EECS2011 Fundamentals of Data Structures

Lecture Notes

Winter 2022

Jackie Wang

Lecture 1

Part A

Measuring Running Time via Experiments

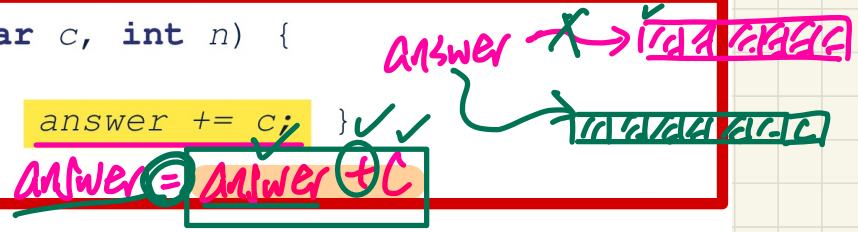
Example Experiment

Computational Problem:

- **Input**: A character *c* and an integer *n*
- **Output**: A string consisting of *n* repetitions of character *c*
  e.g., Given input `'*'` and `15`, output `***************`.

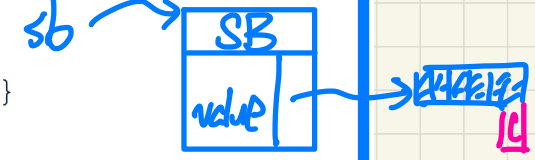
Algorithm 1 using *String* Concatenations:

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



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

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



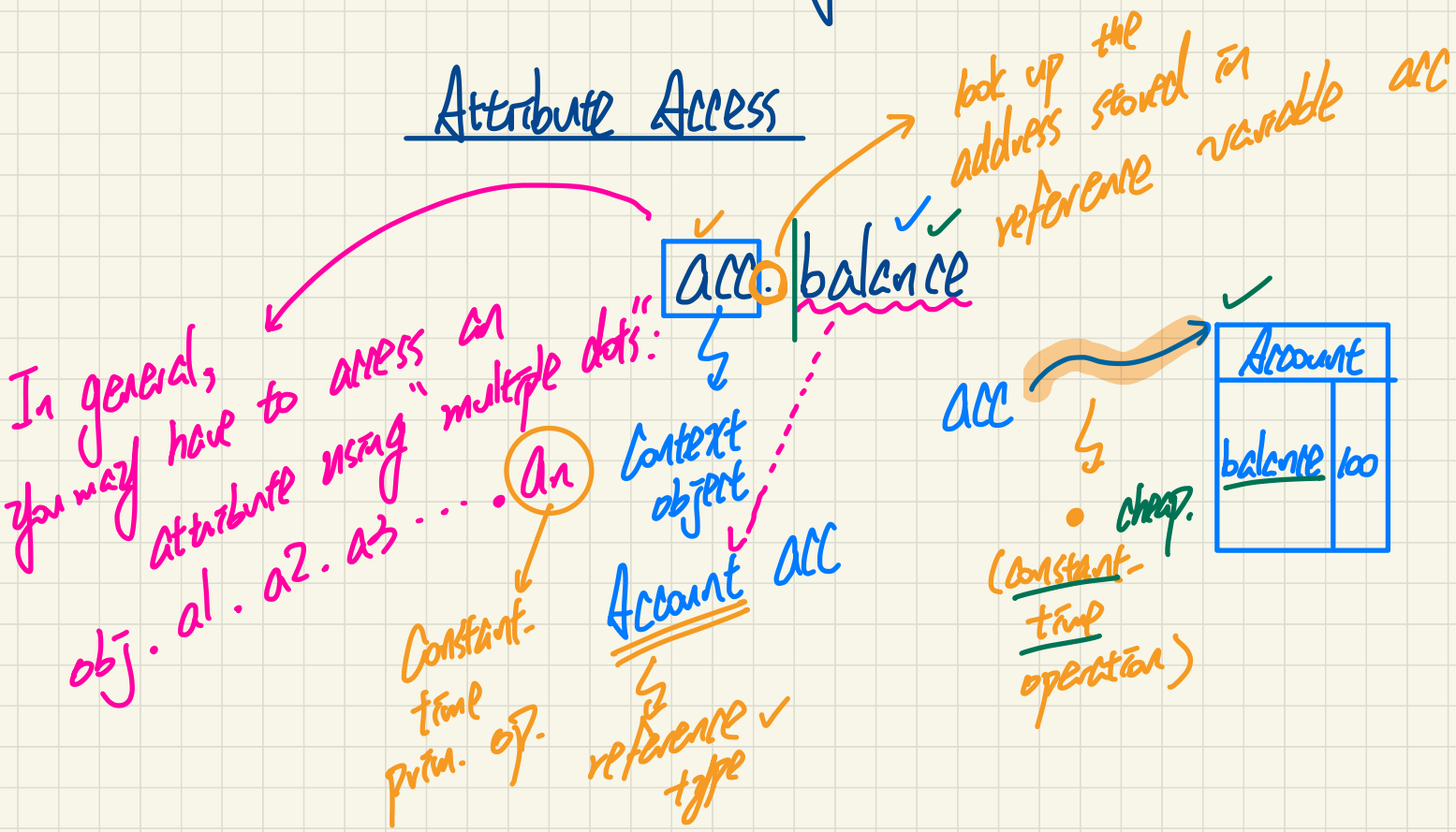
Lecture 1

Part B

Counting Primitive Operations

Primitive Operation (taking constant time)

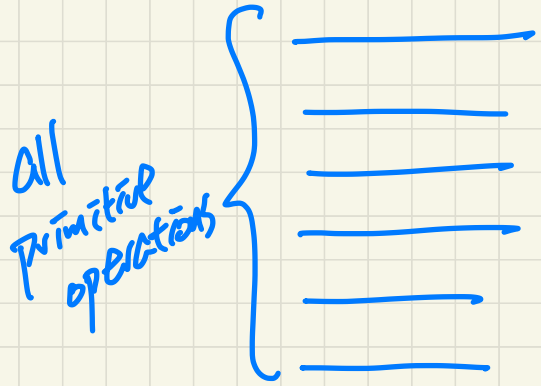
Attribute Access



Method Call obj.m();

Case 1 PO.

```
void m() {
```




```
}
```

Case 2 non-PO.

```
void m() {
.
.
for(int i; i<a.length; i++)
        ⋮
m2();
}
```

```
int findMax(int[] a, int n)
```

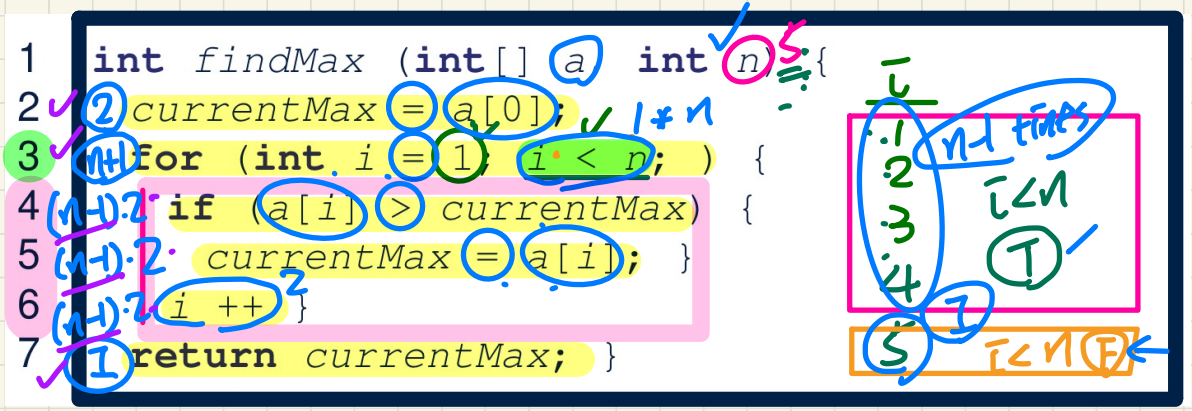
assumed to be a.length

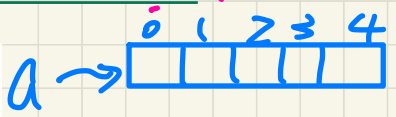
```
int[] seq = {2, -1, 7, 1, 10};
```

```
findMax(seq, 5)
```

seq.length.

Example 1: Counting Number of Primitive Operations





i++

i = i + 1

2 POs.

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

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

$= (n+3) + (6n-6) + 1$

$= 7n - 2$

$(n-1) + 1 = n$

(i<n evaluates to T) (i<n evaluates to F.)

