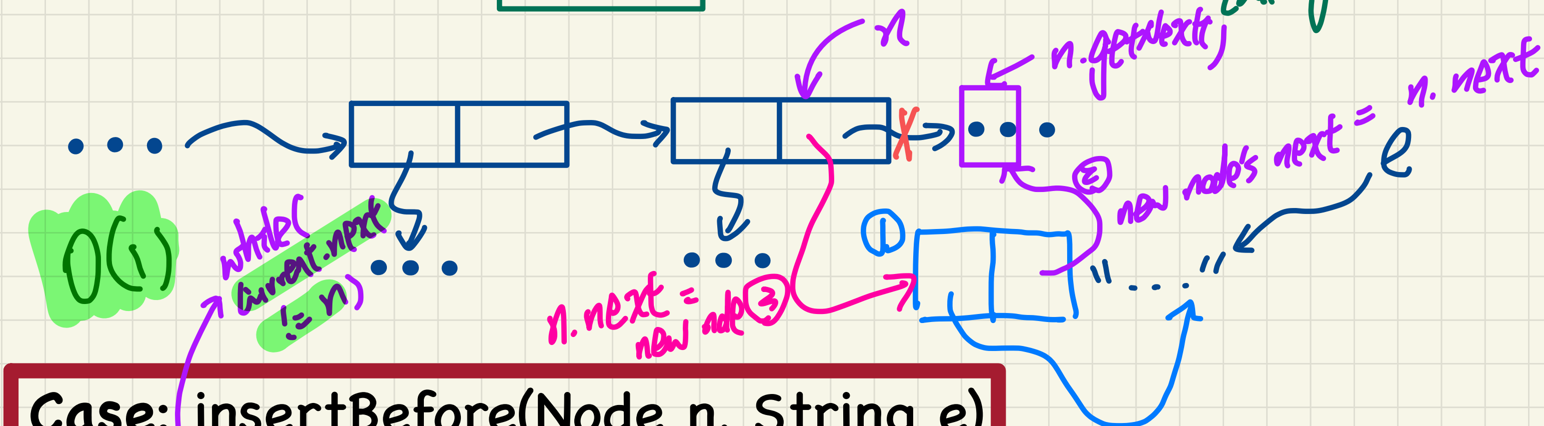
O(n)

size-1: index of last node

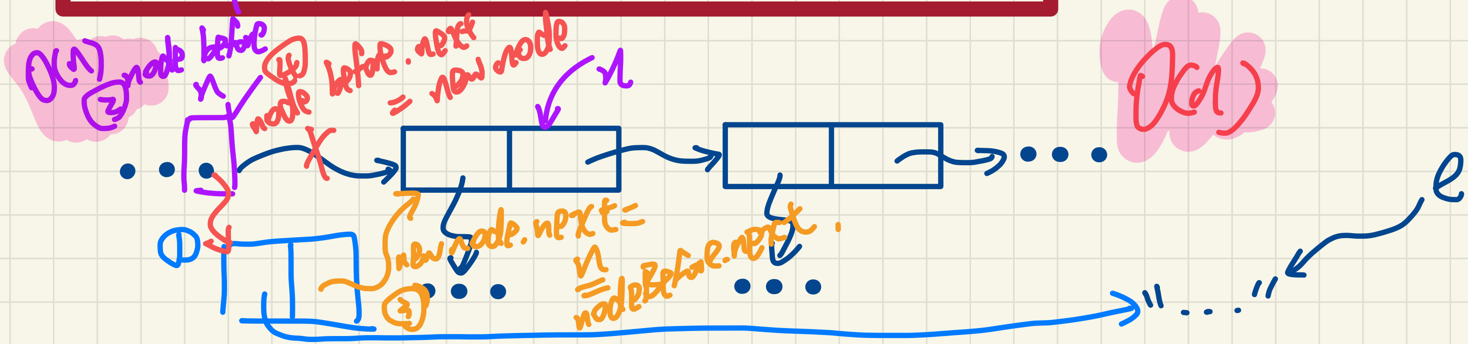
Exercises: insertAfter vs. insertBefore

Case: insertAfter(Node n, String e)

given as input
no need to call getNodeAt



Case: insertBefore(Node n, String e)



Lecture

Arrays vs. Linked Lists

Singly-Linked Lists -
Comparing Arrays and Singly-Linked Lists

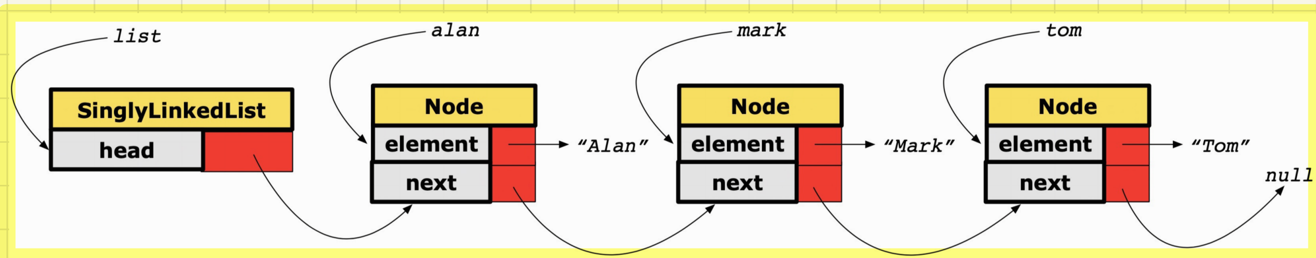
Running Time: Arrays vs. Singly-Linked Lists

<table>
<tr><th colspan="2">Operation \ Data Structure</th><th>Array</th><th>Singly-Linked List</th></tr>
<tr><td colspan="2">get size</td><td colspan="2" rowspan="2">O(1)</td></tr>
<tr><td colspan="2">get first/last element</td></tr>
<tr><td colspan="2">get element at index i</td><td rowspan="2">O(1)</td><td rowspan="2">O(n)</td></tr>
<tr><td colspan="2">remove last element</td></tr>
<tr><td colspan="2">add/remove first element, add last element</td><td rowspan="3">O(n)</td><td rowspan="2">O(1)</td></tr>
<tr><td rowspan="2">add/remove i^{th} element</td><td>given reference to $(i-1)^{th}$ element</td></tr>
<tr><td>not given</td><td>O(n)</td></tr>
</table>

size, head, tail

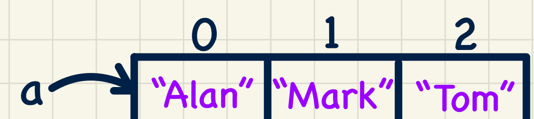
no resizing for array

a[a.length-1] = null



SLL: remove ith node

↳ given the ref to (i-1)th node



Lecture

Arrays vs. Linked Lists

Doubly-Linked Lists - Intuitive Introduction

Why DLL ?

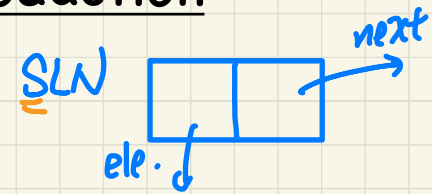
1. performance (e.g. removeLast)

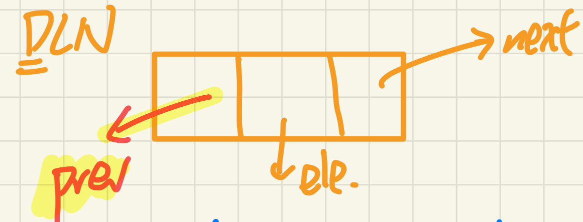
2. code structure

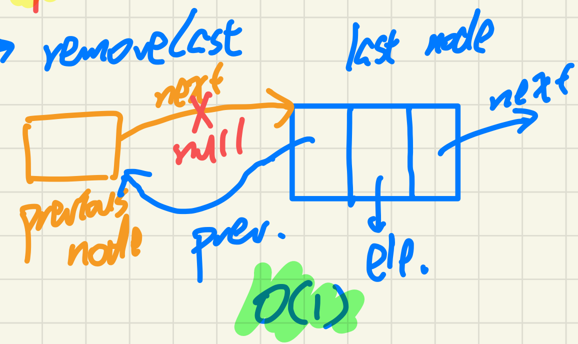
↳ don't need to worry about edge cases

Doubly-Linked Lists (DLL): Visual Introduction

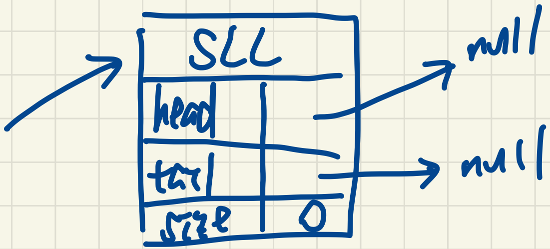
- A chain of bi-directionally connected nodes
- Each node contains:
  DLN
  + reference to a data object
  + reference to the next node
  + reference to the previous node
- A DLL is also a SLL:
  + many methods implemented the same way
  + some method implemented more efficiently
- Accessing a node in a list:
  + Relative positioning: O(n)
    having the prev. ref. does not help.
  + Absolute indexing: O(1)
- The chain may grow or shrink dynamically.
- Dedicated Header vs. Trailer Nodes
  (no head refeference and no tail reference)



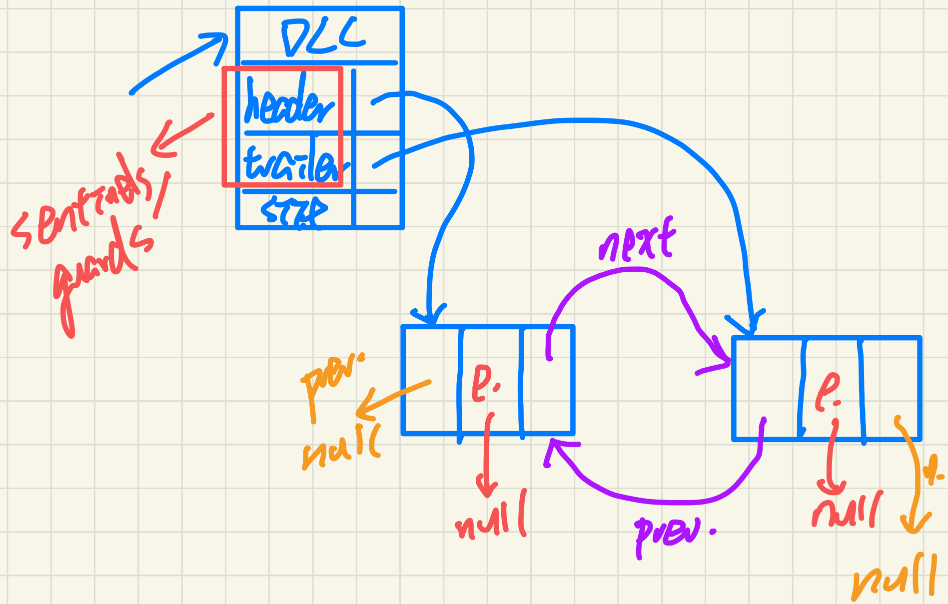




SLL


SLL
head
tail
size
0
null
null

DLL


DLL
header
trailer
size
sentinels /
guards
next
prev.
e.
e.
prev.
null
null
null
null
null

Lecture

Arrays vs. Linked Lists

Doubly-Linked Lists -
Java Implementation: Generic Lists
Initializing a List

Generic DLL in Java: DoublyLinkedList vs. Node

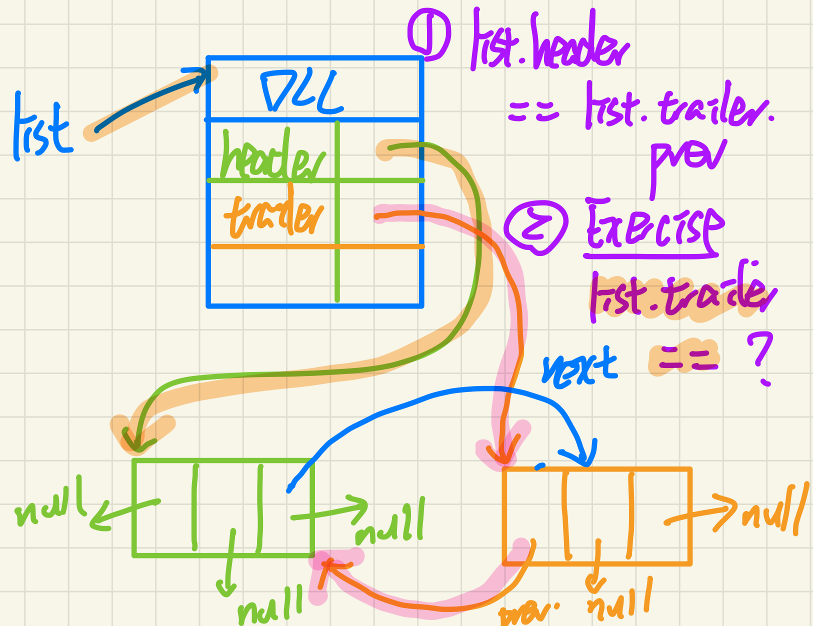
```java
public class DoublyLinkedList<E> {
  private int size = 0;
  public void addFirst(E e) { ... }
  public void removeLast() { ... }
  public void addAt(int i, E e) { ... }
  private Node<E> header;
  private Node<E> trailer;
  public DoublyLinkedList() {
    header = new Node<>(null, null, null);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

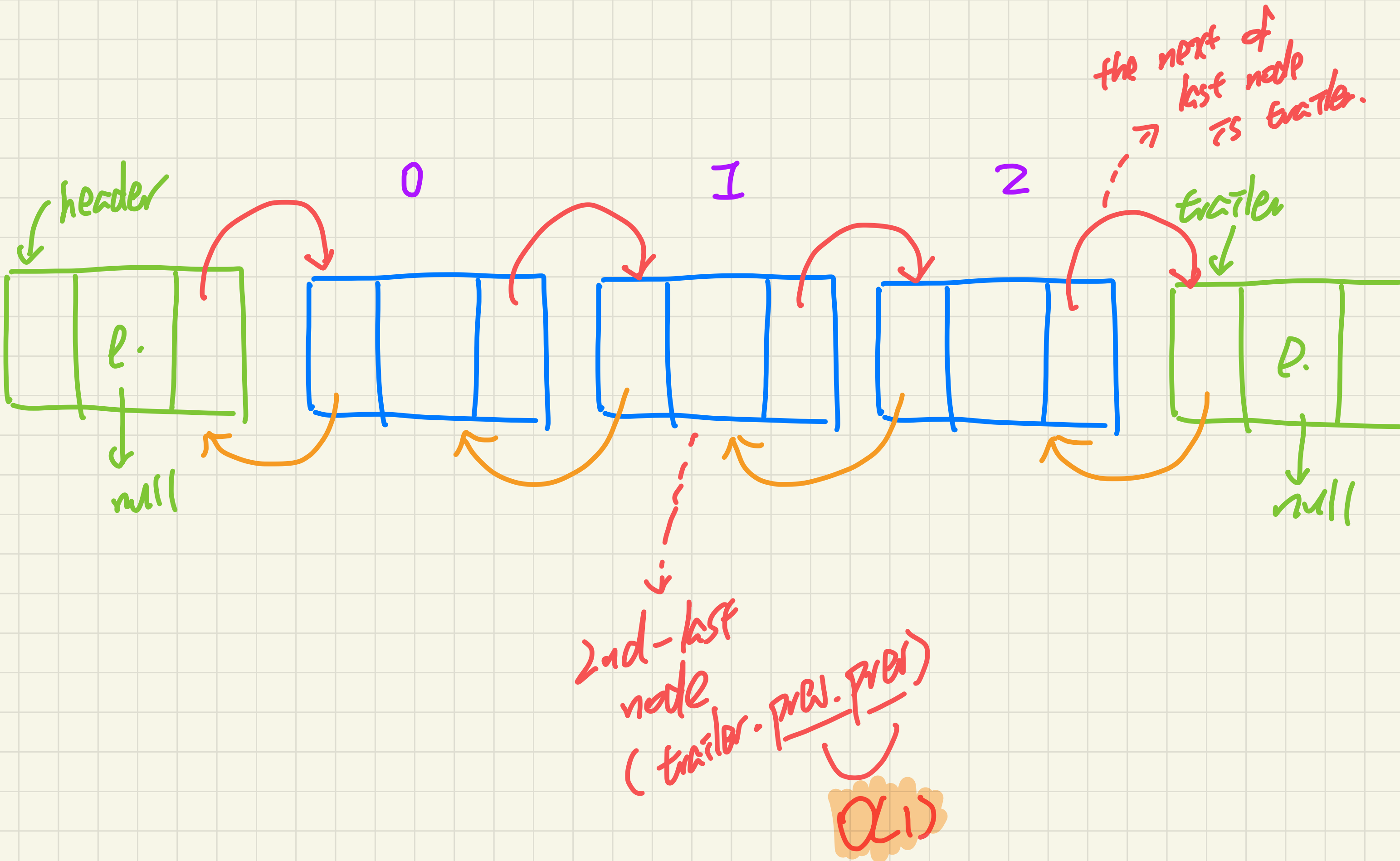
```java
@Test
public void test_String_DLL_Empty_List() {
  DoublyLinkedList<String> list = new DoublyLinkedList<>();
  assertTrue(list.getSize() == 0);
  assertTrue(list.getFirst() == null);
  assertTrue(list.getLast() == null);
}
```

```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;
  }
}
```

DLN

call by value





the next of last node is trailer.
header
0
1
2
trailer
e.
e.
null
null
2nd-last node
(trailer . prev. prev)
O(1)

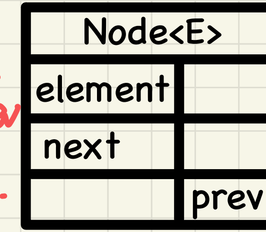
Lecture

Arrays vs. Linked Lists

Doubly-Linked Lists -
Java Implementation: Generic Lists
Operations on a List

Generic DLL in Java: Inserting between Nodes

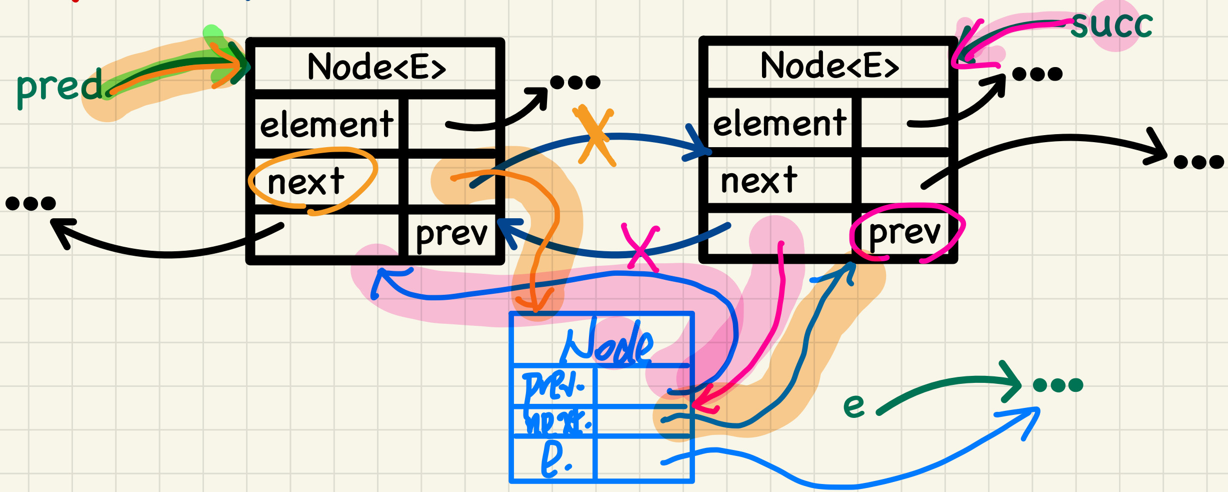
assumption: pred.next == succ, succ.prev == pred.




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

without these two lines, the new node remains unreachable from the head.

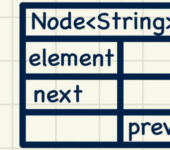
Assumption: pred and succ are directly connected.



Lecture 12 -
Makeup for WrittenTest1

(≈ 90 minutes)

Generic DLL in Java: Inserting to the Front/End

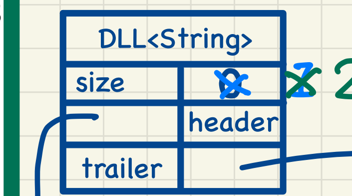


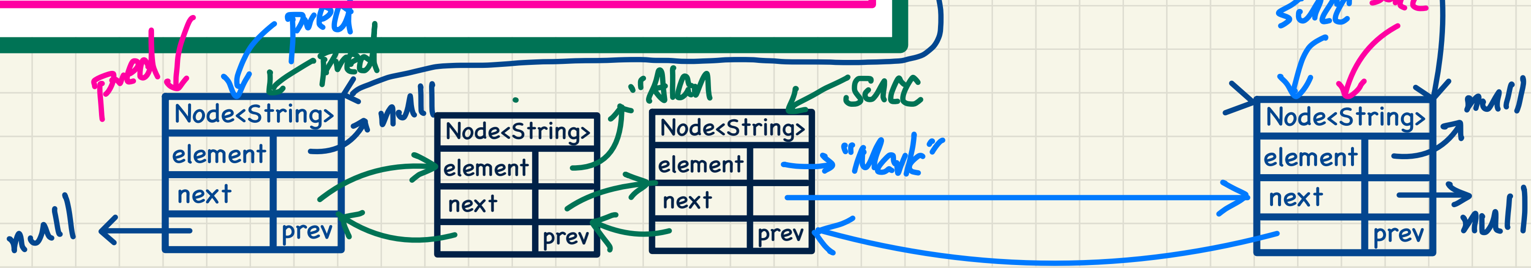

```java
@Test
public void test_String_DLL_Insert_Front_End() {
    DoublyLinkedList<String> list = new DoublyLinkedList<>();
    list.addFirst("Mark");
    list.addFirst("Alan");

    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())
}
```

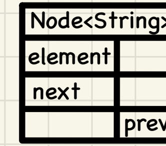
Exercise: Tracing

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

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







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: Tracing.

0

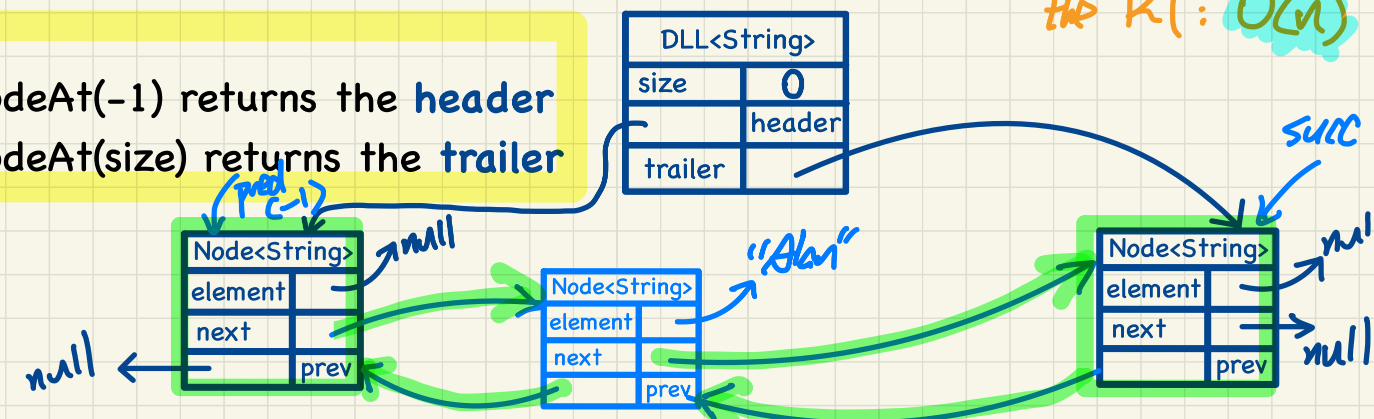
```java
addAt(int i, E e) {
  if (i < 0 || i > size) {
    throw new IllegalArgumentException
  else {
    Node<E> pred = getNodeAt(i - 1);
    Node<E> succ = pred.getNext();
    addBetween(e, pred, succ);
  }
}
```

0-1

still dominates the RT: O(n)

Notes.

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



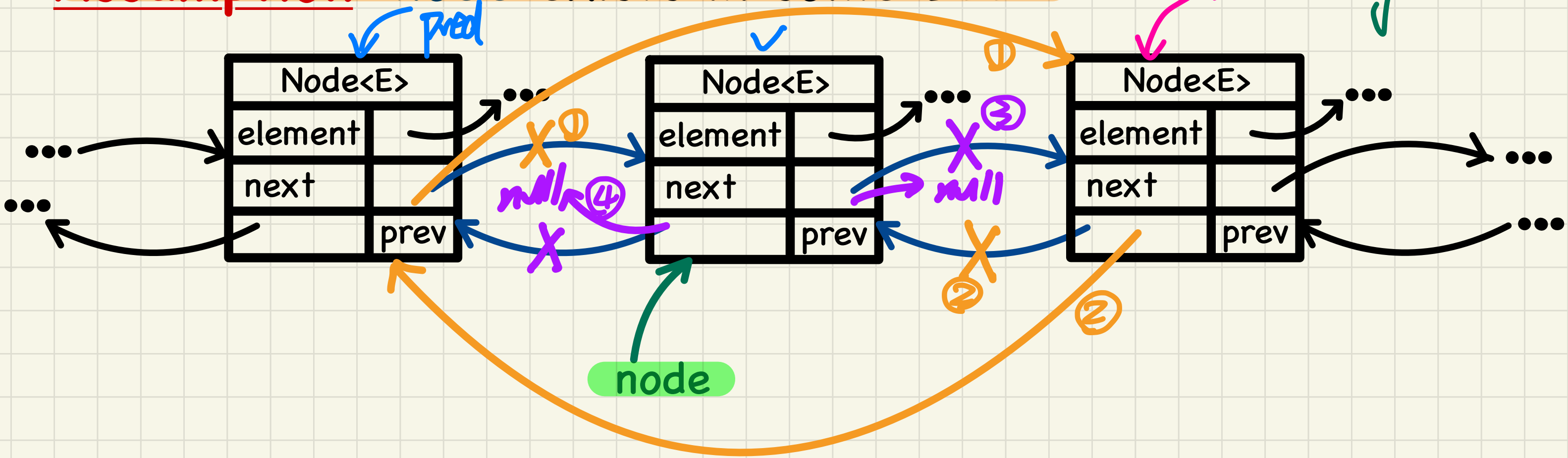
Generic DLL in Java: Removing a Node

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

RT: O(1)

efficient solely because the ref. of the node to remove is given.

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);
}
```

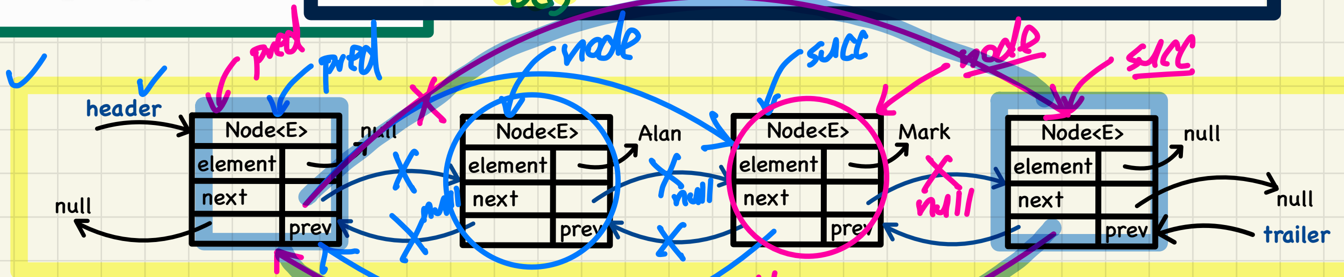
```java
void removeFirst() {
  if (size == 0) { throw new IllegalArgumentException("Empty"); }
  else { remove(header.getNext()); }
}
```

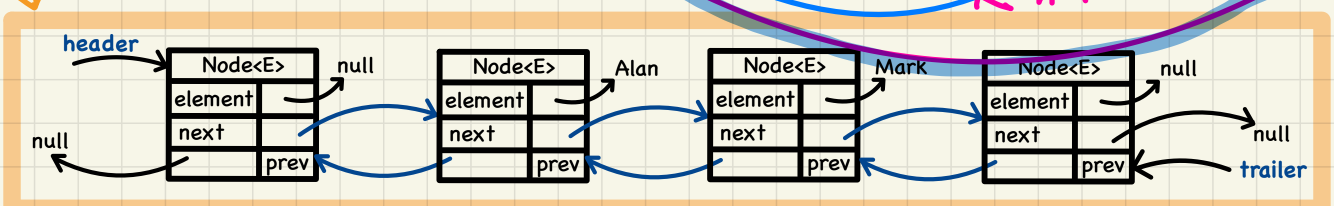
O(1) first node

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

O(1)