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

$n - 1$ → i<n → F → exit from loop

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  }
```

method call (in this case a PO).

String[] names;

i
0
1
2
⋮
names.length - 1

i < names.length (T)

names.length | i < vs.length (F)

Q. # of times Line 3 is executed?

(T) names.length + (F)

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

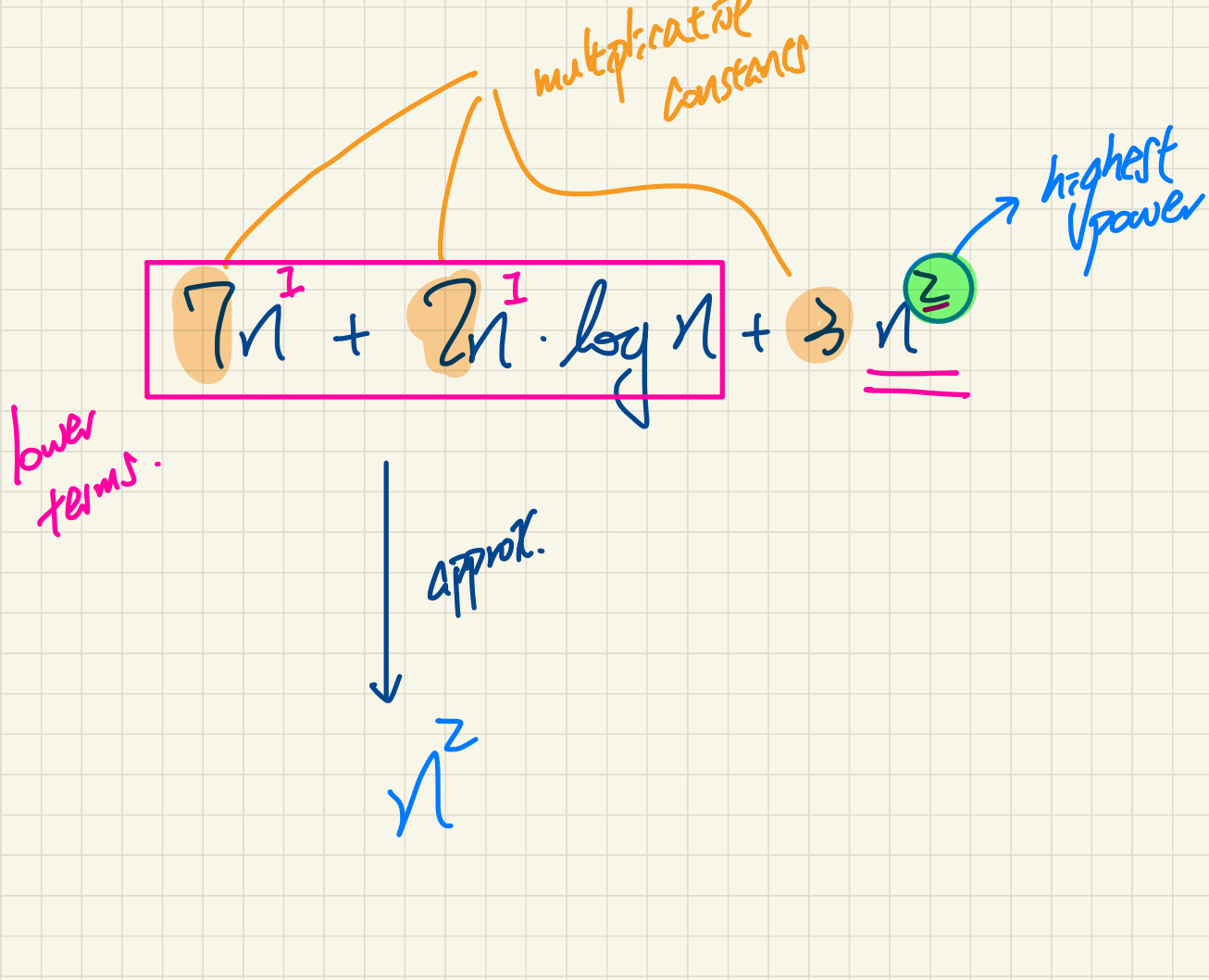
names.length

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

Lecture 1

Part C

Asymptotic Upper Bound


multiplicative constants
$7n^1 + 2n^1 \cdot \log n + 3n^2$
highest power
lower terms.
approx.
n^2

Asymptotic Upper Bound: Big-O

need to be identified in order to prove that $f(n) \in O(g(n))$

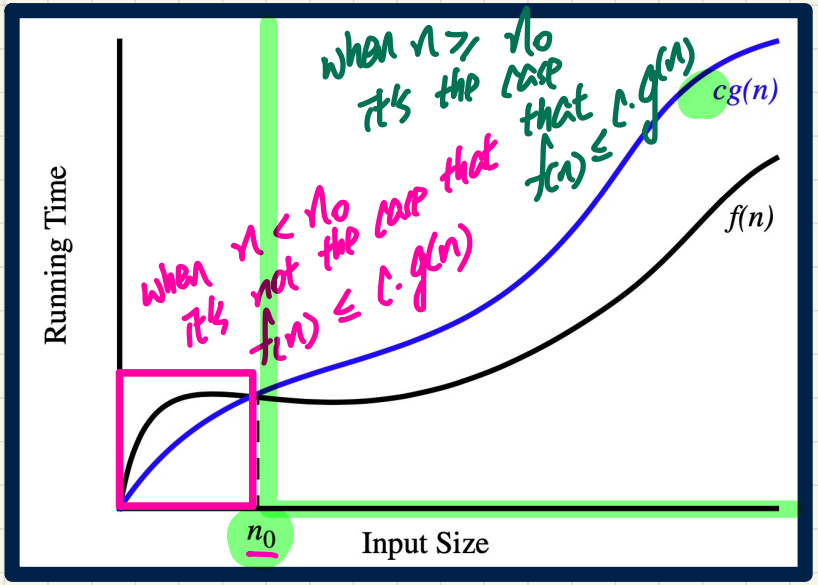
$f(n) \in O(g(n))$ if there are:

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

such that:

upper-bound effect

$f(n) \leq c \cdot g(n)$ for $n \geq n_0$



Example:

f(n) = 8n + 5

g(n) = n

Prove:

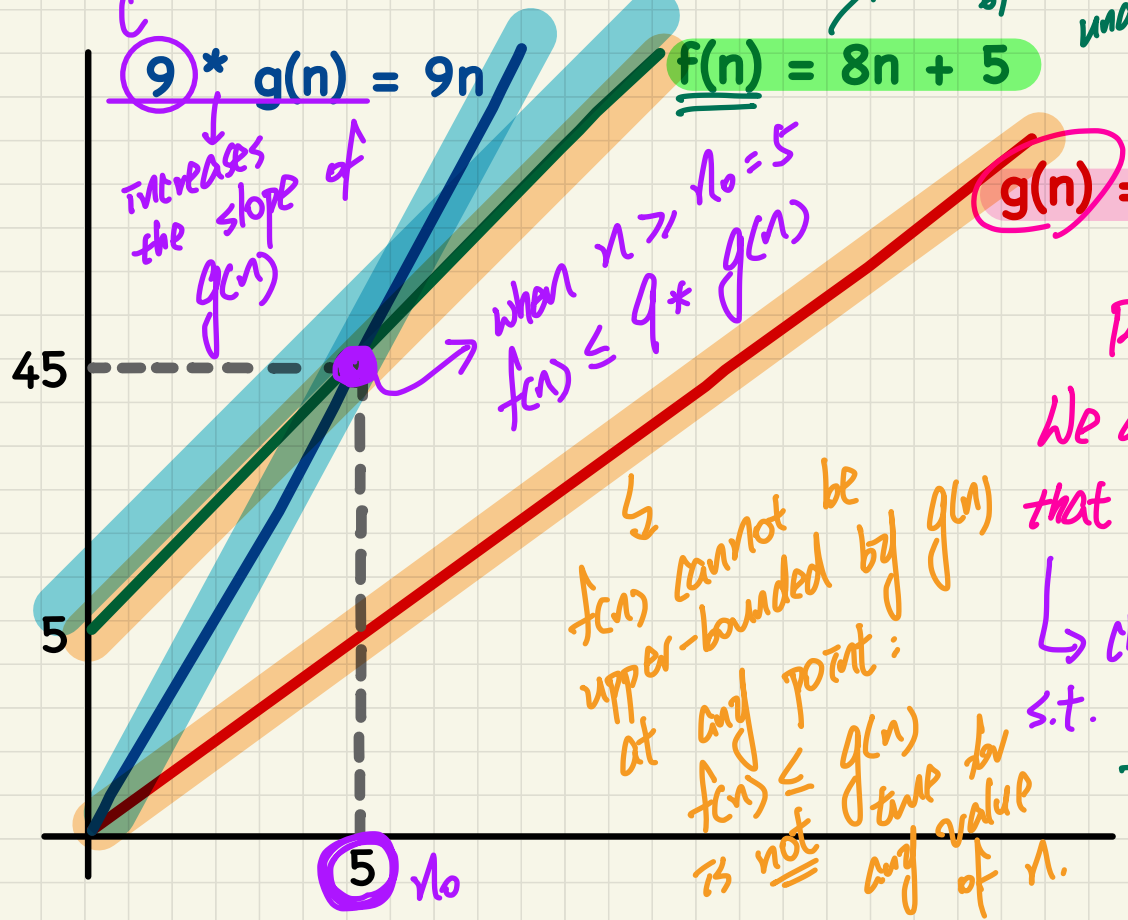
f(n) is O(g(n))

Choose:

c = 9

What about n0?

Asymptotic Upper Bound: Example



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

We prove by choosing

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

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

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

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

① int i $\quad i \le |i|$

② int i $\quad i^x \le i^y$ $\quad x \le y$

$2^3 \le 2^4$

Upper-bound effect: $n_0 = 1$?

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

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

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

Upper-bound effect holds?

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

($n \ge 1 = n_0$, $c = |a_0| + |a_1| + \cdots + |a_d|$)

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

Proper Subset

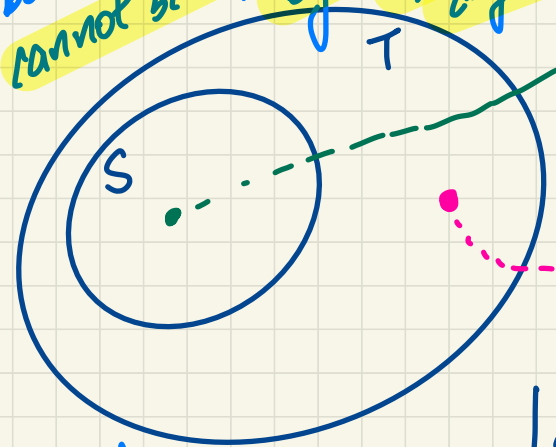
If a function is upper-bounded by n^x (e.g. n)

Can that function be e.g. n^0 upper-bounded by n^y $(y<x)$?

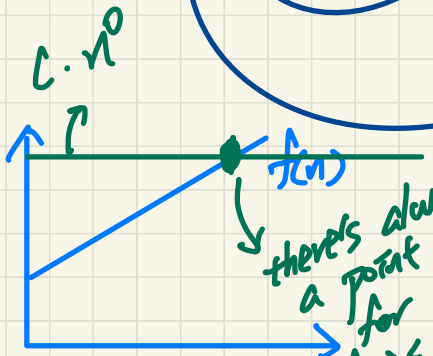
e.g. $f(n) = 3n+1$ is upper bounded by n, but it cannot be upper-bounded by n^0 for any c and n_0.

$S \subset T$

1. Every member of S is also a member of T.
2. There is at least one member of T that's not a member of S. (Not necessarily)

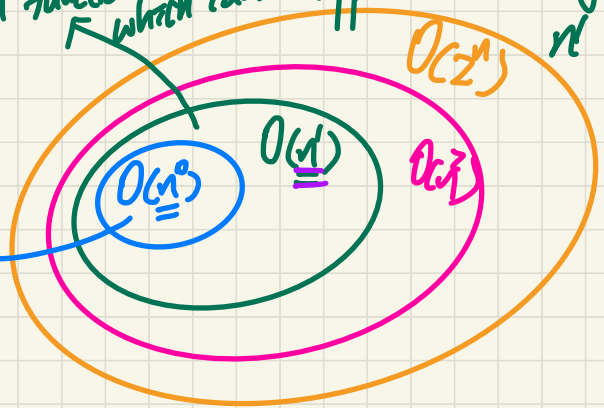


$|S| < |T|$



there's always a point for $f(n) \leq c \cdot n^0$ to be false

all functions which can be upper-bounded by n^0

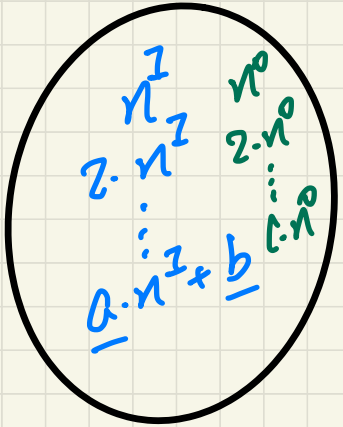


O(g(n)): A Set of Functions

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

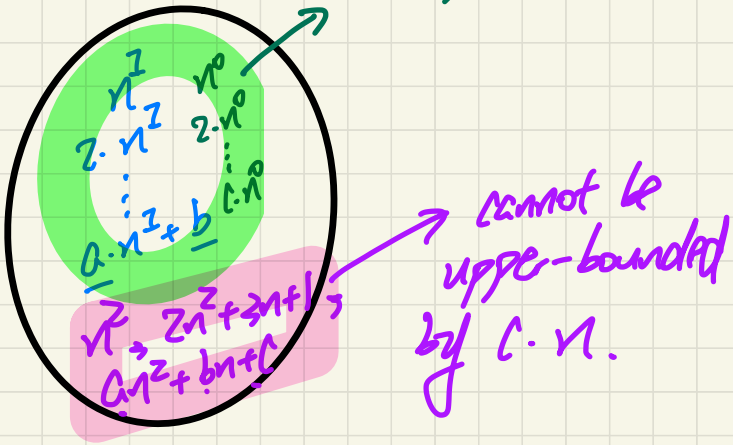
Higest Power of f(n) <= Highest Power of g(n)

O(n^1)



C

O(n^2)



Lecture 1

Part D

Asymptotic Upper Bounds of Math Functions

Asymptotic Upper Bounds: Example (1)

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

$O(n^2)$

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

$n_0 = 1$

$f(n) \leq 15 \cdot n^2$ for $n \geq 1$

Asymptotic Upper Bounds: Example (2)

$20n^3 + 10n \cdot \log n + 5$ is $O(n^3)$

$O(n^3)$

$C = |20| + |10| + |5| = 35$

$n_0 = 1$

$f(n) \leq 35 \cdot n^3$ for $n \geq 1$

Asymptotic Upper Bounds: Example (3)

$3 \cdot \log n + 2$ is $O(\log n)$

$O(\log n)$

$g(n) = 1$

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

$n_0 = 1 \times 2$

$\log 1 = 0$

$\because 2^0 = 1$

$\log 2 = 1$

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

$f(1) \leq 5 \cdot \log 1$

$3 \cdot \log 1 + 2$

2

0

False

$f(2) \leq 5 \cdot \log 2$

$3 \cdot \log 2 + 2$

5

5

True

Asymptotic Upper Bounds: Example (4)

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

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

4

$C = |4| = 4$

$n_0 = 1$

Asymptotic Upper Bounds: Example (5)

$2n^1 + 100 \cdot \log n$ is $O(n)$

$O(n)$

$C = |2| + |100| = 102$

$n_0 = 1$

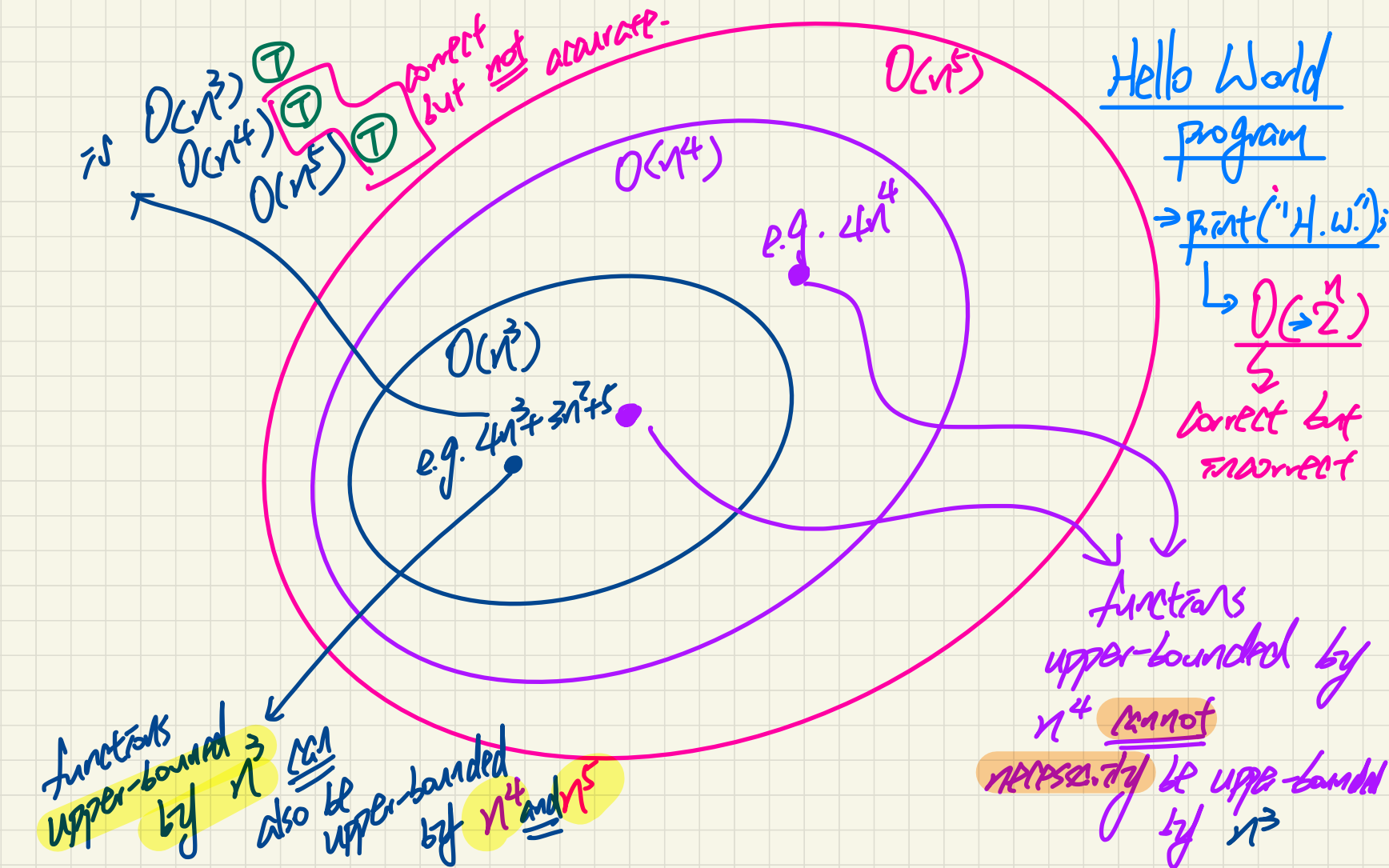
Exercise: Check if the upper-bound effect starts to hold when $n = n_0 = 1$:

$$f(1) \leq 102 \cdot n$$

Lecture 1

Part E

Asymptotic Upper Bounds of Implemented Algorithms


Hello World program
→ Print("H.W.");
↳ O(→2^n)
↳ correct but inaccurate
O(n^5)
O(n^4)
O(n^3)
e.g. 4n^4
e.g. 4n^3+3n^2+5
T
T
T
correct but not accurate
O(n^3)
O(n^4)
O(n^5)
=>
functions upper-bounded by n^4 cannot necessarily be upper-bounded by n^3
functions upper-bounded by n^3 can also be upper-bounded by n^4 and n^5

$4n^2 + 6n + 9 \in O(n^2)$ ✓

$4n^2 + 6n + 9 \in O(4n^2 + 3n)$ ×

Determining the Asymptotic Upper Bound (1)

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

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

$O(1+1+1+1) = O(1)$

$4 = 4 \cdot n^0$

Determining the Asymptotic Upper Bound (2)

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

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

$= O(2 + 3n)$

$= O(n)$

Determining the Asymptotic Upper Bound (3)

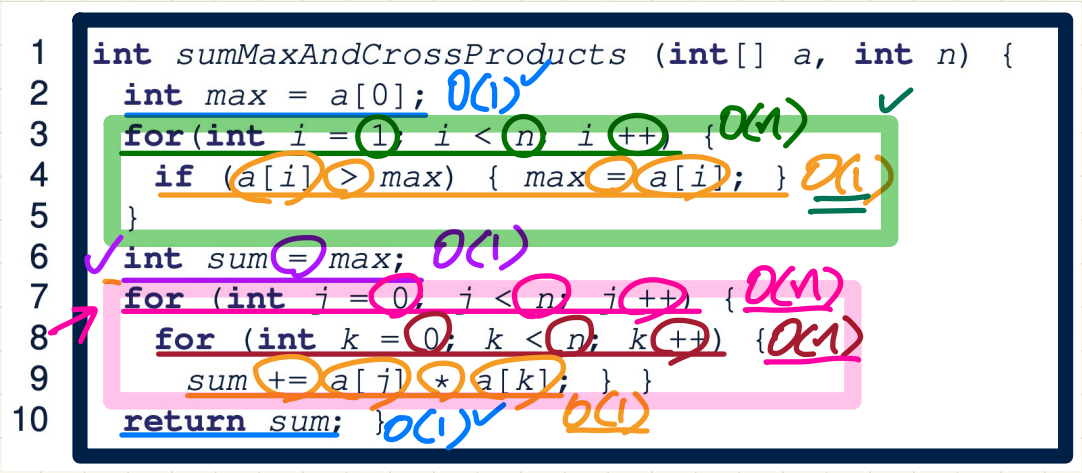
```
1  boolean containsDuplicate (int[] a, int n) {
2    for (int i = 0; i < n; ) {
3      for (int j = 0; j < n; ) {
4        if (i != j && a[i] == a[j]) {
5          return true; }
6        j ++; }
7      i ++; }
8    return false; }
```

$$O(n \cdot (\underbrace{n \cdot (1+1+1)+1}_{3n+1}) + 1)$$

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

$$= O(n^2)$$

Determining the Asymptotic Upper Bound (4)