Exercise: Tracing





Generic DLL in Java: Removing from the Middle

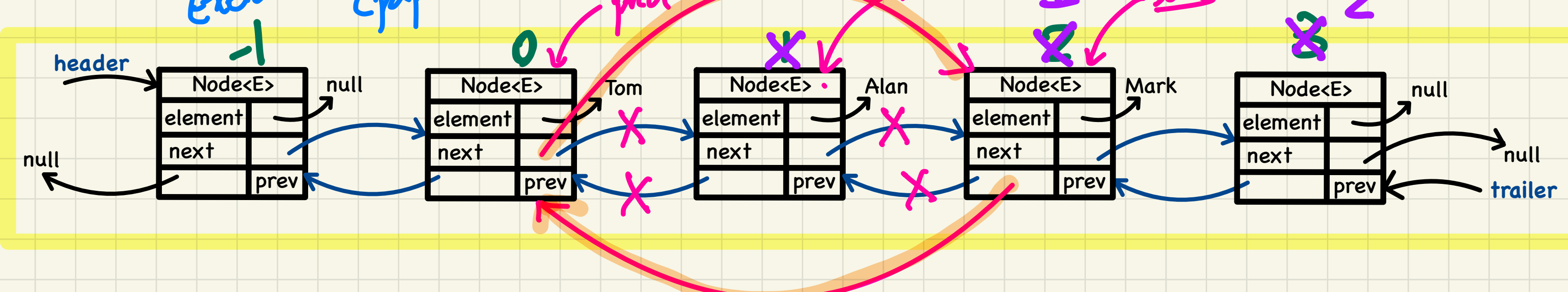
```java
@Test
public void test_String_DLL_removeAt() {
    DoublyLinkedList<String> list = new DoublyLinkedList<>();
    list.addFirst("Mark");
    list.addFirst("Alan");
    list.addFirst("Tom");
    assertTrue(list.getSize() == 3);
    list.removeAt(1);
    assertTrue(list.getSize() == 2);
    list.removeAt(0);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

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

dominates RT: O(n)

Contrast
SLL removeAt:
getNodeAt
(i-1)

Exercise: Tracing (paper & Eclipse).



Lecture 2

Part K

Doubly-Linked Lists -
Comparing Arrays, SLL, and DLL

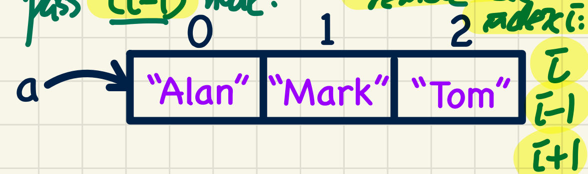
Running Time: Arrays vs. SLL vs. DLL

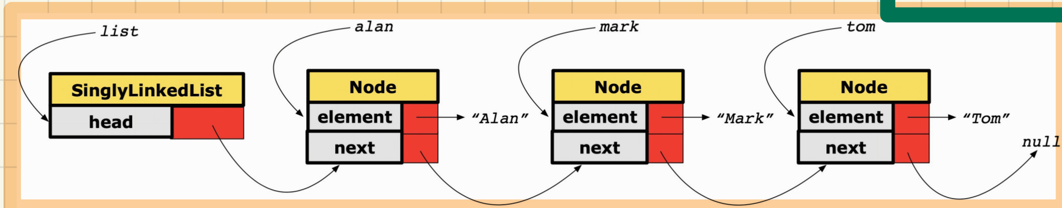
see discussion at the end of SLL.

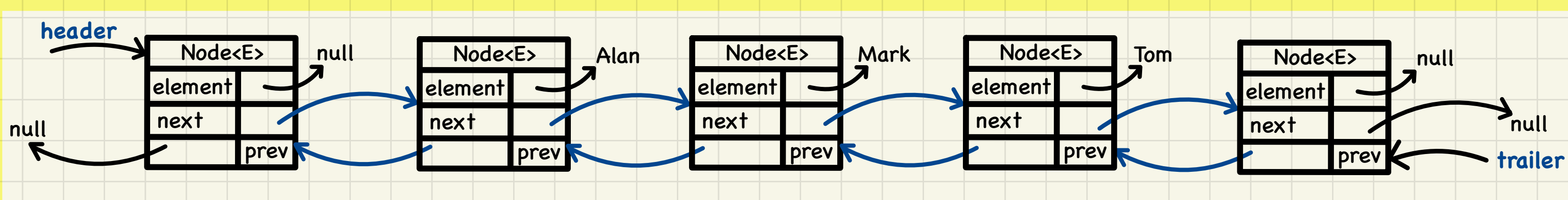
Data Structure / Operation		Array	Singly-Linked List	Doubly-Linked List
size			O(1)	
first/last element				
element at index i		O(1)	O(n)	O(n)
remove last element				
add/remove first element, add last element				O(1)
add/remove i^{th} element	given reference to $(i-1)^{th}$ element	O(n)	O(1) remove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

pass (i-1) node.

remove at index i:







Lecture

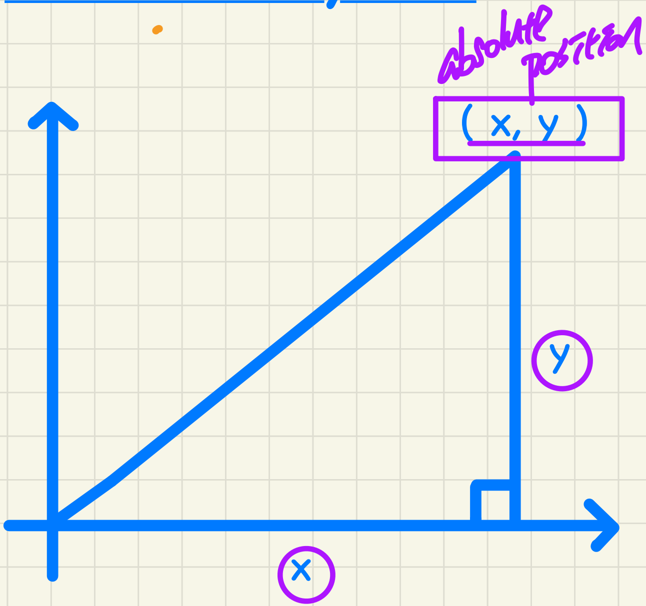
Implementing ADT in Java

Interfaces

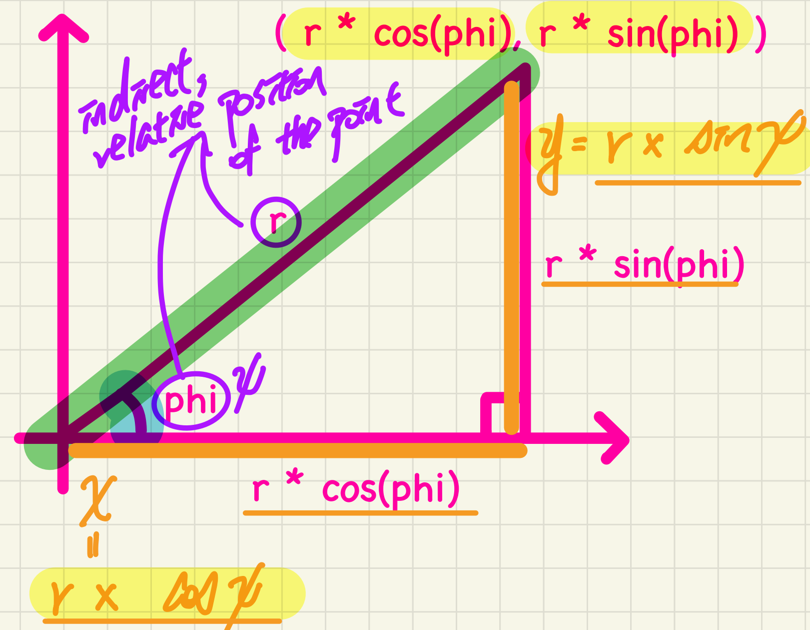
Representations of 2-D Points: Cartesian vs. Polar

Goal: Dynamically, switch between two systems seamlessly.

Cartesian System



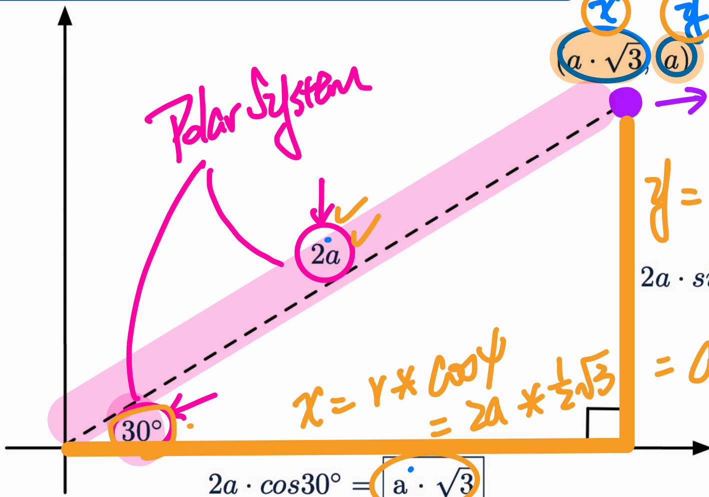
Polar System



Example: Cartesian vs. Polar

Recall: $sin30° = \frac{1}{2}$ and $cos30° = \frac{1}{2} \cdot \sqrt{3}$

Cartesian System

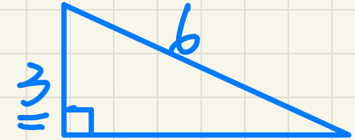


→ point to represent

$y = r \times \sin\psi = 2a \times \frac{1}{2} = \underline{\underline{a}}$

$x = r * \cos\psi = 2a * \frac{1}{2}\sqrt{3} = a \cdot \sqrt{3}$

$a = 3$



$\underline{\underline{3 \cdot \sqrt{3}}}$

$(3)^2 + (3 \cdot \sqrt{3})^2 = 6^2$

We consider the same point represented differently as:

- $r = 2a$, $\psi = 30°$ [polar system]
- $x = 2a \cdot cos30° = a \cdot \sqrt{3}$, $y = 2a \cdot sin30° = a$ [cartesian system]

Interface used as a static type

Interface vs. Implementations

Point p = new Point(); X not valid.

× p.getX() × p.getY()

```
double A = 5;
double X = A * Math.sqrt(3);
double Y = A;
Point p;
p = new CartisianPoint(X, Y); /* polymorphism */
print("(" + p.getX() + ", " + p.getY() + ")");
p = new PolarPoint(2 * A, Math.toRadians(30)); /*
print("(" + p.getX() + ", " + p.getY() + ")");
```

static method.

DT: CartesianPoint

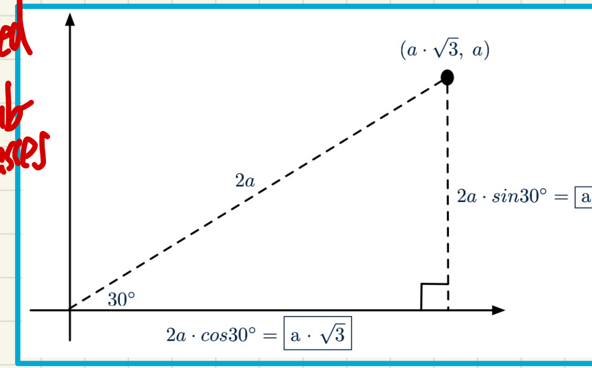
DT: PolarPoint

CartesianPoint	
x	5·√3
y	5

Point p

PolarPoint	
r	10
phi	30°

implementations deferred to sub classes



an abstract class where all methods are abstract

available across packages.

```
public interface Point {
  public double getX();
  public double getY();
}
```

headers of methods

forced to implement

implements

```
public class CartesianPoint implements Point {
  private double x;
  private double y;
  public CartesianPoint(double x, double y) {
    this.x = x;
    this.y = y;
  }
  public double getX() { return x; }
  public double getY() { return y; }
}
```

absolute position

```
public class PolarPoint implements Point {
  private double phi;
  private double r;
  public PolarPoint(double r, double phi) {
    this.r = r;
    this.phi = phi;
  }
  public double getX() { return Math.cos(phi) * r; }
  public double getY() { return Math.sin(phi) * r; }
}
```

relative position

↳ measured in radians.

Lecture 13 - Monday, February 27

Announcements

- Updated semester calendar
- ProgTest1: Guide & PracticeTest
- Makeup Lecture for WrittenTest1
  - Expected to complete by: March 20

Lecture

Stack ADT vs. Queue ADT

Abstract Data Types (ADTs)

Data Structures

trees, binary trees, balanced BT, BST.

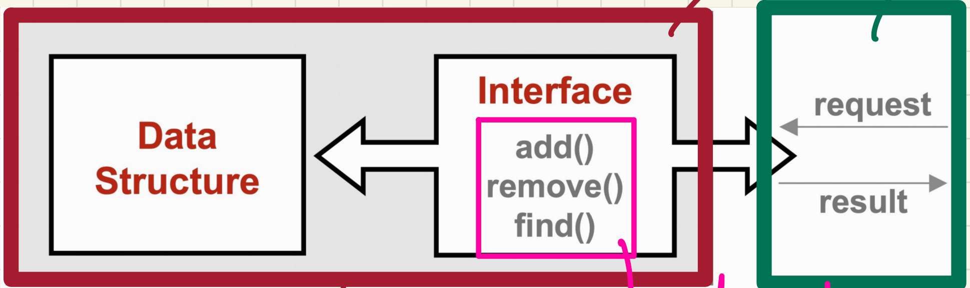
stacks vs queues

Arrays vs. SLLs vs DLLs.

Abstract Data Types (ADTs)

supplier (callee)

client (caller)



1. input types
2. output type
3. description about going from inputs to output

headers of methods

supplier

```
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. */
} }
```

client

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

→ in client MU, use the service "heat" from supplier MW

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

Java API ≈ Abstract Data Types

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>

generic par.

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.

Lecture

Stack ADT vs. Queue ADT

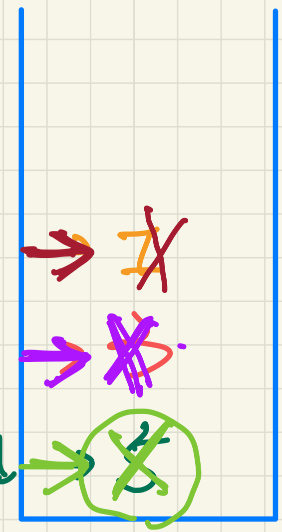
Stack ADT -
Last In First Out (LIFO)
Implementations in Java

Stack ADT: Illustration

	isEmpty	size	top
new stack	T	0	
push(5)	F	1	5
push(3)	F	2	3
push(1)	F	3	1
pop	F	~~3~~ 2	1
pop	F	~~2~~ 1	3
pop	F	~~1~~ 0	5
pop	T	0	

error ∵ precond. not satisfies

order in which elements added: 5, 3, 1



LIFO

order in which elements returned: 1, 3, 5

error ∵ precond. of pop is violated

Implementing the Stack ADT in Java: Architecture

≈ ADT

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

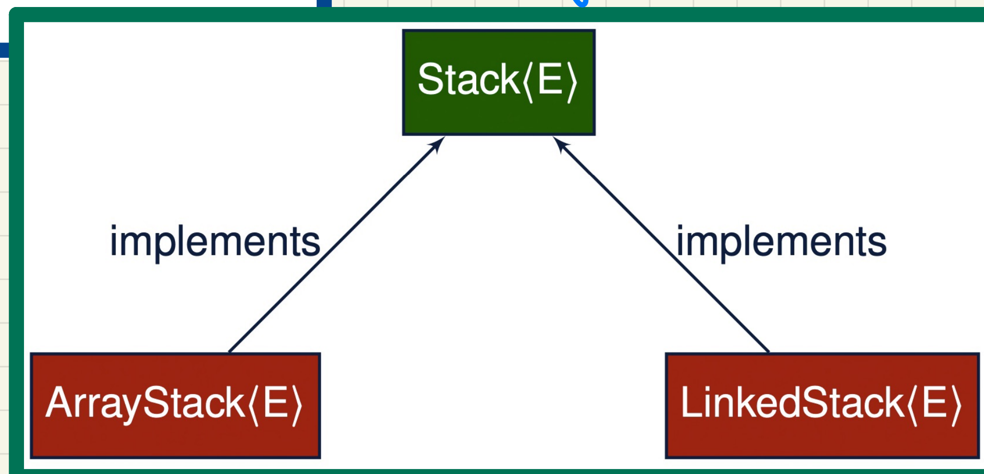
Int. String

generic parameter

wrapper class

Stack<String> s1 = ...

Stack<Integer> s2 = ...



s1.push("A");

s2.push(23);

String v1 = s1.pop();

Integer v2 = s2.pop();

Implementing the Stack ADT using an Array

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

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

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

→ exception to push if stack already full.

→ empty stack → no top.

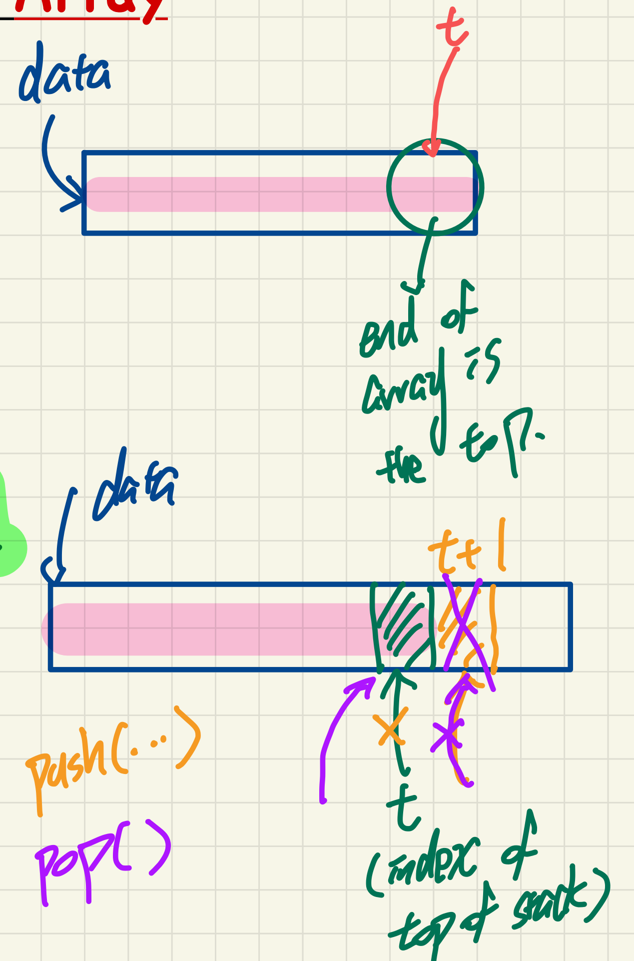
- no loops
- no method calls

↳ O(1) all ops.

→ to relax this constraint ⇒ dynamic array

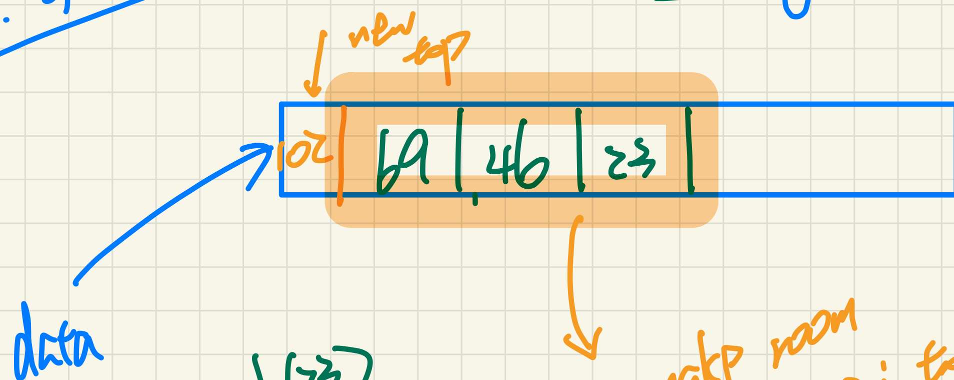
↓ ① doubling strategy

② amortized RT of push: O(1)



alternative imp. of stack using array

beginning of array : top of stack.



push(23)
push(46)
push(69)
push(102).

to make room for the new top, shift all items to right by 1 pos.

$\Rightarrow O(n)$

Implementing the Stack ADT using a SLL

Exercise

- DLL (first is top)
- DLL (last is top)

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

where's the top? first? last?

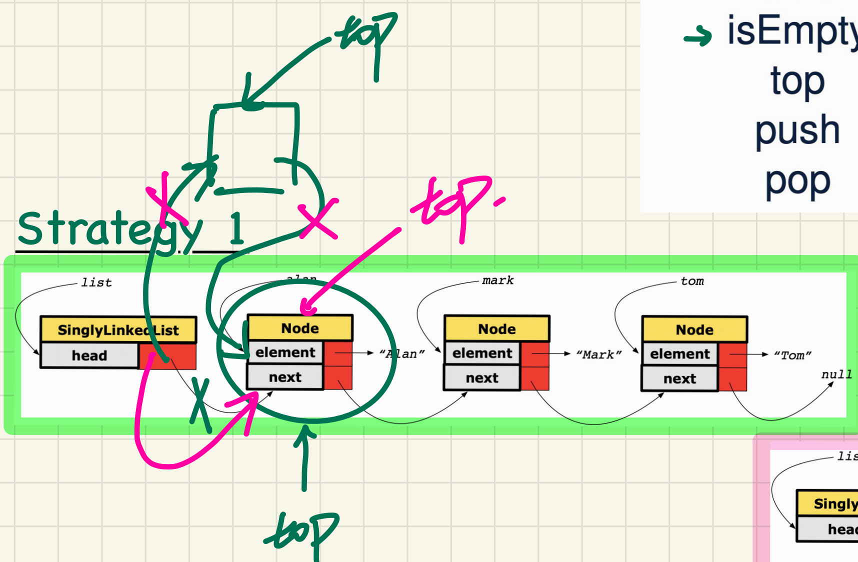
Stack Method	Singly-Linked List Method	
	Strategy 1	Strategy 2
size	list.size O(1)	
isEmpty	list.isEmpty O(1)	
top	O(1) list.first	list.last O(1)
push	O(1) list.addFirst	list.addLast O(1)
pop	list.removeFirst O(1)	list.removeLast O(n)

head

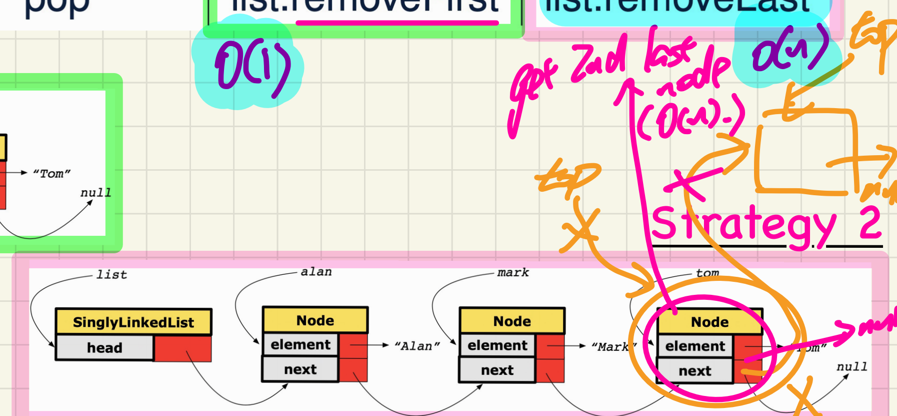
tail

get 2nd last node (O(n))

Strategy 1



Strategy 2



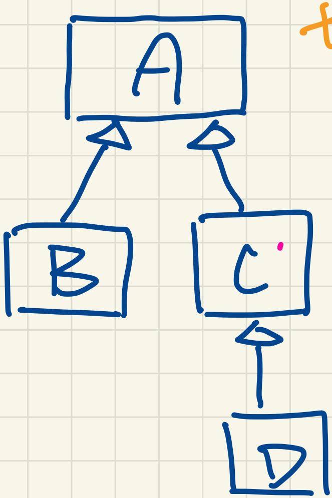
Lecture 14 - Wednesday, March 1

Announcements

- **Makeup Lecture** for WrittenTest1
  + Expected to complete by: March 20
- **A2 solution**: only source code (no solution videos)

Static Type vs. Dynamic Type

declared type



any object of type that's a descendant class of A? is valid

any descendant classes of the ST, including A itself

A oa = new C();

polymorphism

C oc = new D();

static types

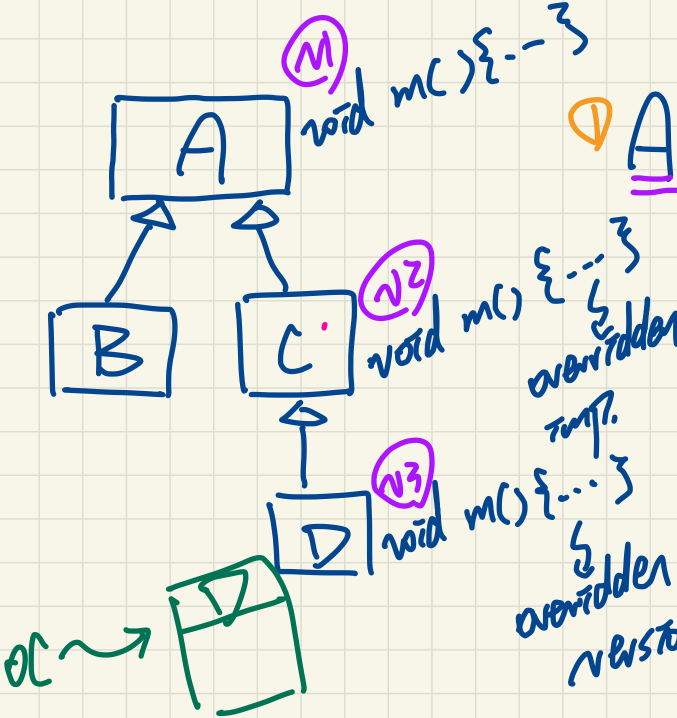
dynamic type

C

oa instance of A (T)

oa instance of C (T)

oa instance of D (F)



polymorphism

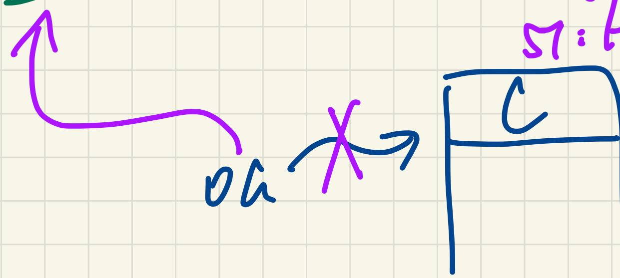
① A oa = new C();

② C oc = new D();

③ oa.m(); invoke version of m in C (dynamic type)

④ oa = oc;

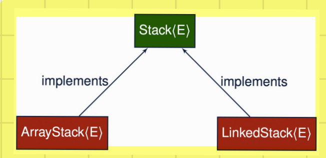
ST: A ST: C



oa.m(); invoke version in D

dynamic binding

Stack ADT: Testing Alternative Implementations



invoke the push imp. in AS class