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

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

$$= O(n^2)$$

How many #s in $[a, b] = b - a + 1$

Determining the Asymptotic Upper Bound (5)

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

outer-loop

i
0
1
2
⋮
n-1

inner loop when i=0: $0, 1, \dots, (n-1)$

inner loop when i=1: $1, \dots, (n-1)$

inner loop when i=2: $2, \dots, (n-1)$

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

(when i=0, when i=1, when i=n-1)

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

Sum of Arithmetic Sequence

$$1 + 2 + 3 + \cdots + n = \frac{(1+n) \cdot n}{2}$$

$$i + (i+c) + (i+2c) + (i+3c) + \cdots + (i + (n-1)c)$$

$i + 0 \cdot c$, $+c$, $+c$, $+c$

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

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

$$= O(n^2)$$

Lecture 2

Part A

Asymptotic Upper Bounds of Array Operations

Inserting into an Array

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

assume: $0 \le i \le a.length$ (n) $- 1$

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

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

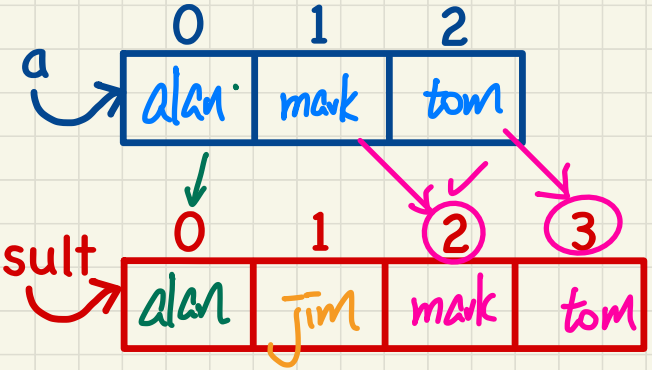
a, n, e, i

n: max # of iterations

when $i = 0$

Example:

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



result[2] = a[2-1]

result[3] = a[3-1]

RT:

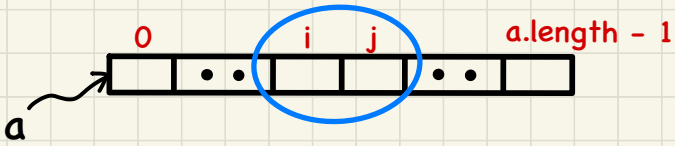
$O(1+n+1+n+1) = O(n)$

Lecture 2

Part B

Asymptotic Upper Bounds
Selection Sort vs. Insertion Sort

Sorting Orders of Arrays

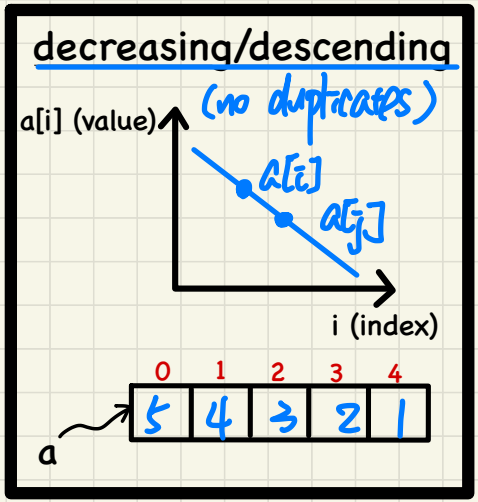


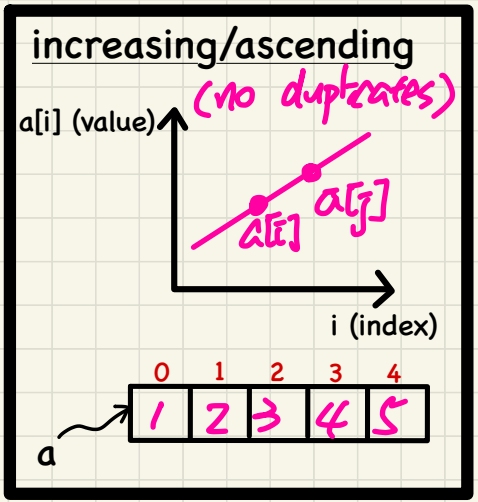
decreasing/descending : $a[i] > a[j]$

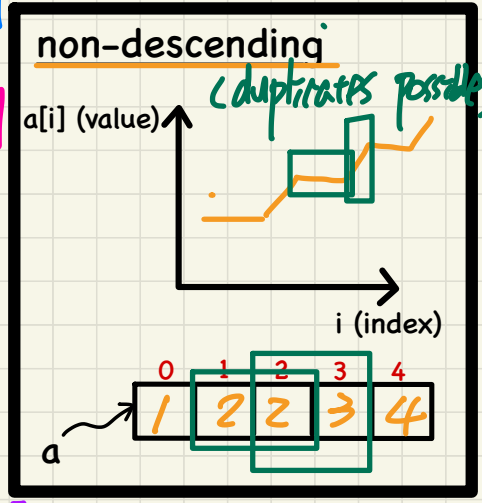
increasing/ascending : $a[i] < a[j]$

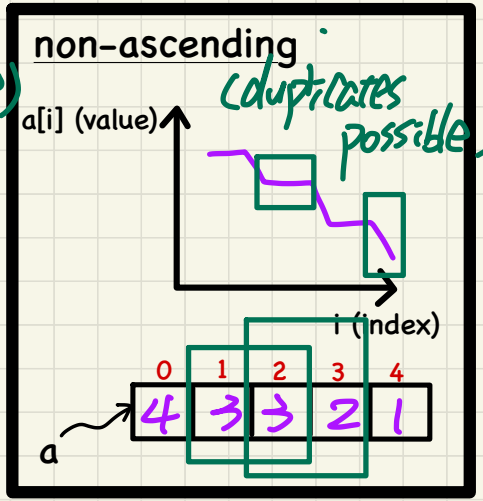
non-descending : $\neg(a[i] > a[j])$
$\equiv a[i] \leq a[j]$

non-ascending : $\neg(a[i] < a[j])$
$\equiv a[i] \geq a[j]$









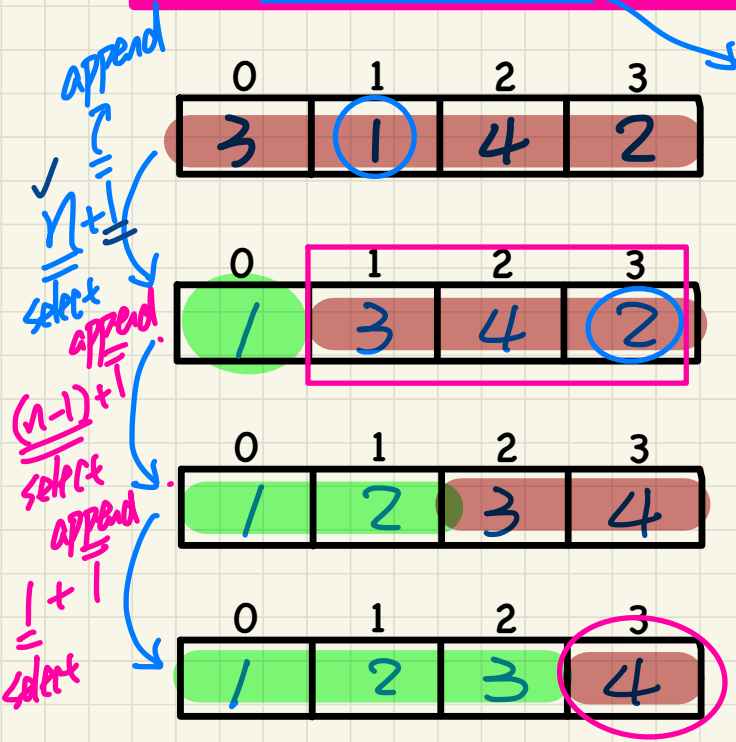
Selection Sort

more expensive

from L to R

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

cheaper

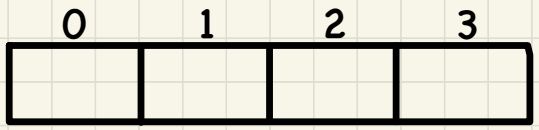


unsorted

sorted

$O(\underbrace{n}_{\text{append to end of sorted portion}} + \underbrace{(n+1)}_{\text{1st selection}} + \underbrace{n}_{\text{2nd selection}} + \cdots + 1)$

$= O(n^2)$

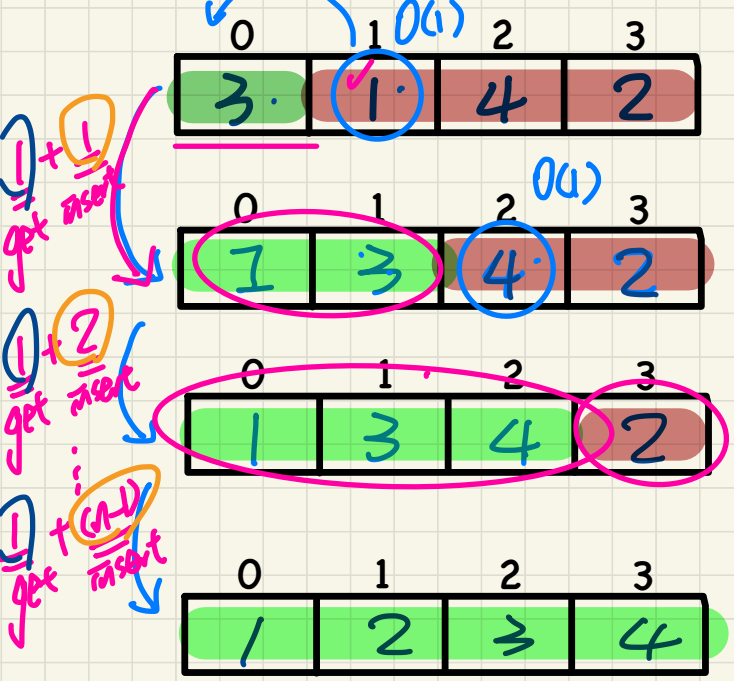


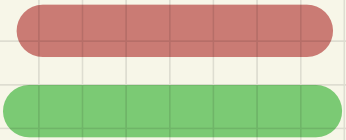
Insertion Sort

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

cheaper

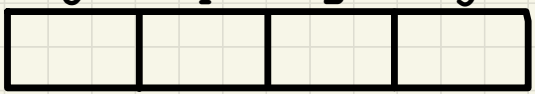
more expensive





$$O(\underbrace{n}_{\text{get}} + (\underbrace{1}_{\text{1st insert}} + \underbrace{2}_{\text{2nd insert}} + \cdots + (n-1)) = O(n^2)$$

get 1st elements of unsorted portion

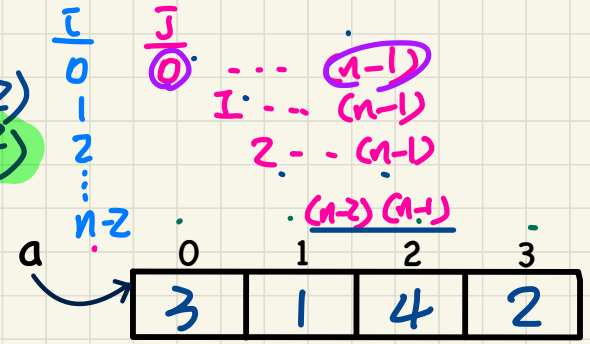


Selection Sort in Java

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

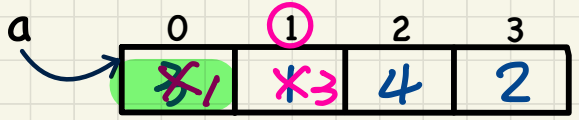
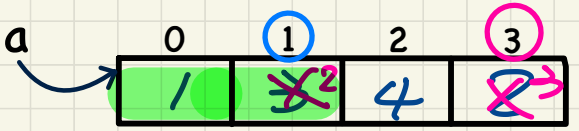
```
1  void selectionSort(int[] a, int n)
2    for (int i = 0; i <= (n - 2); i ++)
3      int minIndex = i;
4      for (int j = i; j <= (n - 1); j ++)
5        if (a[j] < a[minIndex]) { minIndex = j; }
6      int temp = a[i];
7      a[i] = a[minIndex];
8      a[minIndex] = temp;
```

swap a[i] and a[minIndex]



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, (before i++)
a is sorted from a[0] to a[i].

i	inner loop: j from ? to ?	midIndex at L6	after L6 - L8, a becomes?
0	0 1 .. (n-1)	1	
1	1 .. (n-1)	3	

Insertion Sort in Java

$O(2+3+4+\dots+n) = O(n^2)$

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

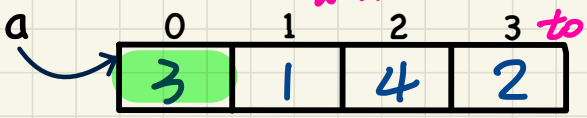
exit when: $\neg(j>0 \wedge a[j-1]>c)$

$= (j \le 0) \vee a[j-1] \le c$

↓ worst case for while-loop to exit

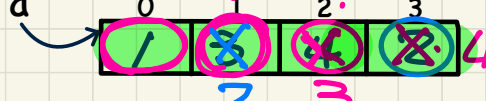
```
1  void insertionSort(int[] a, int n)
2      for (int i = 1; i < n; i ++)
3          int current = a[i];
4          int j = i;
5          while (j > 0 && a[j - 1] > current)
6              a[j] = a[j - 1];
7              j --;
8          a[j] = current;
```

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



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, (before i++)
a is sorted from a[0] to a[i].

i	current after L3	j at L8	after L8, a becomes?
1	a[1] (1)	0	
2	a[2] 4	2	
3	a[3] (2)	1	

I J
1 0
2 1 0
2 1 0
n-1 (n-1) ⋯ 0

In-Place Sorting

↳ Sort by modifying directly the input array (without intermediate storage)

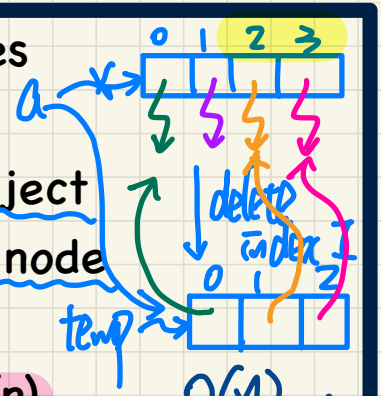
Lecture 2

Part C

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



O(n) always.

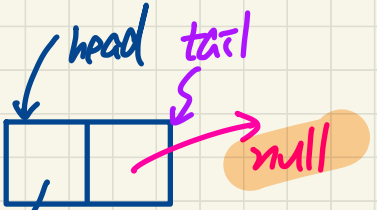
O(1) LL

int[] a = new int[_];

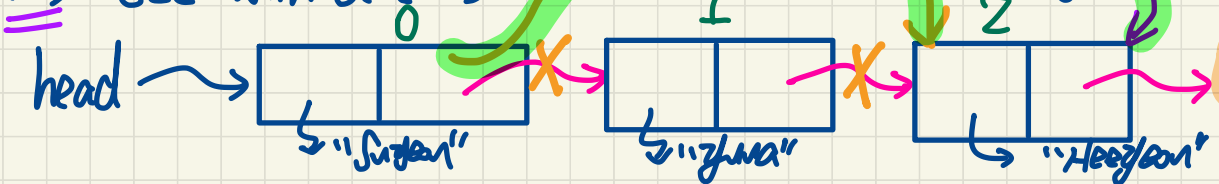
fixed length

(1) Empty SLL

(2) SLL with size 1



(3) SLL with size 3



"Sufyan"

tail Ref. Aliasing

tail == head.next

True.

Lecture 2

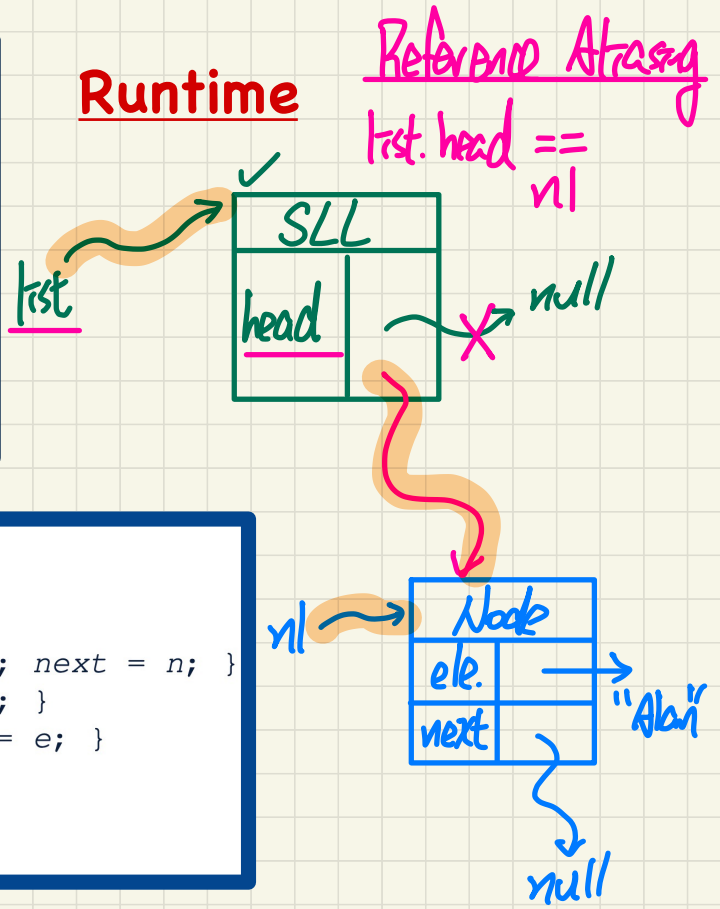
Part D

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

Implementing SLL in Java: SinglyLinkedList vs. Node

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

```
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; }
}
```



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; }
}
```

n = tom

n = mark

this.next

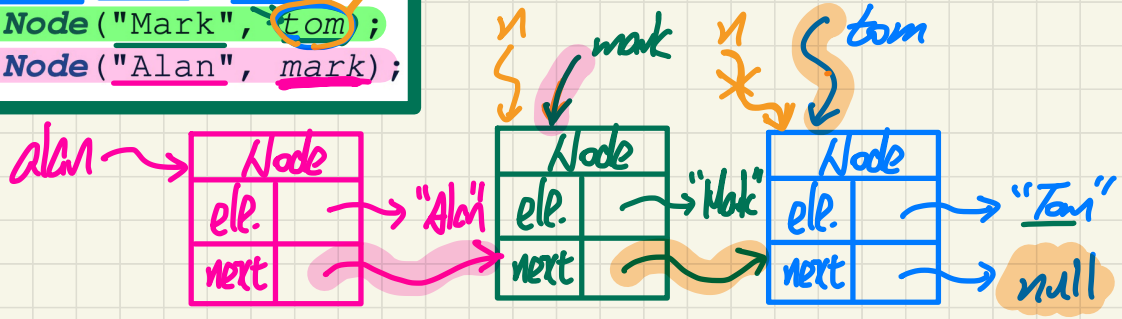
Ref. Aliasing

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

call by value

Approach 1

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



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; }
}
```

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);
```