```java
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;
  }
}
```

ST

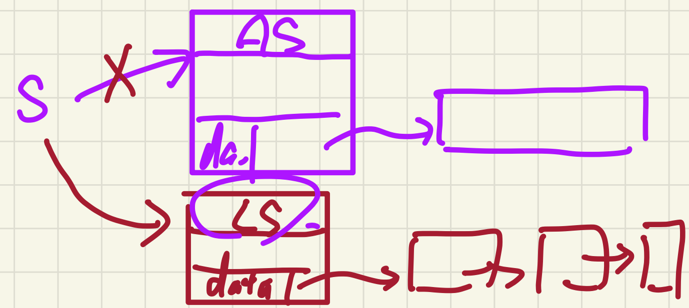
DT1

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

DT2

poly-morphism

invoke the push imp. in LS class



Polymorphic Collection (stack)

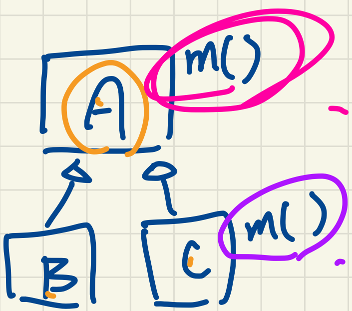
```
public interface Stack<E> {
  public int size();
  public boolean isEmpty();
  public E top();
  public void push(E e);
  public E pop();
}
```

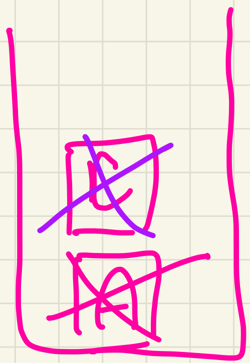
A A A A

Stack<A> s = new ...

* s.push(new A() / new B() / new C()); any descent classes of A

??





s.push(new A());

s.push(new C());

① A obj = s.pop(); ③ obj = s.pop();

DT: C

② obj.m();

DT: A

④ obj.m();

Lecture

Stack ADT vs. Queue ADT

Stack ADT -
Algorithms using the Stack ADT

Algorithm using Stack: Reversing an Array

generic parameter declared at the method level.

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

from L to R

names → | ~~"Alan"~~ | ~~"Mark"~~ | ~~"Tom"~~ |

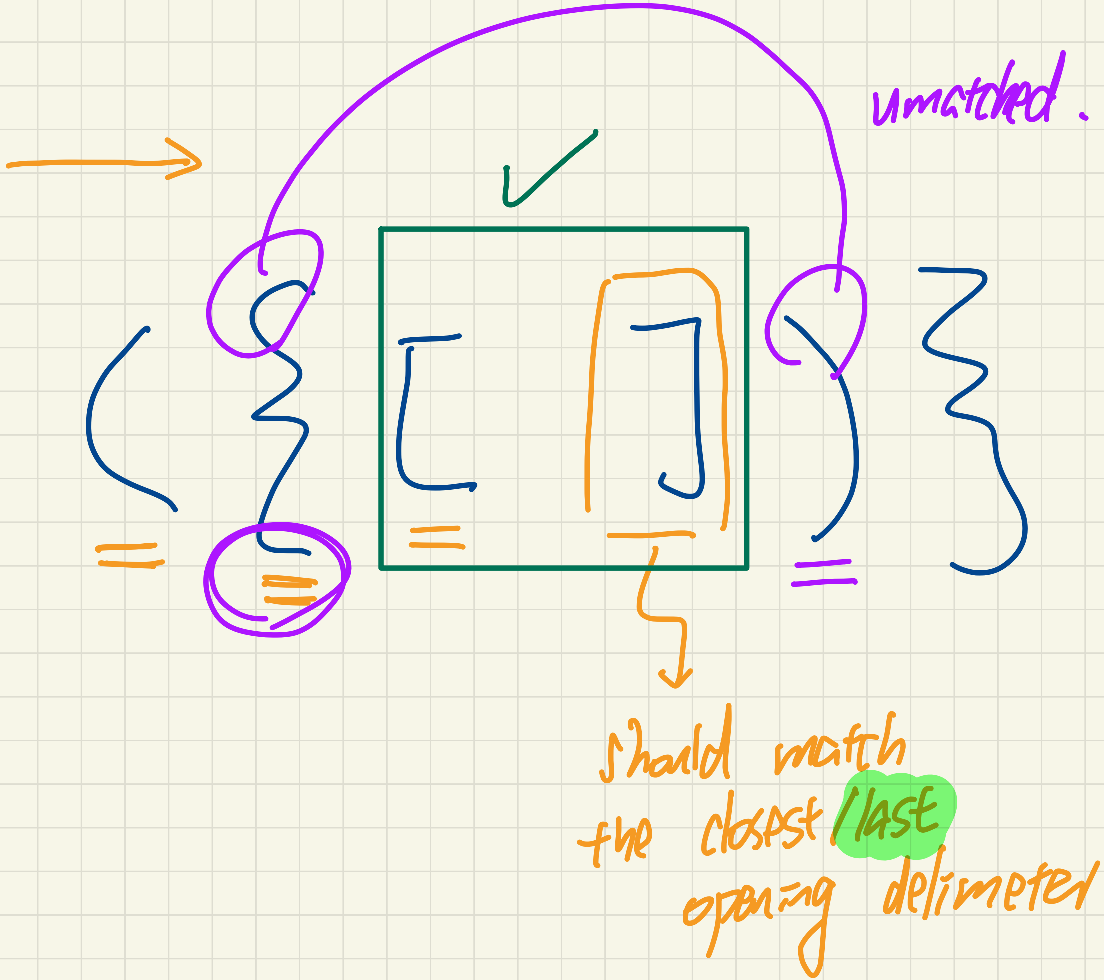
"Tom" "Mark" "Alan"

in-place reverse

~~Tom~~
~~Mark~~
~~Alan~~

buffer

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

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



unmatched.
should match
the closest/last
opening delimeter

Algorithm using Stack: Matching Delimiters

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

0 1 2

RT: O(n)

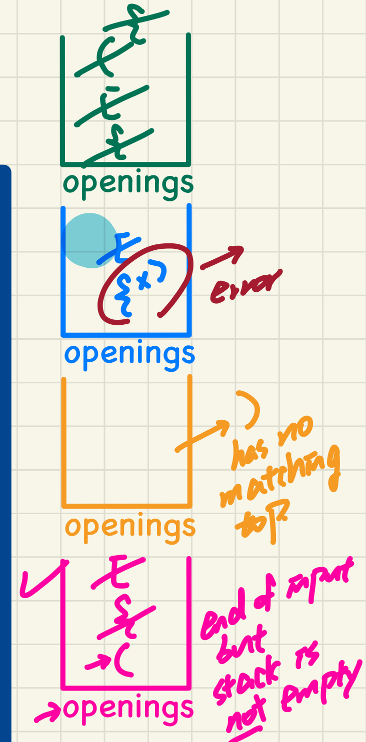
length of input string.

closing not matched by empty stack.

when closing matches open

closing not matched

openings may be non-empty




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

Post-fix notation

operands first, then operator.

$\boxed{3}\ \boxed{4\ 5\ *}\ -$

$3\ \boxed{4\ 5\ -}\ *$

$3 * (4 - 5)$

$\boxed{3\ 4\ -}\ 5\ *$

$(3 - 4) * 5$

$5\ \boxed{3\ 4\ -}\ *$

$5 * (3 - 4)$

$(3 - 4) * 5$

Infix notation

$\boxed{3} - \boxed{(4 * 5)}$

operator

operands

Algorithm using Stack: Calculating Postfix Expressions

Sketch of Algorithm

- When input is an ***operand*** (i.e., a number), ***push*** it to the stack.
- When input is an ***operator***, obtain its two ***operands*** by ***popping*** off the stack **twice**, evaluate, then ***push*** the result back to stack.
- When finishing reading the input, there should be **only one** number left in the stack.

Input 1: 3 4 5 * – ≡ 3 – (4 * 5)

Input 2: 3 4 – 5 * ≡ (3 – 4) * 5

Input 3: 5 2 3 + * + ≡ + 5 * (2 + 3)

Input 4: 5 4 + 6 ≡ 5 + 4 6

34 46 56 * –

? + 25

2 + 3 = 5

5 * 5 = 25

20

4 * 5

LHS RHS

3 – 20

= –17

Lecture

Stack ADT vs. Queue ADT

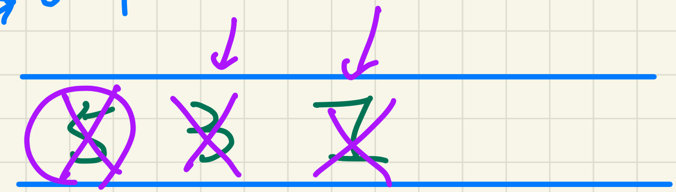
Queue ADT -
First In First Out (FIFO)
Implementations in Java

Queue ADT: Illustration

First-In First-Out

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

exception!

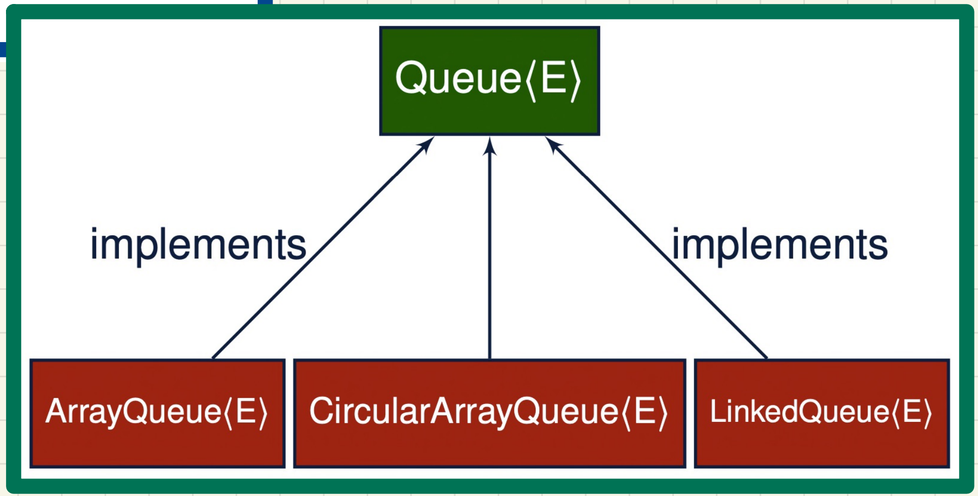


the earlier an element joins a queue, the earlier it gets removed.

exception!

Implementing the Queue ADT in Java: Architecture

```java
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

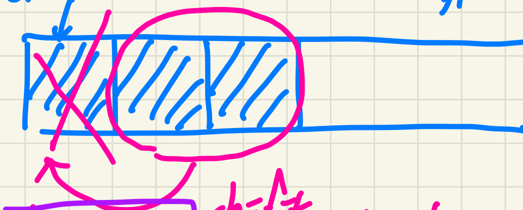
```java
public class ArrayQueue<E> implements Queue<E> {
  private final int MAX_CAPACITY = 1000;
  private E[] data;
  private int r; /* rear index */
  public ArrayQueue() {
    data = (E[]) new Object[MAX_CAPACITY];
    r = -1;
  }
  public int size() { return (r + 1); }
  public boolean isEmpty() { 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(n)

data

(beginning of array → first of Q)



dequeue

shift to left by one position

O(n)

to resolve this → circular array

O(1).

Lecture 15 -
Makeup for ProgTest1

(≈ 90 minutes)

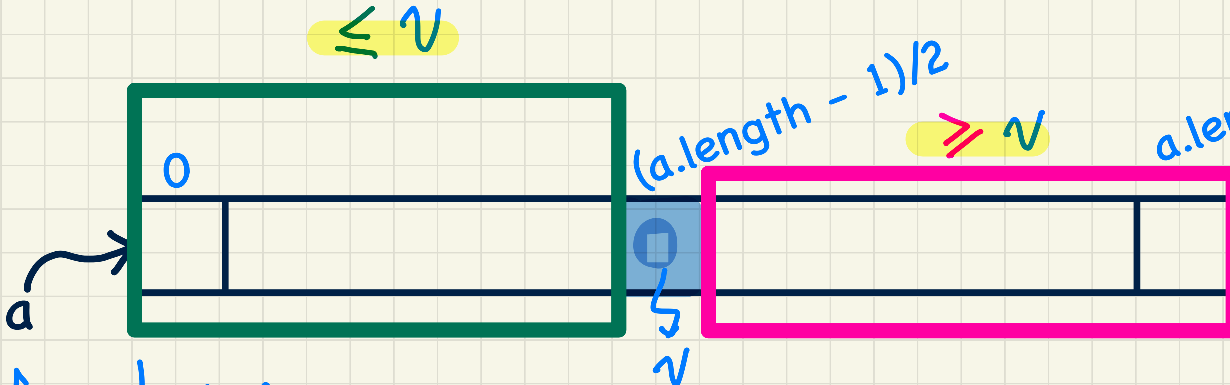
Lecture

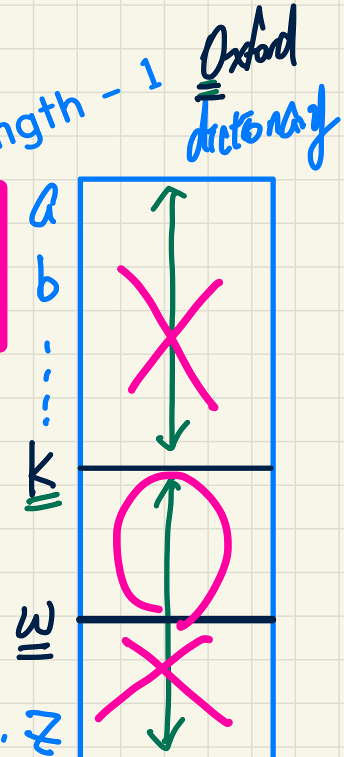
Recursion: Part II

Examples on Recursion
Binary Search

Binary Search: Ideas

Precondition: Array sorted in non-descending order





size of search space
n
n/2
n/4
...
1

log n times to narrow the s.p. to size 1

Search: Does key k exist in array a?

① Keep accessing the middle of the search space (halved each time)

② Recur on the left: k < m.
Recur on the right: k > m

Binary Search in Java

input array sorted

```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; }
  }
}
```

call by value

define the range of indices of the search space.

recursive case

key == middleValue

narrowed search spaces represent a strictly smaller problem to solve.

Binary Search: Tracing

0	1	2	3	4	5	6	7	8
3	6	9	12	15	18	21	24	27

a

initial search space

search space of 1st recursion

search space of 2nd recursion

key

search(a,18)

binarySearchH(a,0,8,18)

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

m.v. $a[4] == 15$

$>$

binarySearchH(a,5,8,18)

m.+1 to.

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

m.v. $a[6] = 21$

binarySearchH(a,5,5,18)

↳ true

from m.-1

Exercise

0	1	2	3	4	5	6	7	8
3	6	9	12	15	18	21	24	27

a

search(a,7)

binarySearchH(a,0,8,7)

binarySearchH(a,0,3,7)

binarySearchH(a,2,3,7)

binarySearchH(a,2,1,7)

Running Time: Ideas

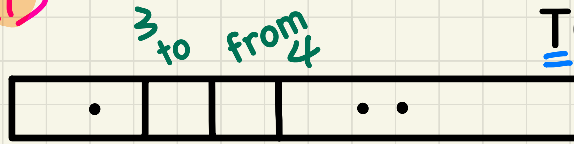
Recurrence Relation

```
boolean allPositive(int[] a) { return allPosH (a, 0, a.length - 1);
boolean allPosH (int[] a, int from, int to) {
  if (from > to) { return true; }
  else if (from == to) { return a[from] > 0; }
  else { return a[from] > 0 && allPosH (a, from + 1, to); } }
```

O(1) O(1) O(1)

subproblem of size n-1

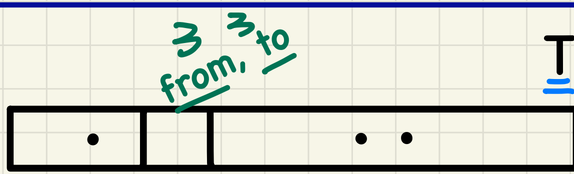
Base Case:
Empty Array



$T(0) = 1$

[4, 3] has 0 numbers

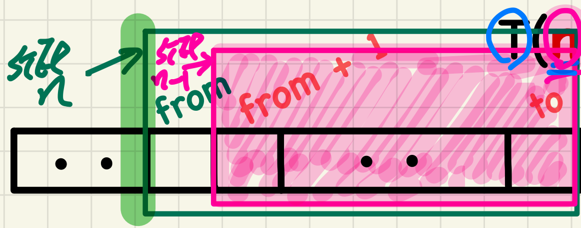
Base Case:
Array of Size 1



$T(1) = 1$

[3, 3] has 1 number

Recursive Case:
Array of size > 1



$T(n) = T(n-1) + 1$

strictly smaller problem

Running Time: Unfolding Recurrence Relation

$$T(0) = 1$$
$$T(1) = 1$$
$$T(n) = T(n-1) + 1$$

recurrence relation derived from Java imp. of recursive algorithm.

$$T(n) = T(n-1) + 1$$

$T(n-1)$

$$= \underbrace{T((n-1)-1) + 1}_{n-2} + 1$$

$T(n-2)$

$$= T(\underbrace{(n-2)-1}_{n-3}) + 1 + 1 + 1$$

$$= \dots \; n-(n-1)$$

$$= T(1) + \underbrace{1 + 1 \dots + 1}_{(n-1)}$$

How many?

$$= 1$$

$$\therefore T(n) = (n-1) + 1$$
$$= n$$

$$O(n)$$

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

Running Time as a Recurrence Relation

$T(0) = 1$

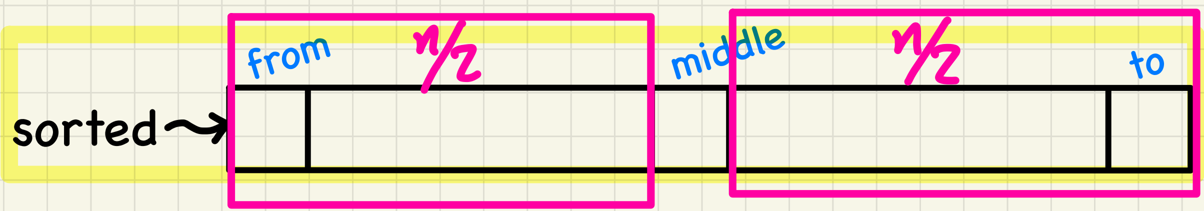
$T(1) = 1$

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

wrong:

$T(n) = \underbrace{T(\frac{n}{2})}_{L} + \underbrace{T(\frac{n}{2})}_{R}$

either L or R but not both



Running Time: Unfolding Recurrence Relation

$T(0) = 1$

$T(1) = 1$

once reaching here, no more unfoldings

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

$\boxed{\text{Assume : } n = 2^x \text{ for } x \geq 0}$

↳ without loss of generality.

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

$T\left(\frac{n}{2}\right)$

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

$$= \left(T\left(\frac{n}{4}\right) + 1\right) + 1$$

$T\left(\frac{n}{4}\right)$

$\frac{n}{2^2}$

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

$T\left(\frac{n}{8}\right)$

$\frac{n}{2^3}$

$$= \left(T\left(\frac{n}{16}\right) + 1\right) + 1 + 1 + 1$$

$\frac{n}{2^4}$

$\vdots$

$$= T(1) \underbrace{+1 +1 \cdots +1}_{\text{How many? } \log n}$$

$O(\log n)$

$2^{\log 8} = 2^3 = 8$

$$\frac{n}{2^{\boxed{?}}} = \frac{n}{n}$$

$\log n$

Lecture 16 - Wednesday, March 8

Announcements

- **ProgTest1** results to be released by Friday, March 17
- **Makeup Lecture** for WrittenTest1, ProgTest1
  + Expected to complete by: March 20

Implementing the Queue ADT using a SLL

Exercise

(1) DLL, first is front of Q

(2) DLL, last is front of Q.

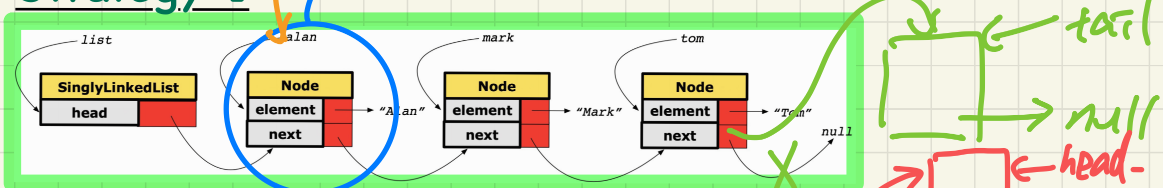
```java
public class LinkedQueue<E> implements Queue<E> {
  private SinglyLinkedList<E> list;
  ...
}
```

Queue Method	Singly-Linked List Method	
	Strategy 1	Strategy 2
size	list.size	
isEmpty	list.isEmpty	
first	list.first O(1)	list.last O(1)
enqueue	✓ list.addLast O(1)	list.addFirst O(1)
dequeue	list.removeFirst O(1)	list.removeLast O(n)

Strategy 1

head

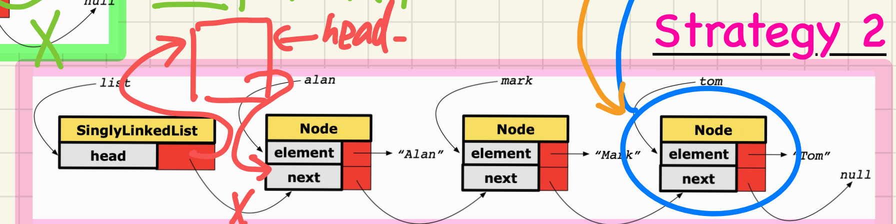
first of Q.



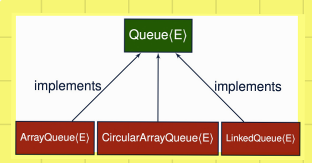
tail

first of Q

Strategy 2



Queue ADT: Testing Alternative Implementations



Polymorphism.

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

dynamic binding.

```java
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;
    }
  }
}
```

Alan Mark Tom

Exercise: Implementing a Queue using Two Stacks

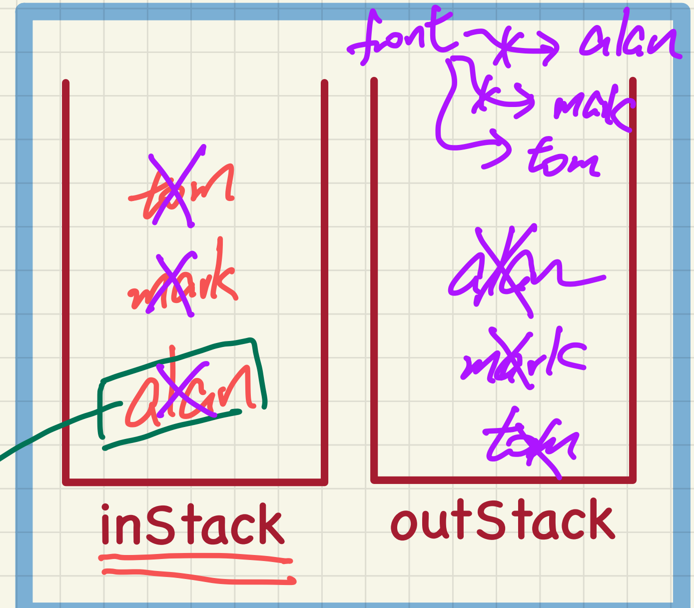
Queue Operation:

```
q.enqueue("alan");
q.enqueue("mark");
q.enqueue("tom");
String front = q.dequeue();
front = q.dequeue();
front = q.dequeue();
```

when the ① dequeue is demanded and ② outStack is empty

front → alan
→ mark
→ tom

front of q.



Queue Operation:

```
q.enqueue("alan");
q.enqueue("mark");
q.enqueue("tom");
String front = q.dequeue();
front = q.dequeue();
front = q.dequeue();
```

LIFO vs FIFO

q.enqueue("Jim");

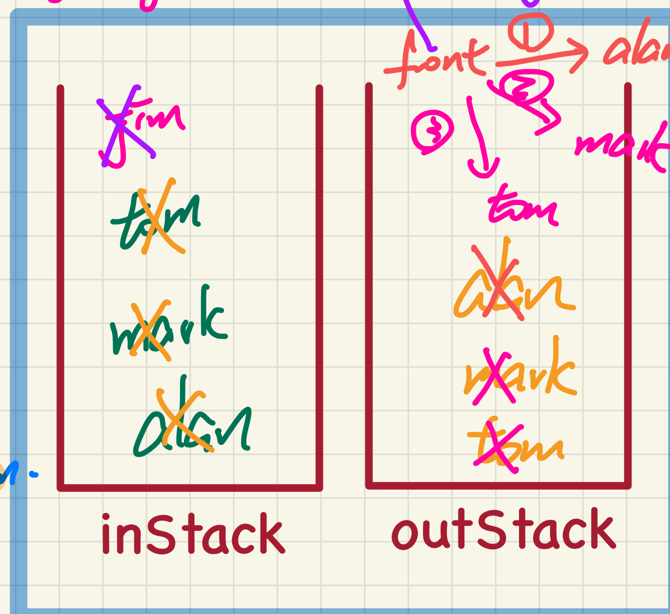
q.dequeue();

top of outStack

Only pop everything off "inStack" and push to "outStack" if:

(1) a "front" or "dequeue" demanded.

(2) "outStack" is empty.



Lecture

General Trees ADT

Terminology, Applications

Trees

a.
1. General Trees
2. Binary Trees (BTs)

b.
3. Binary Search Trees (BSTs)

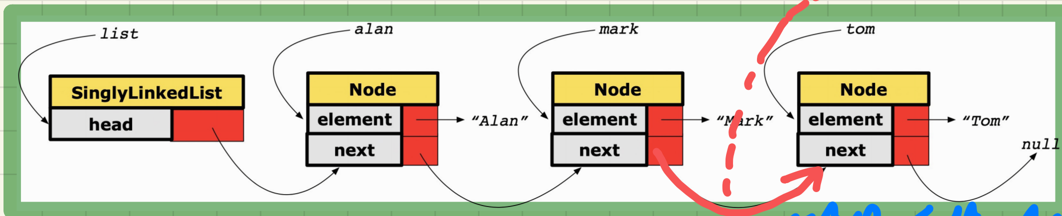
c.
4. Balanced BSTs
5. ADT: Priority Queues
6. Heap Sort

Linear vs. Non-Linear Structures

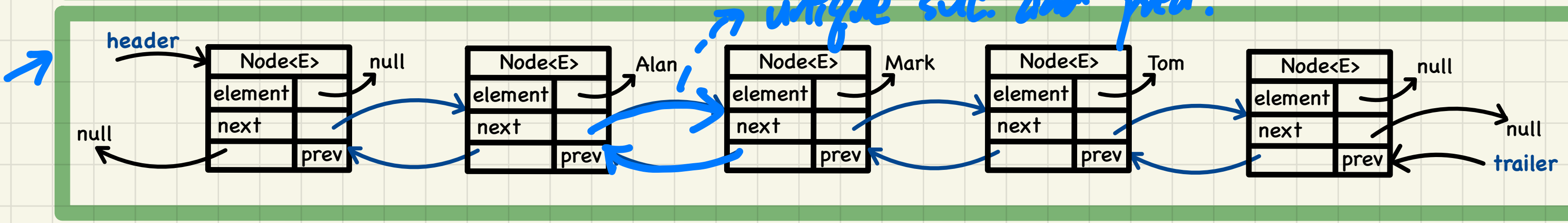
unique successor

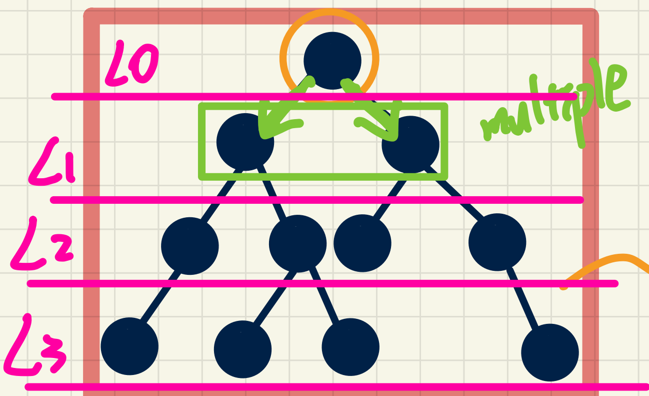
unique predecessor

unique successor



unique suc. and pred.

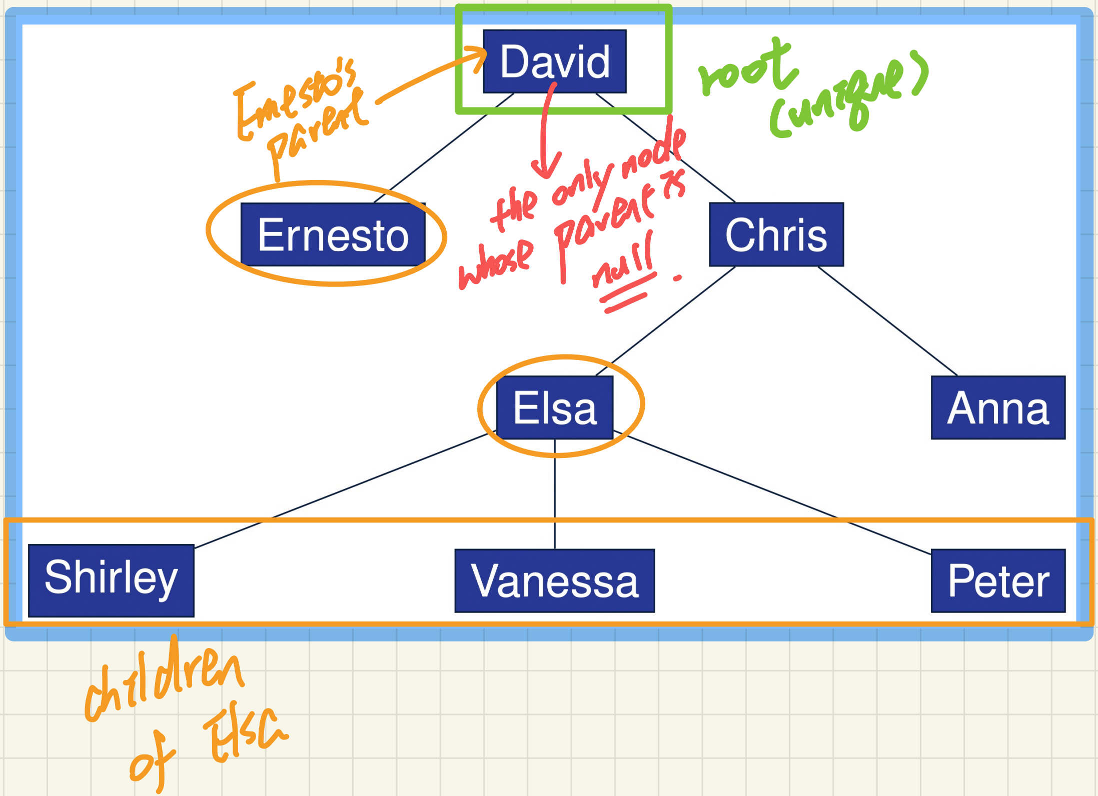




multiple successors, not unique.

hirerarchical structure (levels).

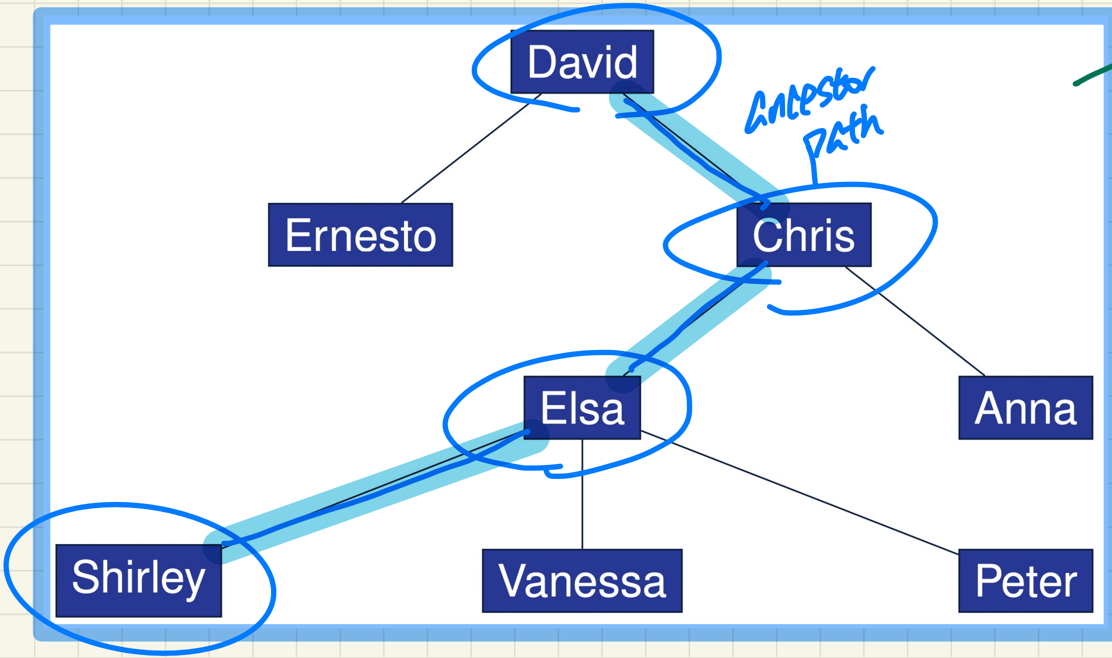
General Trees: Terminology (1)



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

immediately above node

nodes sharing the same parents:
e.g. Ernesto, Chris



Every node has a unique parent.

A node is both its ancestor and descendant.

Ancestors of Shirley : Shirley, Elsa, Chris, David.

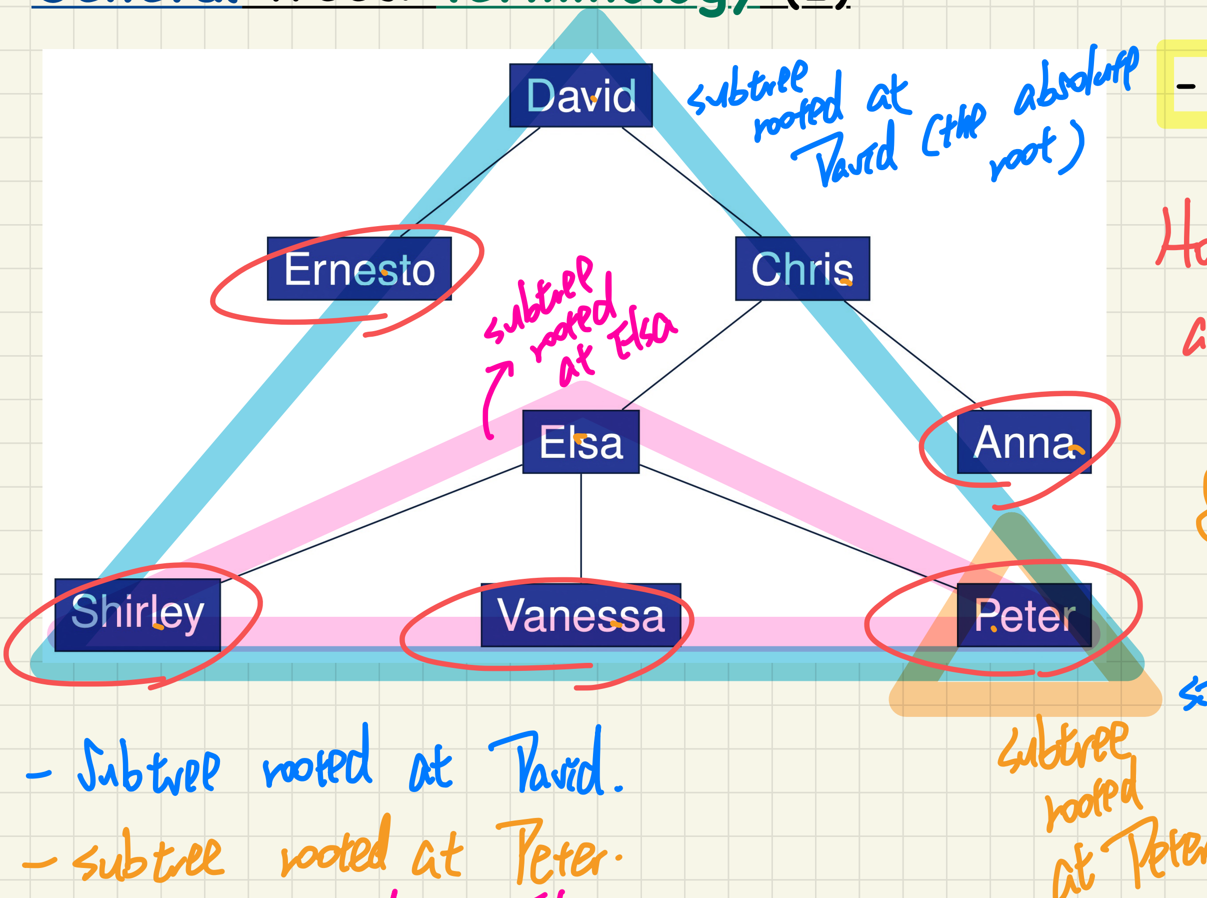
Descendants of the root cover the entire tree.

Descendants of Ernesto : Ernesto

Descendants of Chris : C, Elsa, Anna, S, V, P.

General Trees: Terminology (2)

- subtree



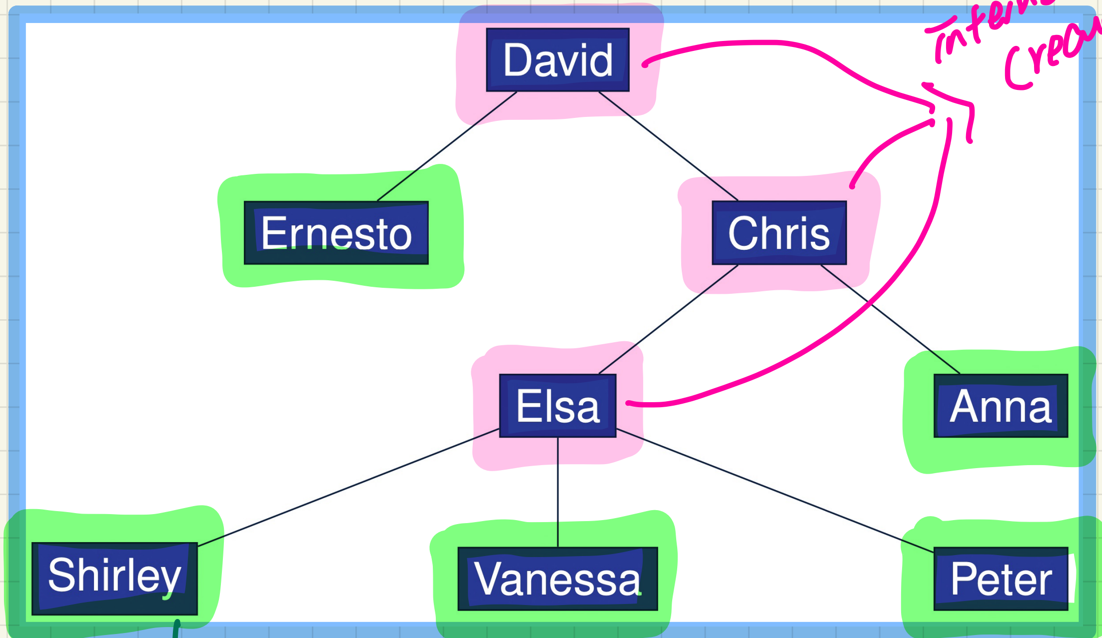
How many subtrees are there in the tree?

8 (# of nodes in the tree).

size \ ST	
1	5 STs ...
2	
⋮	
8	

- Subtree rooted at David.
- subtree rooted at Peter.
- subtree rooted at Elsa.

General Trees: Terminology (3)



internal nodes (recursive cases of recursion on trees)

- external nodes
- internal nodes

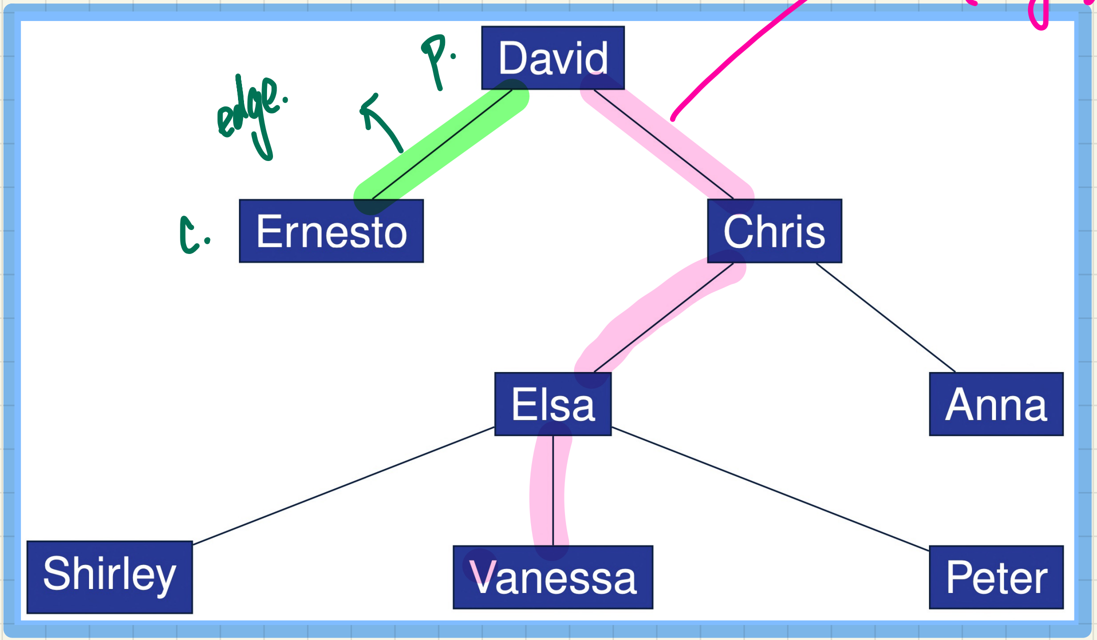
external nodes (base cases of recursion on trees)

Lecture 17 - Monday, March 13

Announcements

- **ProgTest1** results to be released by Friday, March 17
- **Makeup Lecture** for WrittenTest1, ProgTest1
  + Expected to complete by: March 20

General Trees: Terminology (4)



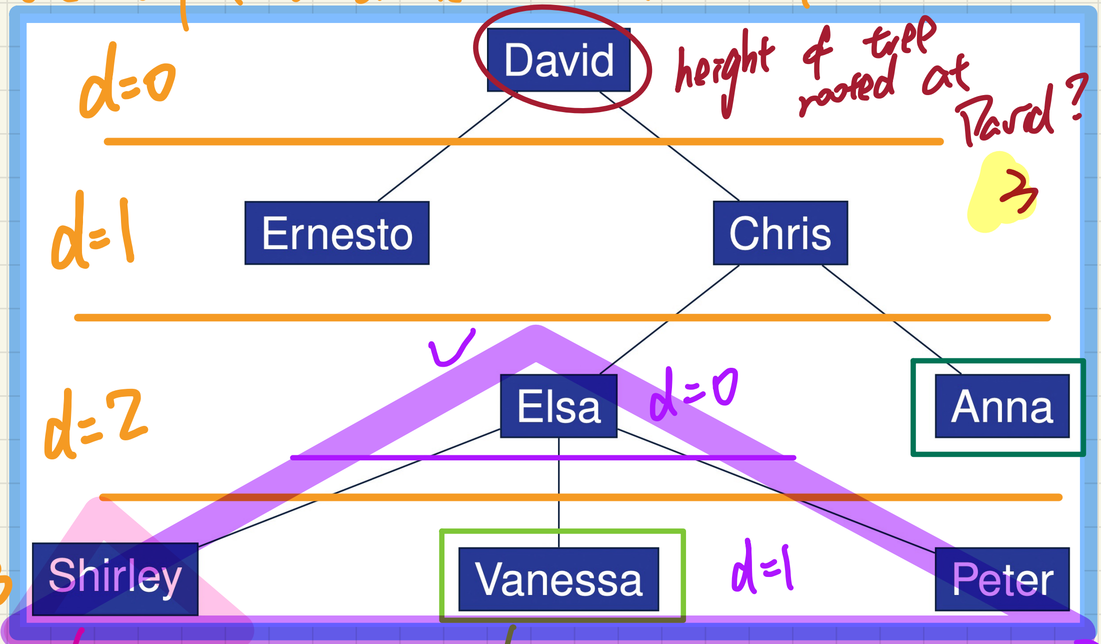
- edge
- path
- depth
- height

a path (In general, as short as 1 single edge; as long as the height of tree)

Use "depth" to divide tree node into levels.

depth of a node

height of a (sub)tree



d=2

d=3

height of subtree rooted at Shirley. 0

d = 3

① # of edges to the root

② # of ancestors − 1

height of subtree rooted at Elsa? 1

General Trees: Recursive Definition

- root
- size

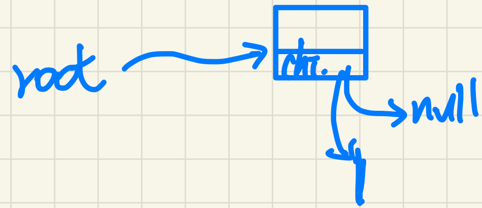
Case 0: Empty tree

ϕ

root → null

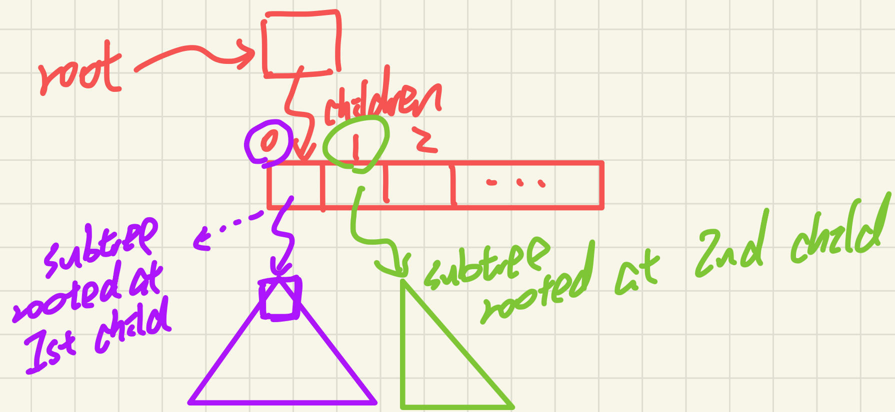
size = 0

Case I: A singleton tree

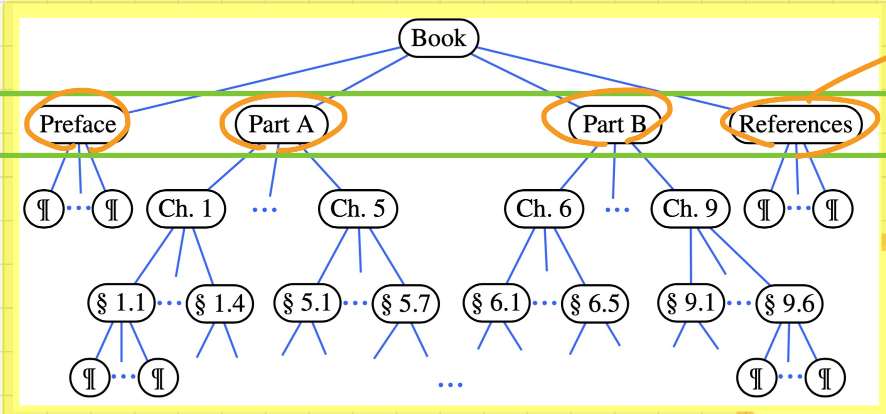


$\boxed{size = 1}$.

Case 2: > 1 nodes

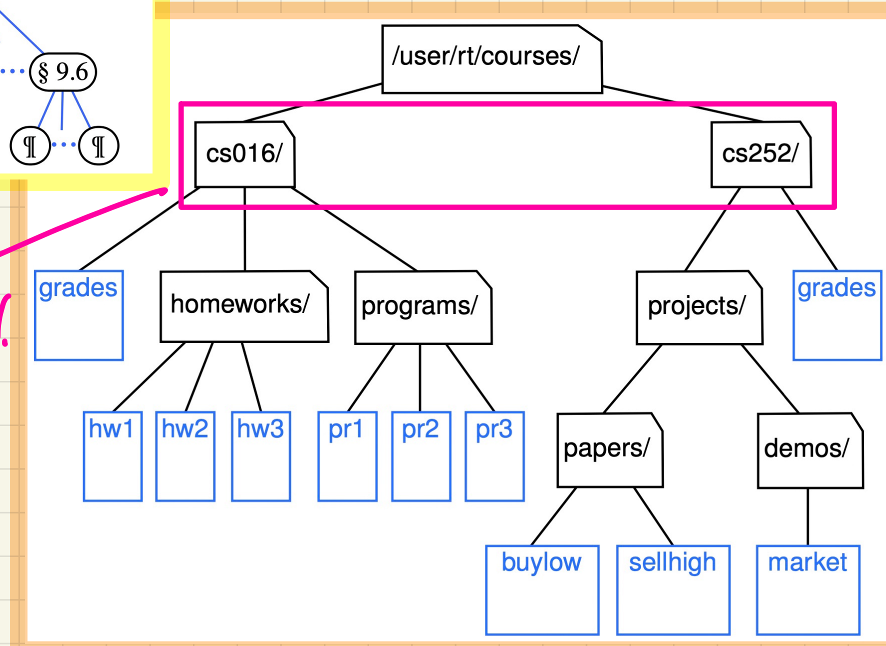


General Trees: Ordered vs. Unordered Trees



d=1

there's a linear order among children at the same level.



unordered!

red

cd

tree.

Lecture

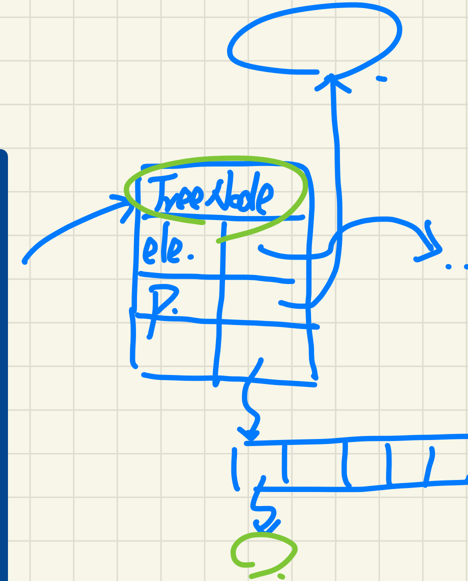
General Trees ADT

Implementing a Generic Tree in Java

Generic, General Tree Nodes

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

\# of child nodes



Compare:

+ prev ref.
+ next ref.

in a DLN.

Instantiating Generic Structures

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

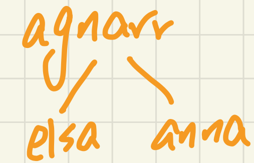
alt. TreeNode<E>.getClass()

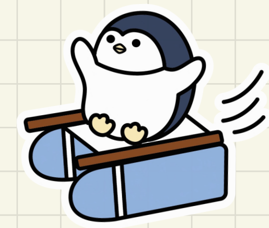
```
class TreeNode<E> {
    private TreeNode<E>[] c;
    public TreeNode<E>() {
        c = (TreeNode<E>[]) new Object[..];
        c = (TreeNode<E>[]) Array.newInstance(this.getClass(), ...);
```

E is wrapped within TN

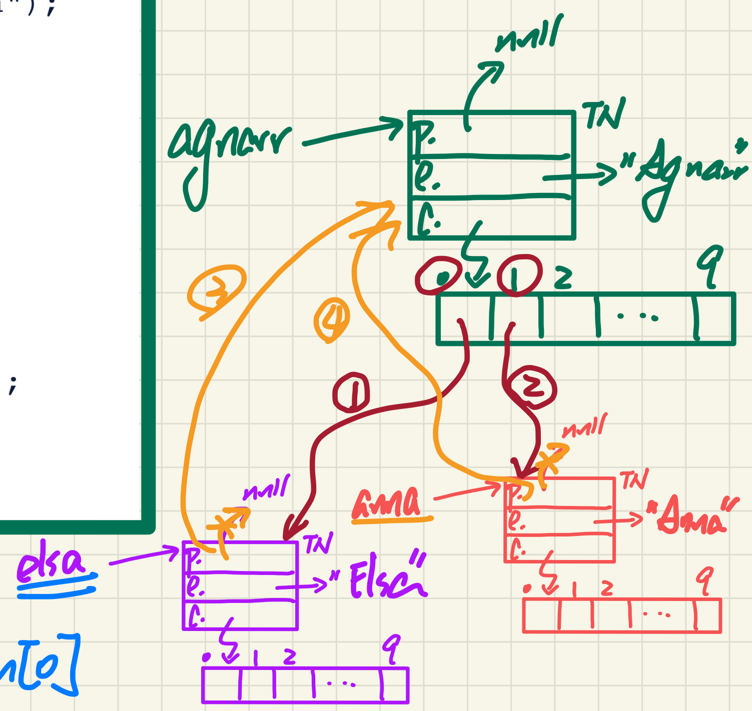
?

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);
}
```



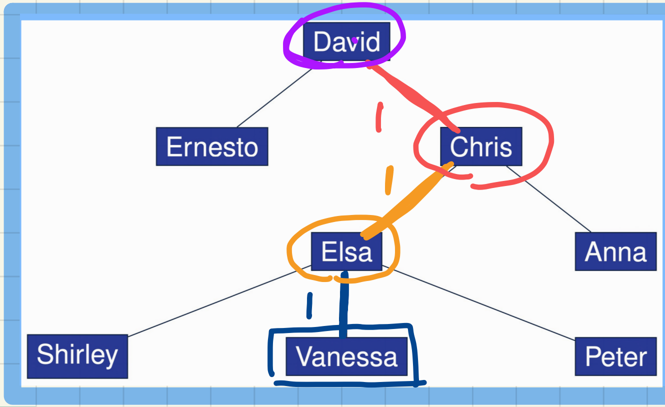
Aliasing
1 agnarr.getChildren[0]
2 elsa
3 elsa.getParent().getChildren[0]

Tracing: Computing a Node's Depth

arbitrary node in tree.

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

n is the root




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

depth(vanessa)

= 1 + depth(Elsa)

→ strictly smaller problem: closer to root!

= 1 + 1 + depth(Chris)

= 1 + 1 + 1 + $\boxed{\text{depth(David)}}$

0 → base case

= 3.

Tracing: Computing a Tree's Height

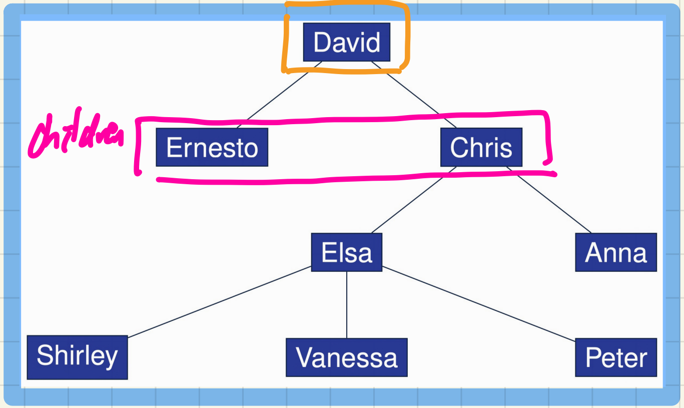
```java
public int height(TreeNode<E> n) {
  TreeNode<E>[] children = n.getChildren();
  if(children.length == 0) { return 0; }
  else {
    int max = 0;
    for(int i = 0; i < children.length; i ++) {
      int h = 1 + height(children[i]);
      max = h > max ? h : max;
    }
    return max;
  }
}
```

recursive cal. of height

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

height

children



height(chris)

Lecture 18 - Wednesday, March 15

Lecture

Binary Trees ADT

Definition, Terminology, Properties

Binary Trees: Recursive Definition

- root
- size

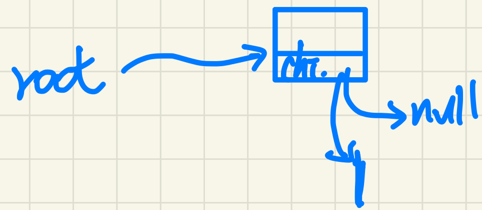
Case 0: Empty tree

ϕ

root → null

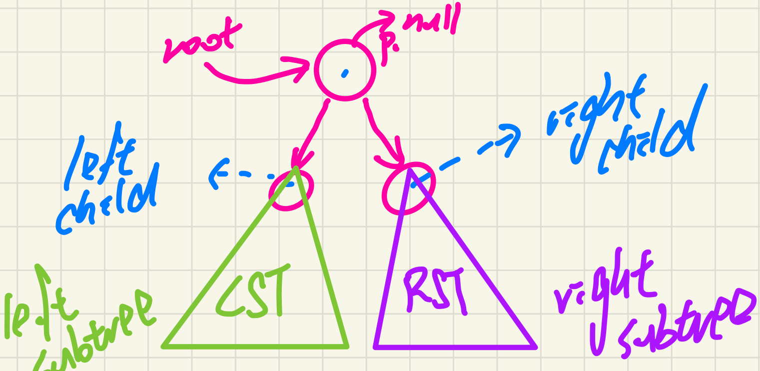
size = 0

Case 1: A singleton tree



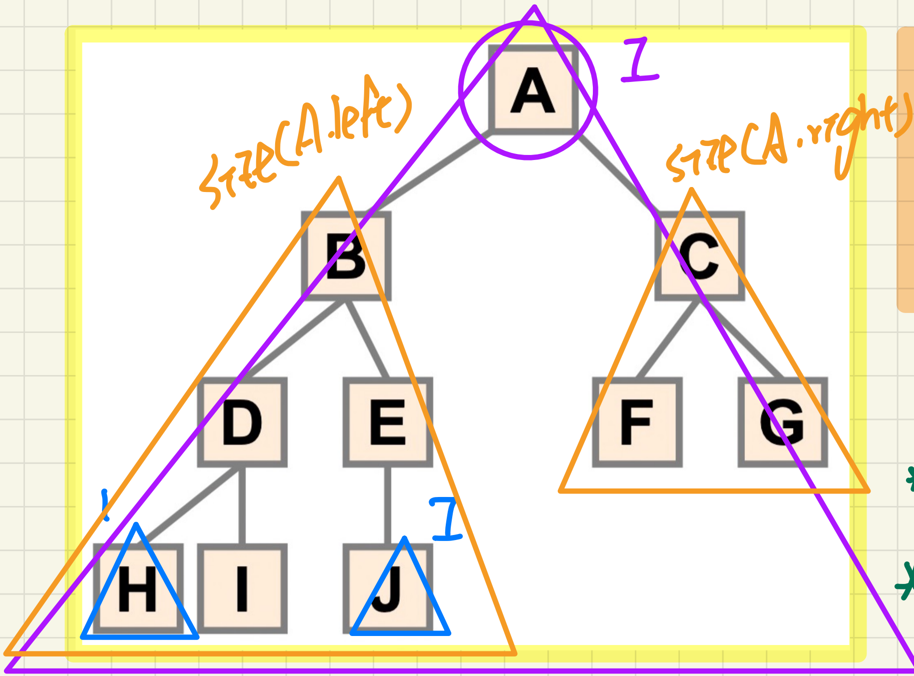
$\boxed{size = 1}$.

Case 2: $\geq$ 2 nodes



BT Terminology: LST vs. RST

search(target, int. n) = n.equals(t)
|| search(t, n.left)
|| search(t, n.right)



Strategy of Recursion on BT:

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

e.g.,

* + counting size

** + searching item

** search(target, ext. n) = n.equals(t)

e.g. x, F

* size(external n.) = 1

left right null

size(internal n.) = 1 + size(n.left) + size(n.right)

e.g. A

Deriving the Sum of a Geometric Sequence

Initial Term: I

Common Factor: r

Number of Terms: k

2^0

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

$*2 \quad *2$

c.f.

2^{10}

$I \cdot r^0$

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

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

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

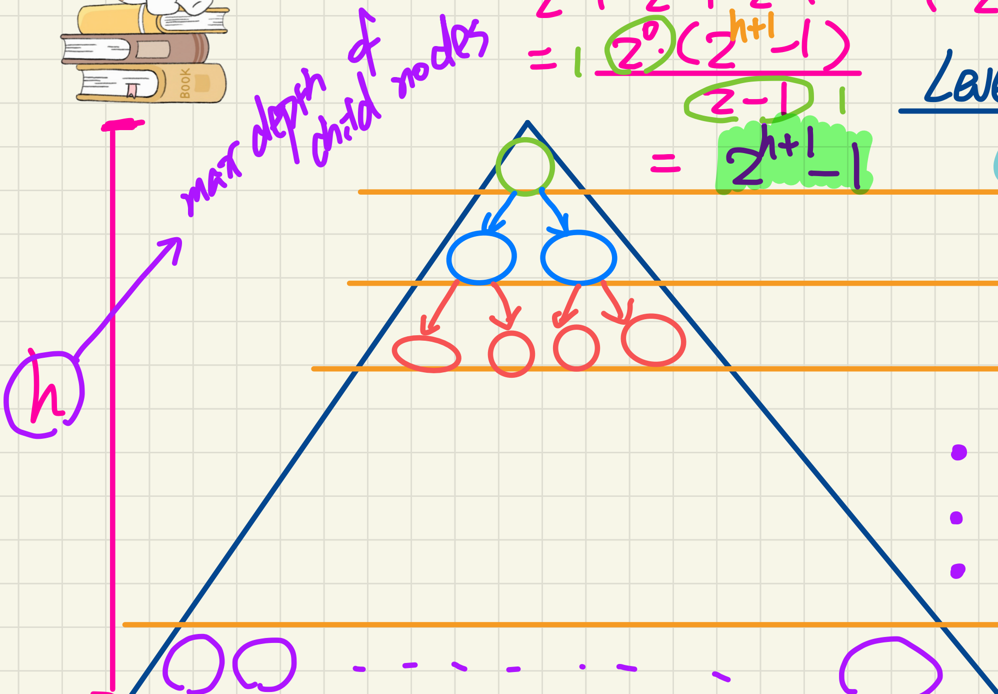
$$S_k = \frac{I \cdot (r^k - 1)}{r-1}$$

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

BT Terminology: Depths, Levels, Max # of Nodes

Max # of nodes in a BT with height h. $h \geq 0$

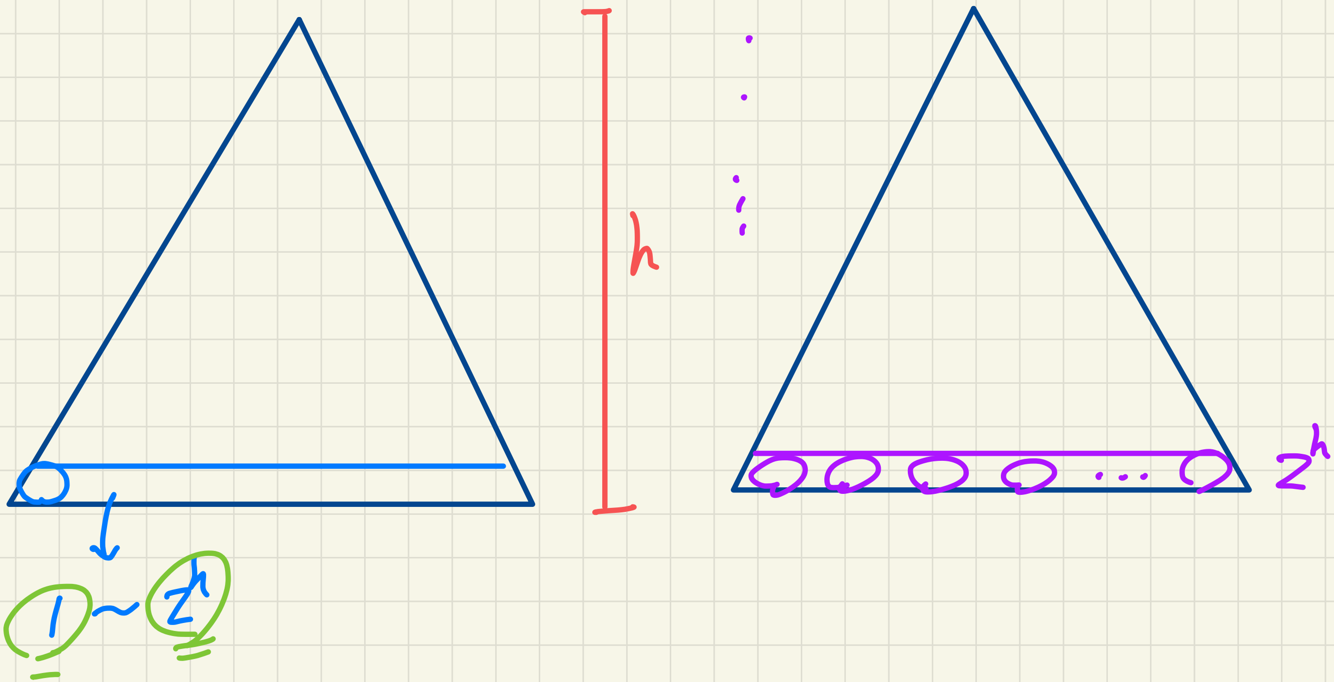
$$2^0 + 2^1 + 2^2 + \cdots + 2^h = \frac{2^0 \cdot (2^{h+1} - 1)}{2 - 1} = 2^{h+1} - 1$$



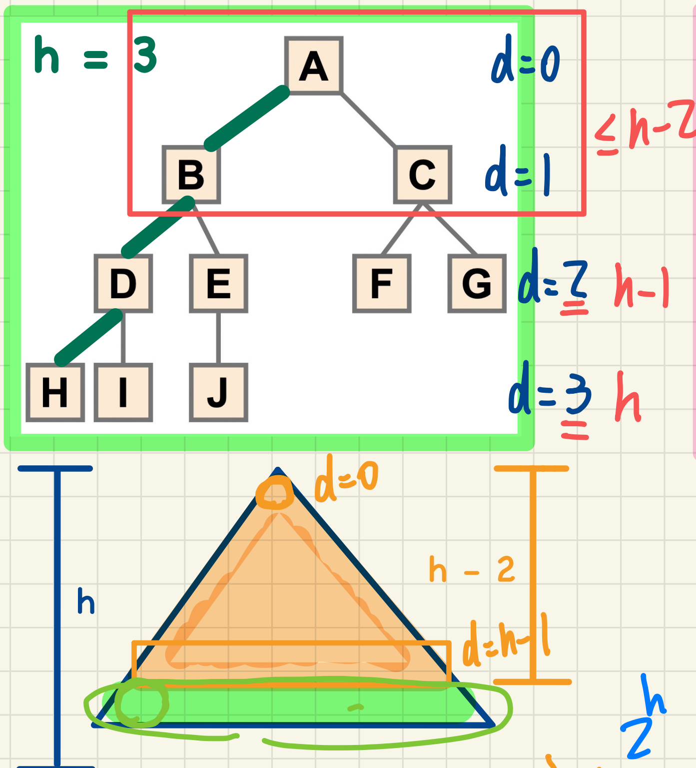
Level ? (d)	Max # Nodes at Level ?
0	$1 = 2^0$
1	$2 = 2^1$
2	$4 = 2^2$
⋮	
h	? 2^h

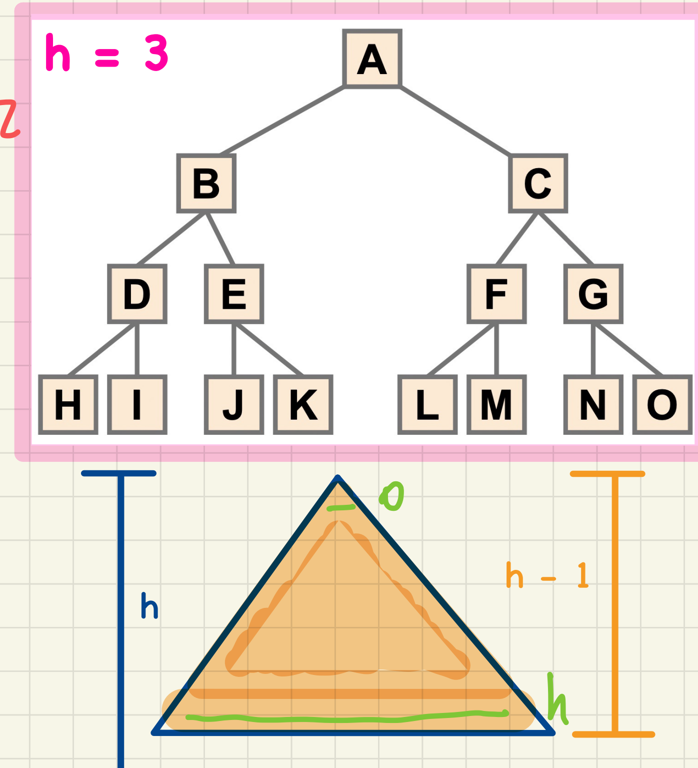
Exercise: Max # of nodes from Level 0 to Level $h-1$?