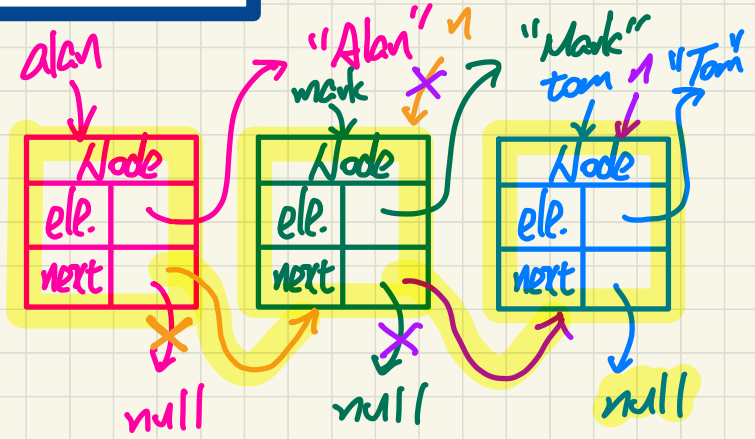
context object

call by value

alan.next = mark

n = mark

n = tom



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

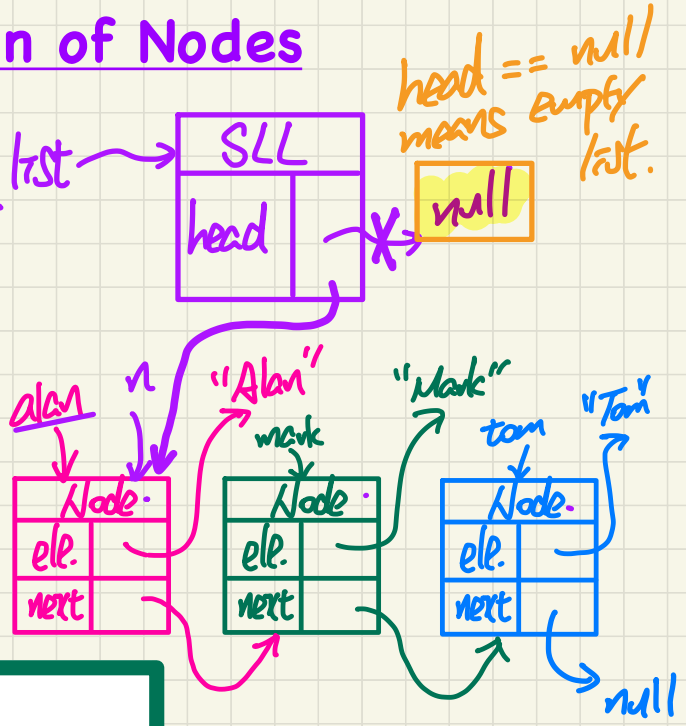
```
public class SinglyLinkedList {
   private Node head = null;
   public void setHead(Node n) { head = n; }
   public int getSize() { ... }
   public Node getTail() { ... }
   public void addFirst(String e) { ... }
   public Node getNodeAt(int i) { ... }
   public void addAt(int i, String e) { ...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

n = alan last this.head = n.

Approach 1

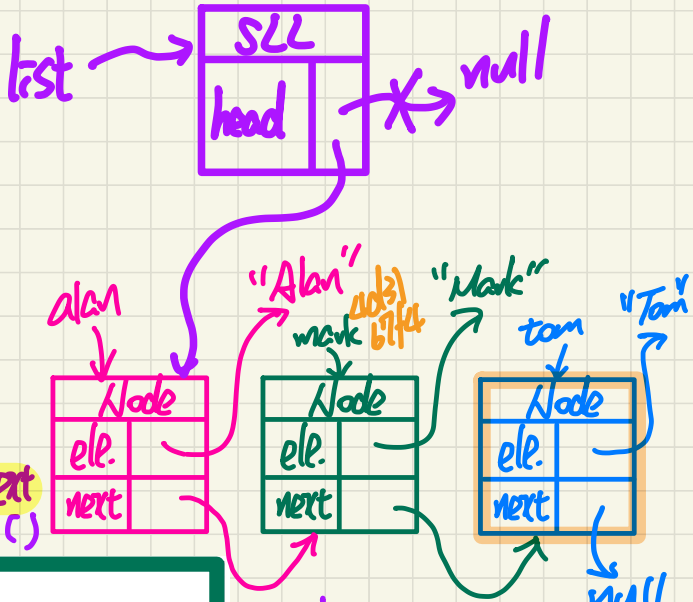
```
Node tom = new Node("Tom", null);
Node mark = new Node("Mark", tom);
Node alan = new Node("Alan", mark);
SinglyLinkedList list = new SinglyLinkedList();
list.setHead(alan);
```

call by value



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

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



Approach 2

④ list.getFirst().getNext().getNext()

```
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);
```

Q. How many paths to reach "Tom" object?

① tom
② mark.getNext()
③ alan.getNext().getNext()

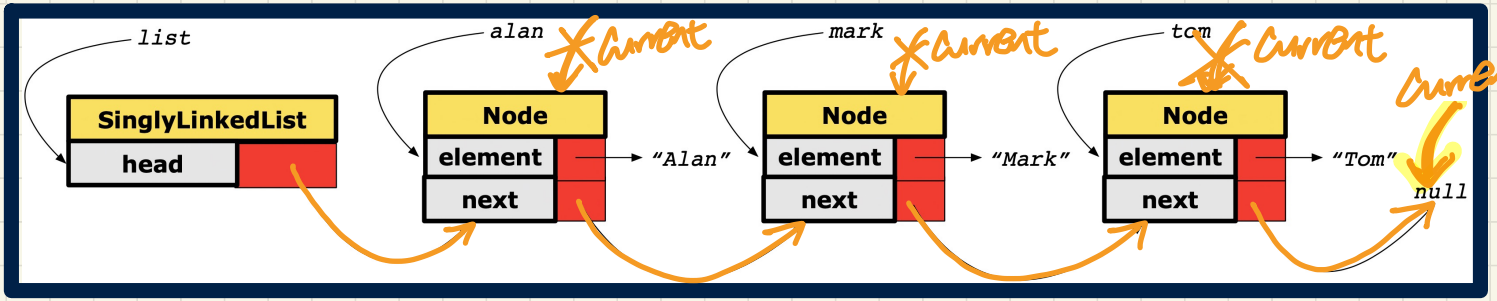
Lecture 2

Part E

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

SLL Operation: Counting the Number of Nodes

i = i + 1




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

Exit: current == null

O(1)

starting from 1st node in chain, we keep calling getNext() until current becomes null

Trace: list.getSize()

current	current != null	End of Iteration	size
alan	true	1	1
mark	true	2	2
tom	true	3	3
null	false	O(1)	

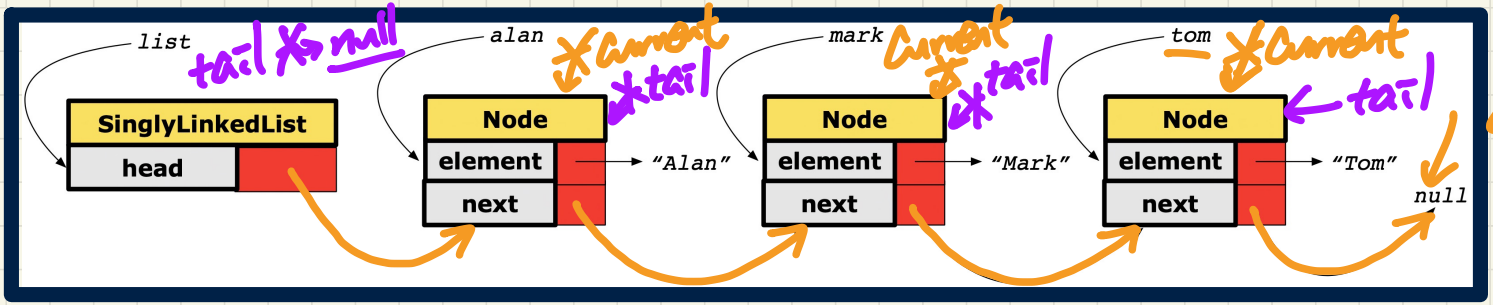
reaching null. (n iterations)

1. Empty list
2. Non-empty list

head → null

Empty SLL

SLL Operation: Finding the Tail of the List

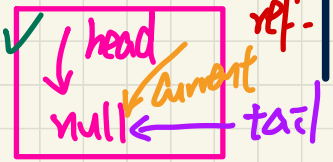



```
1  Node getTail() {
2      Node current = head;
3      Node tail = null;
4      while (current != null) {
5          tail = current;
6          current = current.getNext();
7      }
8      return tail;
9  }
```

O(n)

how many iterations before getting null

1. Empty list
2. Non-empty list



Trace: list.getTail()

current	current != null	End of Iteration	tail
alan	true	1	alan
mark	true	2	mark
tom	true	3	tom
null	fake		

SLL Operation: Inserting to the Front of the List

Exercise: Use debugger to trace.

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

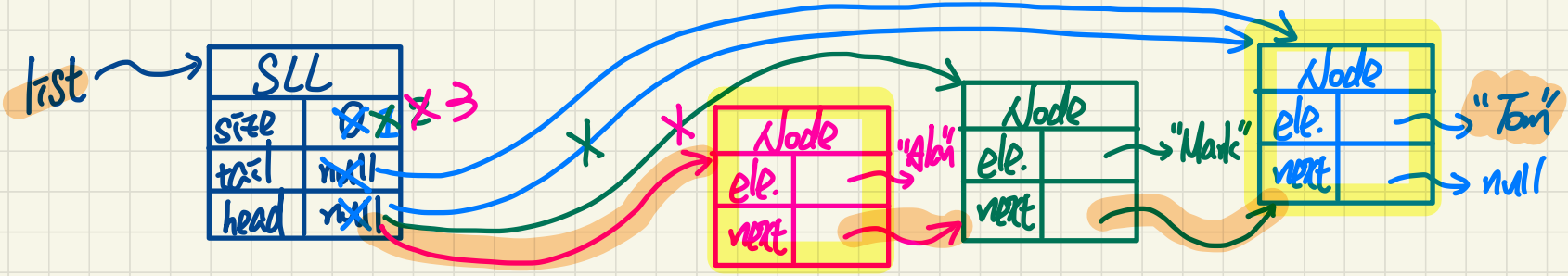
return values of att. size and head ⇒ O(1)

user/caller friendly
Alt: Node n

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

O(1)

overhead of declaring size as an attribute.



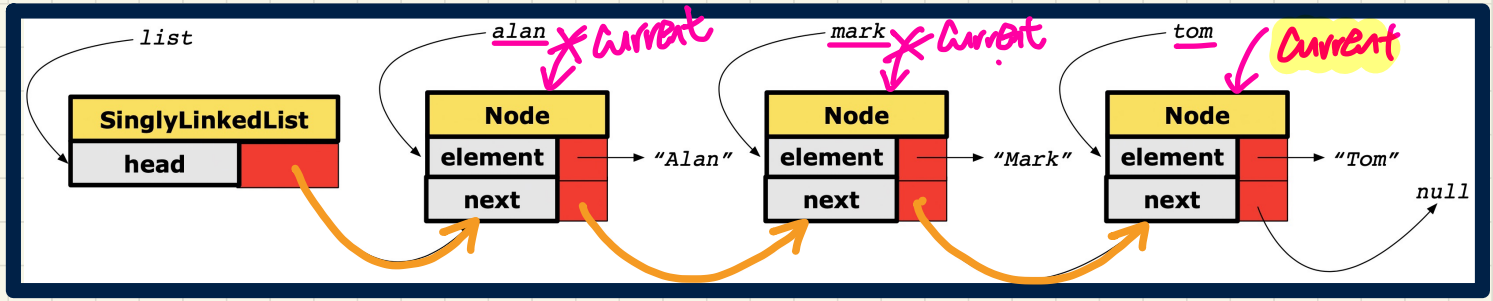
Lecture 2

Part E (continued)

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

Exercises on Non-Empty list: lst.getNodeAt(-1) vs. lst.getNodeAt(3)

SLL Operation: Accessing the Middle of the List




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

Trace: list.getNodeAt(2)

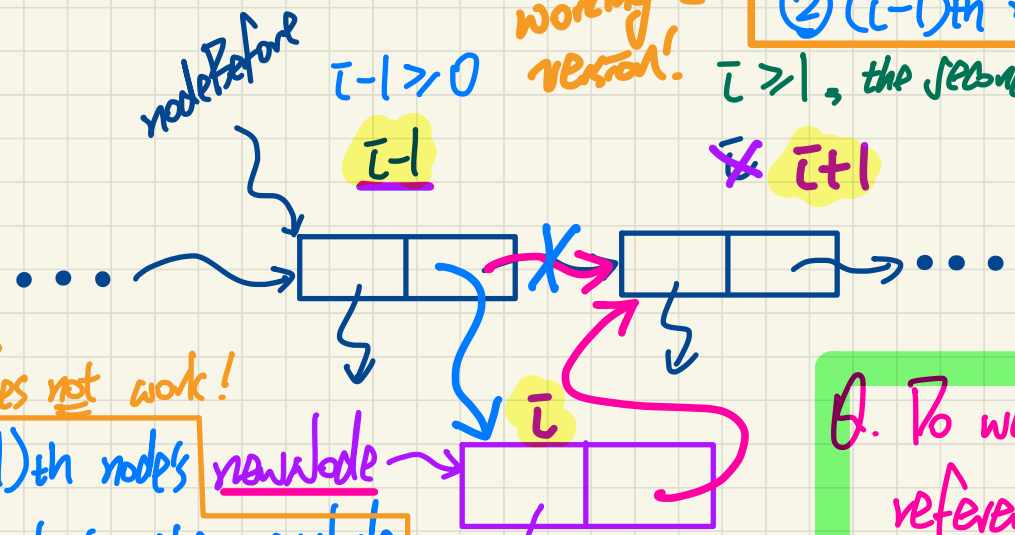
current	index	index < 2	Start of Iteration
alan	0	T	1
mark	1	T	2
tom	2	F	

1. Empty list
2. Non-Empty list

lst.getNodeAt(0);

Idea of Inserting a Node at index i

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



① The newNode's next is set to (i-1)th node's next.
② (i-1)th node's next set to newNode.

seq. of code does not work!

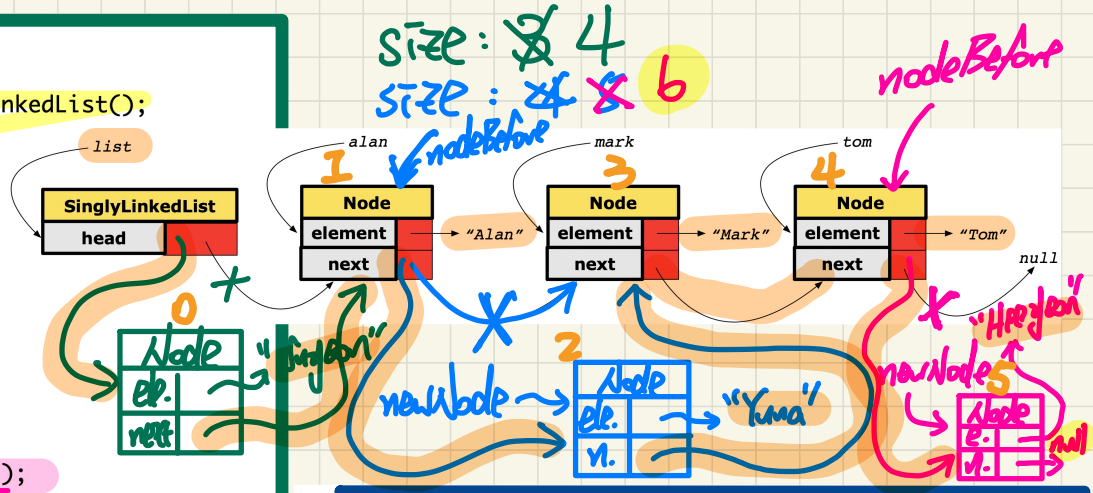
① The (i-1)th node's newNode next is set to the newNode
② The newNode's next is set to the old ith node.

Q. Do we need the reference to the (i-1)th node?

YES!

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

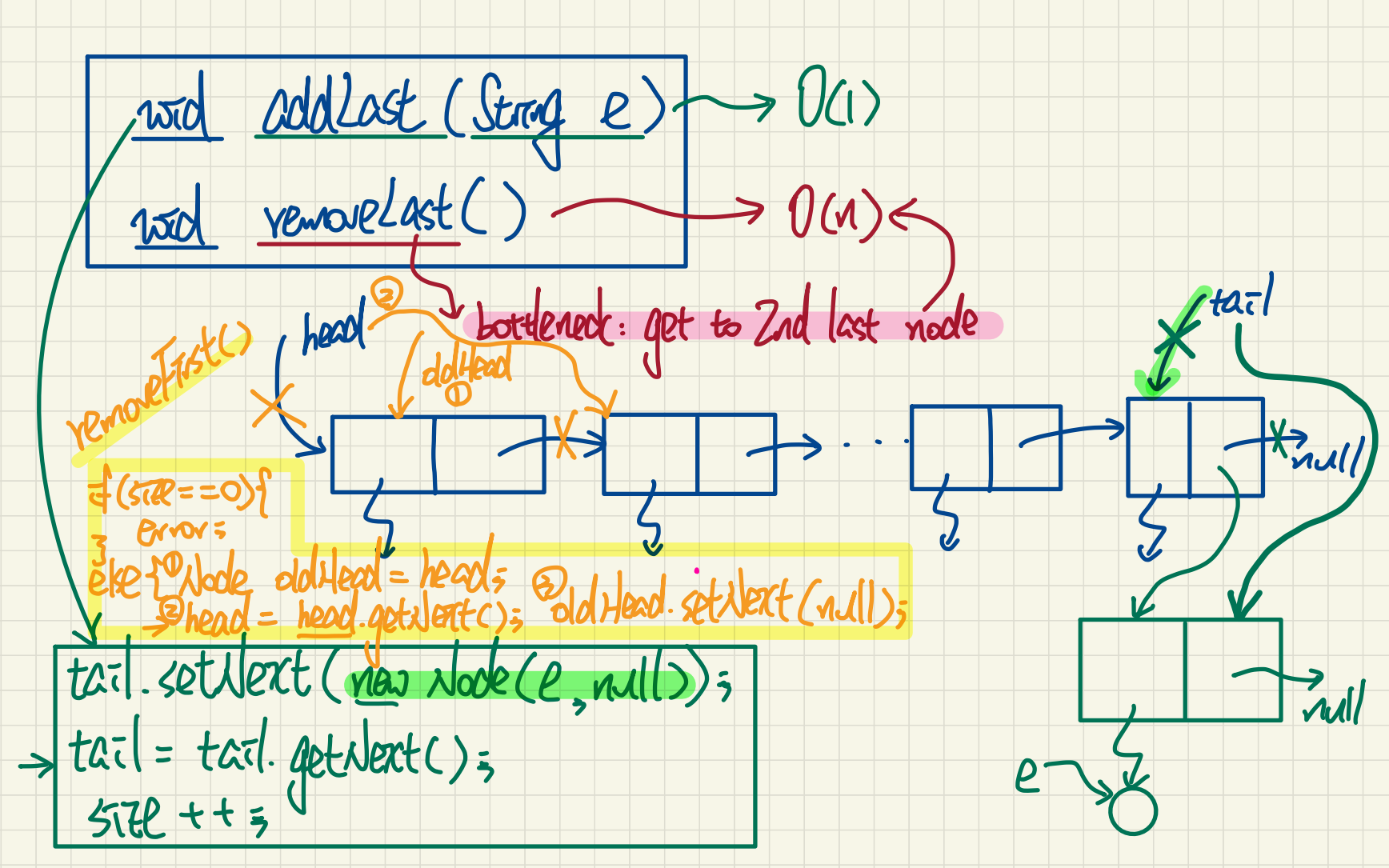



```java
1  void addAt (int i, String e) {
2      if (i < 0 || i > size) {
3          throw new IllegalArgumentException("Invalid Index.");
4      }
5      else {
6          if (i == 0) {
7              addFirst(e);
8          }
9          else {
10             Node nodeBefore = getNodeAt(i - 1);
11 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

O(n)?

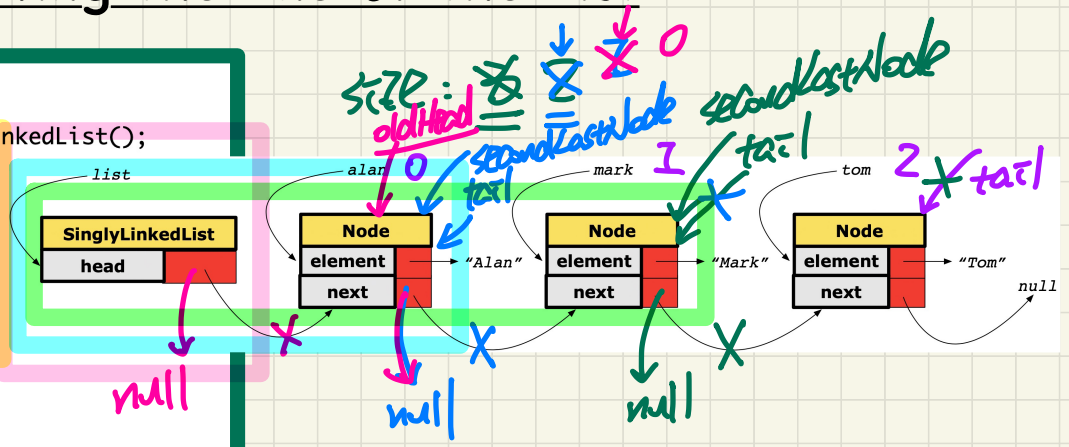
4 O(n)

O(n) - dominated by getNodeAt


void addLast(String e)
O(1)
void removeLast()
O(n)
bottleneck: get to 2nd last node
head
oldHead
removeFirst()
if(size==0){
error;
}else{
Node oldHead = head;
head = head.getNext();
oldHead.setNext(null);
tail.setNext(new Node(e,null));
tail = tail.getNext();
size++;
tail
null
null
e

SLL Operation: Removing the End of the List

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

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

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

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

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




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

O(n) - dominating term

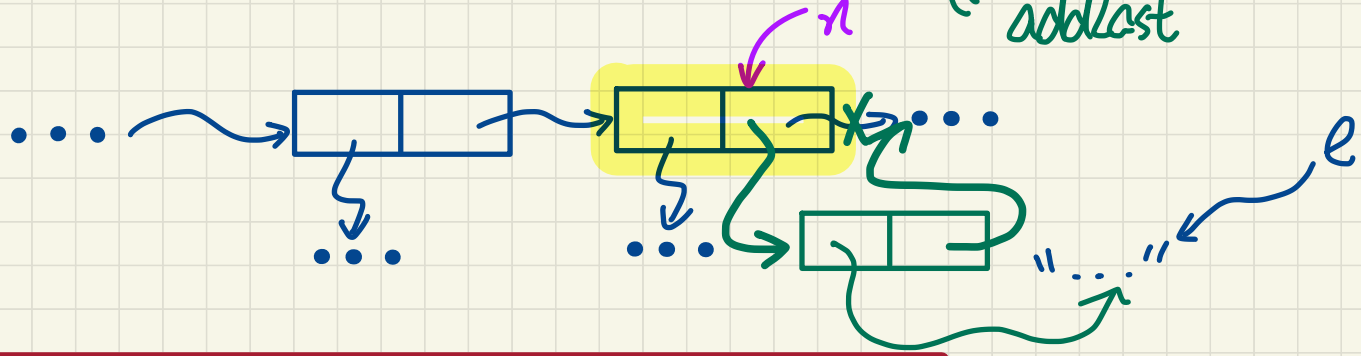
Exercises: insertAfter vs. insertBefore

reference node

Case: insertAfter(Node n, String e)

O(1)

≈ addLast

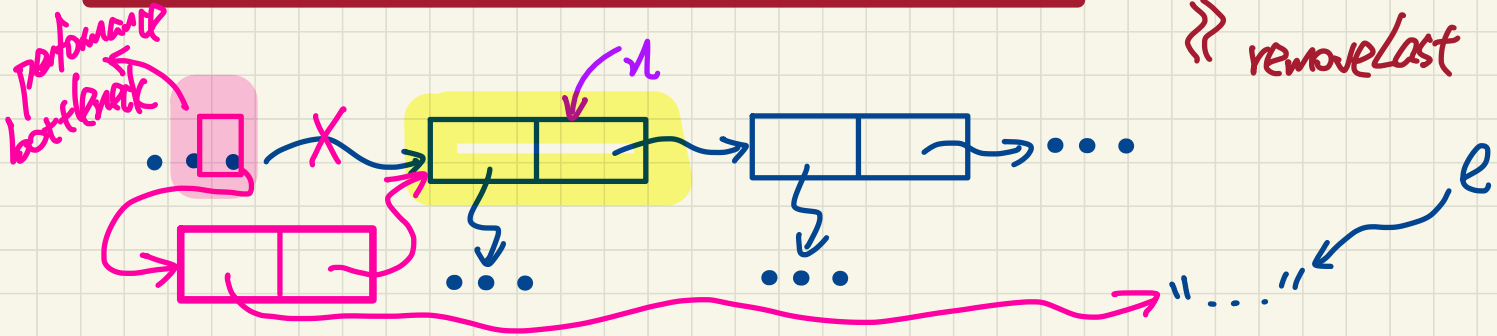


Case: insertBefore(Node n, String e)

O(n)

≫ removeLast

performance bottleneck



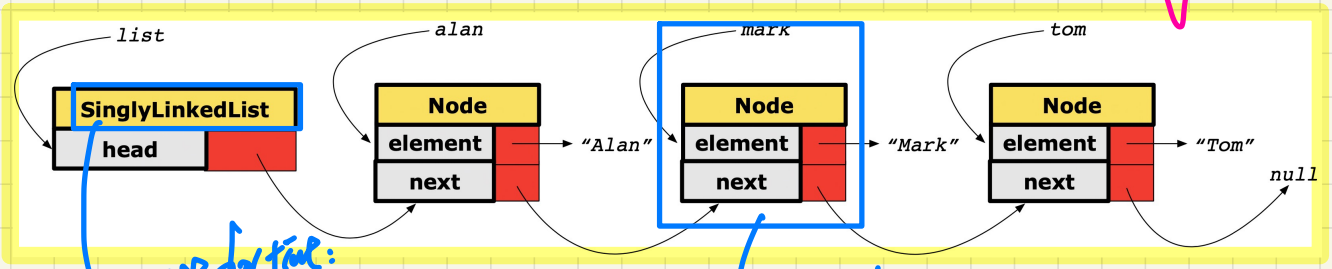
Lecture 2

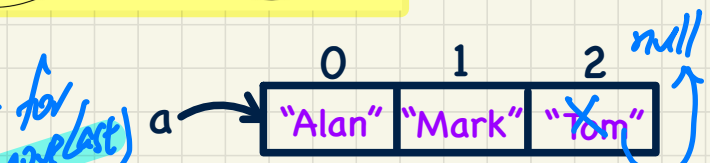
Part F

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

Running Time: Arrays vs. Singly-Linked Lists

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





Lecture 2

Part G

Singly-Linked Lists -
Implementing Generic Lists in Java

Non-Generic Classes: Node vs. SinglyLinkedList

as an implementor, you must commit to a type for node elements

```
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; }
}
```

Everything else is tied to this String type accordingly.

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

not compatible

```
Node n1 = new Node("Alan", null);
Node n2 = new Node("Mark", n1);
Node n3 = new Node(23, null);
SLL list = new SLL();
list.setHead(n2);
list.addAt(0, "Tom");
list.addAt(1, 23);
Node n4 = list.getNodeAt(1);
String e = n4.getElement();
```

X

inconsistent with the committed String type.

X

Generic Classes: Node and SinglyLinkedList

As an implementer, we leave the choice of element type to whoever uses these classes for declarations.

declaration of a generic param.