Complete BT vs. Full BT



BT Terminology: Complete vs. Full BTs





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

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

Min # nodes?

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

BT Properties: Bounding # of Nodes

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

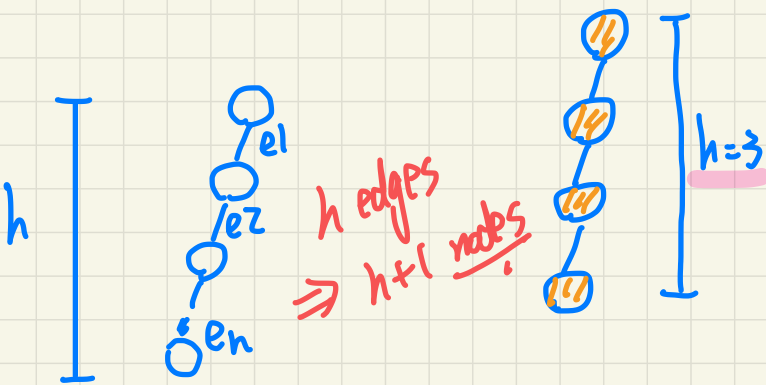
$$h + 1 \leq n \leq \boxed{2^{h+1} - 1}$$

For example, say h = 3

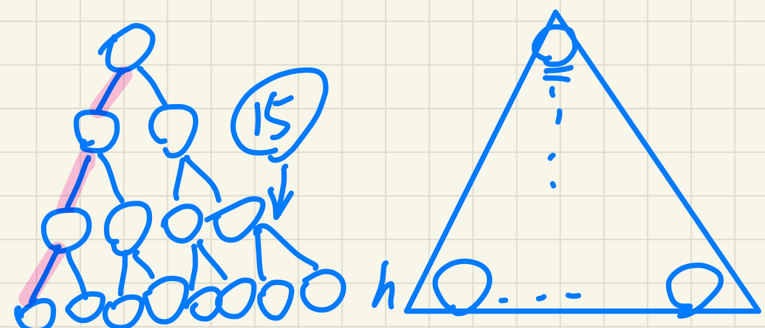
min: $3 + 1 = 4$

max: $2^{3+1} - 1 = 15$

Minimum # of nodes



Maximum # of nodes



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

BT Properties: Bounding Height of Tree

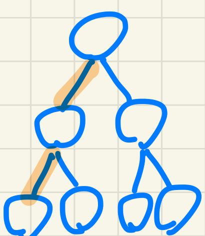
Given a *binary tree* with *n* ***nodes***, the ***height*** *h* is bounded as:

$$\boxed{\log_2(n+1)-1} \le h \le \boxed{n-1}$$

For example, say n = 7

Minimum height

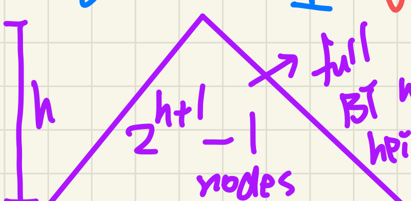
full BT



$\#$ nodes $= 7$

$h = 2$.

$\log_2(7+1) - 1 = 2$



full BT with height h

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

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

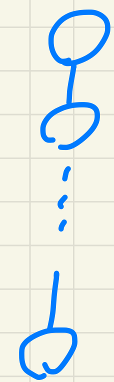
$\log_2(n+1) = h + 1$

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

Maximum height

7 nodes

$\Rightarrow$ 6 edges



n nodes

$\hookrightarrow (n-1)$ edges

BT Properties: Bounding # of External Nodes

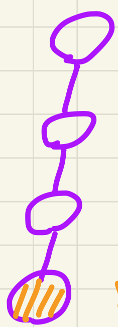
Given a *binary tree* with ***height*** h, the ***number of external nodes*** n_E is bounded as:

n_E

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

For example, say h = 3

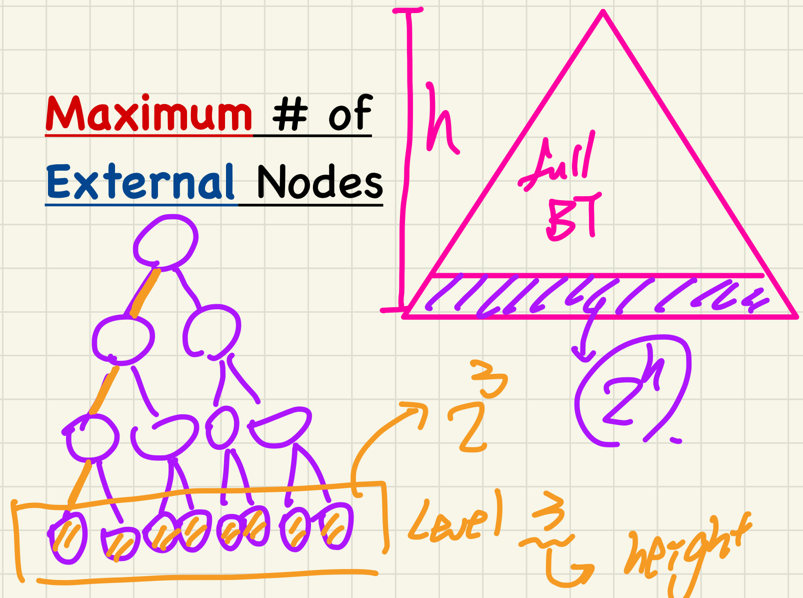
Minimum # of External Nodes



$h=3$

$n_E = 1$

Maximum # of External Nodes



BT Properties: Bounding # of Internal Nodes

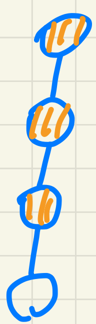
Given a *binary tree* with ***height*** h, the ***number of internal nodes*** n_I is bounded as:

n_I

$$h \le n_I \le 2^h - 1.$$

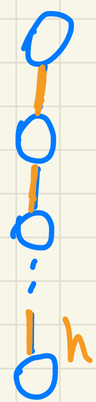
For example, say h = 3

Minimum # of Internal Nodes



$h = 3$

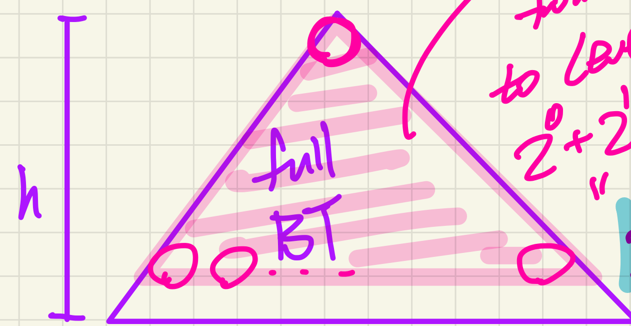
$n_I = 3 \rightarrow h$



\# edges $= h$

↳ $\underline{h}$ internal nodes

Maximum # of Internal Nodes



\# nodes from Level 0 to Level $h-1$:

$2^0 + 2^1 + \dots + 2^{h-1}$

$= 2^h - 1$

Lecture 19 -
Makeup for WrittenTest2

(≈ 90 minutes)

<u>Lecture</u>

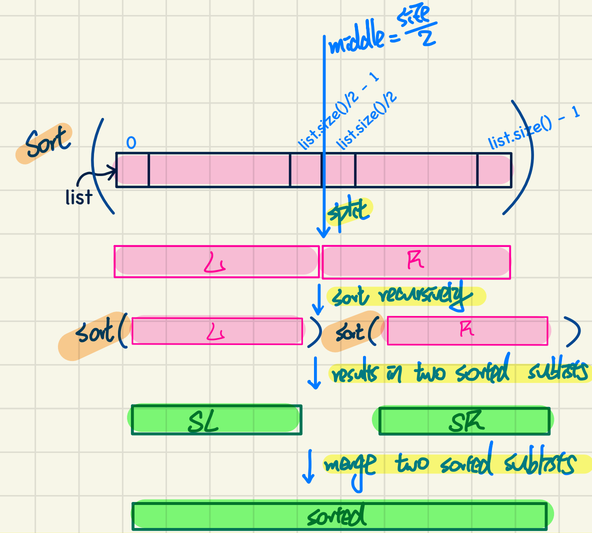
<u>Recursion: Part II</u> (<u>continued</u>)

Merge Sort

Merge Sort in Java

```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;
}
```

base cases



left

0	1	2	3
24	45	63	85

right

0	1	2	3
17	31	50	96

Precondition

L and R sorted

merge

17	24	31	45	50	63	85	96

```java
/* 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;
}
```

(a) # iterations: min(L.size(), R.size())

O(1)

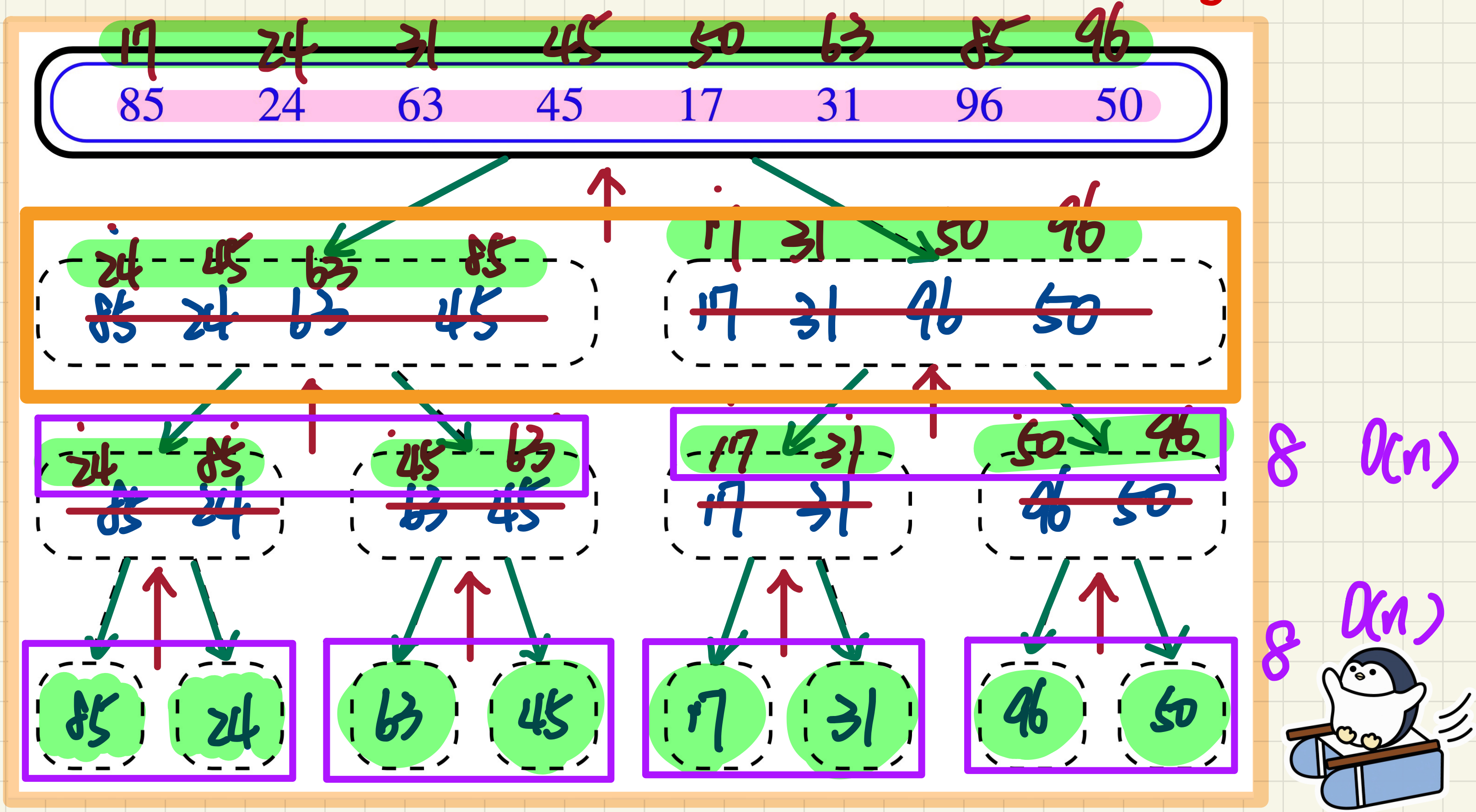
O(1)

(b) # iterations: remaining # of items to loop over in the longer list

(a) + (b) = L.size() + R.size()

Merge Sort: Tracing

→ split
→ merge



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;
}
```

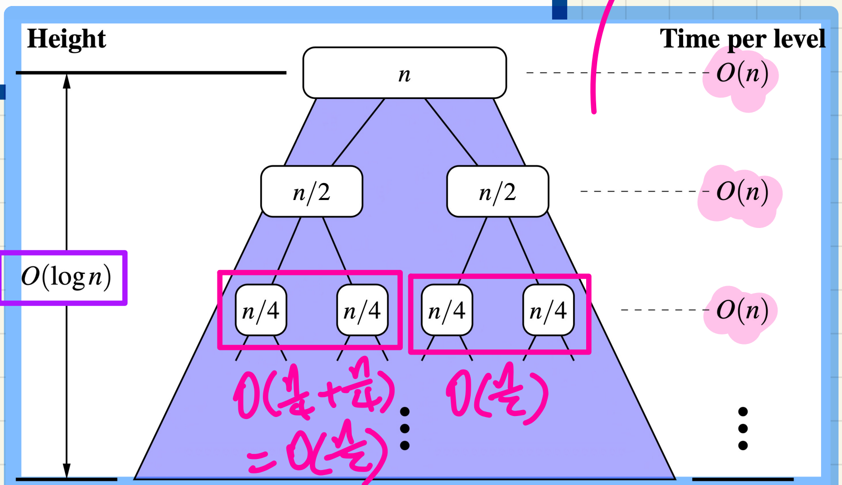
size = $\frac{n}{2}$

sort(left)
sort(right)

$O(1)$

$O(n)$

recursion tree is a full BT



Total RT:
$O(\log n \times n)$
$= O(n \cdot \log n)$

height of balanced BST

Running Time as a Recurrence Relation

$T(0) = 1$

$T(1) = 1$

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

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

Running Time: Unfolding Recurrence Relation

$T(0) = 1$

$T(1) = 1$

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

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

$n=8$

$2^{\log 8} = 8$

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

$$= 2 \cdot \left(2 \cdot \boxed{T\left(\frac{n}{4}\right)} + \frac{n}{2}\right) + n \qquad \left[4 \cdot T\left(\frac{n}{4}\right) + 2n\right] \quad 4 = 2^2$$

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

$$\vdots$$

$$= \underbrace{\boxed{2^{\log n}}}_{n} \cdot \overbrace{\boxed{T(1)}}^{1} + \log n \cdot n = n + n \cdot \log n$$

$$T(1) \rightarrow \frac{n}{2^{\log n}}$$

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

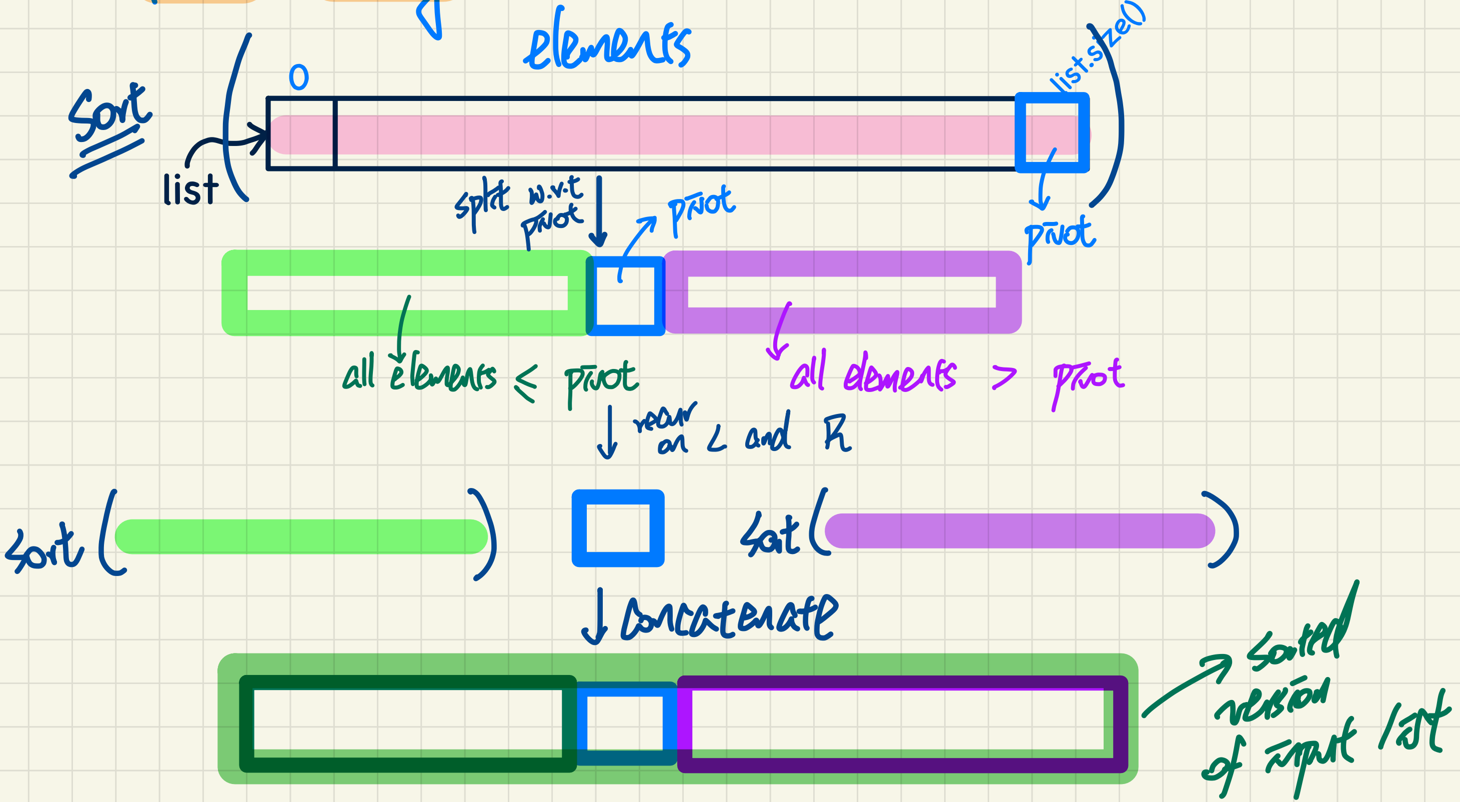
<u>Lecture</u>

<u>Recursion: Part II</u> (<u>continued</u>)

Quick Sort

Quick Sort: Ideas

pivot: ideally the median value of the list elements



Quick Sort in Java

```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 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;
}
```

base cases $O(1)$

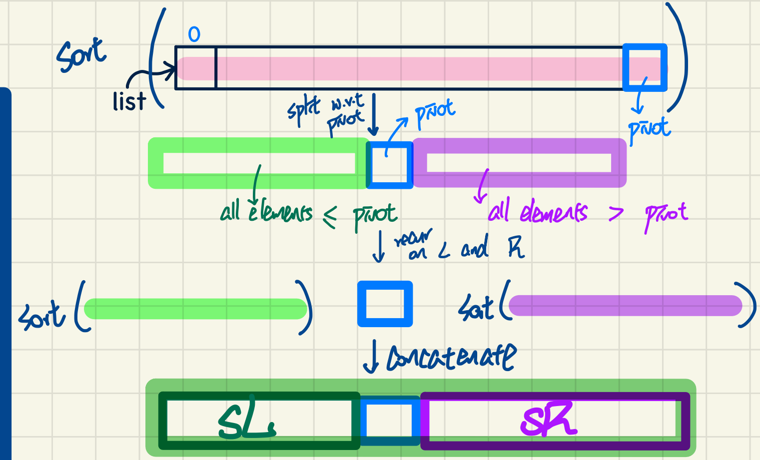
$O(1)$ $O(n)$ $O(1)$

1. Best case: pivot is s.t. $|L| \approx |R|$

2. Worst case: e.g. $|L| = n-1$, $|R| = 0$

$|L| \ll |R|$ or $|R| \ll |L|$

$O(n)$




```java
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;
}
```

$O(n)$

list

85	24	63	45	17	31	96	50

left

24	45	17	31

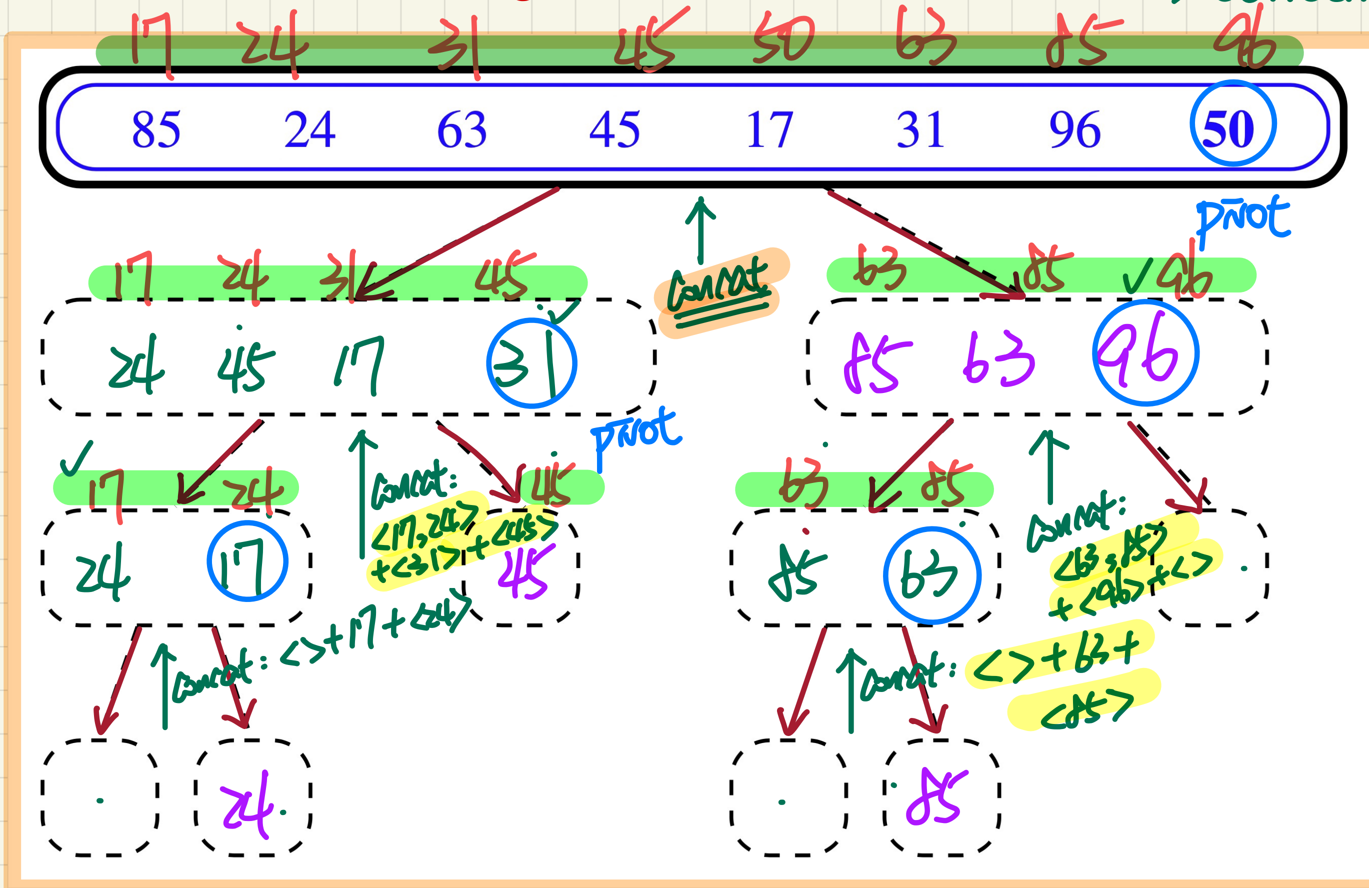
right

85	63	96

Quick Sort: Tracing

split

concatenate



Quick Sort: Worst-Case 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 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;
}
```

$O(n)$

| 9 | 7 | 5 | 3 | 1 |

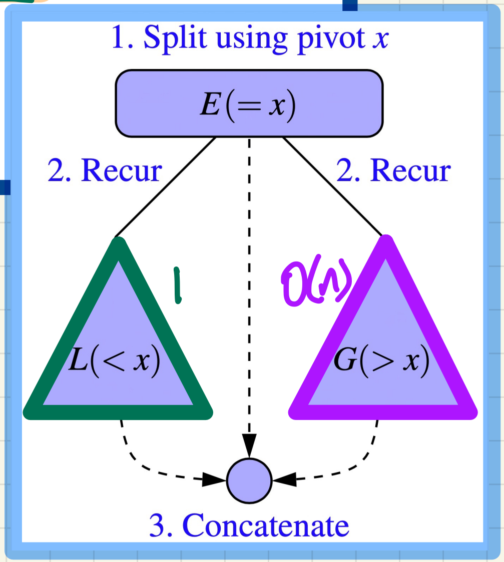
1 | 9 | 7 | 5 | 3 |

1 | 9 | 7 | 5 |

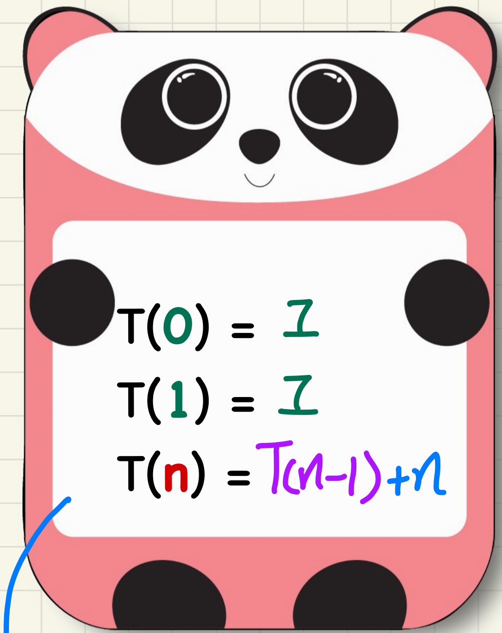
1 | 9 | 7 |

1 | 9 |

splits: 4 $O(n)$



Running Time as a Recurrence Relation



Exercise: Solve by unfolding

Quick Sort: Best-Case Running Time

$\log_2 n$

```
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;
}
```

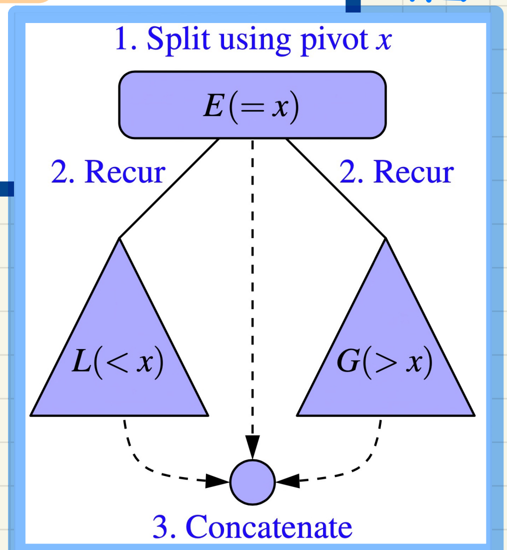
$O(1)$

$O(n)$

median

$\leq$ $>$

sizes equal



n

$\frac{n}{2}$ $\frac{n}{2}$

$\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$ $\frac{n}{4}$

$\vdots$

1 1 $\cdots$ 1 1

Running Time as a Recurrence Relation

$T(0) = 1$

$T(1) = 1$

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

Exercise 1: Solve by Unfolding.