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

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

list1
list2

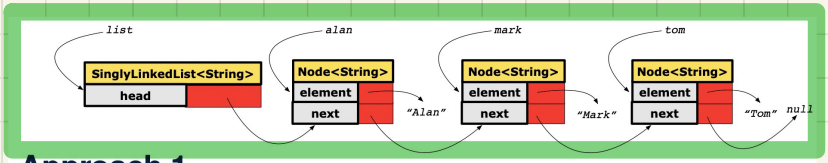
```
Node<String> n1 = new Node<>("Alan", null);
Node<String> n2 = new Node<>("Mark", n1);
Node<Integer> n3 = new Node<>(23, null);
Node<Integer> n4 = new Node<>(46, n3);
Node<Integer> n5 = new Node<>("Tom", null);
Node<Integer> n6 = new Node<>(46, n2);

SLL<String> list1 = new SLL<>();
list1.setHead(n2);
list1.addAt(0, "Tom");
Node<String> n7 = list1.getNodeAt(1);
String e1 = n7.getElement();

SLL<Integer> list2 = new SLL<>();
list2.setHead(n4);
list2.addAt(0, 68);
Node<Integer> n8 = list2.getNodeAt(1);
Integer e2 = n8.getElement();
```

expecting Node<Integer> but getting Node<String>

Node<E>: generic type parameter

list2.setHead(n1) ✗ expecting <Node<I>> but getting <Node<S>>

List Constructions

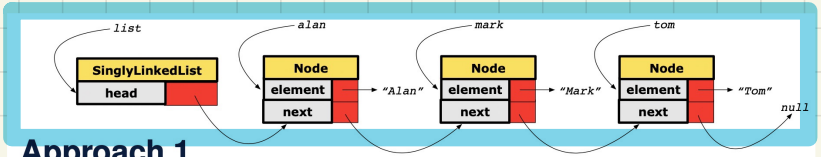


Approach 1

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

Approach 2

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



Approach 1

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

Approach 2

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

Generic List

```
Node<String> alan = new Node<String>(...);
                    new Node<>(...);
```

Non-Generic List

List Methods

class SLL <E> {

SLL <String>
SLL <Person>

Generic List

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

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

Non-Generic List

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

```
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;
   }
}
```

3

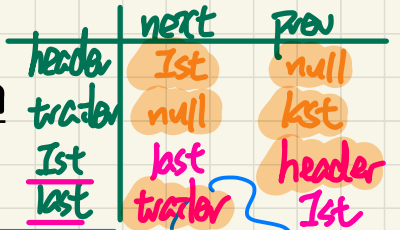
Lecture 2

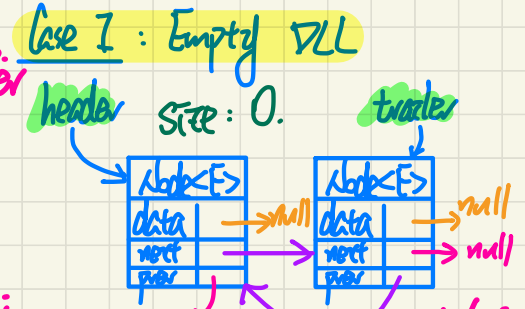
Part H

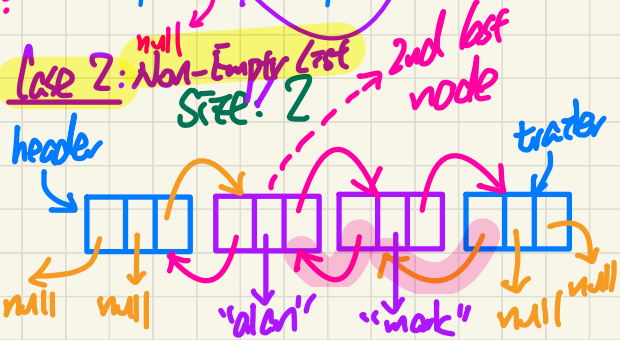
Doubly-Linked Lists - Intuitive Introduction

Doubly-Linked Lists (DLL): Visual Introduction

- A chain of bi-directionally connected nodes
- Each node contains:
  - + reference to a data object
  - + reference to the next node
  - + reference to the previous node
- A DLL is also a SLL:
  - + many methods implemented the same way
  - + some method implemented more efficiently
- Accessing a node in a list:
  - + Relative positioning: O(n)
  - + Absolute indexing: O(1)
- The chain may grow or shrink dynamically.
- Dedicated Header vs. Trailer Nodes
  (no head refeference and no tail reference)







Lecture 2

Part I

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

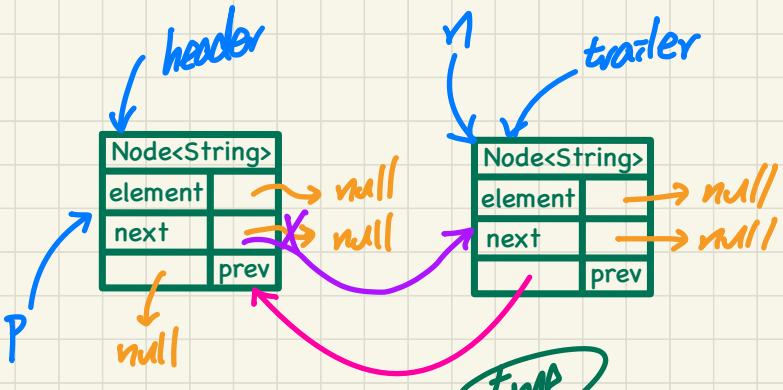
Generic DLL in Java: DoublyLinkedList vs. Node

```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
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;
   }
}
```

call by value

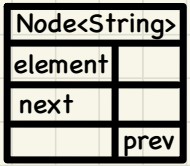
```java
@Test
public void test_String_DLL_Empty_List() {
    DoublyLinkedList<String> list = new DoublyLinkedList<>();
    assertTrue(list.getSize() == 0);
    assertTrue(list.getFirst() == null);
    assertTrue(list.getLast() == null);
}
```



header.getNext() == trailer

trailer.getPrev() == header

true



Lecture 2

Part J

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

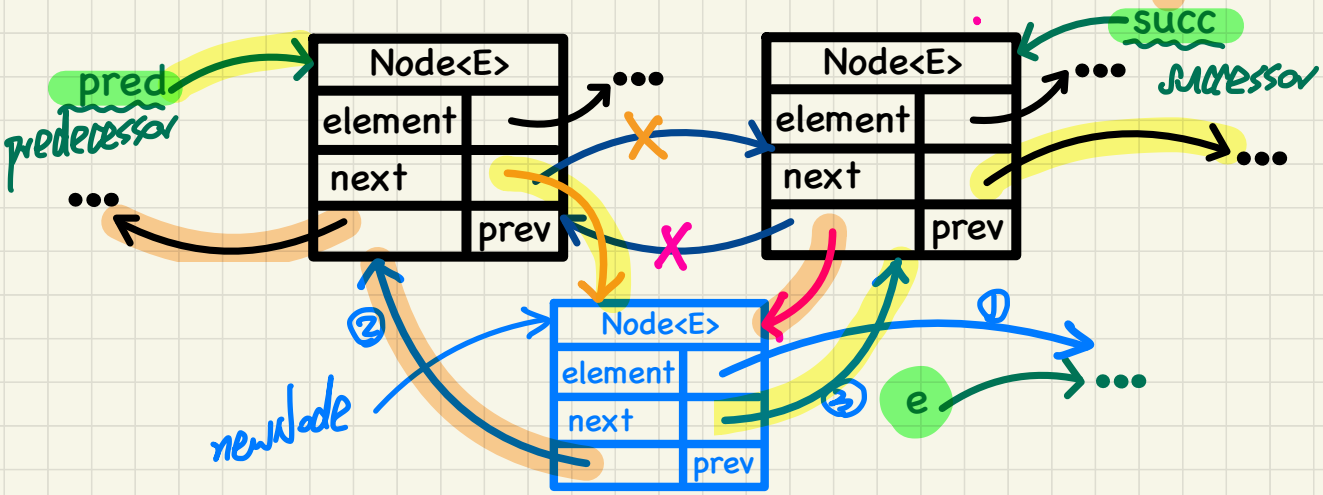
Generic DLL in Java: Inserting between Nodes

Node<E>	
element	
next	
	prev

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

RT: O(1)

Assumption: pred and succ are directly connected.



Generic DLL in Java: Inserting to the Front/End

```
@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())
}
```

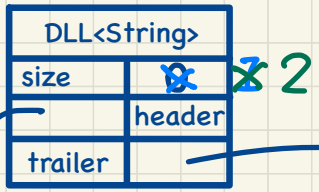
Exercise: Tracing

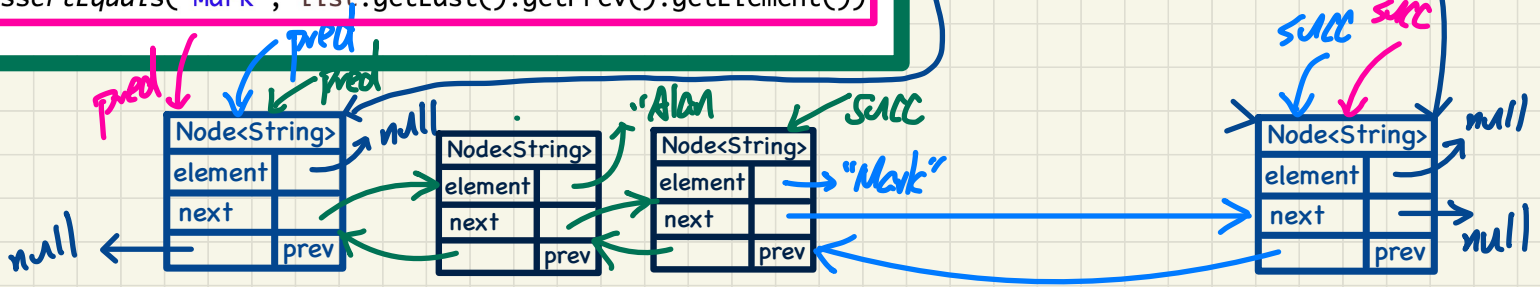
```
void addFirst(E e) {
    addBetween(e, header, header.getNext());
}
```

pred succ

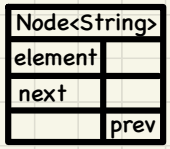
```
void addLast(E e) {
    addBetween(e, trailer.getPrev(), trailer);
}
```

pred succ





Generic DLL in Java: Inserting to the Middle




```java
@Test
public void test_String_DLL_addAt() {
    DoublyLinkedList<String> list = new DoublyLinkedList<>();
    list.addAt(0, "Alan");
    list.addAt(1, "Tom");
    list.addAt(1, "Mark");

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

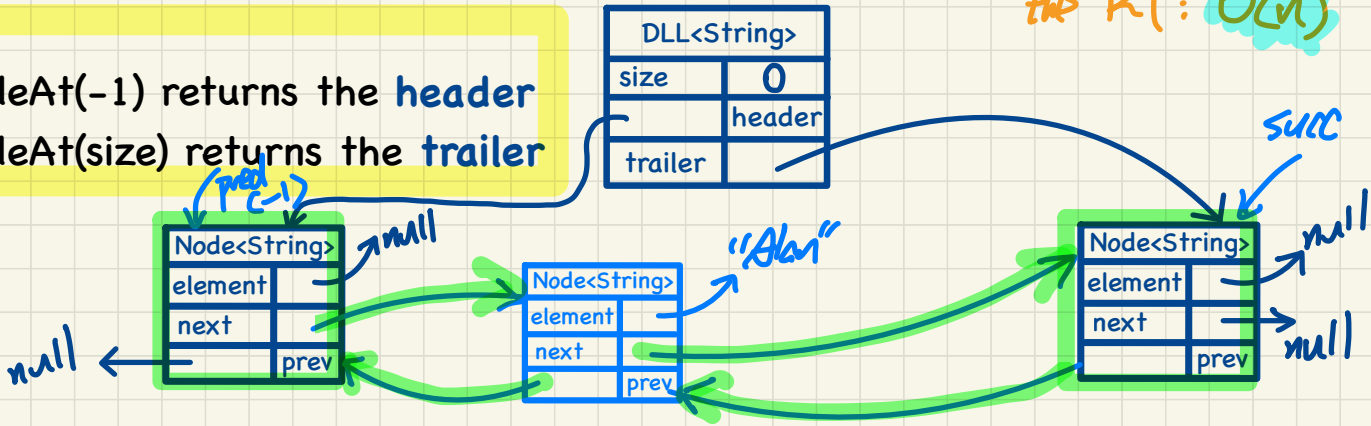
Exercise: Tracing

```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);
  }
}
```

still dominates the RT: O(n)

Notes.

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



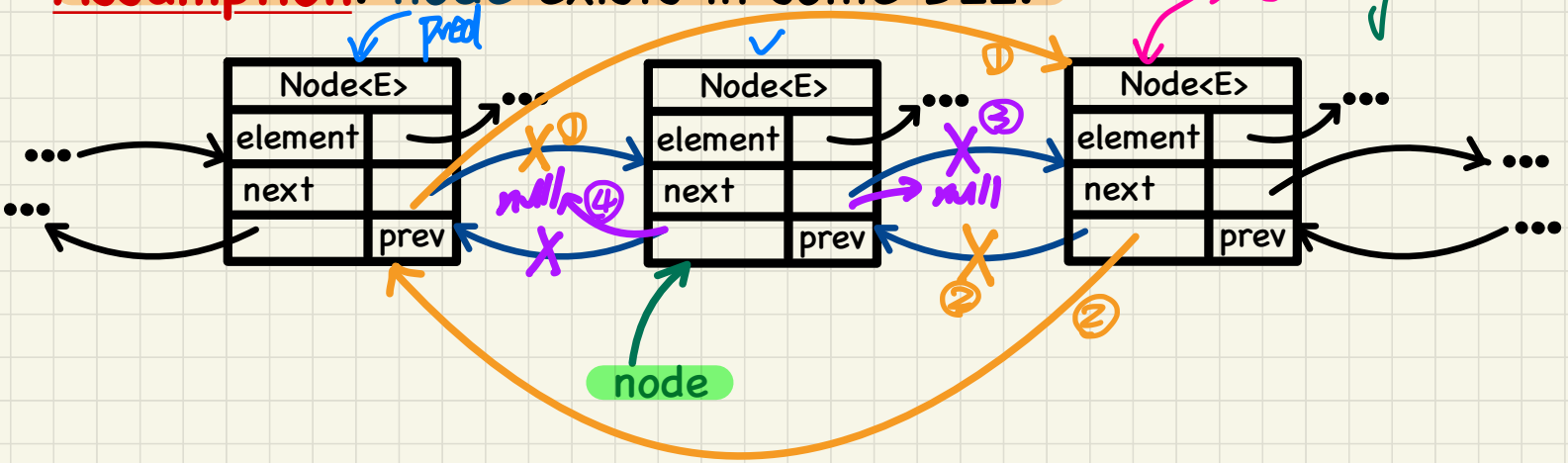
Generic DLL in Java: Removing a Node

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

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()); }
}
```

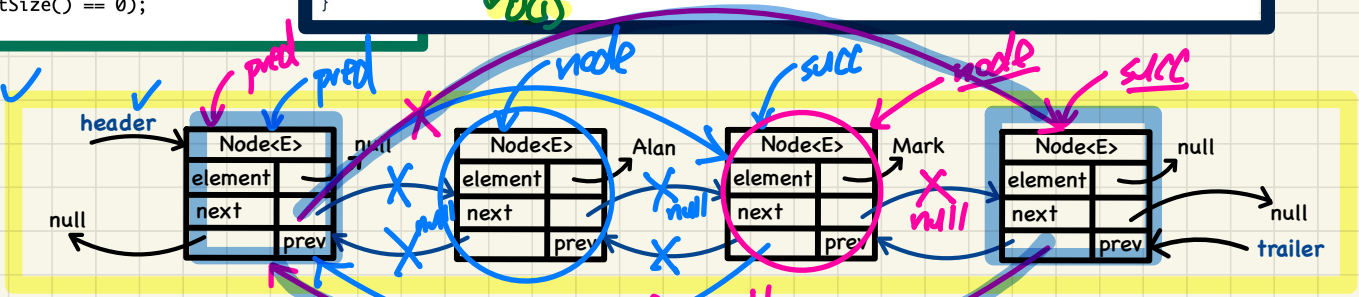
O(1)

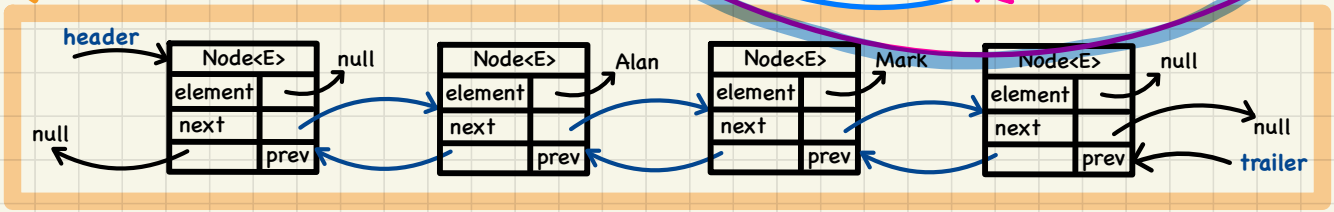
first node

```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

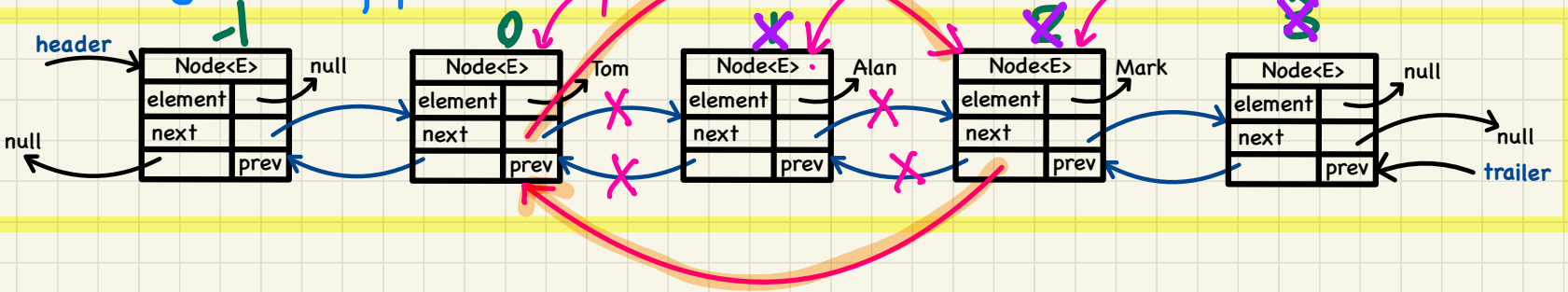
```java
@Test
public void test_String_DLL_removeAt() {
    DoublyLinkedList<String> list = new DoublyLinkedList<>();
    list.addFirst("Mark");
    list.addFirst("Alan");
    list.addFirst("Tom");
    assertTrue(list.getSize() == 3);
    list.removeAt(1);
    assertTrue(list.getSize() == 2);
    list.removeAt(0);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

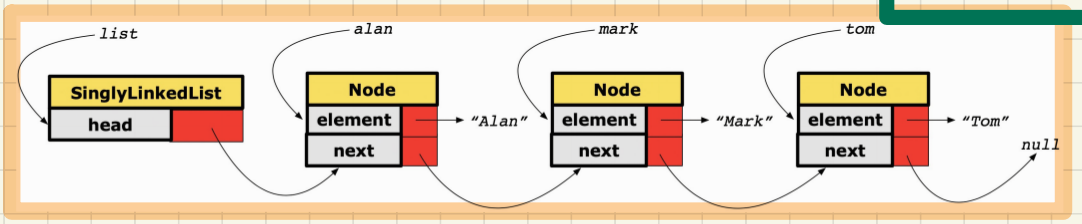
Part K

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

Running Time: Arrays vs. SLL vs. DLL

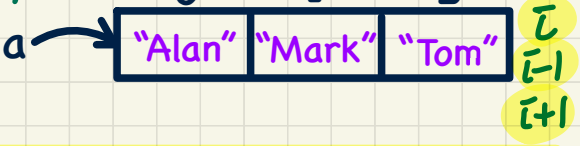
see discussion at the end of SLL.

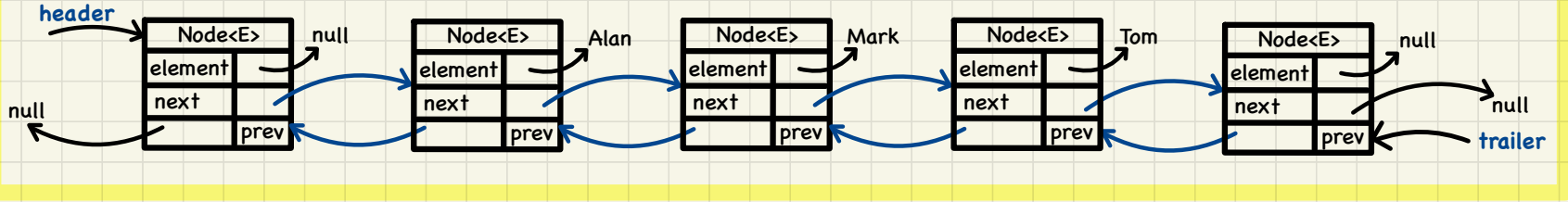
Operation \ Data Structure	Array	Singly-Linked List	Doubly-Linked List
size	O(1)		
first/last element	O(1)		
element at index i	O(n)	O(n)	O(n)
remove last element	O(n)	O(1) remove at index i:	O(1)
add/remove first element, add last element			
add/remove i^{th} element	given reference to $(i-1)^{th}$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



pass (i-1) node.

remove at index i:





Lecture 3

Part A

Modularity, Abstract Data Types (ADTs) - Definition & Terminology

Supplier vs. Client in OOP

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

supplier class (callee)

client class (caller)

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

client fulfilling obligation

contractual relation in effect

given client fulfilling their obligations, supplier must fulfil their obligation.

supplier method/service used in the context of the client class.

pre-stat (pre-execution of supplier method)

m.heat(obj);

client's obligation must be fulfilled

post-stat (post-execution of supplier method)

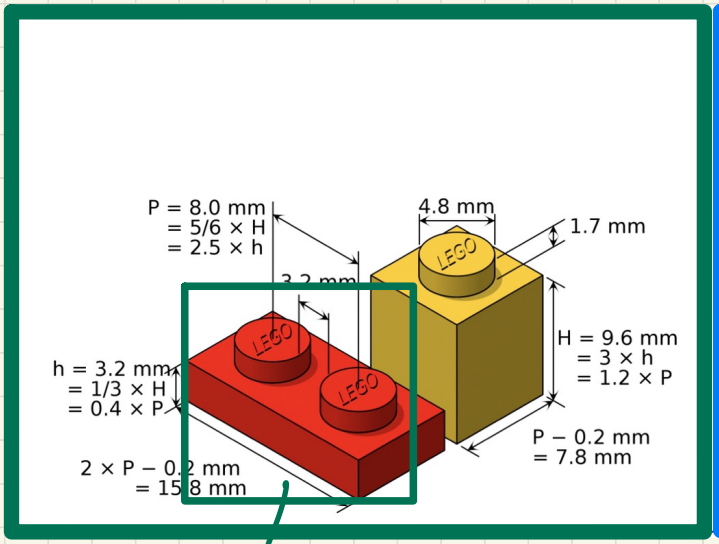
supplier's obligation must be fulfilled.

postcondition (conditions for supplier to satisfy).

precondition (conditions for client to satisfy in order to use the supplier's method).

in Java: exceptions.

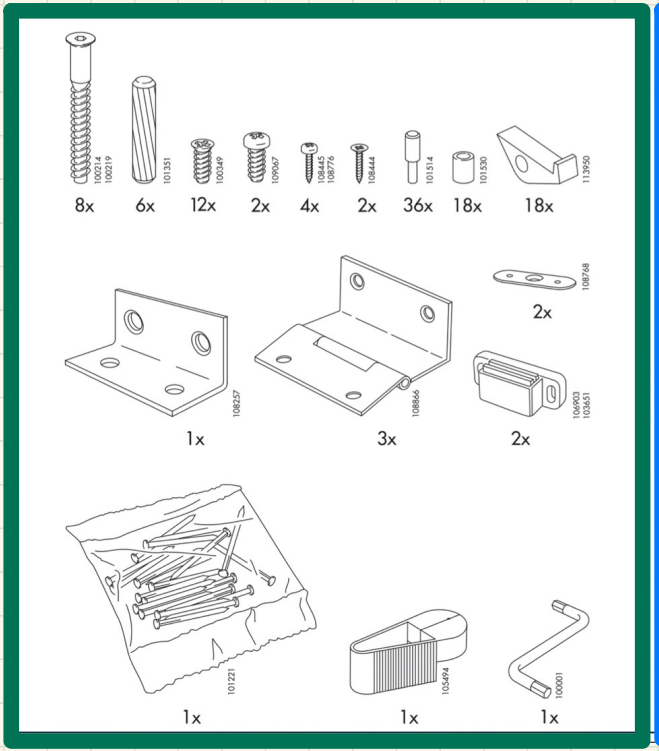
Modularity: Childhood Activities

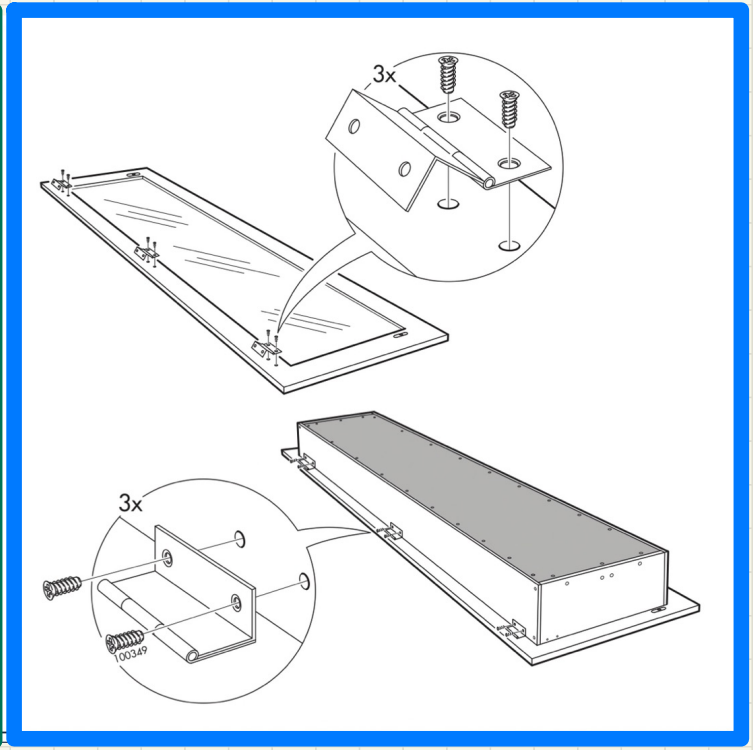


interface specification (of a module)

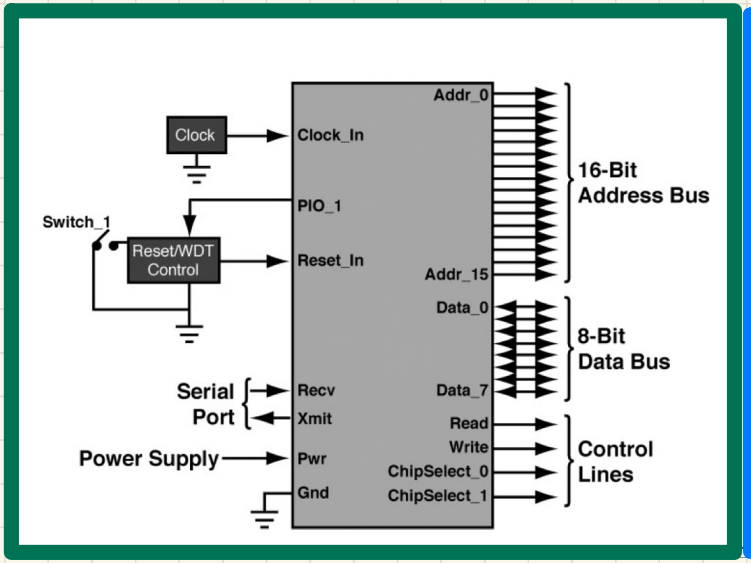
architecture (assembly) of building blocks

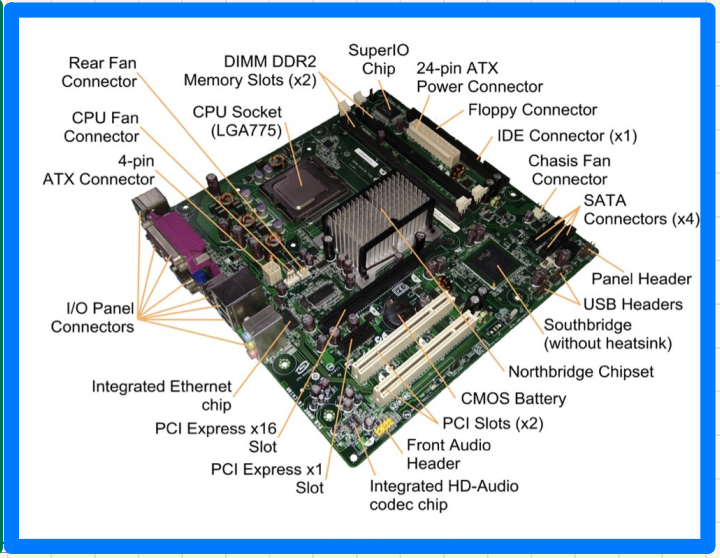
Modularity: Daily Constructions



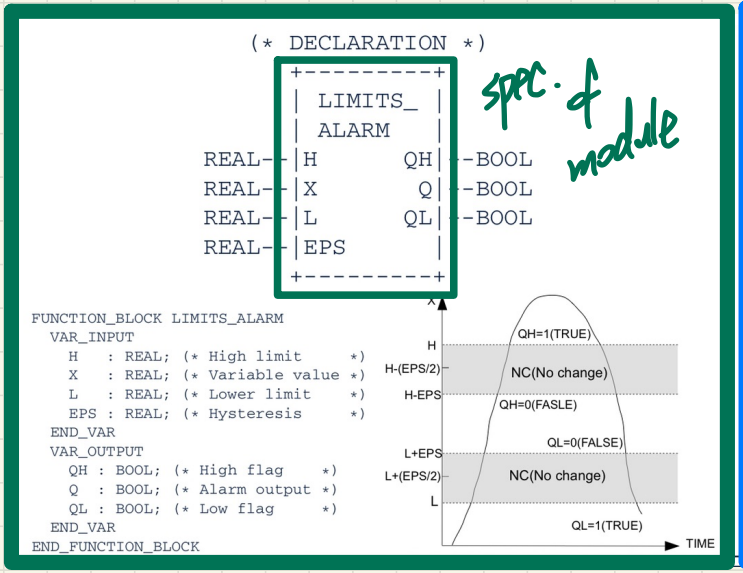


Modularity: Computer Architectures

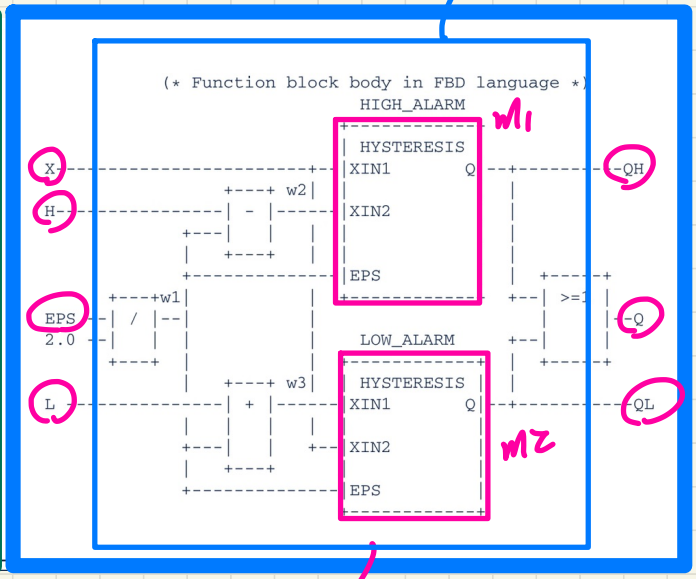




Modularity: System Developments



spec. of module



a bigger module

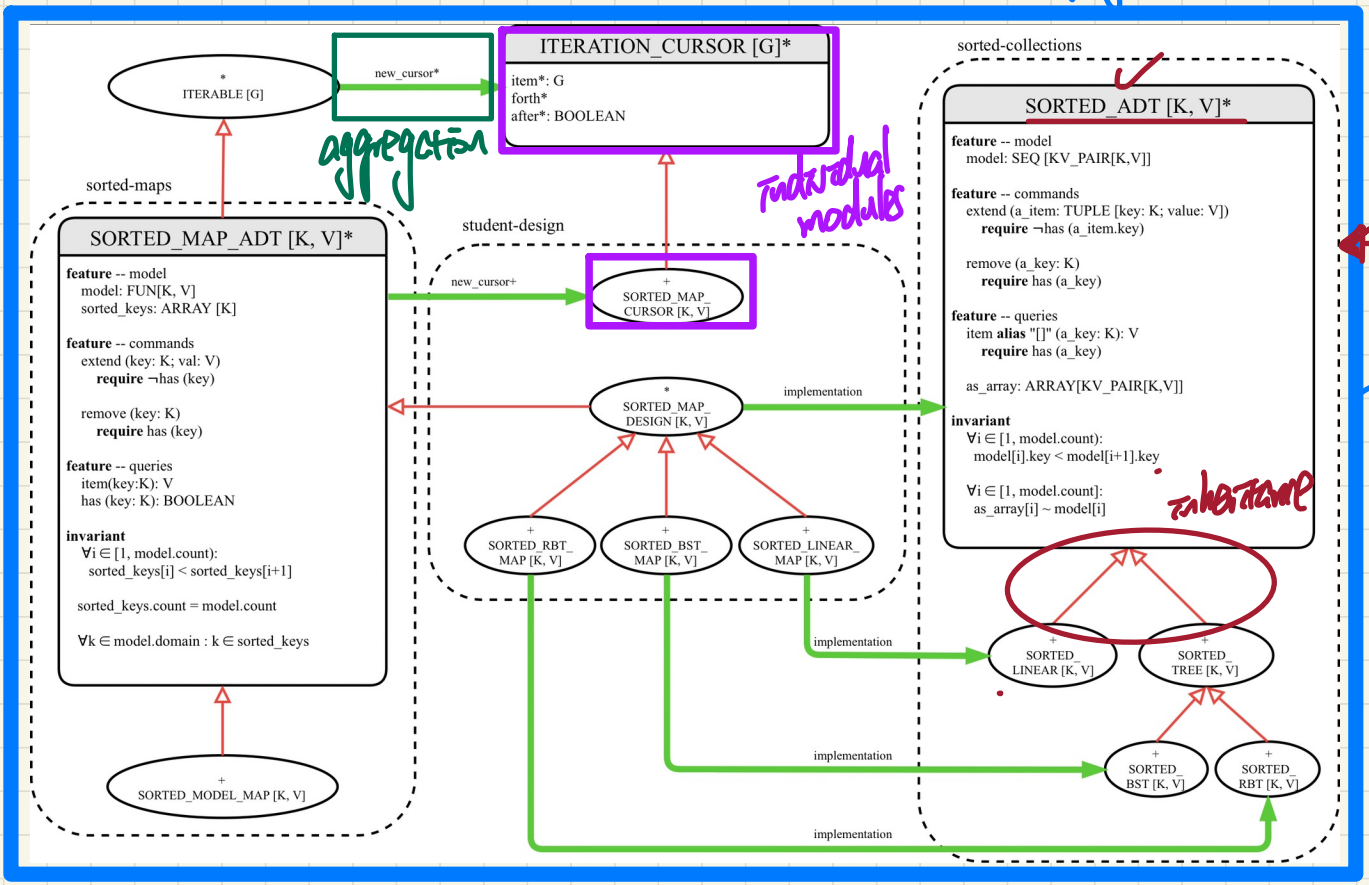
M1

M2

assembly as a Composition of well-specified modules

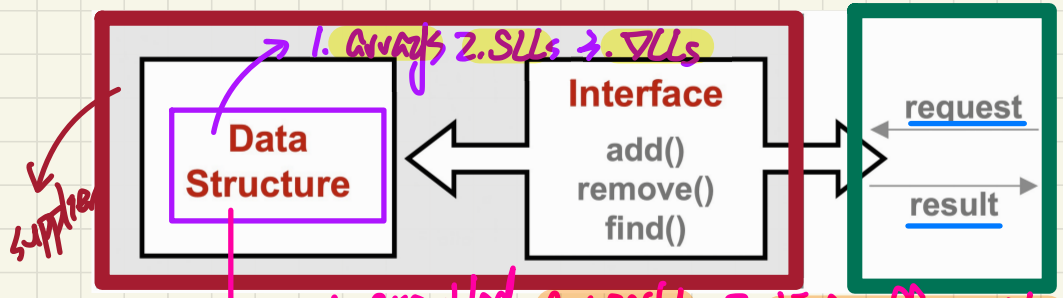
Modularity: Software Design

In OOP, assemble classes via:
1. aggregations
2. compositions
3. inheritance



architectural diagram.

Abstract Data Types (ADTs)



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

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

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

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

Java API ≈ Abstract Data Types

NT is subject to ambiguities & contradictions.

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

set

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

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

Parameters:

index - index of the element to replace

element - element to be stored at the specified position

Returns:

the element previously at the specified position

Throws:

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

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

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

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

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

Interface List<E>

Type Parameters:

E - the type of elements in this list

All Superinterfaces:

Collection<E>, Iterable<E>

All Known Implementing Classes:

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

---

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

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

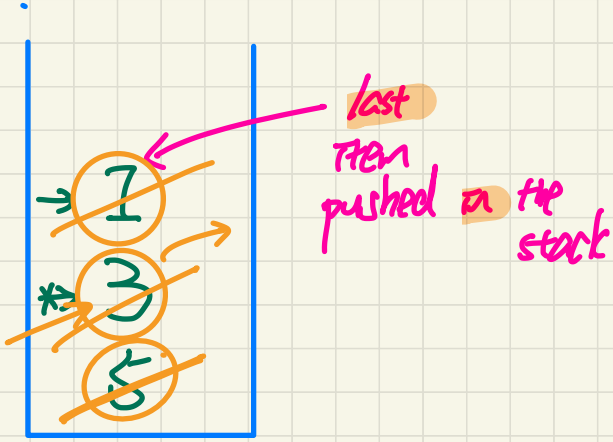
Lecture 3

Part B

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

Stack ADT: Illustration

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



last pushed element

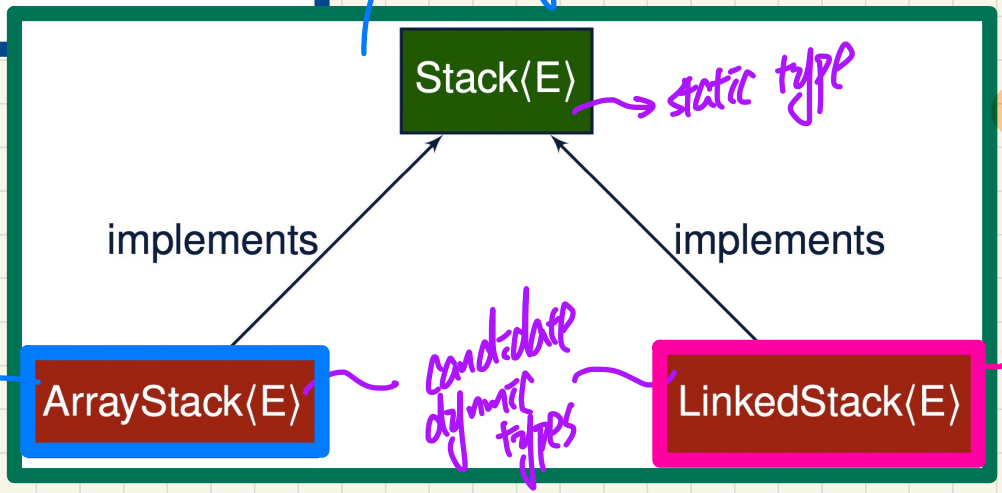
2nd last pushed element

(LIFO)

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

Implementing the Stack ADT in Java: Architecture

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



Implementing the Stack ADT using an Array

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

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

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

limitation: fixed size

ArrayStack<String>

↳ instantiates E for Stack:

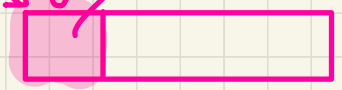
Stack<String>

(E[]) Object[____]

↳ what you have to write in Java.

efficient top. of stack

goal: treat this first item as the top.



Implementing the Stack ADT using a SLL

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

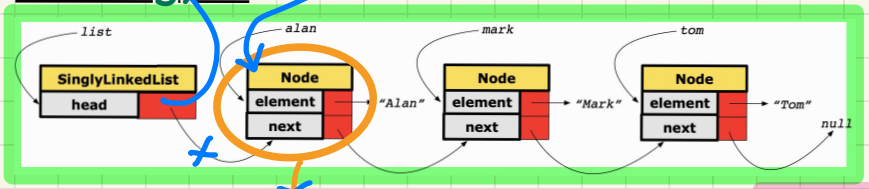
Stack Method	Singly-Linked List Method	
	Strategy 1	Strategy 2
size	list.size	
isEmpty	list.isEmpty	
top	list.first	list.last
push	list.addFirst	list.addLast
pop	list.removeFirst	list.removeLast

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

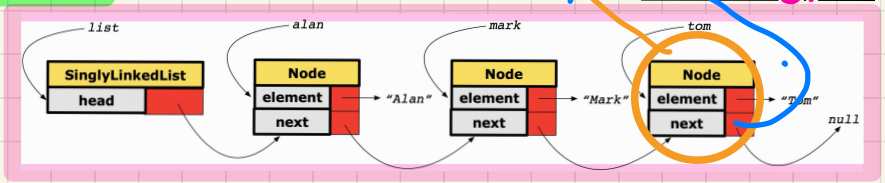
O(1)

O(1)

Strategy 1



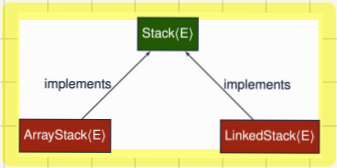
Strategy 2



Stack ADT: Testing Alternative Implementations

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

✗ interface can't be a DT.




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

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

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

static type

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

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

dynamic type

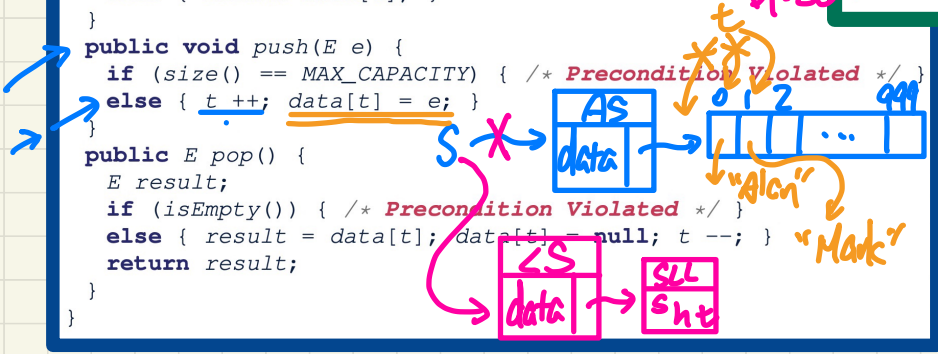
DT: AS

DT: LS

version in AS.

DT changes

version in LS. class of Stack?



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

Lecture 3

Part C

Stack ADT –
Algorithms using the Stack ADT

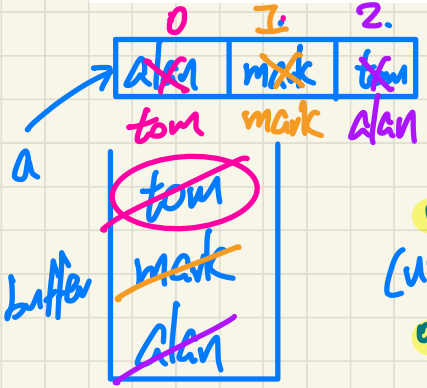
Algorithm using Stack: Reversing an Array

```
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();
  }
}
```

generic parameter at the method level

reverse({"alan", "mark"})
String[]

reverse({23, 46})



handling an array of strings (using a stack of strings)

```
@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);
}
```

handling an array of ints (using a stack of ints)

Algorithm using Stack: Matching Delimiters

```
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();
}
```

opening: ([{
closing:)] }
0 1 2

exit: foundError || i >= exp.length

c is opening

c is closing

more closing than opening

2 == 0

closing not matching opening

!false true

more opening than closing


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

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

2 + 3 = 5

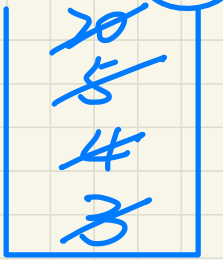
5 * 5 = 25

? + 25

(insufficient operands)

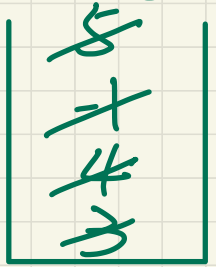
4 * 5 = 20

3 - 20 = -17

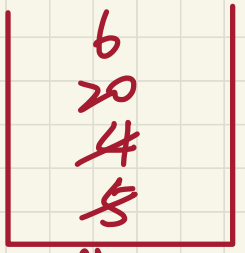


3 - 4 = -1

-1 * 5 = -5



5 + 4 = 20



(insufficient operator)

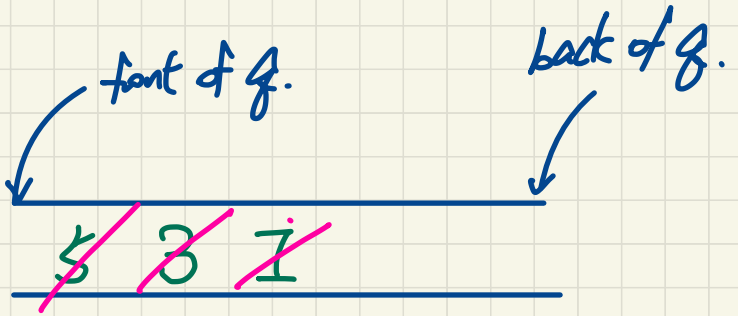
Lecture 3

Part D

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

Queue ADT: Illustration

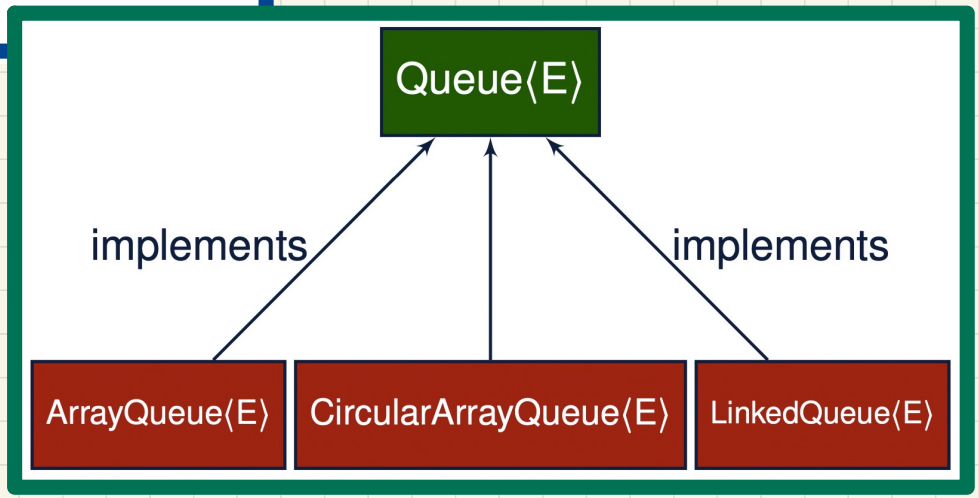
	isEmpty	size	first
new queue	T	0	n.a.
enqueue(5)	F	1	5
enqueue(3)	F	2	5
enqueue(1)	F	3	5
dequeue rtn. 5	F	2	3
dequeue rtn. 3	F	1	1
dequeue rtn. 1	T	0	n.a.



→ First-In First-Out (FIFO)

Implementing the Queue ADT in Java: Architecture

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



Implementing the Queue ADT using an Array

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

O(1)

O(1)

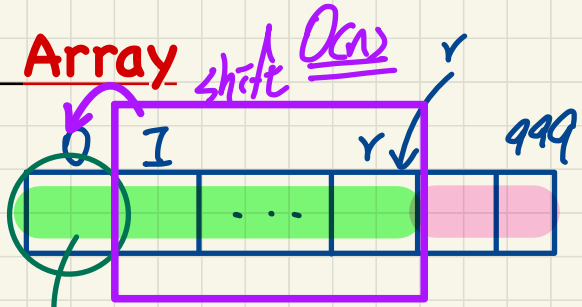
O(1)

front index

O(1)

O(n)

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



limitation: no resizing.

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

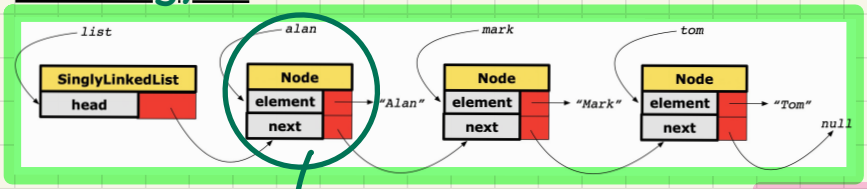
Implementing the Queue ADT using a SLL

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

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

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

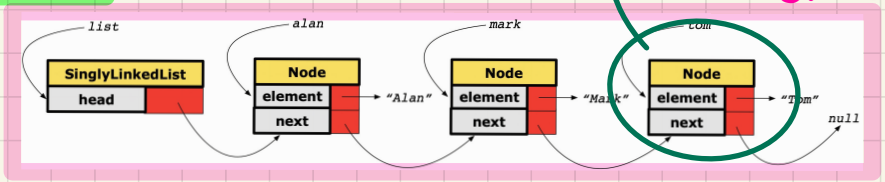
Strategy 1



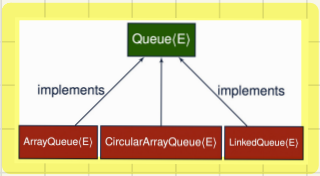
first of queue

first of queue.

Strategy 2



Stack ADT: Testing Alternative Implementations




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

dynamic binding.

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

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

Polymorphism

Lecture 3

Part D

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

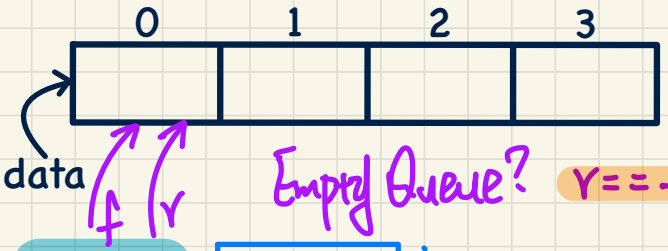
size of Queue vs. size of array | % modulo

Implementing the Queue ADT using a Circular Array

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

Assume: A circular array of length 4.

Phase 0: Empty Queue q



f=0;
r=0;

Empty Queue? r==f

Phase 1: enqueue 3 elements

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

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

(r+1)%4 = 3

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

empty slots before f

slots before this index r

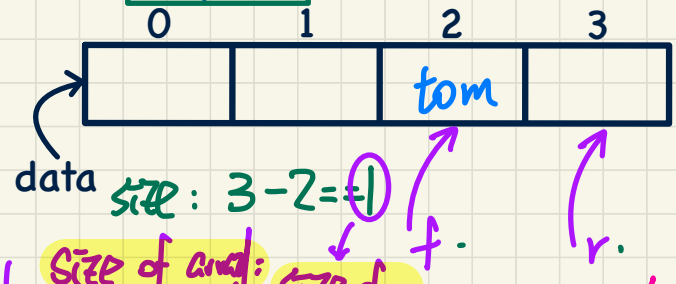
size: r>=f → r-f

3 = 1+(4-2)

size: r<f → r+(N-f)

Phase 2: dequeue 2 times

data[f]=null; f++;



size: 3-2=1

size of array: 4 (fixed) → size of 4.