Ex. 2

$T(0)=1$

$T(1)=1$

$T(n) = T(\frac{n}{10}) + T(\frac{9n}{10}) + n$

Lecture 20 - Wednesday, March 22

Announcements

- WrittenTest2 results to be released by FRI, March 24
- Assignment 3, ProgTest2
- Makeup Lecture for WrittenTest2
  + Expected to complete by: Exam Day

TreeNode

SLLNode

recursion

Lecture

Binary Trees ADT

Definition, Terminology, Properties

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

n_E, n_I before ext.

n_E', n_I' after the 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: $\boxed{n_E = n_I + 1}$ ✓

# of external nodes

internal nodes

Induction on Size of Proper BT

Base Case: 1-node tree

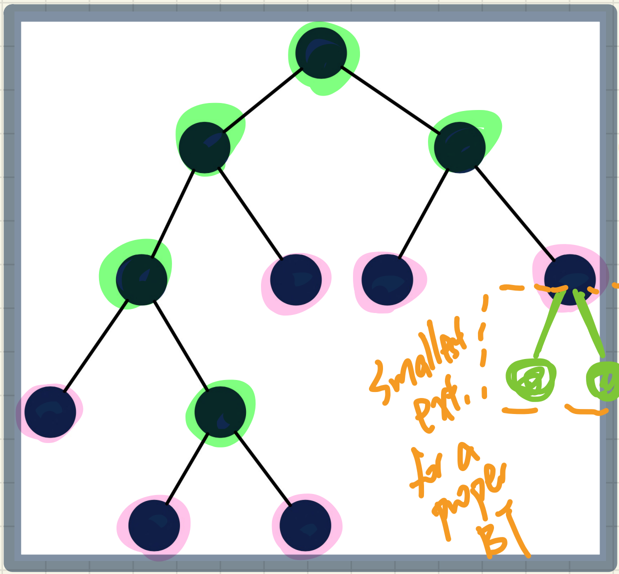
$n_E = 1$

$n_I = 0$

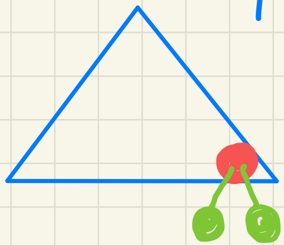
* By I.H.: $n_E' = n_E + 1 = (n_I + 1) + 1 = n_I' + 1$

$n_E = n_I + 1$ ✓

Inductive Hypothesis: for a proper BT with > 1 nodes: $n_E = n_I + 1$



Given a proper BT, extend it by turning an ext. node into an int. node



$n_E' = n_E - 1 + 2 = n_E + 1$

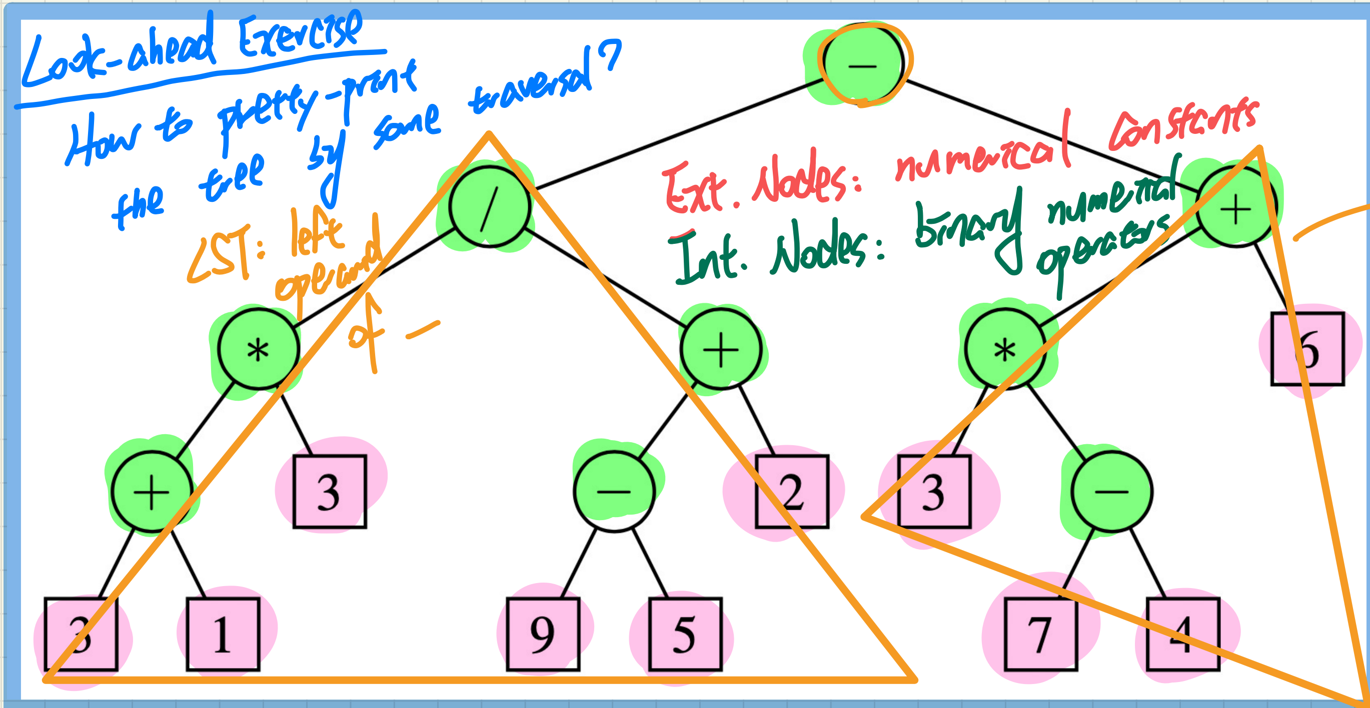
$n_I' = n_I + 1$

Lecture

Binary Trees ADT

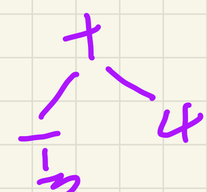
Applications

Applications of Binary Trees: Infix Notation



Q. Is the binary tree necessarily **proper**?

unary op. ← $\boxed{-3}$ + 4

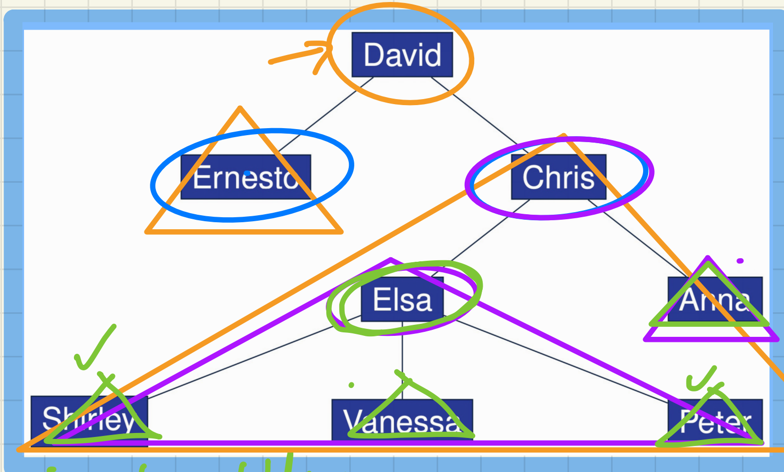


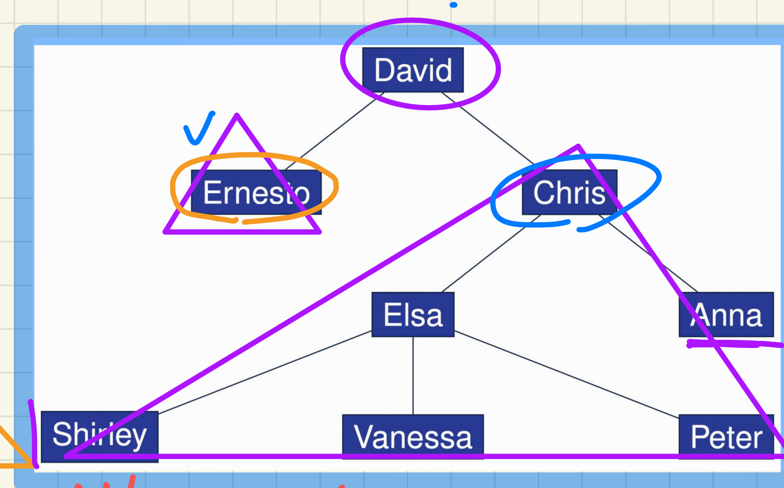
Lecture

Binary Trees ADT

Tree Traversals
Pre-Order, In-Order, Post-Order

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





parent, then children

Pre-Order Traversal

from the Root

pre-order(David)

David Ernesto Chris Elsa Shirley Vanessa Peter Anna

po(Elsa) po(Anna)

→po(Ernesto) po(Chris)

children, parent

Post-Order Traversal

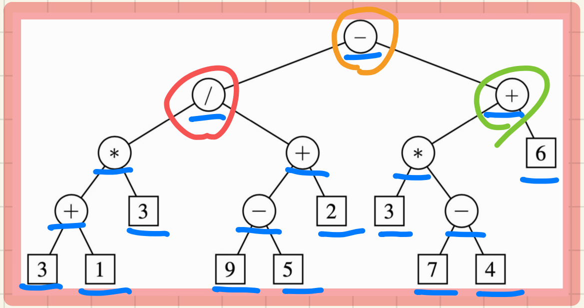
from the Root

post-order(David)

Ernesto S. V. P. Elsa A. C. David

PO(Ernesto) PO(Chris)

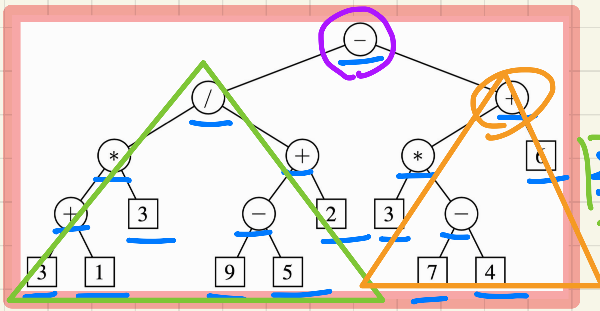
Binary Tree Traversals



Pre-Order Traversal

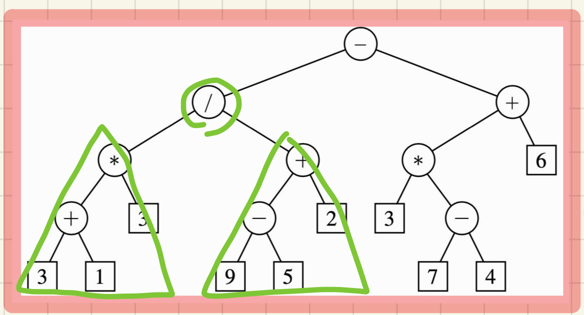
− / * + 3 1 3 + − 9 5 2 + * 3 − 7 4 6

po(/)   po(+)



In-Order Traversal

3 + 1 * 3 / 9 − 5 + 2 − 3 * 7 − 4 + 6



Post-Order Traversal

postfix notation!

3 1 + 3 * 9 5 − 2 + / 3 7 4 − * 6 + −

Lecture 21 - Monday, March 27

Announcements

Q1
Q2

- Assignment 3 due soon
- $\boxed{\text{ProgTest2}}$ guide & practice questions released
- Makeup Lecture for WrittenTest2
  - Expected to complete by: Exam Day

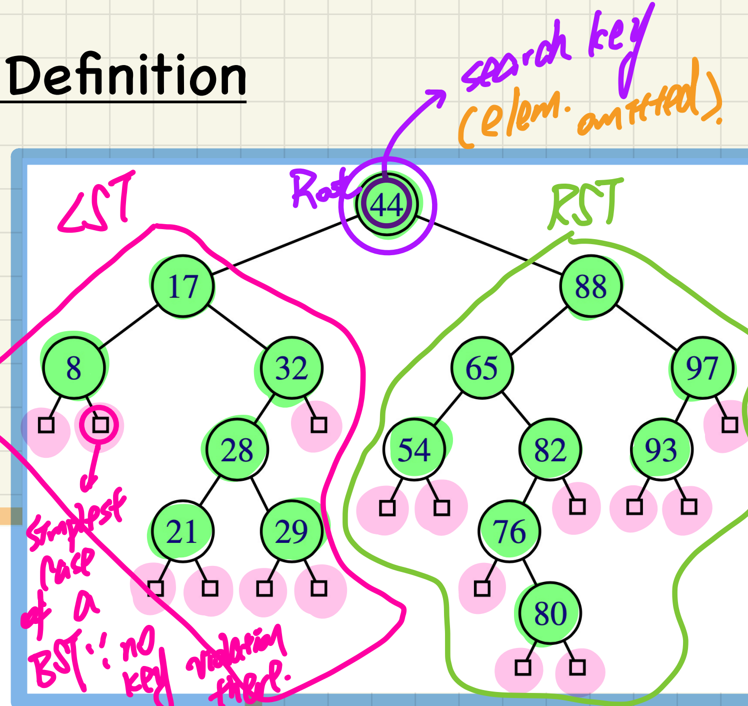
Lecture

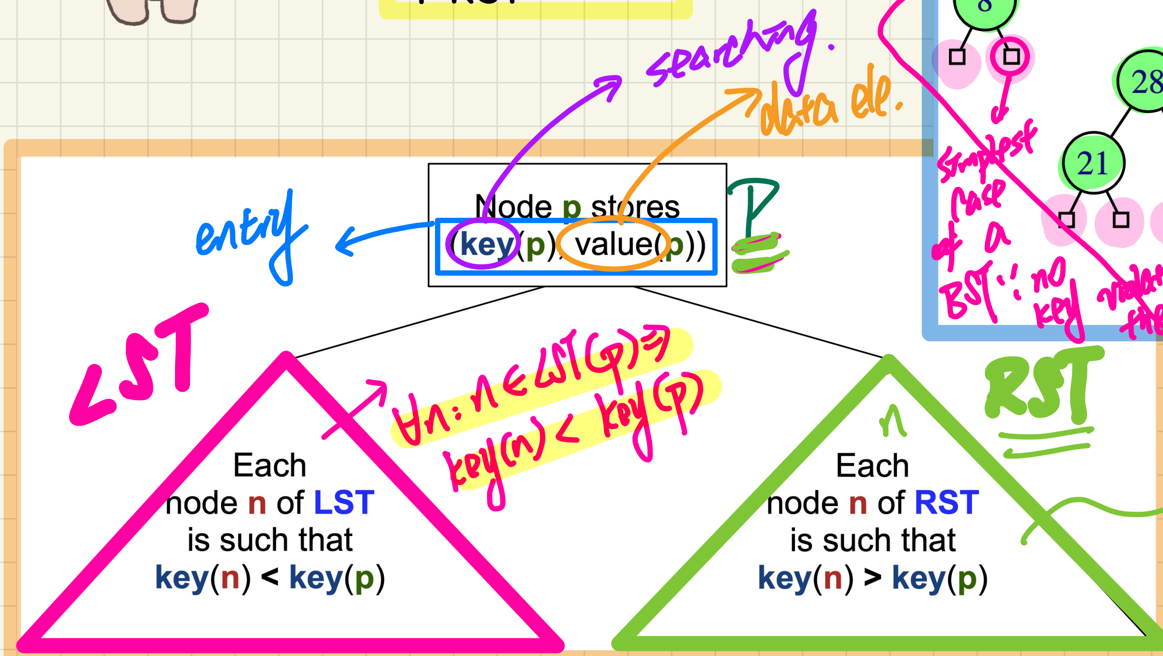
Binary Search Tree (BST)

Definition and Property

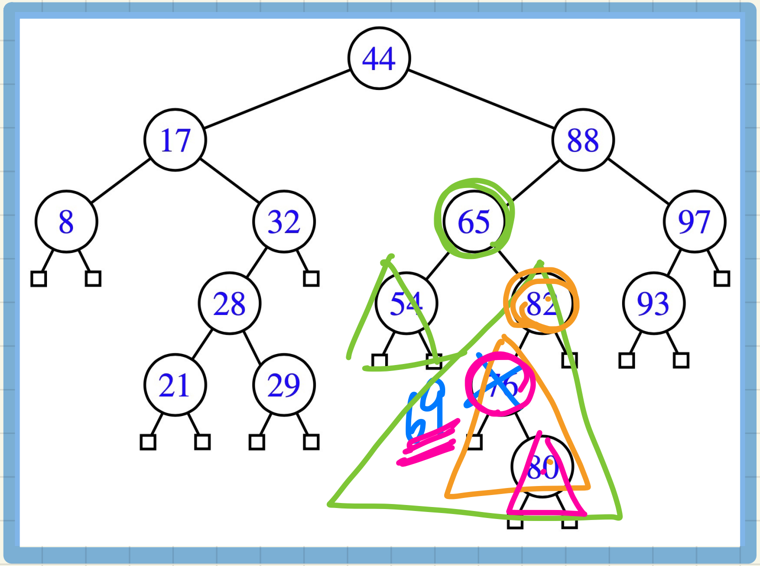
Binary Search Trees: Recursive Definition

- external node
- internal node
  + LST
  + RST





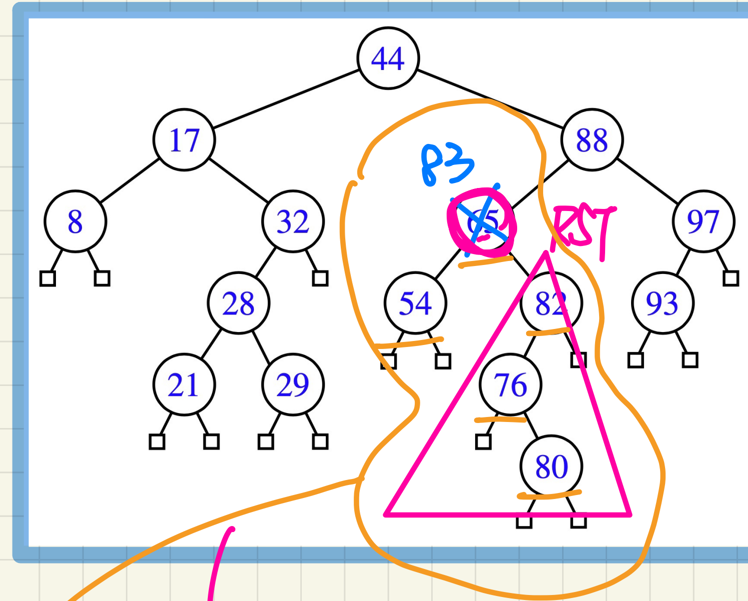
Q1.



✓ still a BST.

Tot:
54 83 76 82 80
not sorted

Q2.



not a BST
∵ 82 in 83's RST is
not greater than it.

Binary Search Trees: Sorting Property

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

In-order traversal of 44's LST

I.o.T. of 44's RST

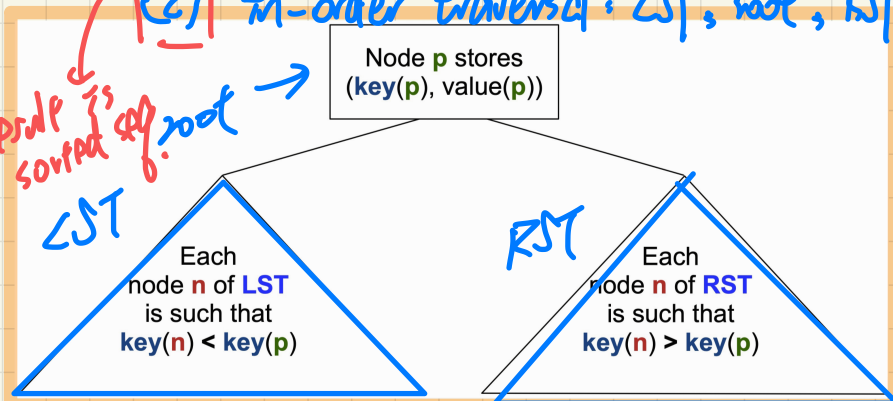
In-order traversal : 8 17 21 28 29 32 44 54 65 76 80 82 88 93 97

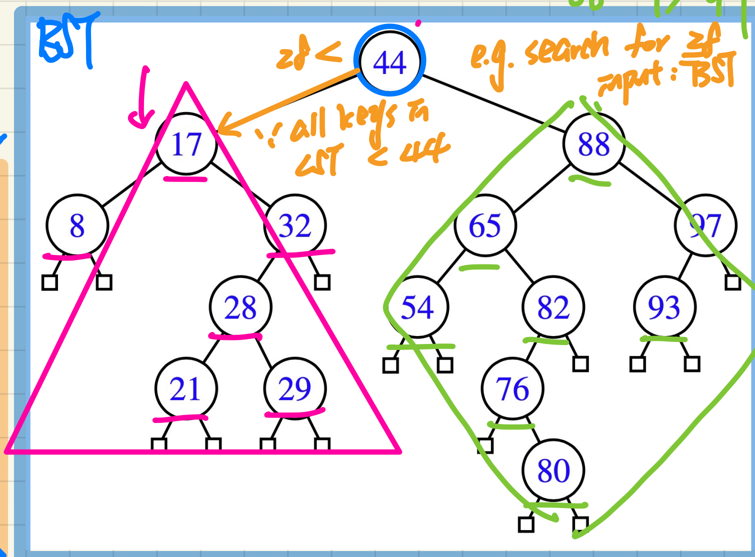
Given a T. that's a BST:

(1) search prop. satisfied

(2) in-order traversal: LST, root, RST

result is a sorted seq.





Lecture

Binary Search Tree (BST)

Implementing a Generic BST in Java
Tree Construction and Traversal

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) { ... }
}
```

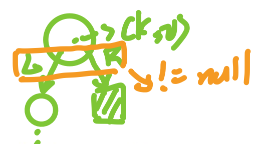
att.

for creating ext. nodes

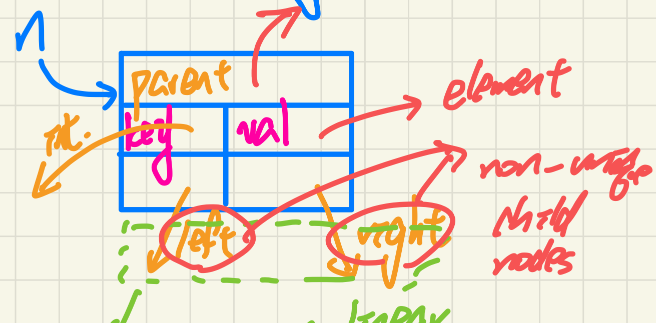
for creating int. nodes.

ext. node.

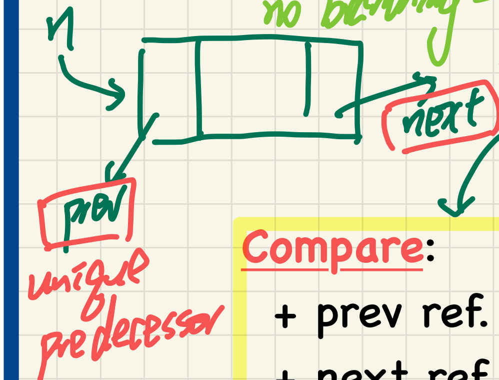
int. node.



BSTNode<String> n



no branching ⇒ linear.



Compare:

+ prev ref.
+ next ref.

in a DLN.

Generic, Binary Tree Nodes - Traversal

iot result:

ArrayList root ArrayList

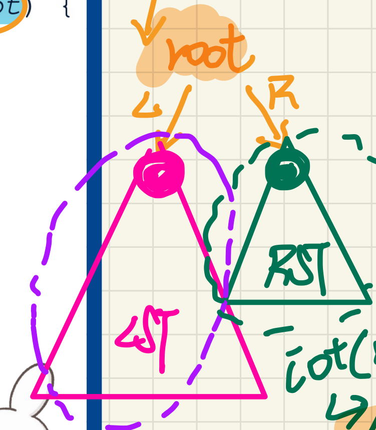
pre-order t.: B2, B1, B3

```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;
  }
}
```

B1

B2

B3



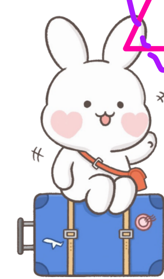
iot(root.getR())

→ ArrayList

iot(root.getLeft())

↓

ArrayList



Exercise

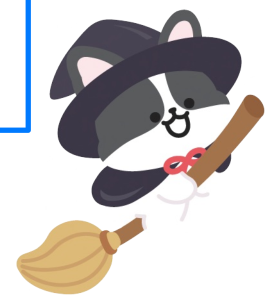
1. BSTNode<E>[]
2. SLLNode<BSTNode<E>>

pre-order?

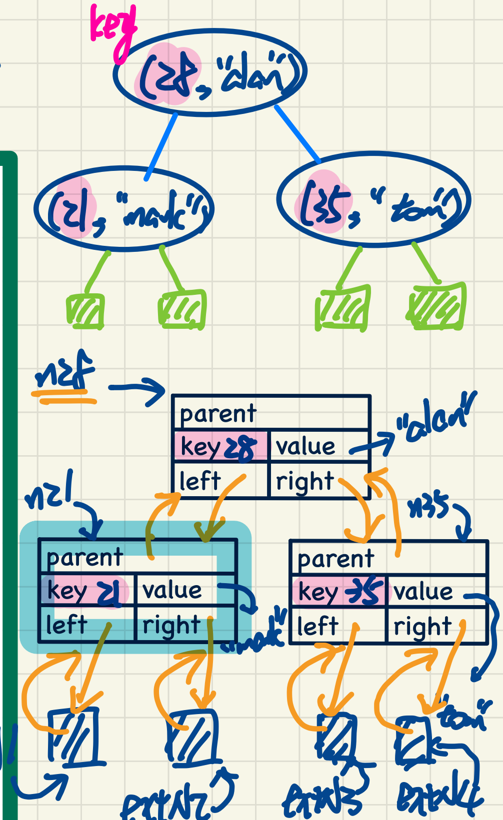
post-order?

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());
}
```



sorting property



aliasing

- n21
- n28.getLeft()
- extN1.getP()
- n28.getL().getR().getP()

Lecture 22 - Wednesday, March 29

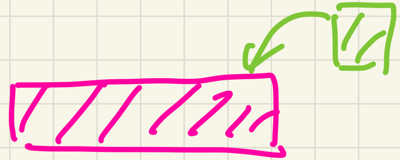
Announcements

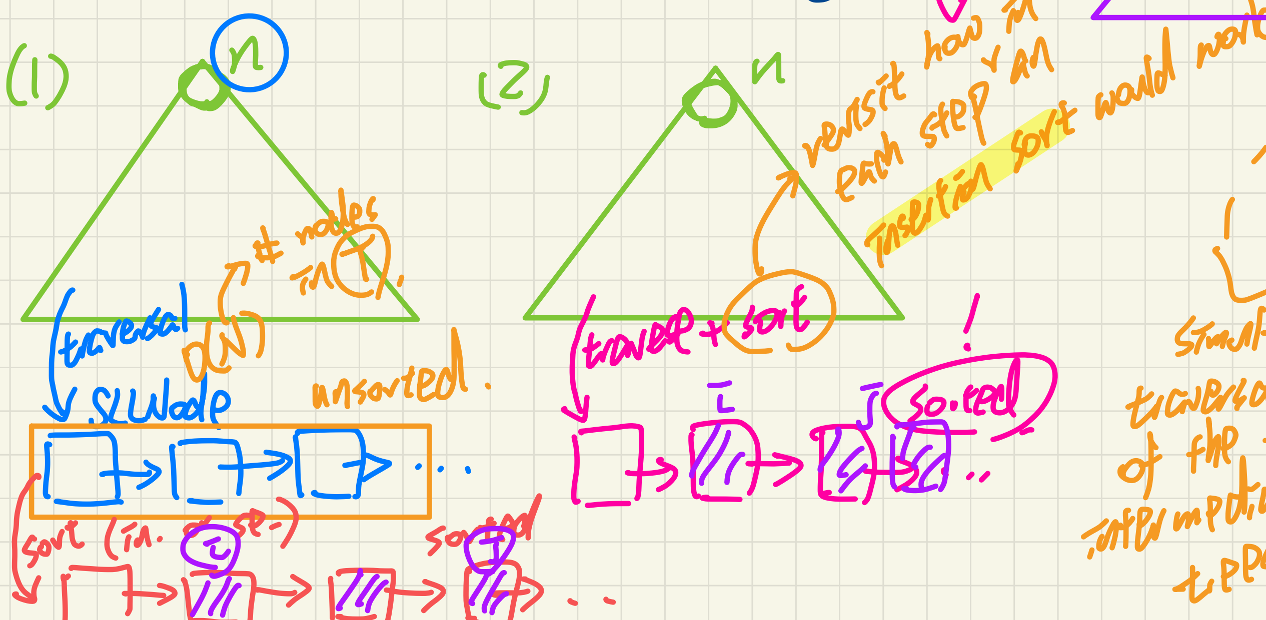
- Bonus Opportunity - Course Evaluation
- ProgTest1: Jackie (Office Hour)
- Assignment3 solution released, ProgTest2

Assignment 3

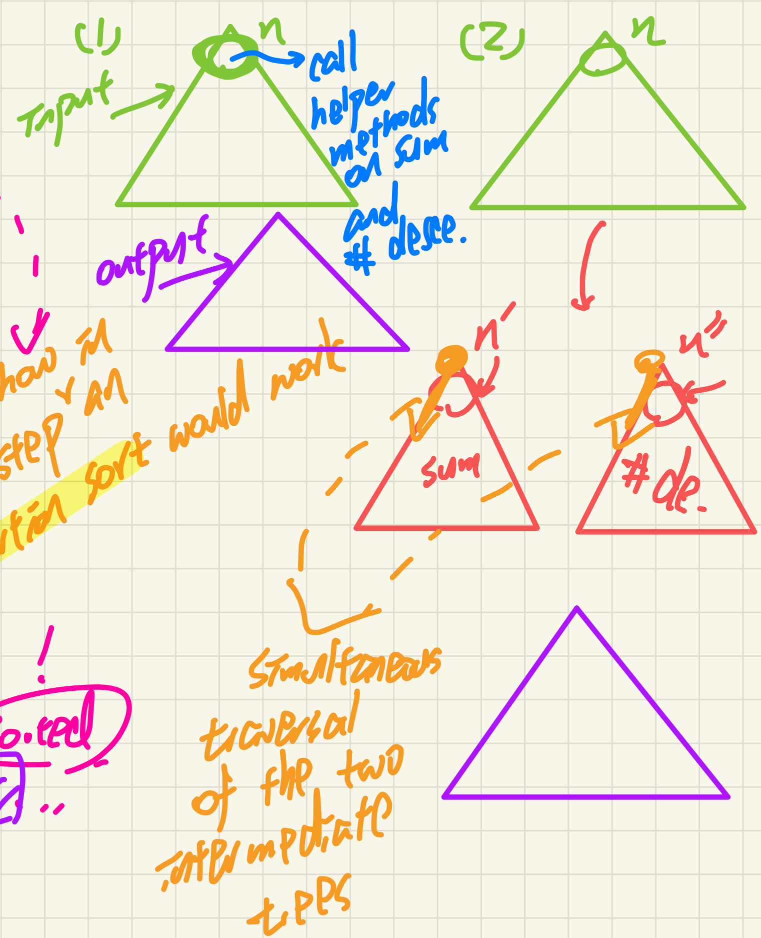
Task 1 : Rank

task1 (TN n , int i , int j)





Task 2 : Stats



Lecture

Binary Search Tree (BST)

Implementing a Generic BST in Java
Searching

BST operations :

1. Input : a BST

↳ search property

2.1 Searching

2.2 Insertion

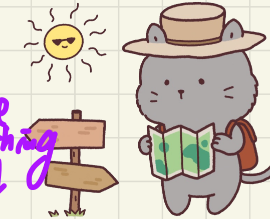
2.3 Deletion

Critical

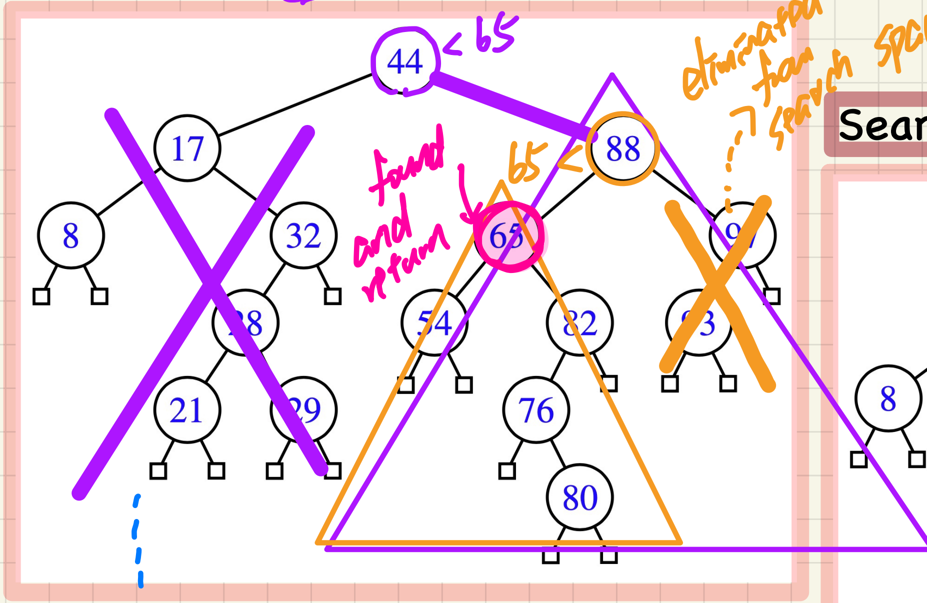
↳ Search property maintained in the output : it remains a BST.