(3+1)%4 = 0

Phase 3: enqueue 2 elements

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

[0, r-1] → (r-1)-0+1 = r

[f, N-1] → (N-1)-f+1 = N-f

Lecture 3

Part E

Implementing Stack and Queue - Dynamic Arrays: Const. Increments vs. Doubling

total RT / # ops.

Amortized Analysis: Dynamic Array with Const. Increments

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

worst-case: $O(n)$

this only occurs once in a while

Resizing loop: $O(n)$; `capacity = capacity + C`: $O(1)$

Sum of Arith. Seq. $a_1 + a_2 + \dots + a_k = \frac{(a_1 + a_k) \cdot k}{2}$

$* \; I + (k-1) \cdot C = n$

$k = \frac{n - I}{C} + 1$ (# of resizing steps)

initial array: I

1st resizing: I C — I

2nd resizing: I C C — $I + C$

3rd resizing: I C C C — $I + 2 \cdot C$

⋮

Last resizing: I C C C ... C C — $* \; I + (k-1) \cdot C$ (n)

$I+0 \cdot C$, $I+1 \cdot C$

$\underbrace{I}_{1st} + \underbrace{(I+C)}_{2nd} + \underbrace{(I+2C)}_{3rd} + \dots + \underbrace{n}_{kth}$

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

$= \frac{n^2 + C \cdot n + C \cdot I - I^2}{2C}$

$O(n^2)$ total RT

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

W.L.O.G, assume: n pushes and the last push triggers the last resizing routine.

Deriving the Sum of a Geometric Sequence

Initial Term: I

Common Factor: r

Number of Terms: k

$$S_k = \underbrace{I}_{1st} + \underbrace{I \cdot r}_{2nd} + \underbrace{I \cdot r^2}_{3rd} + \underbrace{I \cdot r^3}_{4th} + \cdots + \underbrace{I \cdot r^{k-1}}_{k\text{-th}}$$

$I \cdot r^0$

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

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

Avg: total RT / # of.

Amortized Analysis: Dynamic Array with Doubling

P1 $2^x = y \Rightarrow x = \log y$ $\quad 2^3 = 8 \Rightarrow 3 = \log 8$

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

resizing: O(n)

O(1)

O(n)

Sum of Geo. Seq: $S_k = \frac{I \cdot (r^{\#\text{terms}} - 1)}{r - 1}$

P3 $2^{\log x} = x$ $\quad 2^{\log 8} = 2^3 = 8$

initial array: [I]

1st resizing: [I I] I

2nd resizing: [I I I I] 2·I

⋮

Last resizing: [I I ⋯ I I | I I ⋯ I I] n n n

P2 $x^{y+z} = x^y \cdot x^z$

$$\text{Total RT} = \underbrace{I}_{1\text{st}} + \underbrace{2 \cdot I}_{2\text{nd}} + \underbrace{2^2 \cdot I}_{3\text{rd}} + \cdots + \underbrace{2^{k-1} \cdot I}_{k\text{th}} = n$$

$$= \frac{I \cdot (2^{\log \frac{n}{I} + 1} - 1)}{2 - 1} \quad \left(\cdot \frac{n}{I}\right)$$

$$= I \cdot \left(\frac{n}{I} \cdot 2 - 1\right) = 2n - I$$

$O(n)$

$2^{k-1} \cdot I = n$

$2^{k-1} = \frac{n}{I}$

$k - 1 = \log \frac{n}{I}$

$k = \log \frac{n}{I} + 1$

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

and the last push triggers the last resizing routine.

Amortized / Average RT: $O(\frac{n}{n}) = O(1)$

push/enqueue

Exercise ArrayList
↳ implemented by ?

	Const. increments	doubling
Worst-case RT	O(n)	O(n)
Amortized/avg. RT	O(n)	O(1)

O(1) due to that doubling the size each time makes it substantially less frequent to resize

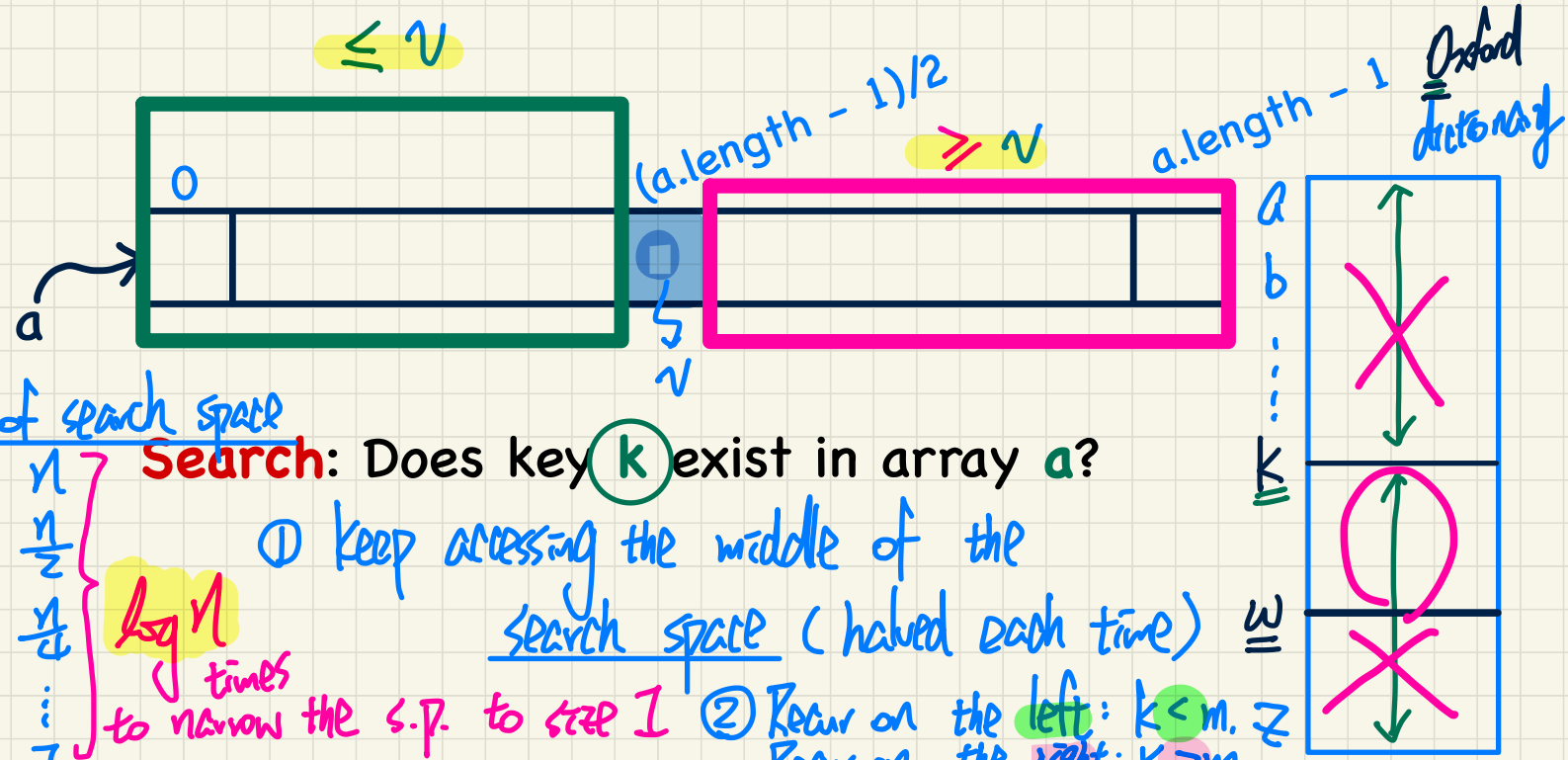
Lecture 4

Part B

Examples on Recursion
Binary Search

Binary Search: Ideas

Precondition: Array sorted in non-descending order



Binary Search in Java

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

input array sorted

call by value

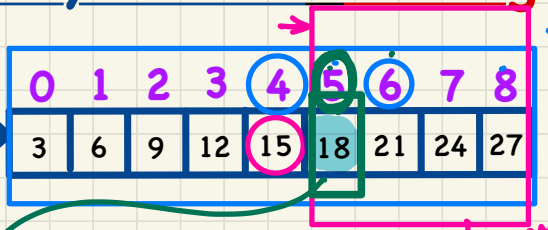
define the range of indices of the search space.

recursive case

narrowed search spaces represent a smaller problem to solve.

key == middleValue

Binary Search: Tracing



initial search space

search space of 2nd recursion

search(a,18)

key

search space of 1st recursion

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

a[m] == 15

m.v. 7

binarySearchH(a,0,8,18)

m.+1 to.

binarySearchH(a,5,8,18)

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

a[6] = 21

m.v.

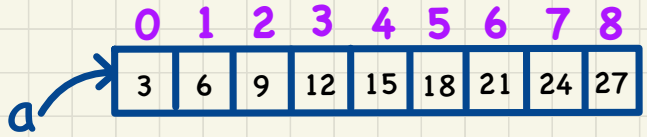
binarySearchH(a,5,5,18)

↳ true

from

m.-1

Exercise



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

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

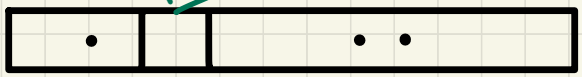
O(1) O(1) O(1) subproblem of size n-1

Base Case:
Empty Array

$T(0) = 1$

[4, 3] has 0 numbers

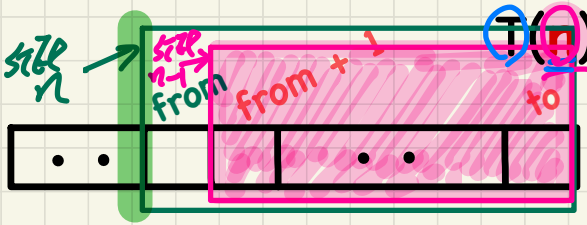
Base Case:
Array of Size 1



$T(1) = 1$

[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.

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

$= T((n-1)-1) + 1 + 1$ ($T(n-1)$; $n-2$)

$= T((n-2)-1) + 1 + 1 + 1$ ($T(n-2)$; $n-3$)

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

$= T(1) + 1 + 1 + \cdots + 1$ (How many? $(n-1)$)

$= 1$

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

$= n$

$O(n)$

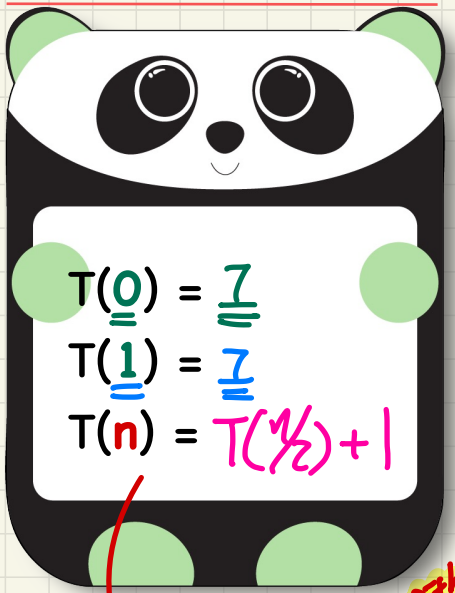
Binary Search: Running Time

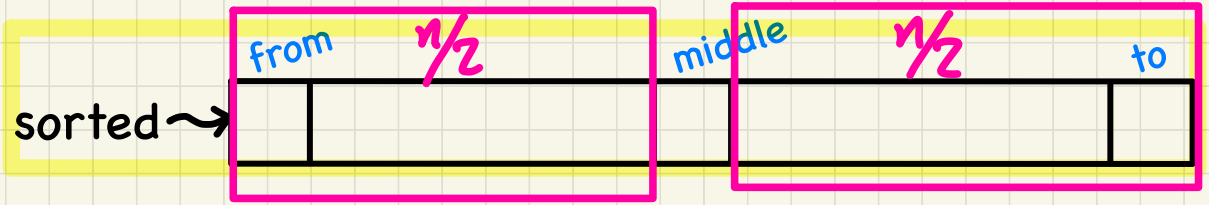
```
boolean binarySearch(int[] sorted, int key) {
  return binarySearchH(sorted, 0, sorted.length - 1, key);
}
boolean binarySearchH(int[] sorted, int from, int to, int key) {
  if (from > to) { /* base case 1: empty range */
    return false; }
  else if(from == to) { /* base case 2: range of one element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

O(1)

O(1)

Running Time as a Recurrence Relation





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

L R

either L or R but not both

Running Time: Unfolding Recurrence Relation

$T(0) = 1$

$T(1) = 1$

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

once reaching here, no more unfoldings

Assume: $n = 2^x$ for $x \geq 0$

↳ without loss of generality.

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

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

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

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

$\vdots$

$= T(1) + 1 + (\dots) + 1$

How many? $\log n$

$O(\log n)$

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

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

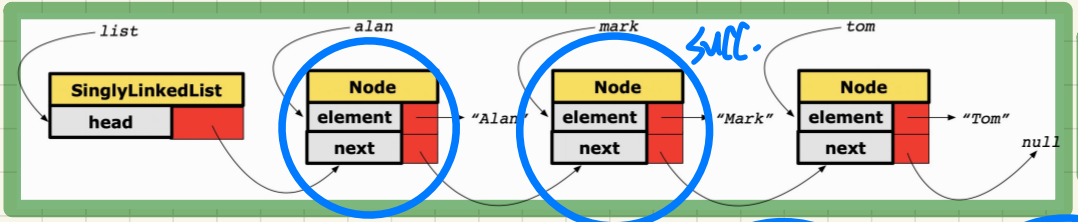
Lecture 5a

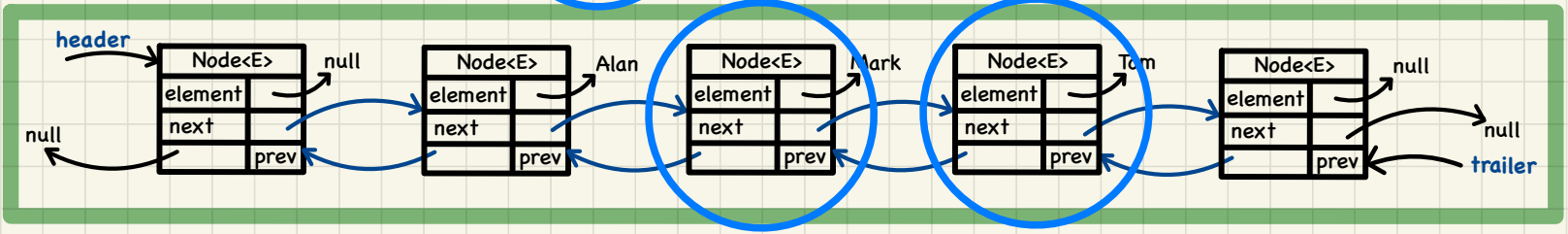
Part A

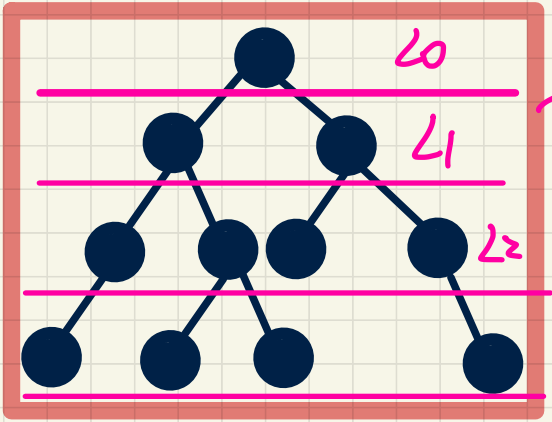
General Trees
Terminology, Applications

Linear vs. Non-Linear Structures

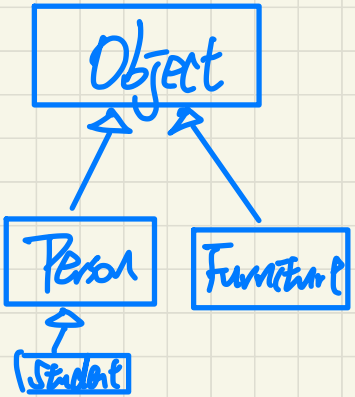
linear DS (can't branch out when traversing)



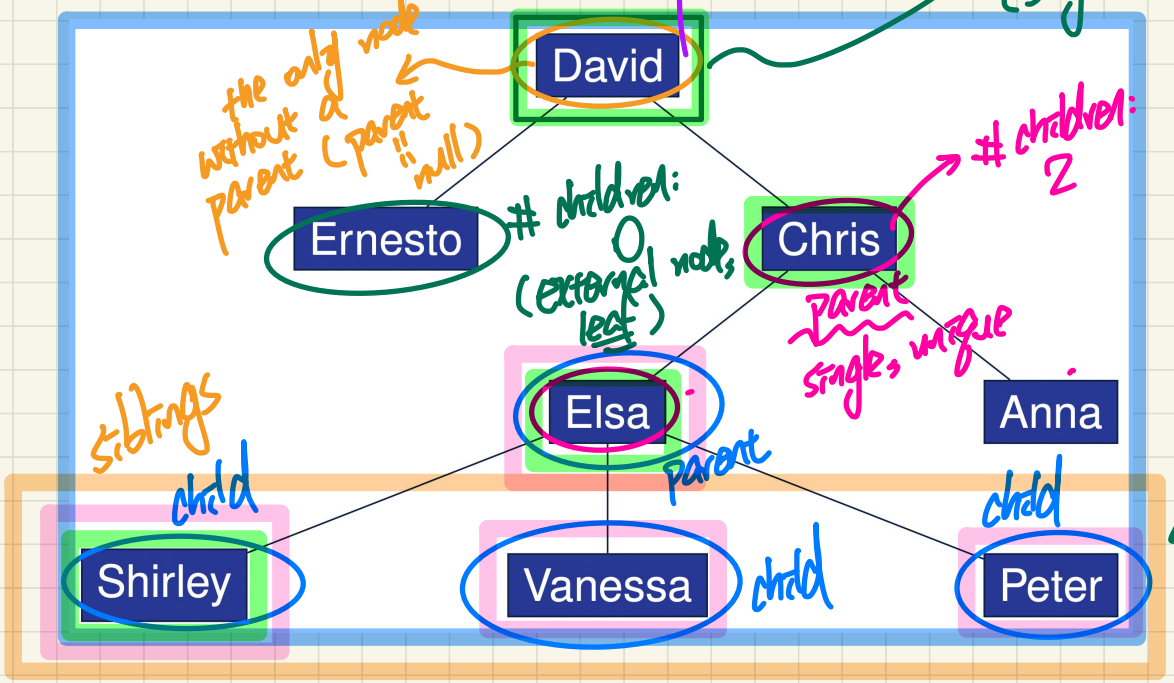




→ non-linear DS (levels = branching out structure) & hierarchical (e.g. inheritance)



General Trees: Terminology (1)



descendants of root covers the entire tree.

root (single, unique)

the only node without a parent (parent = null)

children: 0 (external node, leaf)

children: 2

parent single, unique

siblings

child

parent

child

child

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

ancestors of shirley:
shirley, elsa, chris
david

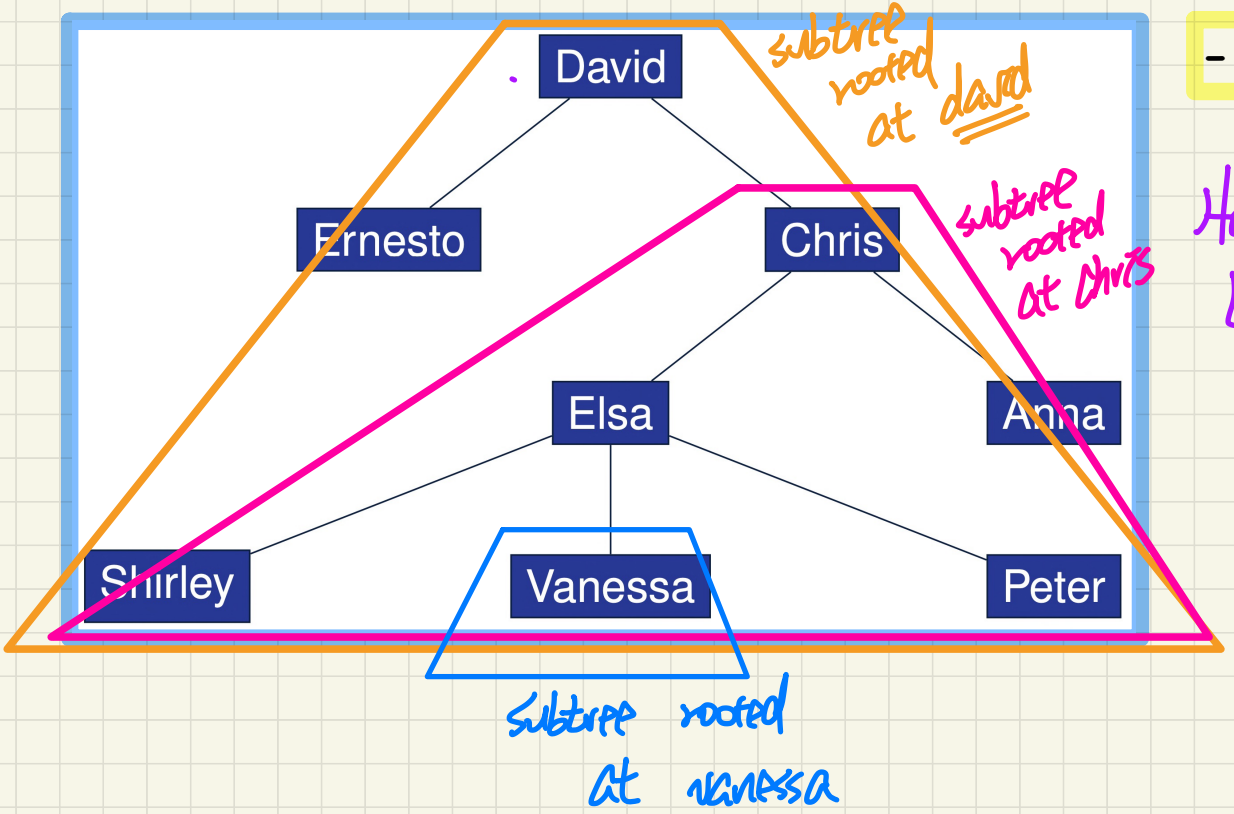
descendants of elsa:
elsa, shirley, vanessa, peter

Elsa is parent of vanessa
Chris is parent of Elsa
⇒ Chris is parent of vanessa ✗

General Trees: Terminology (2)

subtrees ≈ sub-arrays.

- subtree

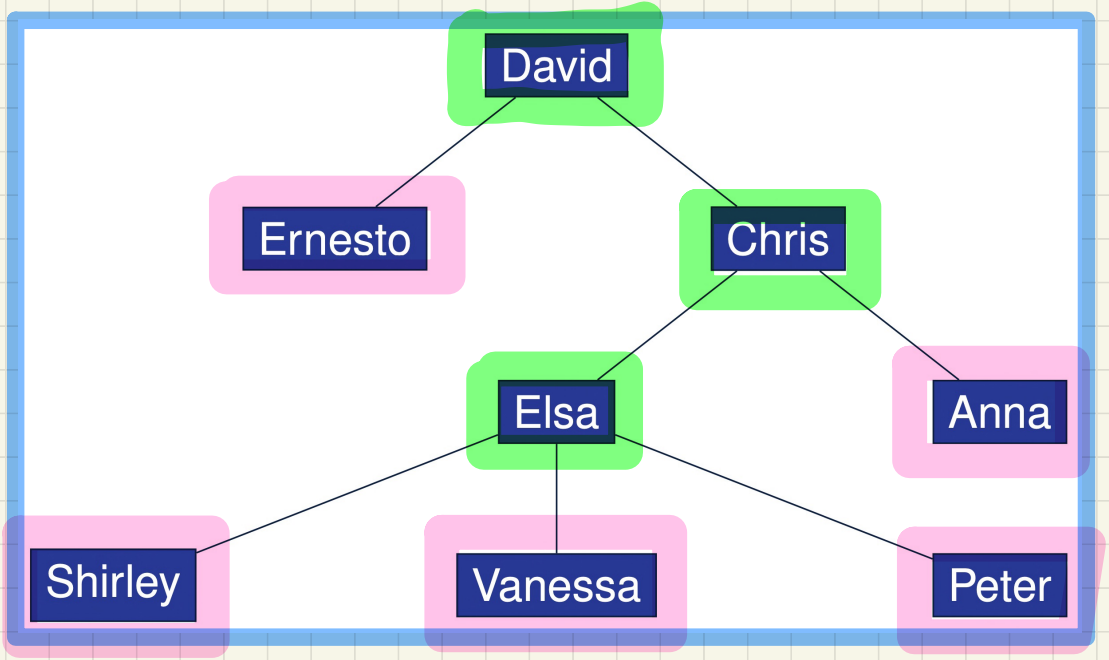


How many subtrees?

↳ count # node

8

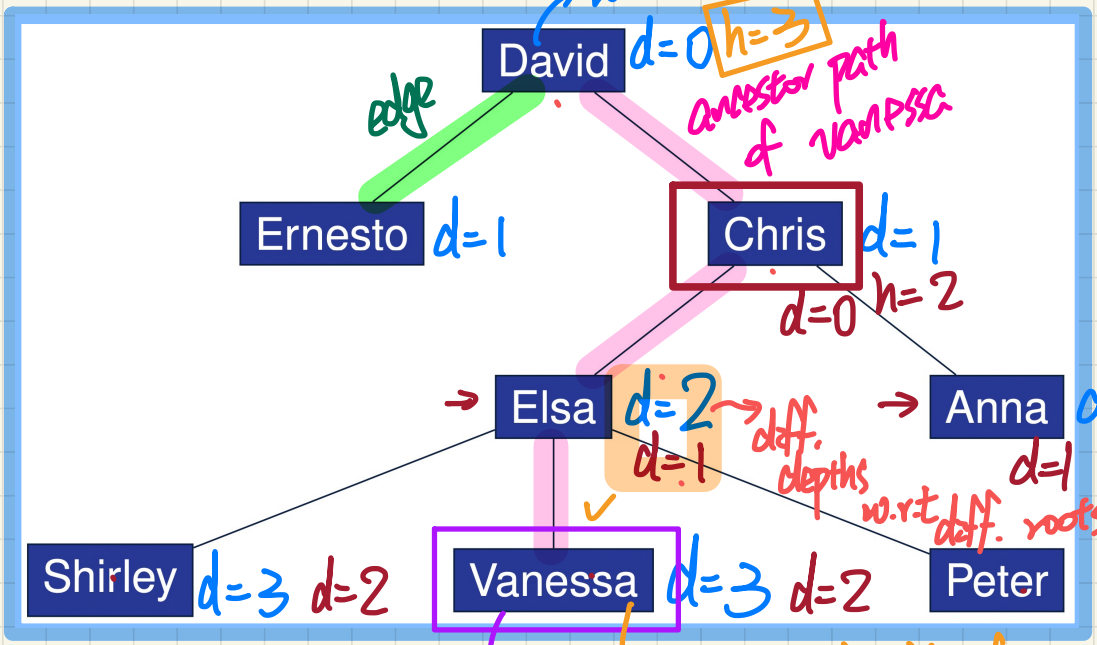
General Trees: Terminology (3)



- external nodes (5)
- internal nodes (3)

not necessarily at the bottom level.

General Trees: Terminology (4)



height of subtree rooted at David.

ancestor path of vanessa

- edge
- path
- depth → # edges from root
- height → max depth among all nodes

diff. depths w.r.t diff. roots

d=0 & h=0 → relative to the subtree root (vanessa)

height of subtree rooted at external node vanessa

* heights of subtrees rooted at external nodes are 0.

General Trees: Recursive Definition

- root
- size

Case 1: Empty Tree

$\emptyset$ root → null

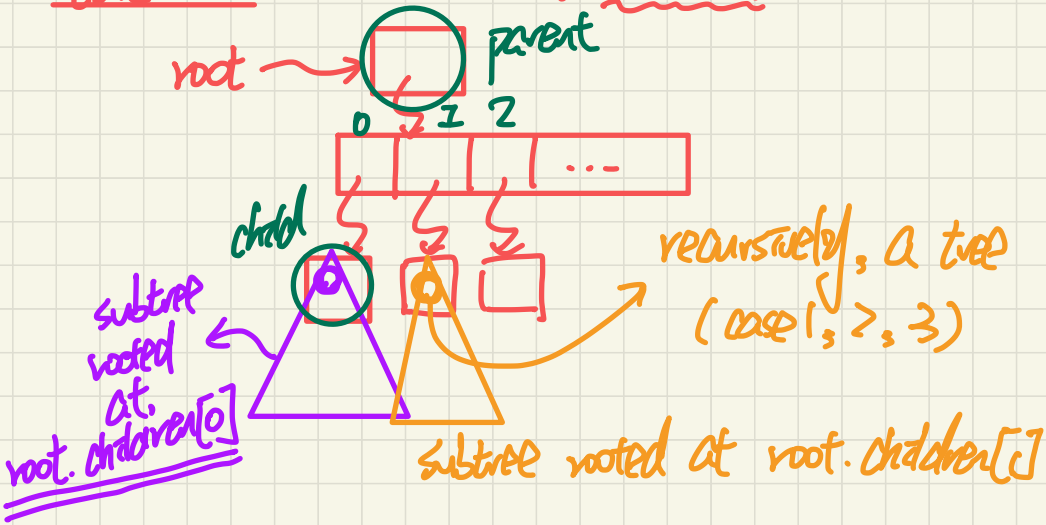
size = 0

Case 2

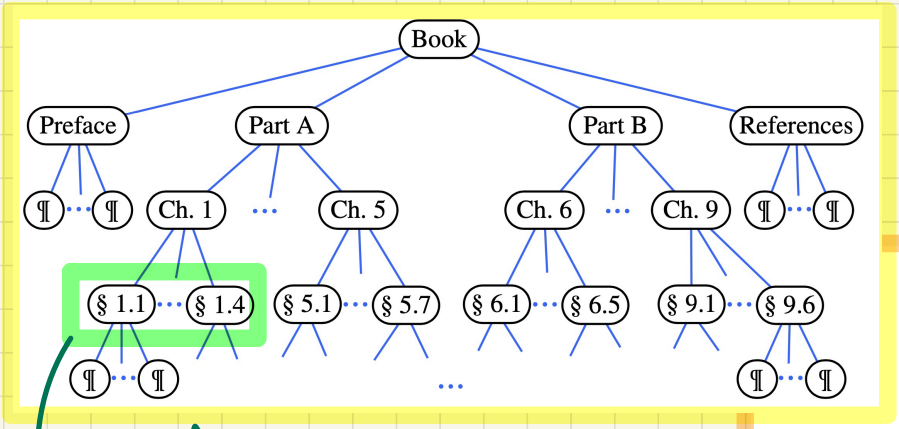
root → "alan"

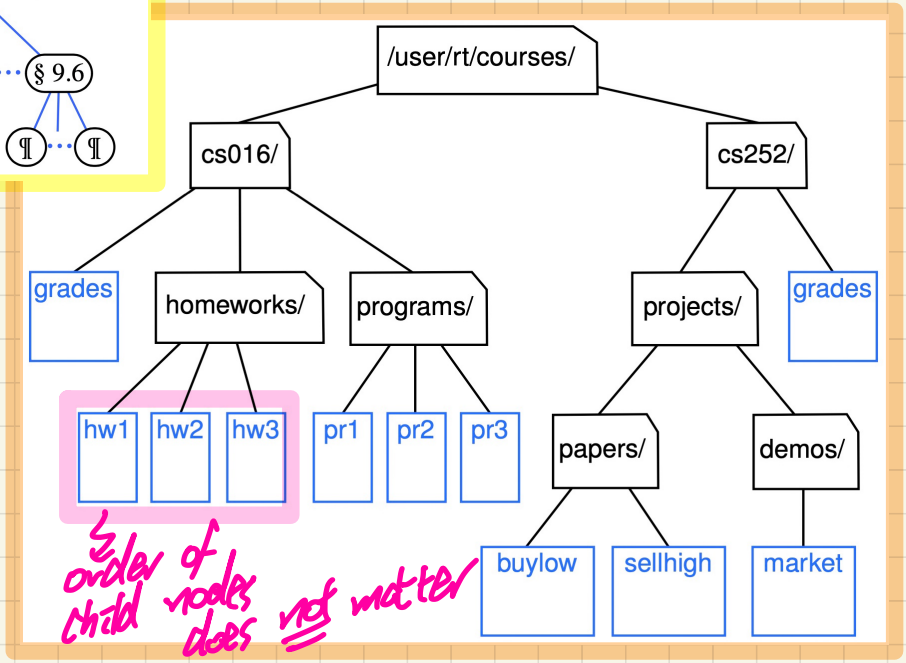
size = 1

Case 3 (Recursive Case) size > 1



General Trees: Ordered vs. Unordered Trees





Lecture 5a

Part B

General Trees
Implementing a Generic Tree in Java

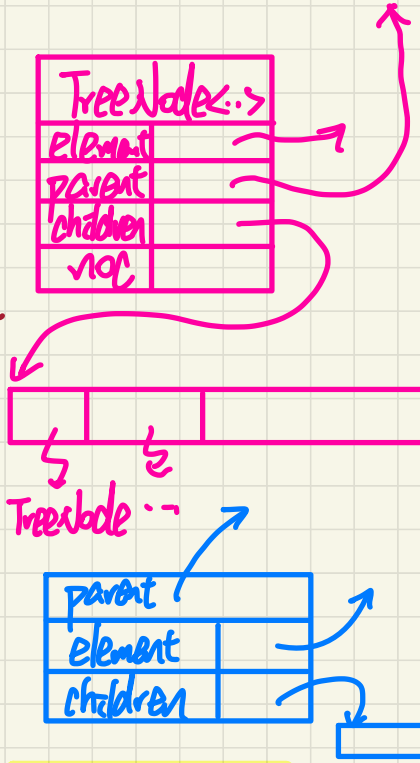
Generic, General Tree Nodes

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

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

Class Cast Exception

TreeNode<E>



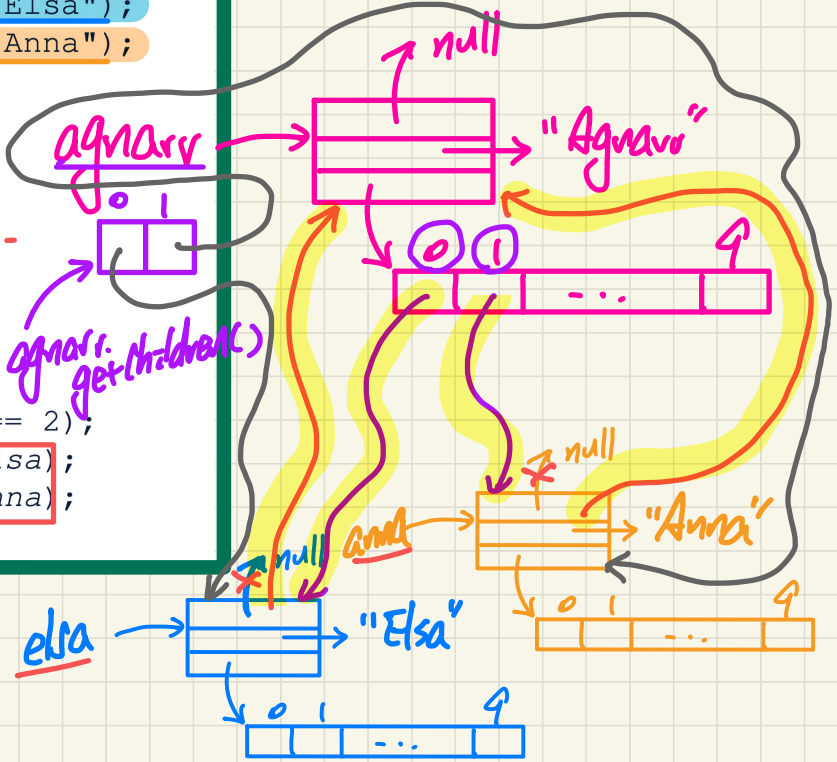
Compare:

+ prev ref.
+ next ref.

in a DLN.

Tracing: Constructing a Tree

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

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

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



Tracing: Computing a Node's Depth

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

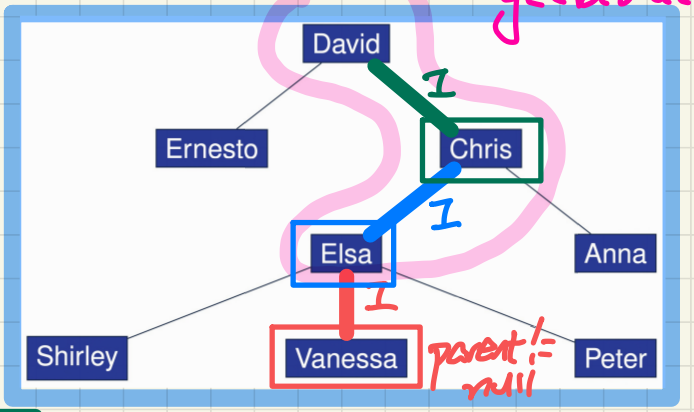
n is the root

edge from n to its parent

strictly smaller subproblem