BST Operation: Searching a Key

searching { successful
↳ ret. int. node with matching key
unsuccessful
↳ ret. ext. node that can store the searching key

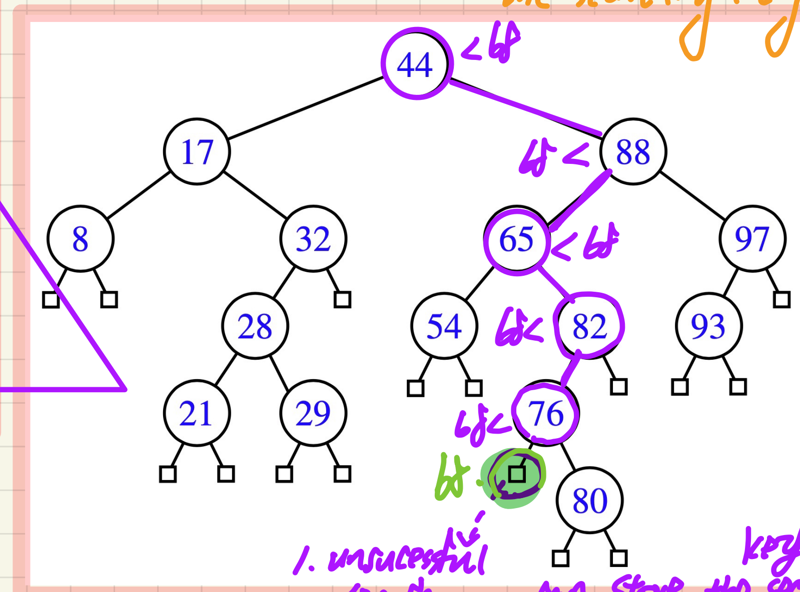


Search key 65



eliminated from the search space

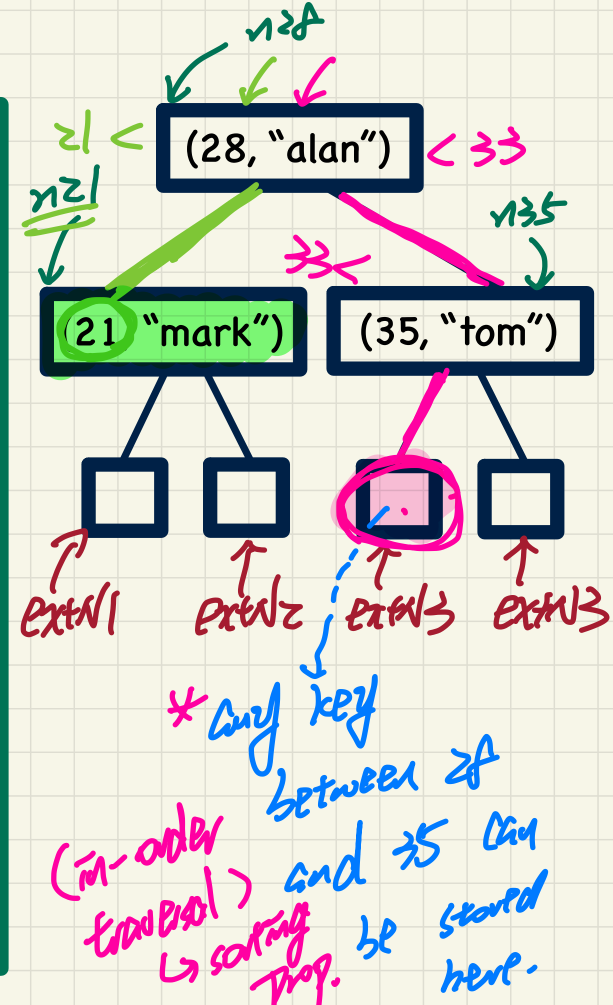
Search key 68



1. unsuccessful search
2. return the ext. node that can store the search key

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* */
}
```



Running Time: Search on a BST

* Worst: given N nodes, $\lfloor \log N \rfloor \le h \le N$

internal or external

```
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;
}
```

root

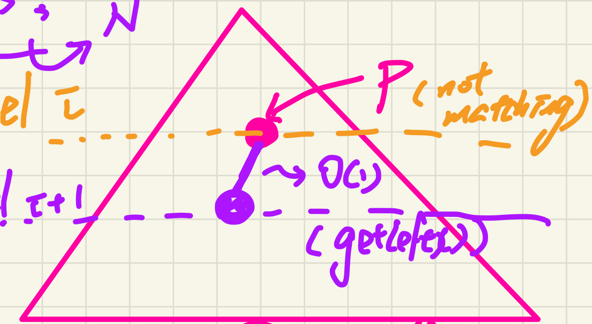
key to search

base case.

R1

R2

recursive cases

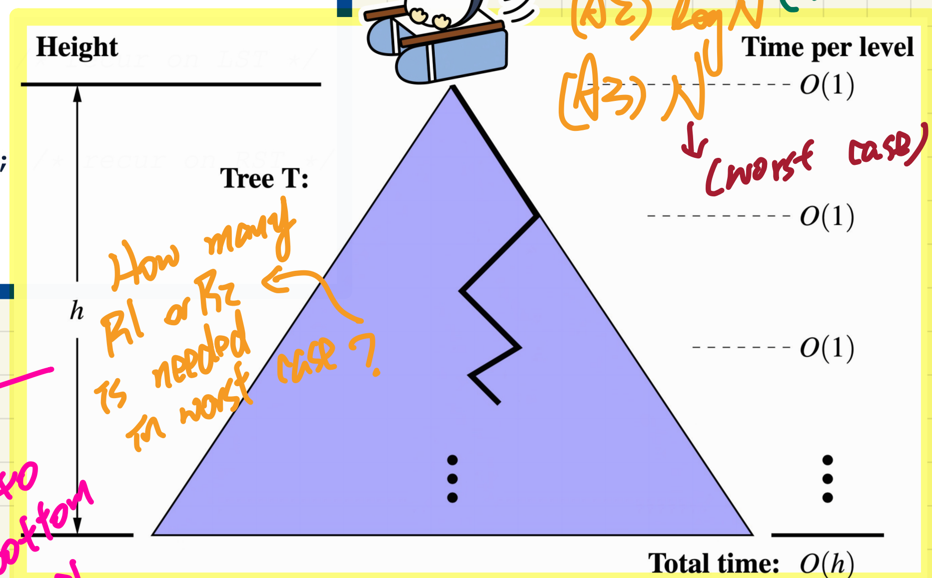


(A1) h accurate.

(A2) log N (best case)

(A3) N

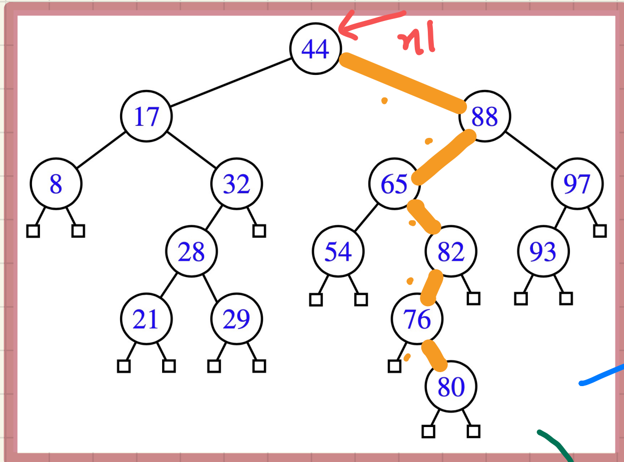
(worst case)



How many R1 or R2 is needed in worst case?

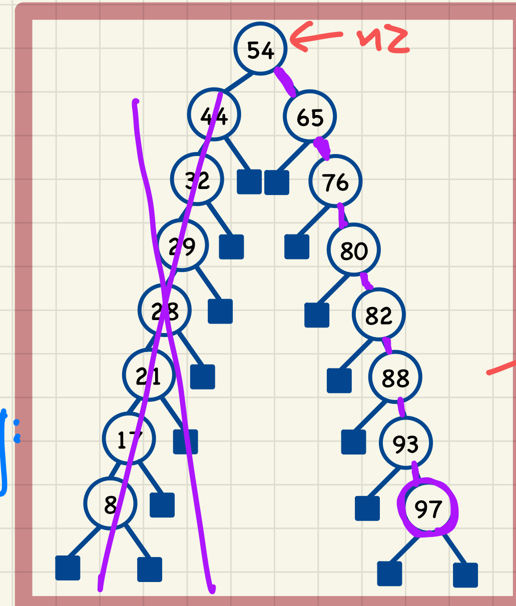
worst case: unsuccessful search ⇒ have to get down to the bottom level.

Binary Search: Non-Linear vs. Linear Structures



$N = 15$

$h = 5 \leq \log N$



$N = 15$

$h = 7 \approx \frac{N}{2}$

→ worst case: $O(N)$

best case of searching: $O(\log N)$

In-order(n1) = In-order(n2) = a

search(97)

corresponds to

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

Lecture

Binary Search Tree (BST)

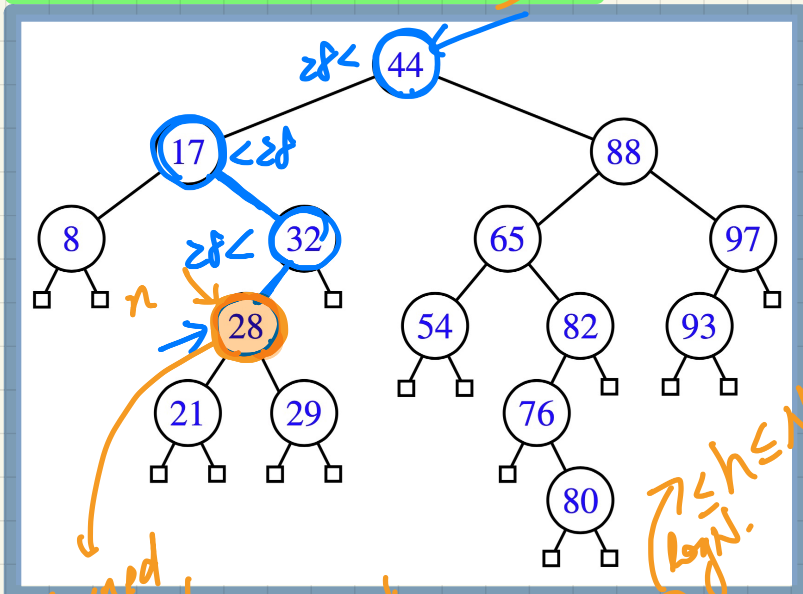
Implementing a Generic BST in Java Insertion

Visualizing BST Operation: Insertion

Insert Entry (28, "suyeon")

should be non-existing.

↳ otherwise, do some replacement



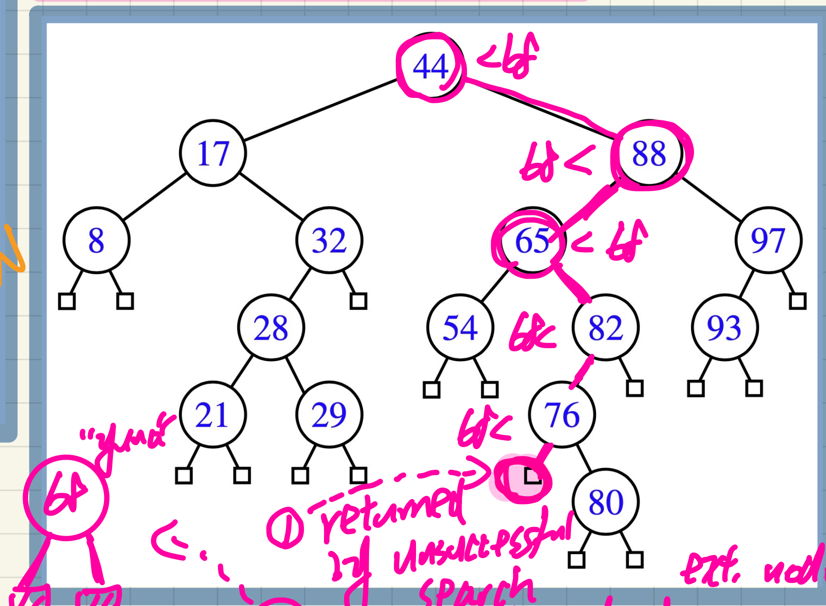
① returned by the search method

② set n's element to "suyeon"

RT: O(h)

↳ searching.

Insert Entry (68, "yuna")



① returned by unsuccessful search

② store key and ele. to this ext. node

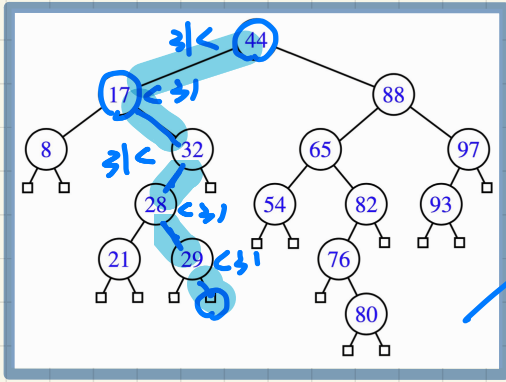
Lecture

Binary Search Tree (BST)

Implementing a Generic BST in Java
Deletion

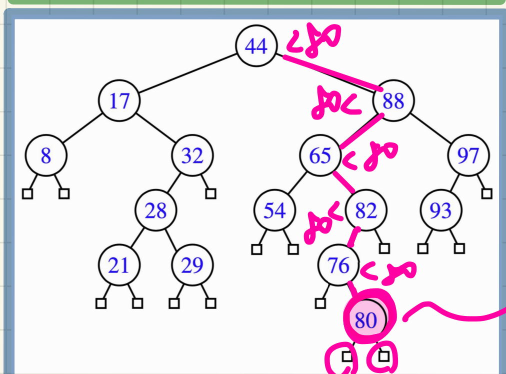
Visualizing BST Operation: Deletion

Case 1: Delete Entry with Key 31



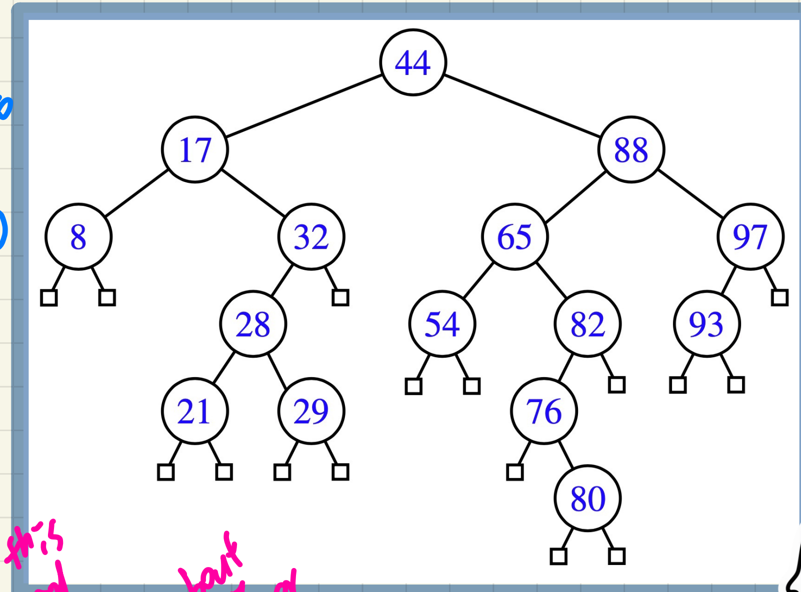
nothing to delete
RT: O(h)

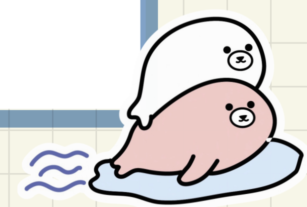
Case 2: Delete Entry with Key 80



can delete this right away without worrying about any LST or RST
⇒ RT: O(h)

Case 3: Delete Entry with Key 32





Lecture 23 - Wednesday, April 5

Announcements

- ProgTest1: Jackie (Office Hour)
- Assignment 4 released
- Exam guide to be released

BST

- search property
- sorting property (in-order traversal)
- searching
  - recursive
  - RT: accurate $O(h)$
    best case $O(\log n)$
    worst case $O(N)$
- Insertion (searching)

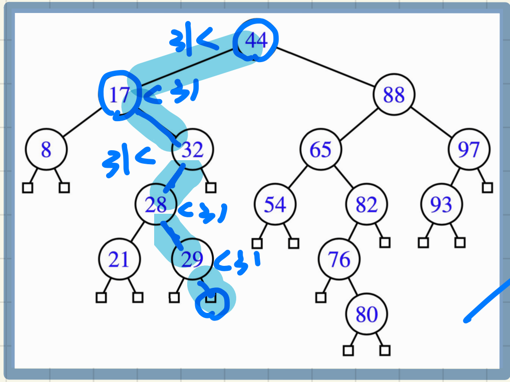
Visualizing BST Operation: Deletion

→ Exercise: Implement Case 3

Before deleting 32: 8 17 21 28 29 ~~32~~ 44

After deleting 32: 8 17 21 28 29 44

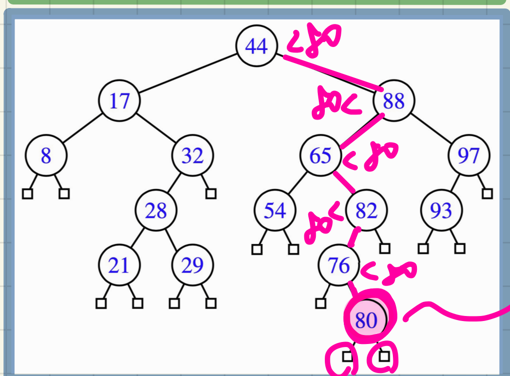
Case 1: Delete Entry with Key 31



nothing to delete

RT: O(h)

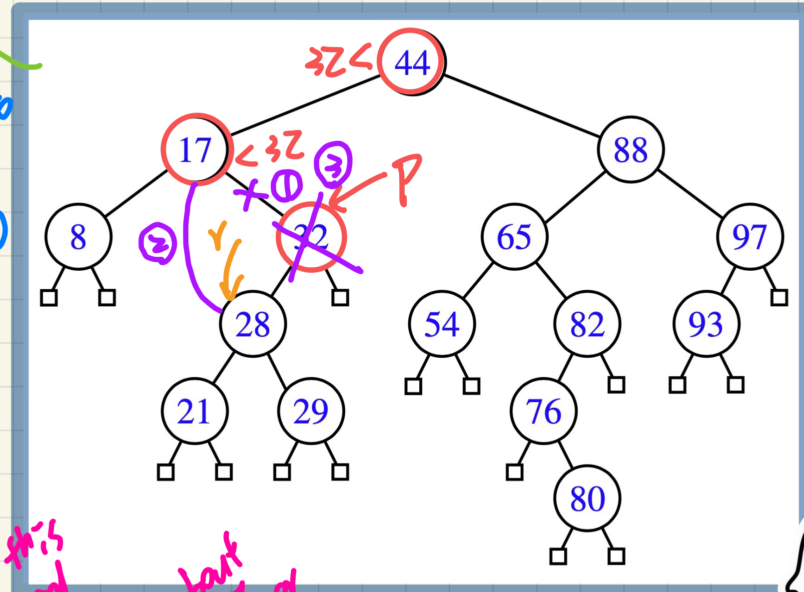
Case 2: Delete Entry with Key 80



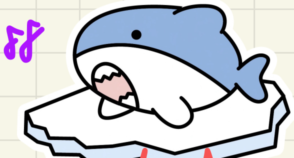
can delete this right away without worrying about any LST or RST

⇒ RT: O(h)

Case 3: Delete Entry with Key 32



Visualizing BST Operation: Deletion

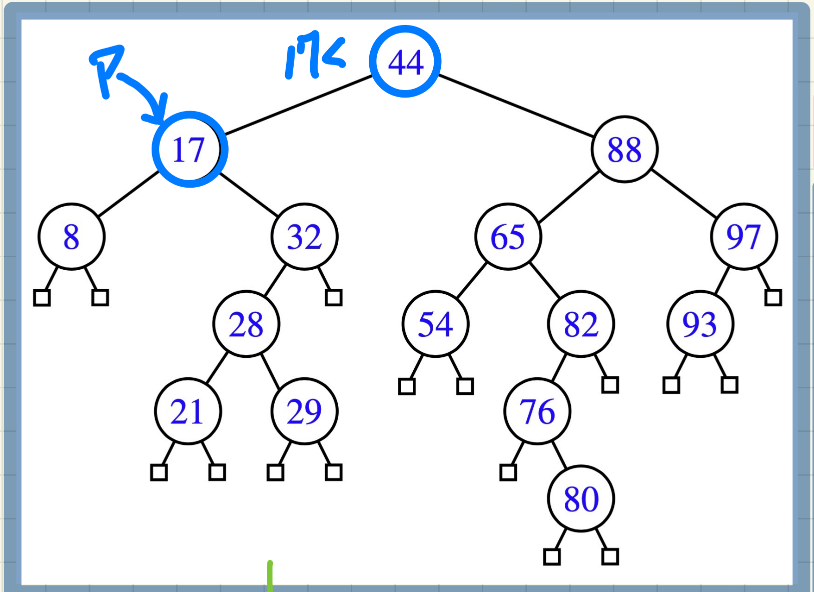


largest key that's smaller than 88

Before deleting 88: two keys before and after 88

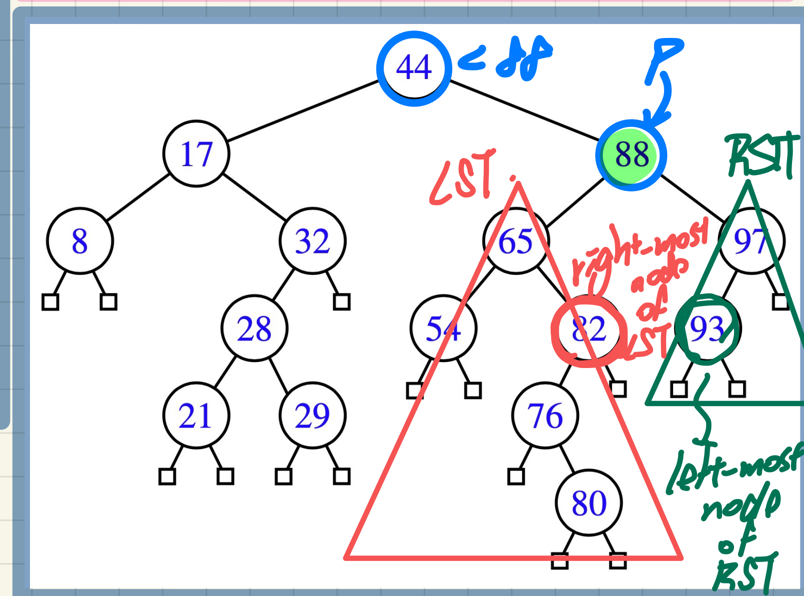
... 44 54 65 76 80 82 88 93 97

Case 4.1: Delete Entry with Key 17

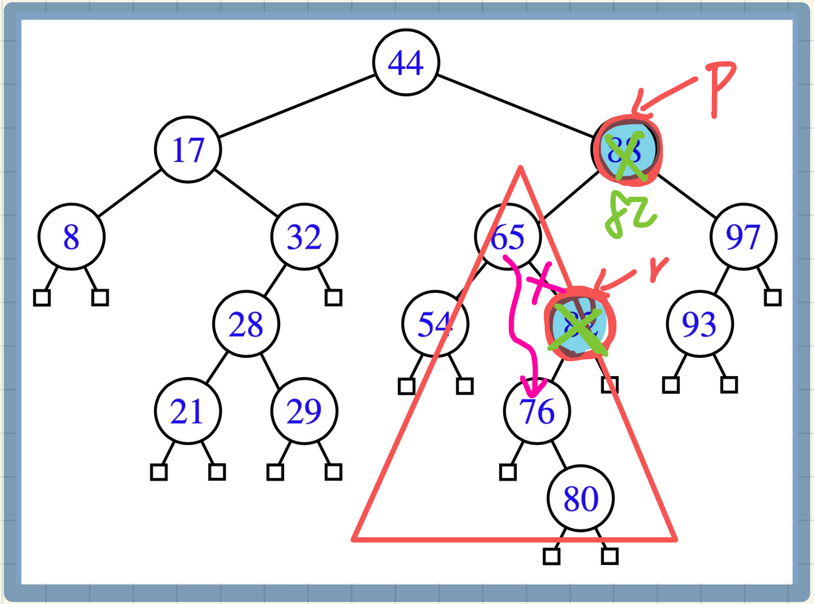


Exercise

Case 4.2: Delete Entry with Key 88



Case 4.2: Delete Entry with Key 88



p: to delete
r: largest key < 88

Choosing 82 or 93 works:
the resulting in-order traversals are identical.

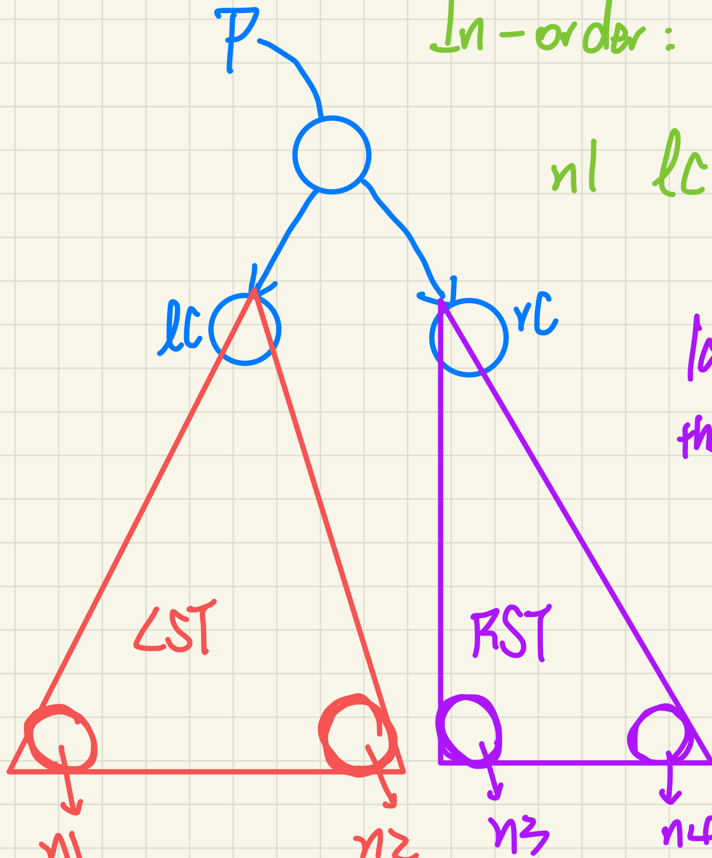
① Replace 88 by 82

② Connect 76 as the child of 65.

Exercise

Compare the in-order traversal results before & after the deletion steps.

BST



In-order :

n1 lc $\boxed{n2}$ p $\boxed{n3}$ rc n4

n2: largest that's < p

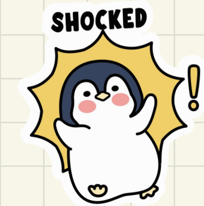
n3: smallest that's > p

Lecture

Balanced Binary Search Tree

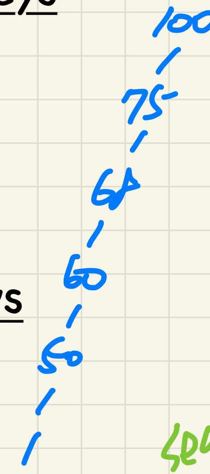
Motivation and Property

Worst-Case RT: BST with Linear Height



Example 1: Inserted Entries with Decreasing Keys

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



BST with height O(N)

searching/insertion/deletion can be O(N)

Example 2: Inserted Entries with Increasing Keys

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

exercises

Example 3: Inserted Entries with In-Between Keys

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

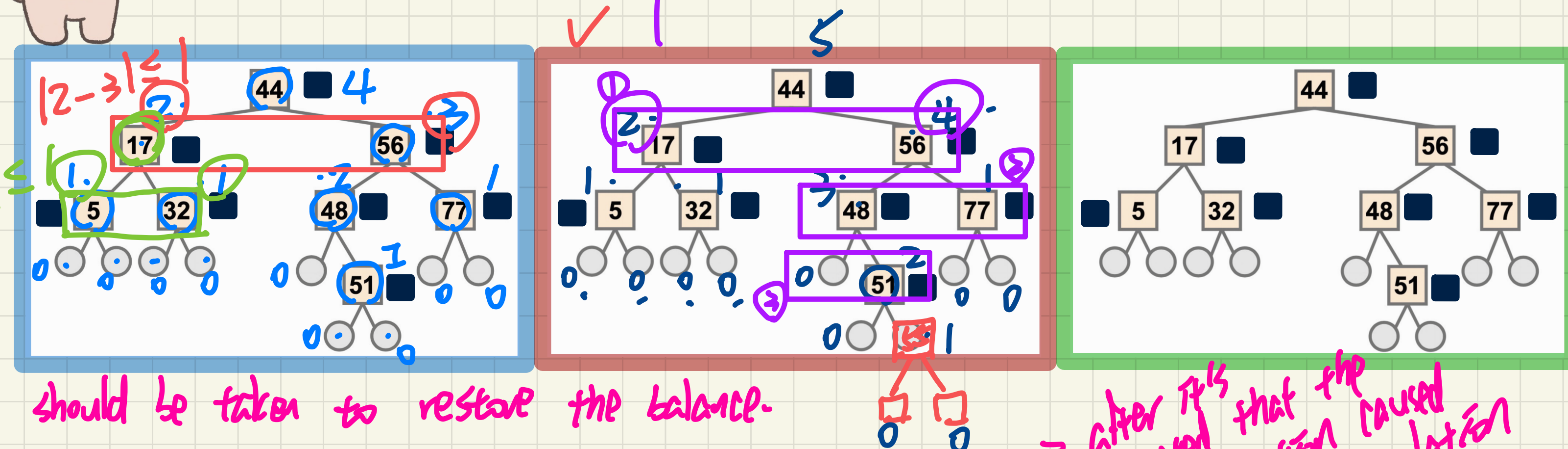
Balanced BST: Definition

- internal node
- height
- height balance

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

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

① violates height-balance prop.
$|2-4| > 1$



* should be taken to restore the balance.

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

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

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

after it's detected that the insertion caused a violation of the height-bal prop., some steps*

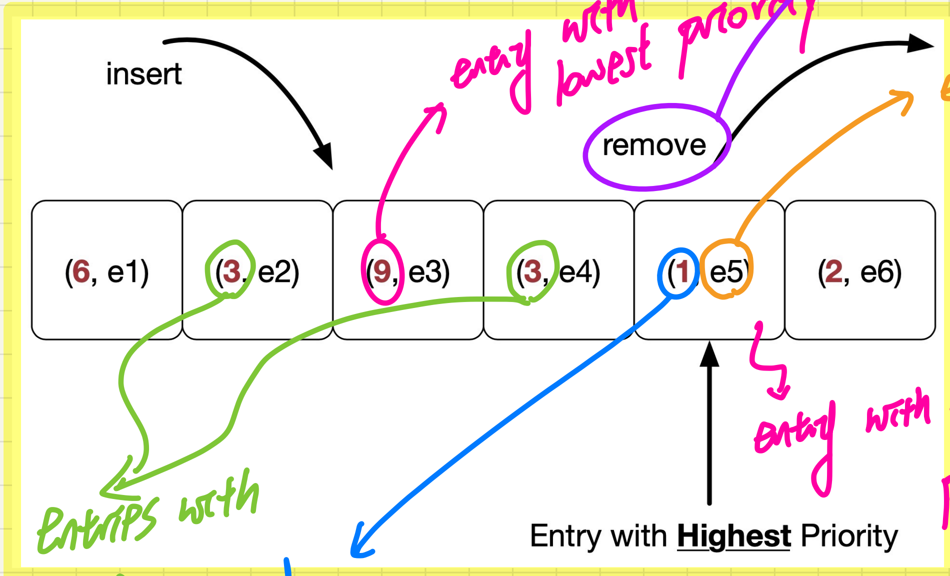
exercise

Lecture

Priority Queue

Intro & List-Based Implementations

What is a Priority Queue (PQ)



entry with lowest priority

remove entry with the highest priority (i.e., lowest key value)

element

entry with highest priority

entries with the same priority. (when needed, doesn't matter which one is chosen)

key, denoting the priority

Compare PQ with FIFO queue

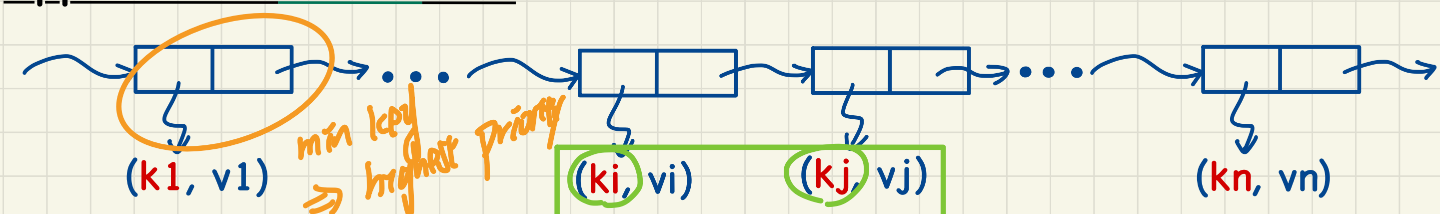
- entries in a FIFO queue is returned in chronological order.
- entries in a PQ is returned according to key val.

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)*

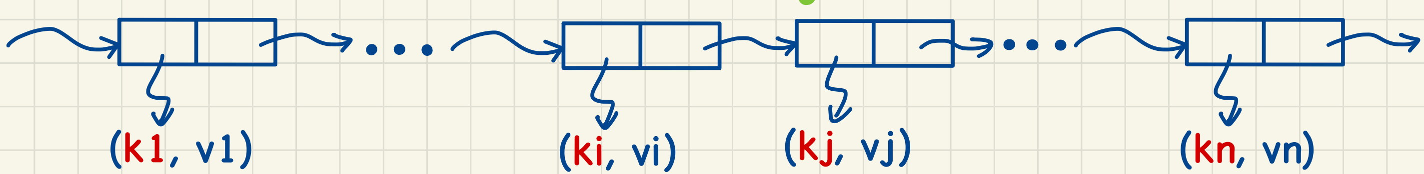
highest priority

Approach 1: Sorted List



$k_i \le k_j$

Approach 2: Unsorted List



Lecture 24 -
Makeup for ProgTest2

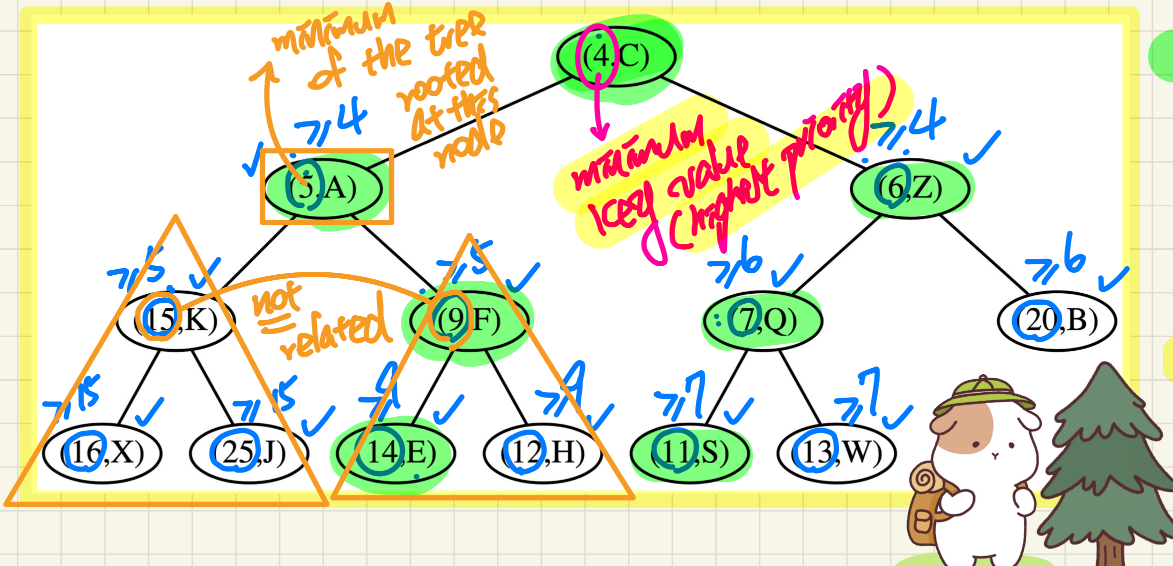
Lecture

Priority Queue

Heaps -
Examples and Properties

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.

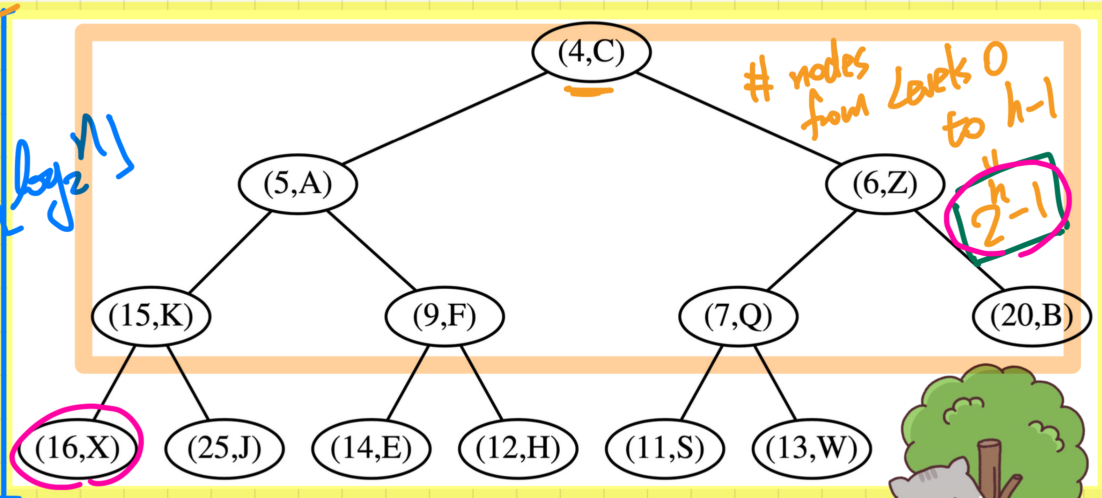
P2. the minimum key exists in the root entry.

P3. key values between LST and RST are not related.

Heaps: Structural Properties of Nodes

Property: The tree is a complete Binary Tree

$h = \lfloor \log_2 n \rfloor$



$n = 13$

$\lfloor \log_2 n \rfloor = \lfloor 3. \cdots \rfloor = 3$

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

Max # of nodes: $(2^h - 1) + 2^h$

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

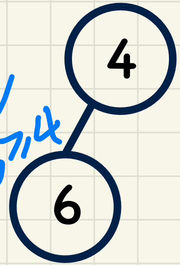
# of nodes at level $h = n - (2^h - 1)$

Example Heaps < relational / structural

Example 1

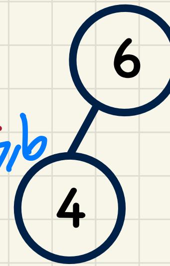
✓ smallest possible one-node heap

Example 2



✓ $6 \geq 4$ ✓

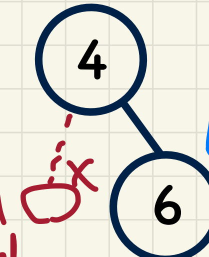
Example 3



✗ $4 \geq 6$ ✗

violating the relational property

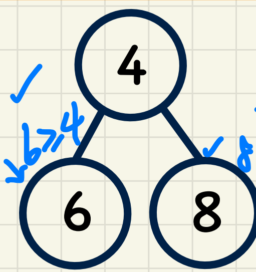
Example 4



✓ $6 \geq 4$

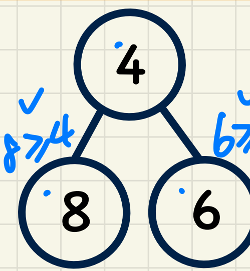
violating structural property

Example 5



✓ $6 \geq 4$ ✓ $8 \geq 4$

Example 6



✓ $8 \geq 4$ ✓ $6 \geq 4$ ✓

full BTs ⇒ complete BTs

Lecture

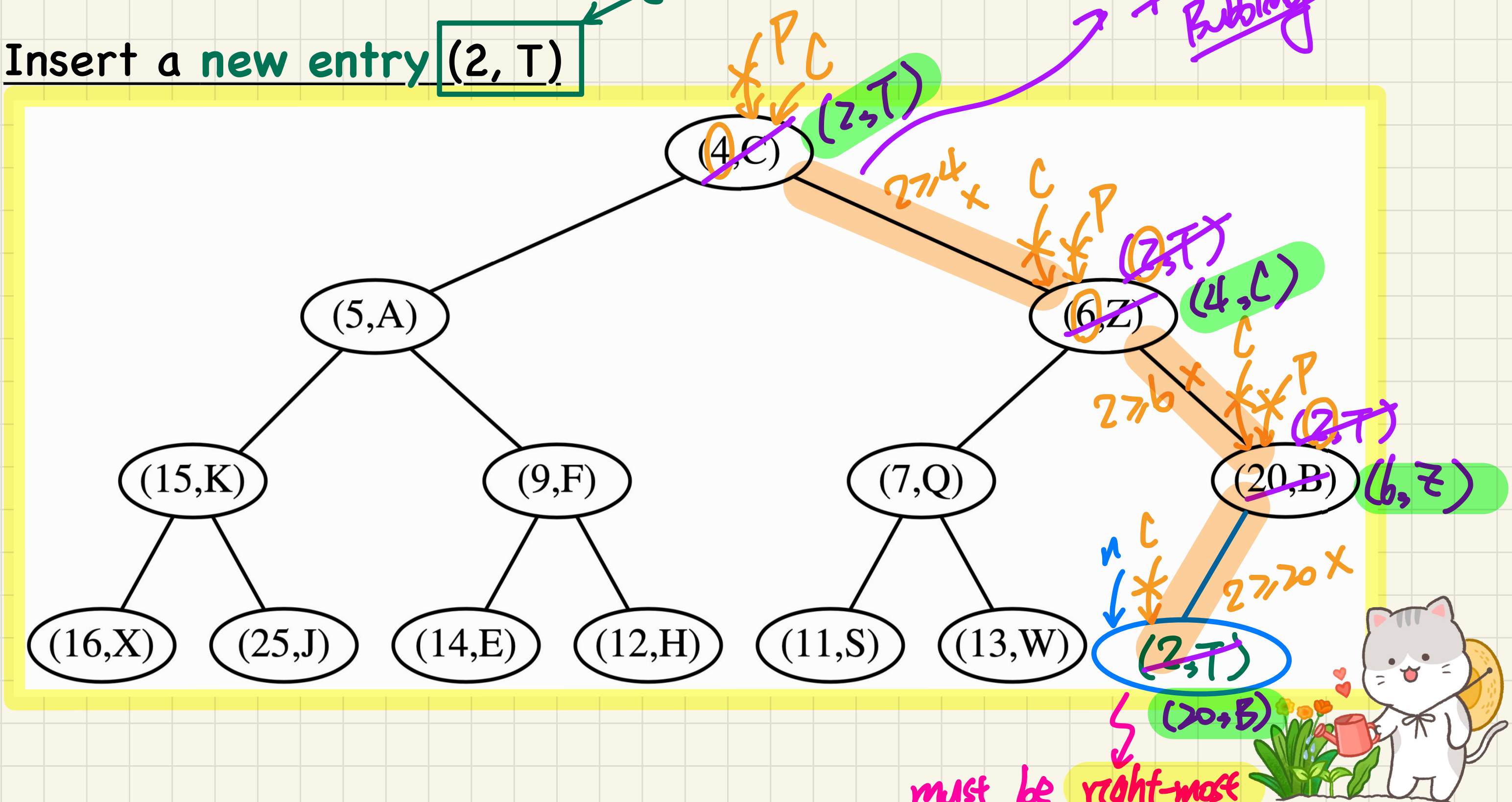
Priority Queue

Heaps - Insertions

Heap Operations: Insertion

Insert a new entry $\boxed{(2, T)}$ ← e

Up-Heap Bubbling.



must be right-most at level h in order to preserve structural property

Lecture

Priority_Queue

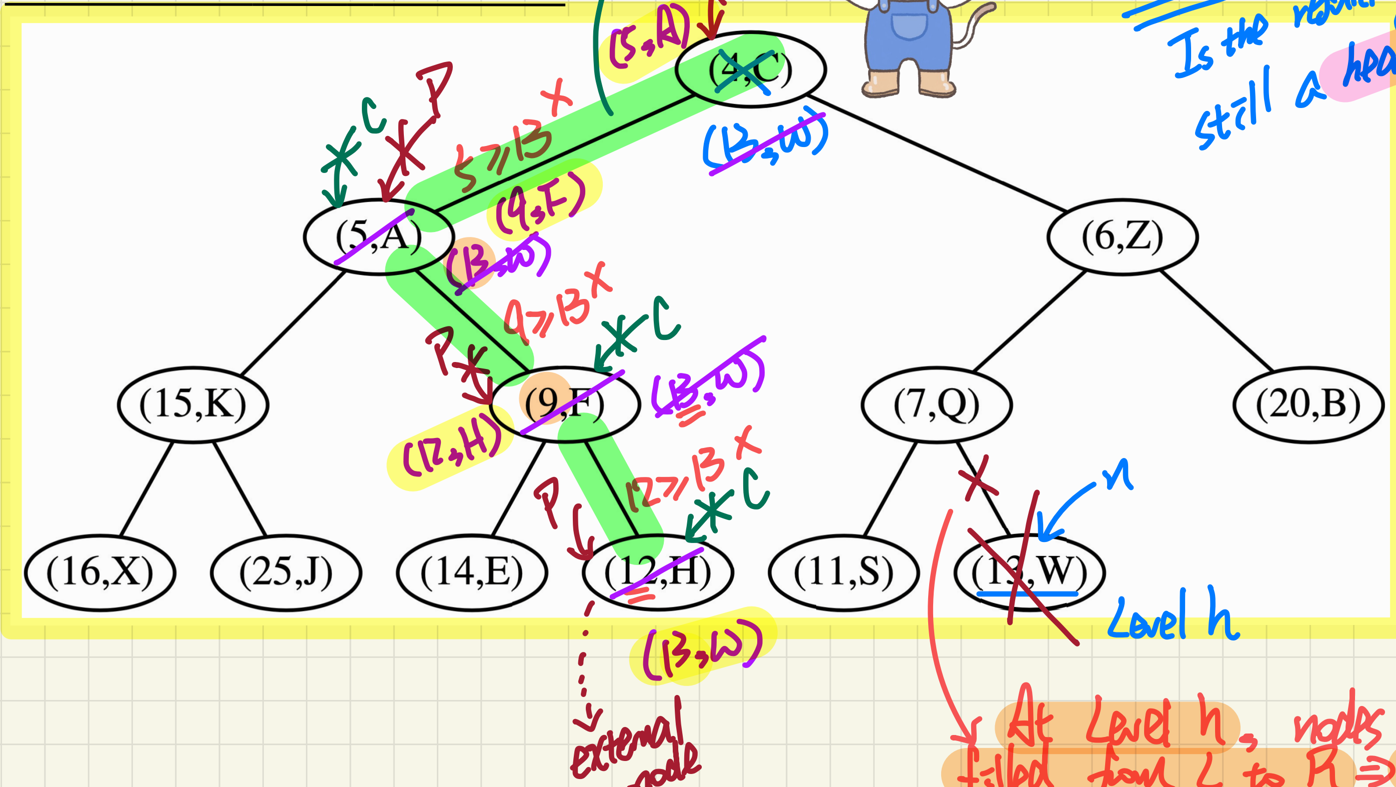
Heaps -
Deletions

Heap Operations: Deletion

root-to-leaf path (down-heap bubbling)

Exercise: Is the resulting tree still a heap?

Delete the root/minimum



At Level h, nodes are still filled from L to R ⇒ complete BT

Lecture

Priority Queue

Heaps - Top-Down Heap Construction

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***.

first inserted to level 2

RI : # nodes at level, # upheap bubbling steps

$$* \quad \underset{\text{root}}{1} + 2^1 \cdot 1 + 2^2 \cdot 2 + \dots + 2^h \cdot h$$

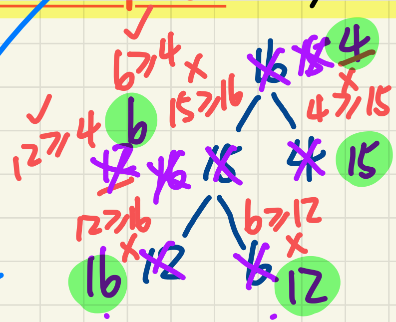
$\le \log n$, $\le \log n$, $\log n$

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 one at a time.

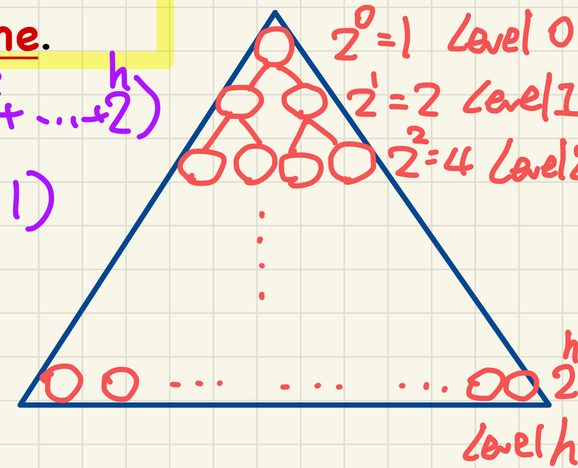
first inserted to level 1



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

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

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



Exercise: Complete inserting the remaining keys to the heap.

Lecture

Priority Queue

Heaps -
Bottom-Up Heap Construction

Bottom-Up Heap Construction

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

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

- **Assume**: The resulting heap will be *completely filled* at **all** levels.
  $\Rightarrow N = 2^{h+1} - 1$ for some *height* $h \geq 1$ [$h = (\log(N+1)) - 1$]
  15 = 3
- Perform the following steps called *Bottom-Up Heap Construction*:

  Step 1: Treat the first $\boxed{\frac{N+1}{2^1}}$ list entries as heap roots. 8

  $\therefore \frac{N+1}{2^1}$ heaps with height 0 and size $\boxed{2^1 - 1}$ constructed.

  Step 2: Treat the next $\boxed{\frac{N+1}{2^2}}$ list entries as heap roots. 4

  ◇ 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 $\boxed{2^2 - 1}$ constructed. 3

  25%

  12.5%

  Step 3: 4 heaps

  $\frac{16}{2^{3+1}} = 1$

  ...

  Step $h+1$: Treat next $\boxed{\frac{N+1}{2^{h+1}}} = \frac{(2^{h+1}-1)+1}{2^{h+1}} = 1$ 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.

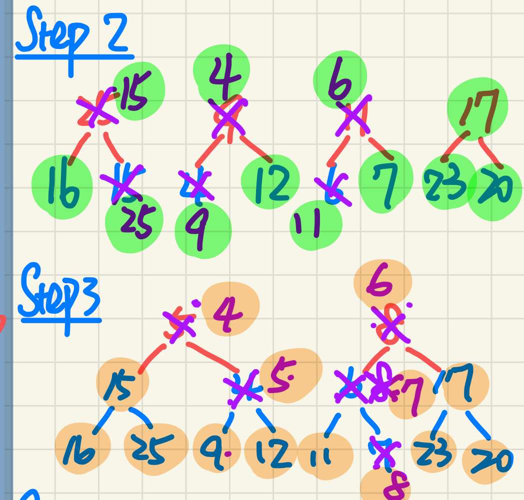
  $\therefore \boxed{\frac{N+1}{2^{h+1}} = 1}$ heap, each with height h and size $\boxed{2^{h+1} - 1}$ constructed.

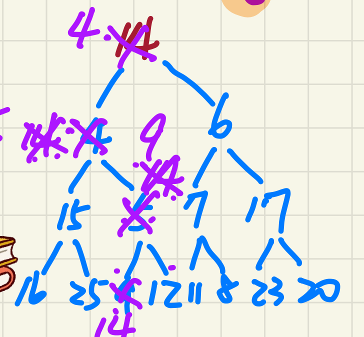
  $2^{3+1} - 1 = 15$

① $\frac{N+1}{2^3}$

② size of each heap: $2^3 - 1 = 7$

③ height of each heap: 2





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**.

Lecture

Priority Queue

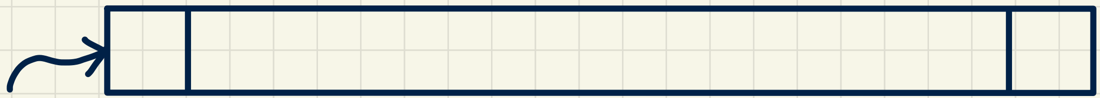
Heaps -
Heap Sort Algorithm

Heap Sort: Ideas

$O(N \cdot \log N)$

N entries

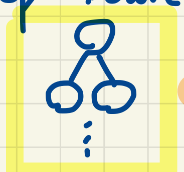
0 a.size() - 1



a

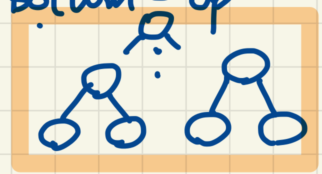
Construct a heap out of N entries

(A) Top-Down



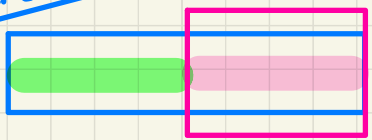
$O(N \cdot \log N)$

(B) Bottom-Up



$O(N)$

SelectionSort

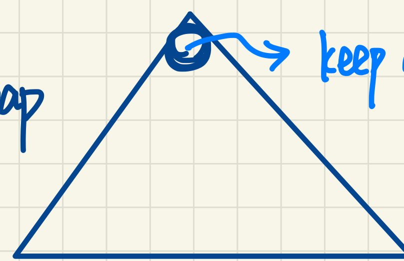


select the min from unsorted portion & put it to the front/end of the list

≈ Selection Sort

exploit the HOP (relational property): root stores entry with min key

heap



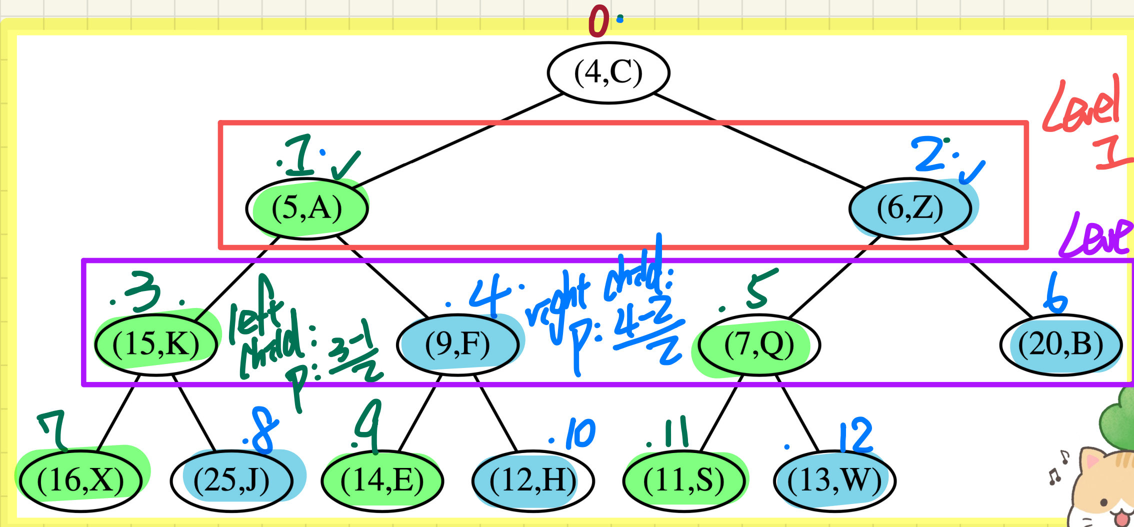
keep deleting the root until the heap is empty. N deletions, each $O(\log N) \Rightarrow O(N \cdot \log N)$

Lecture

Priority Queue

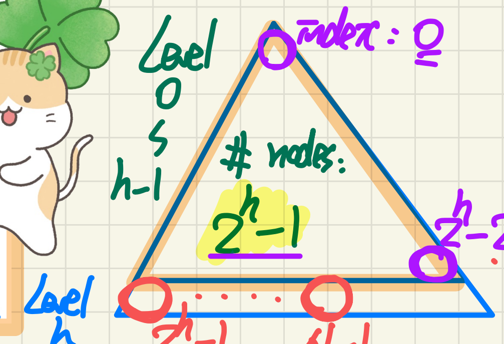
Heaps -
Array-Based Implementation

Array-Based Representation of a Complete BT



Exercise

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



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

L1 L2

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 !