```
@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));
}
```

trace: depth(anna)



depth(vanessa)

= 1 + depth(elsa)

= 1 + 1 + depth(chris)

= 1 + 1 + 1 + depth(david)

depth(david) = 0

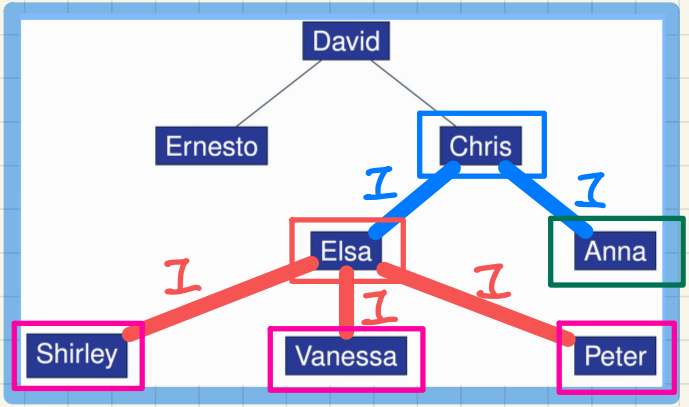
= 3

Tracing: Computing a Tree's Height

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

n is external (leaf)

↳ strictly small problem: height of a child node.




```
@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));
}
```

Exercise

height(chris)

$= 1 + \text{MAX}(\text{height(elsa)}, \text{height(anna)})$

$= 1 + \text{MAX}(1 + \text{MAX}(h(s), h(v), h(p)), 0)$

$= 2$

Lecture 5a

Part C

Binary Trees
Definition, Terminology, Properties

Binary Trees: Recursive Definition

- root
- size

Case 1: Empty BT

size = 0

$\emptyset$ root → null

Case 2: one-node BT

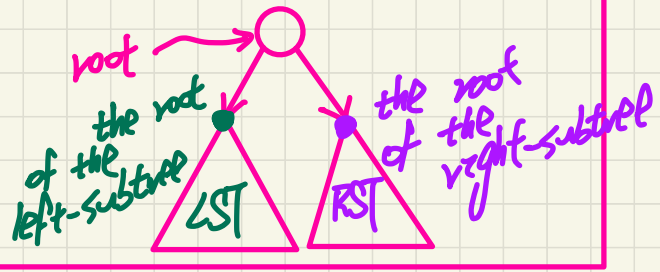
size = 1

root → ?

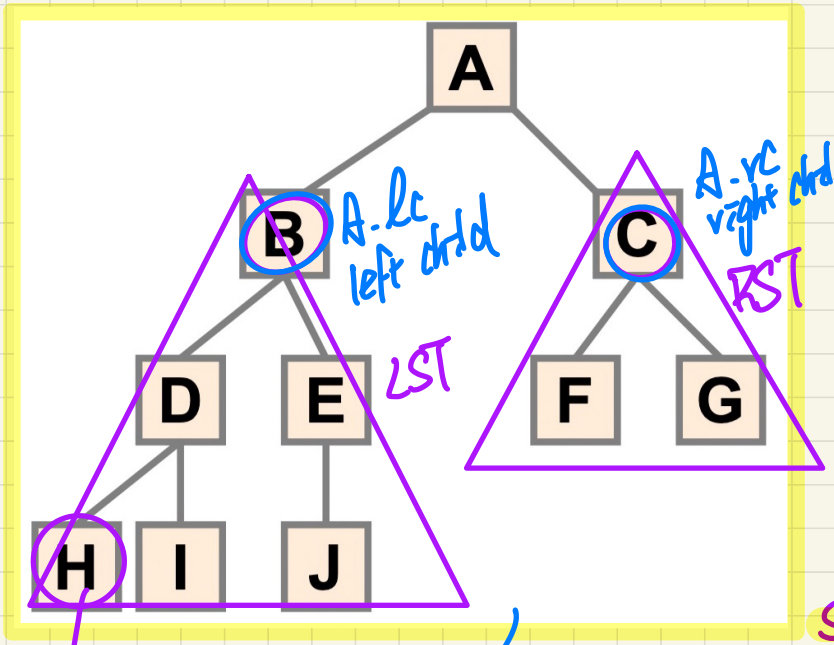
algorithm on a BT should be recursive

given that a BT's structure is recursive.

Case 3: size of BT > 1



BT Terminology: LST vs. RST



Strategy of Recursion on BT:

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

e.g.,

+ counting size
+ searching item

size(A) = 1 + size(A.lc) + size(A.rc)

size(H) = 1

ext. node (no further recursion necessary).

BT without satisfying the "search property"

search(A, item) = A.equals(item) || search(A.lc, item) || search(A.rc, item)

search(H, item) = H.equals(item)

BT Terminology: Depths, Levels, Max # of Nodes

Max # of nodes in a tree with height h:

$2^0 + 2^1 + 2^2 + \cdots 2^h$

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

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

Level? $\stackrel{d}{=}$ depth

Max # Nodes at Level?

$1 = 2^0$

r

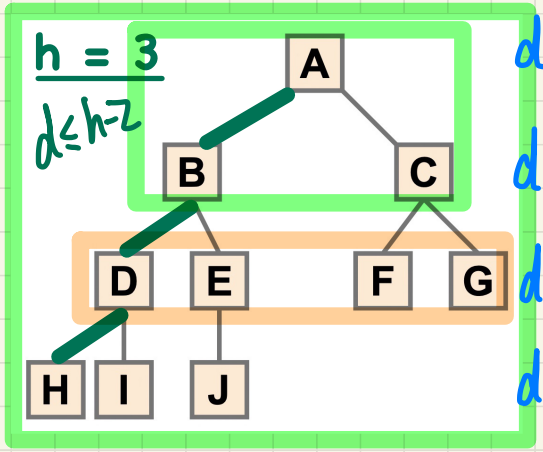
$1 * 2 = 2 = 2^1$

$2 * 2 = 4 = 2^2$

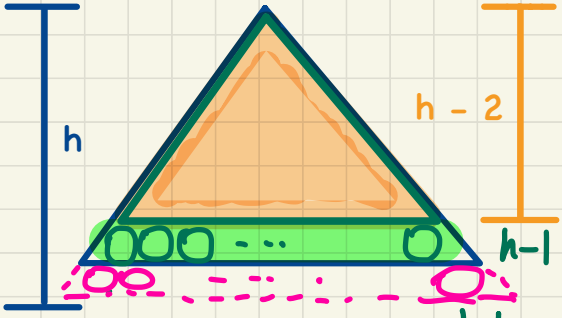
$4 * 2 = 8 = 2^3$

2^h

BT Terminology: Complete vs. Full BTs

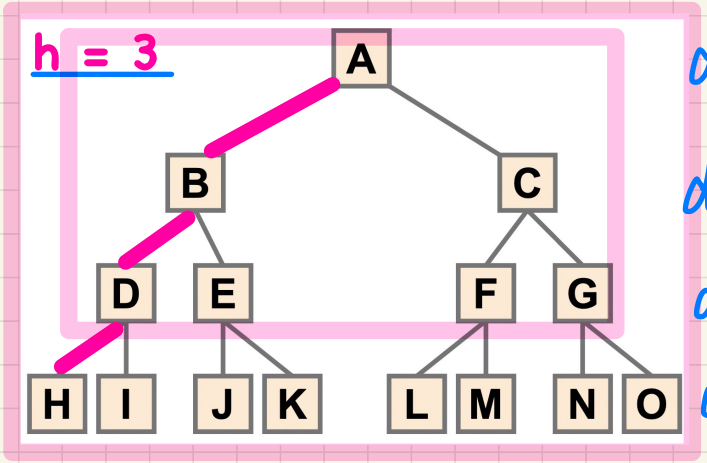


d=0
d=1 h-2
d=2 h-1
d=3 h

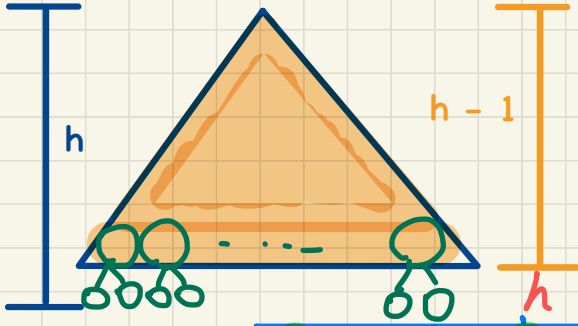


Min # nodes? $(1+2+4\cdots+2^{h-1})+1=2^h$

Max # nodes? $(1+2+4\cdots+2^{h-1})+2^h=(2^h-1)+2^h=2^{h+1}-1$



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



Min # nodes?

Max # nodes?

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

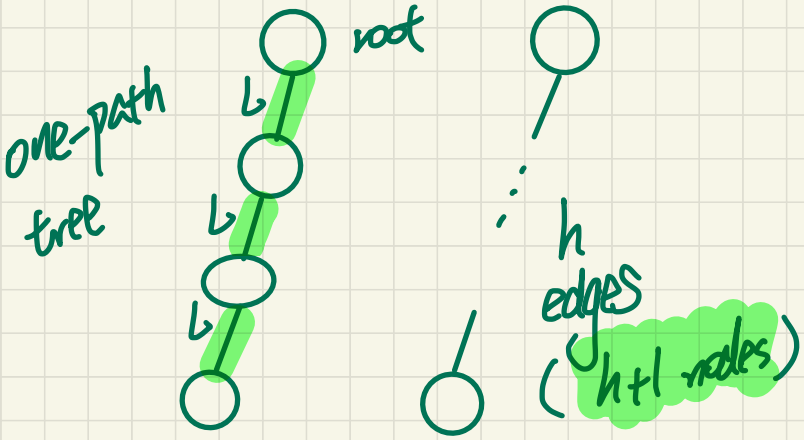
BT Properties: Bounding # of Nodes

Given a *binary tree* with ***height*** $h \geq 0$, the ***number of nodes*** n is bounded as:

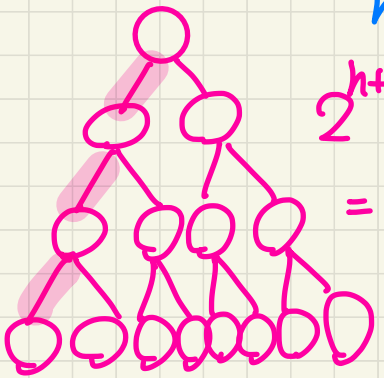
$$h + 1 \leq n \leq 2^{h+1} - 1$$

For example, say h = 3

Minimum # of nodes



Maximum # of nodes

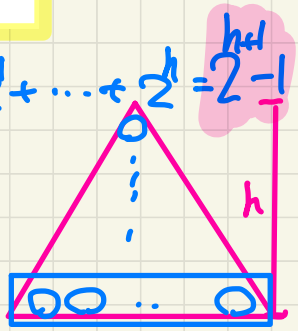


$2^{h+1} - 1$

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

$= 15$

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



BT Properties: Bounding Height of Tree

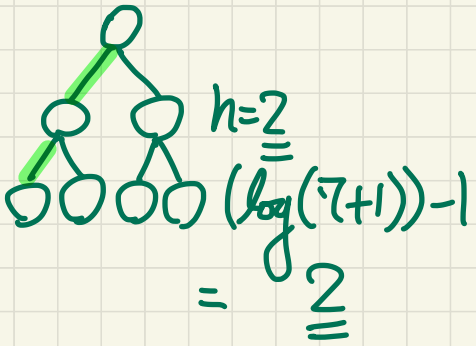
Given a *binary tree* with $n \geq 0$ ***nodes***, the ***height*** h is bounded as:

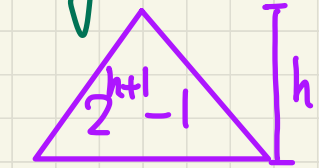
$$\log(n+1) - 1 \leq h \leq n - 1$$

For example, say n = 7.

Minimum height

(branch out two ways whenever possible)





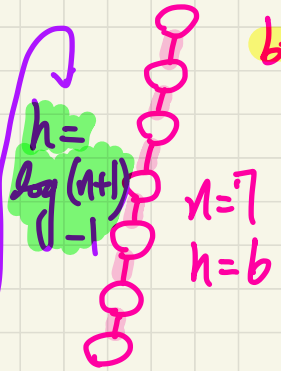
$n = 2^{h+1} - 1$

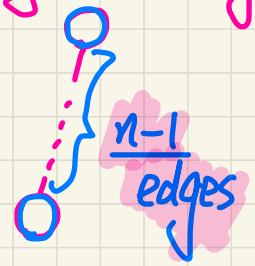
$n+1 = 2^{h+1}$

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

Maximum height

(don't waste any nodes branching two ways)





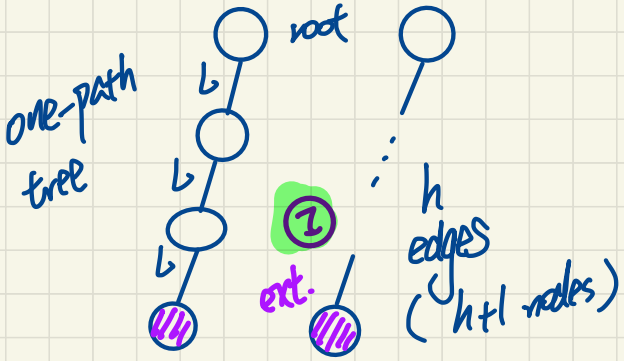
BT Properties: Bounding # of External Nodes

Given a *binary tree* with *height* h (≥ 0) the *number of external nodes* n_E is bounded as:

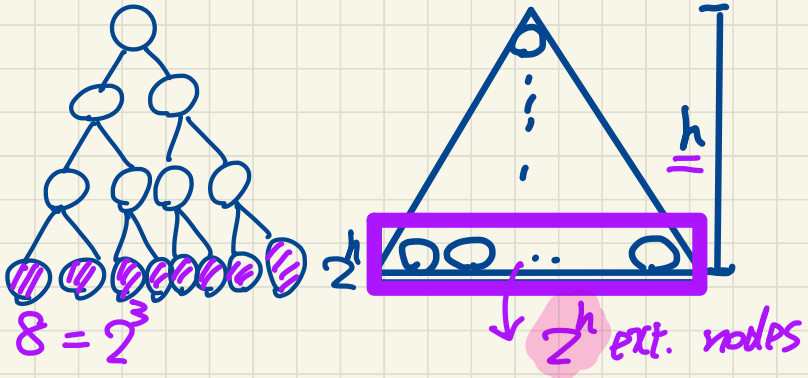
$$1 \leq n_E \leq 2^h$$

For example, say h = 3

Minimum # of External Nodes



Maximum # of External Nodes



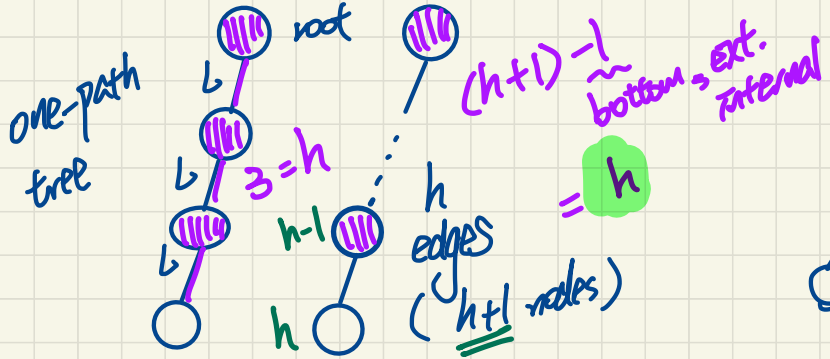
BT Properties: Bounding # of Internal Nodes

Given a *binary tree* with *height* $h \geq 0$, the *number of internal nodes* n_I is bounded as:

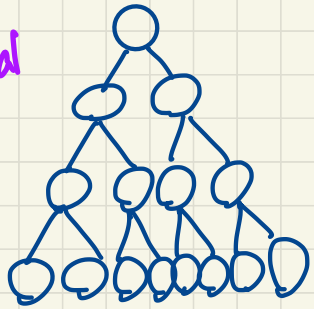
$$h \leq n_I \leq 2^h - 1$$

For example, say h = 3

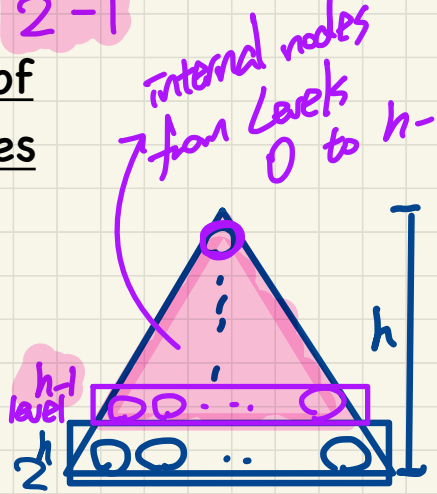
Minimum # of Internal Nodes



Maximum # of Internal Nodes



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



Lecture 5a

Part C

Binary Trees
Definition, Terminology, Properties
(continued)

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

Given a *binary tree* that is:

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

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

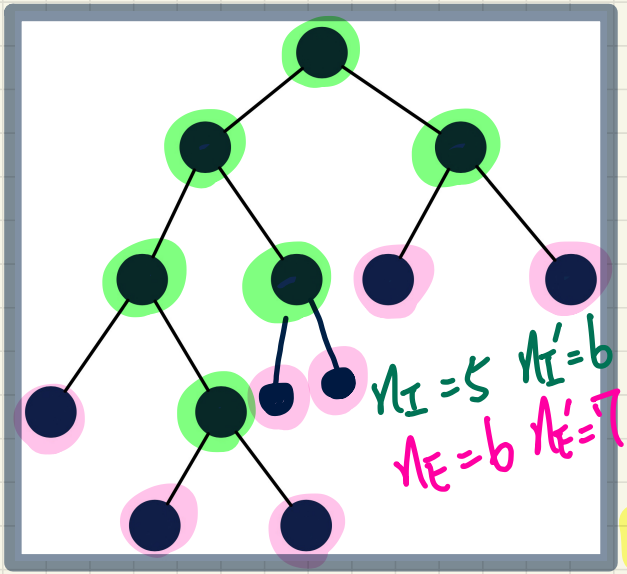
Induction on Size of Proper BT

$* \quad n_E' = n_E + 1$

$= \{\text{ind. hypoth.}\}$

$(n_I + 1) + 1$

$= n_I' + 1$ ∎



Base Case: A proper BT with size 1

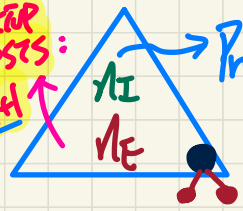
no internal node

⇒ "proper" property is satisfied

(without violation witness)

Inductive Recursive Case

Inductive hypothesis: $n_E = n_I + 1$



→ Proper BT with size > 1

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

$n_I' = n_I + 1$ *

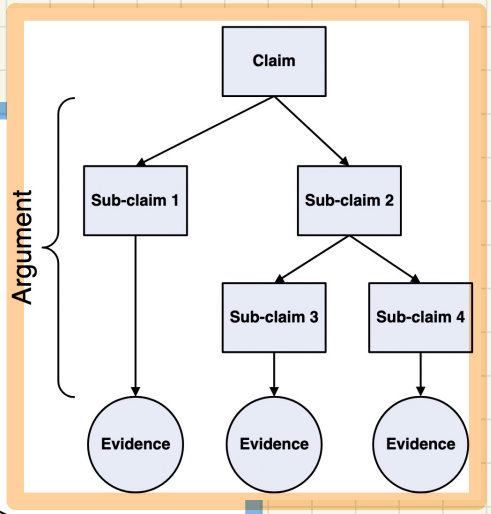
Lecture 5a

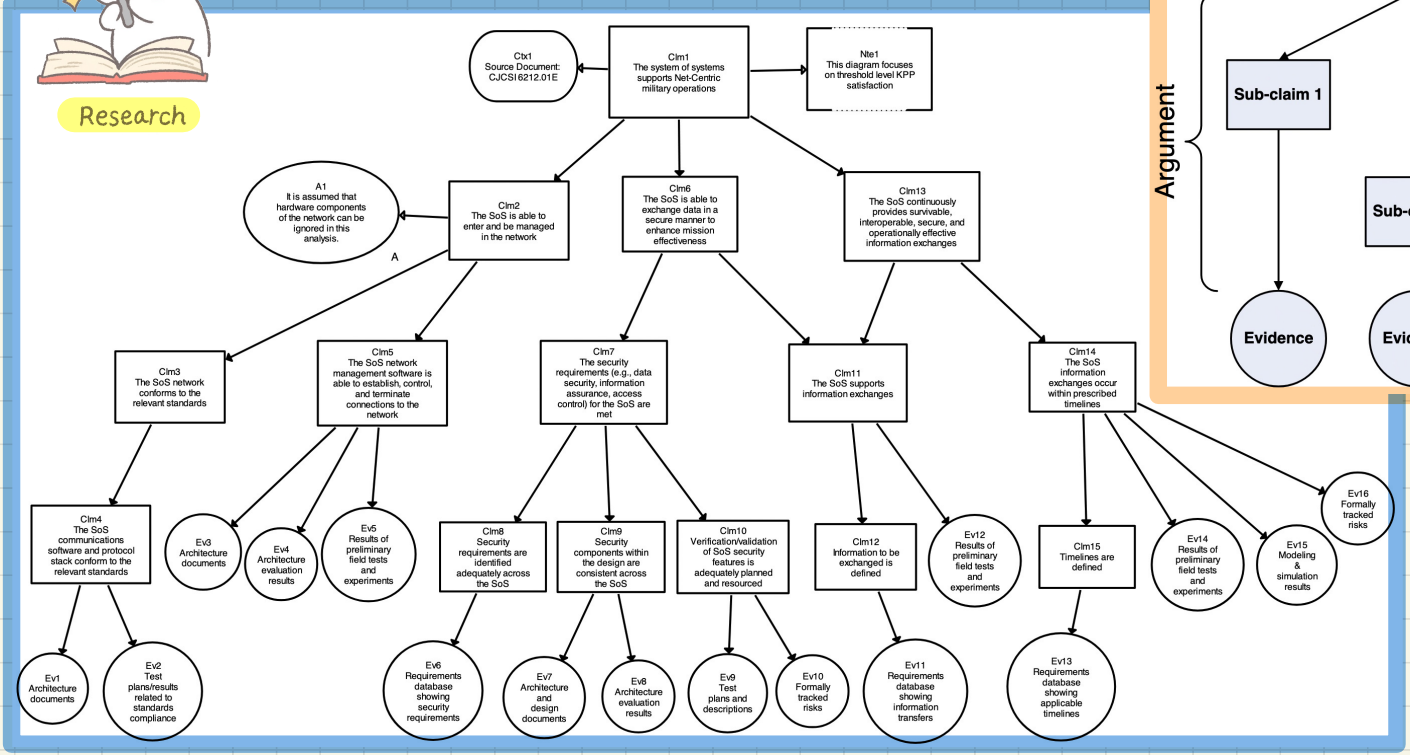
Part D

Binary Trees
Applications

Applications of General Trees: Assurance Cases

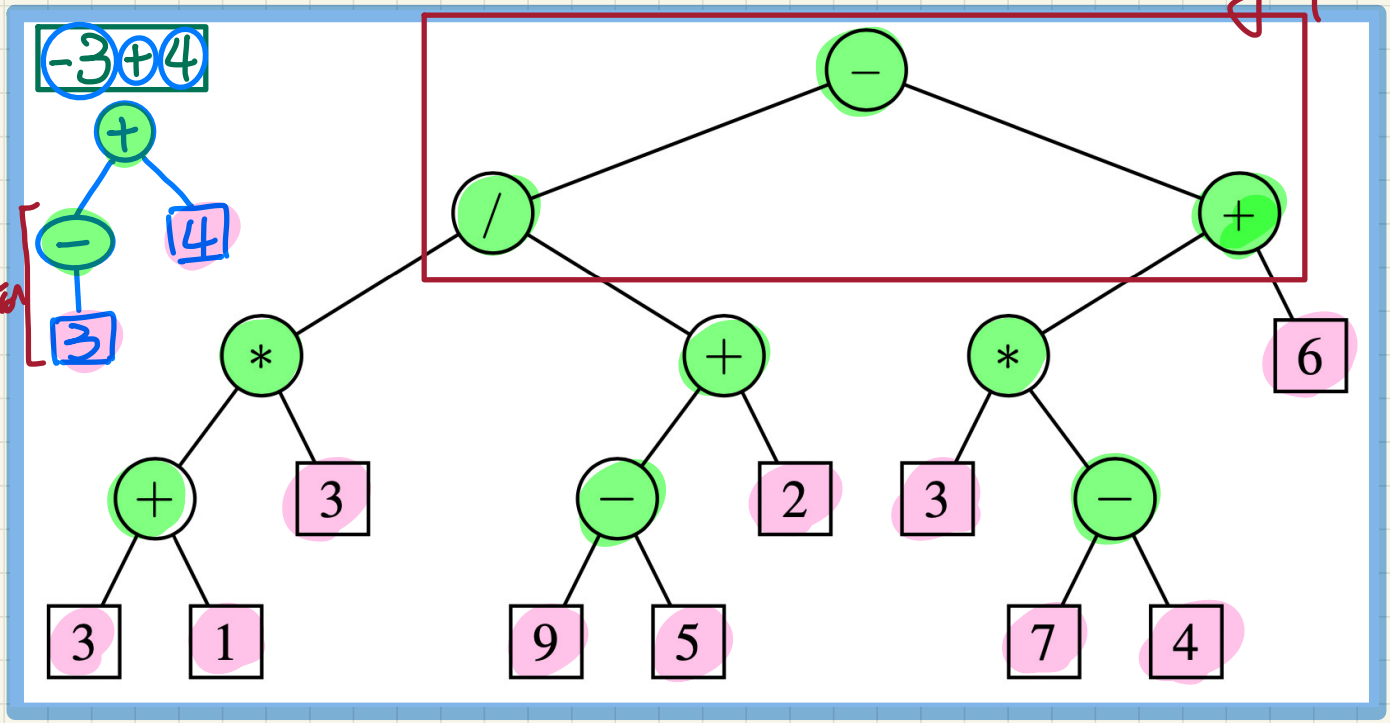
Research on "Assurance Cases" if interested!





Source: https://resources.sei.cmu.edu/asset_files/whitepaper/2009_019_001_29066.pdf

Applications of Binary Trees: Infix Notation



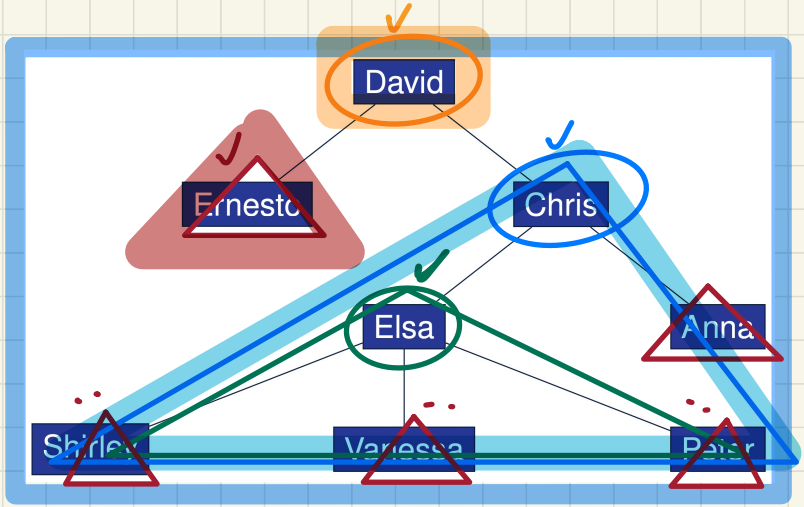
Q. Is the binary tree necessarily proper?

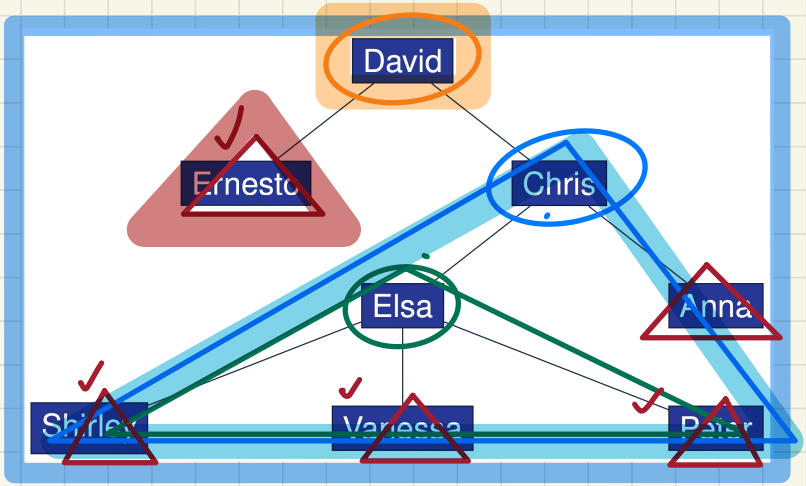
Lecture 5a

Part E

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

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





Pre-Order Traversal from the Root

Parent, pre-order(child nodes)

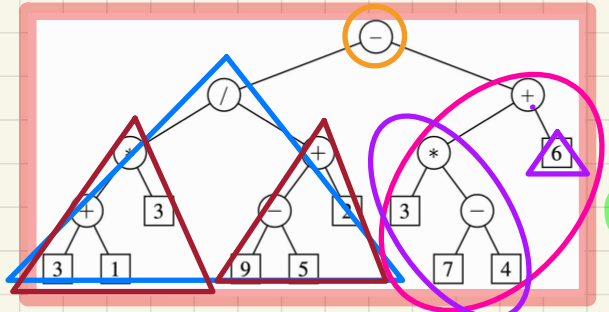
David | Ernesto | Chris | Elsa | Shirley | Vanessa | Peter | Anna

Post-Order Traversal from the Root

post-order(child nodes), Parent

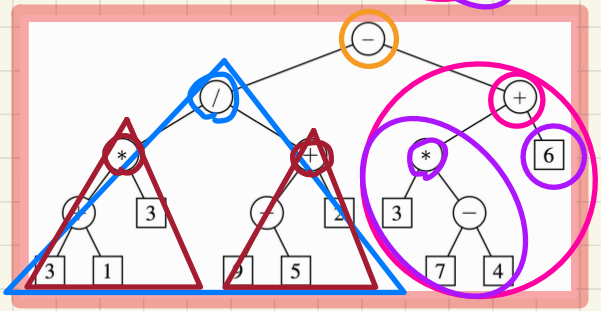
Ernesto | Shirley | Vanessa | Peter | Elsa | Anna | Chris | David

Binary Tree Traversals



Pre-Order Traversal

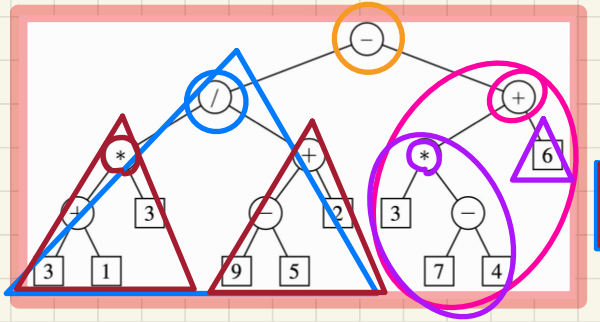
− / * + 3 1 3 + − 9 5 2 + * 3 − 7 4 6



In-Order Traversal

In-Order(LST) Parent In-Order(RST)

3 + 1 * 3 / 9 − 5 + 2 − 3 * 7 − 4 + 6



Post-Order Traversal

3 1 + 3 * 9 5 − 2 + / 3 7 4 − * 6 + −

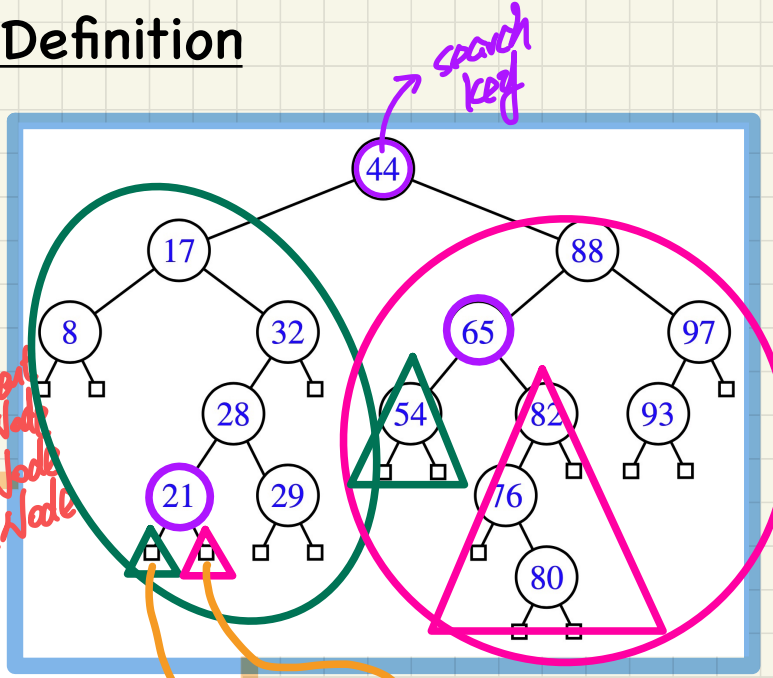
Lecture 5b

Part A

Binary Search Tree - Definition and Property

Binary Search Trees: Recursive Definition

- external node
- internal node
  + LST
  + RST



$\forall x \cdot x \in \emptyset \Rightarrow P(x)$

internal node

≈ element of a Trivially SLLNode/DLLNode

Node p stores (key(p), value(p))

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

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

can't find a violation of search prop.

External node satisfies search property ∵ its LST or RST contains no nodes

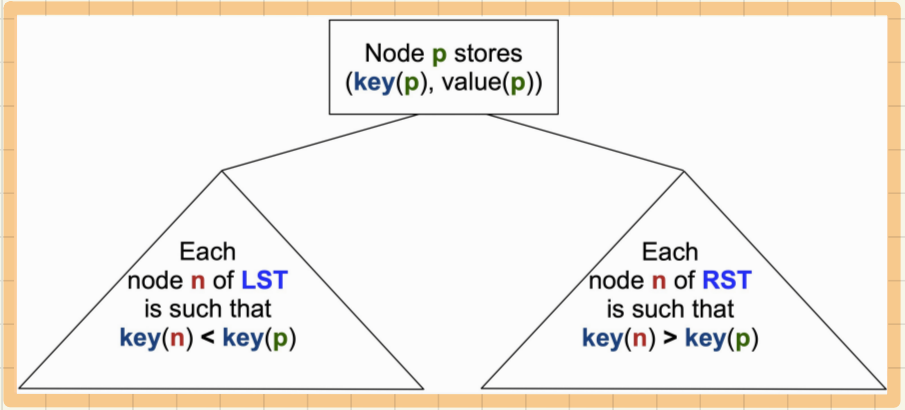
Binary Search Trees: Sorting Property

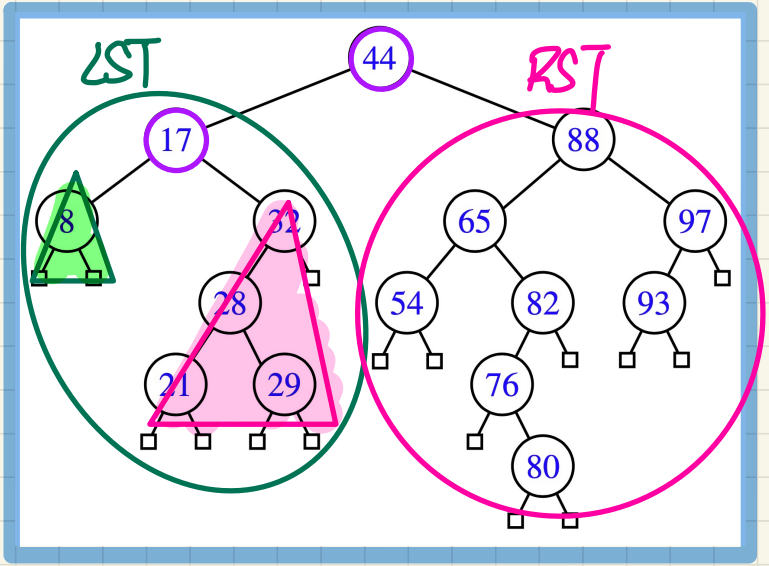
- BST: Non-Linear Structure
- In-Order Traversal

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

in-order of RST

in-order of LST





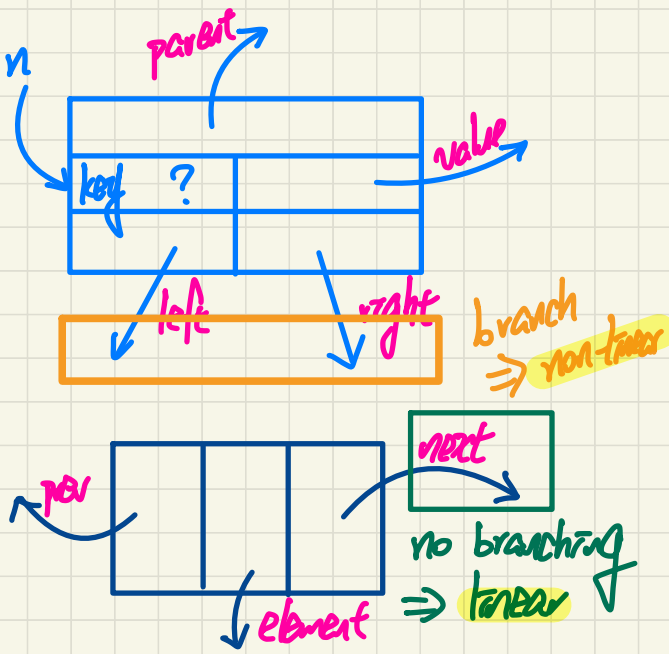
Lecture 5b

Part B

Binary Search Trees -
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) { ... }
}
```



Compare:

+ prev ref.
+ next ref.

in a DLN.

Generic, Binary Tree Nodes - Traversal

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

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

      result.add(root);

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

otherwise, root. isExternal ⇒ return null

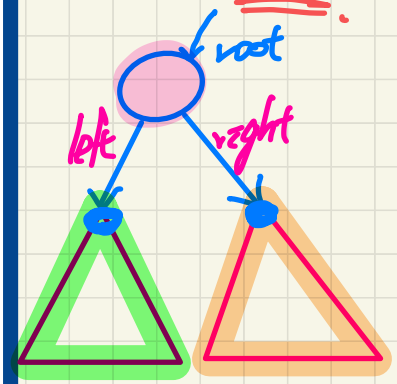
assumed to be the root of some BST

accumulate the returned result from LST

accumulate the returned result from RST

external node

↳ in-order tra. returns null.



Exercises: change to

1. BSTNode<E>[]
2. SLLNode<E>
3. pre-order
4. post-order

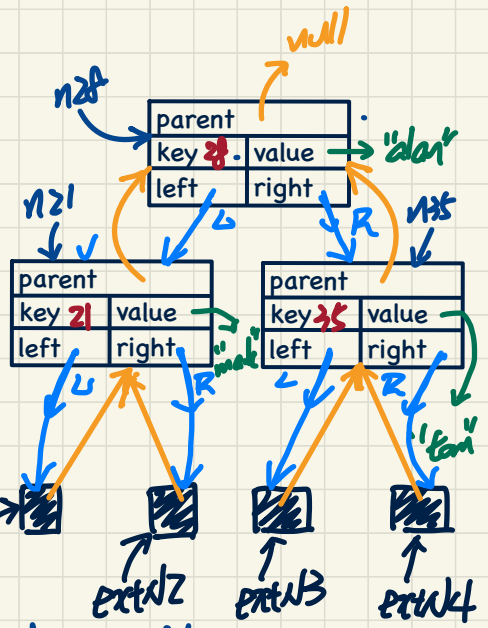
Tracing: Constructing and Traversing a BST

```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());
}
```

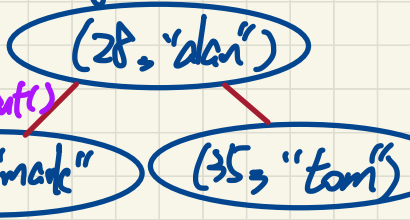
internal nodes

external nodes

sorting properties on keys.



alternatively



tracing

① n28 == n28.getLeft().getParent() T

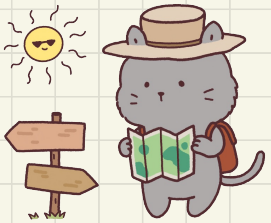
② n28.getLeft() == n28.getLeft().getRight().getParent() T

Lecture 5b

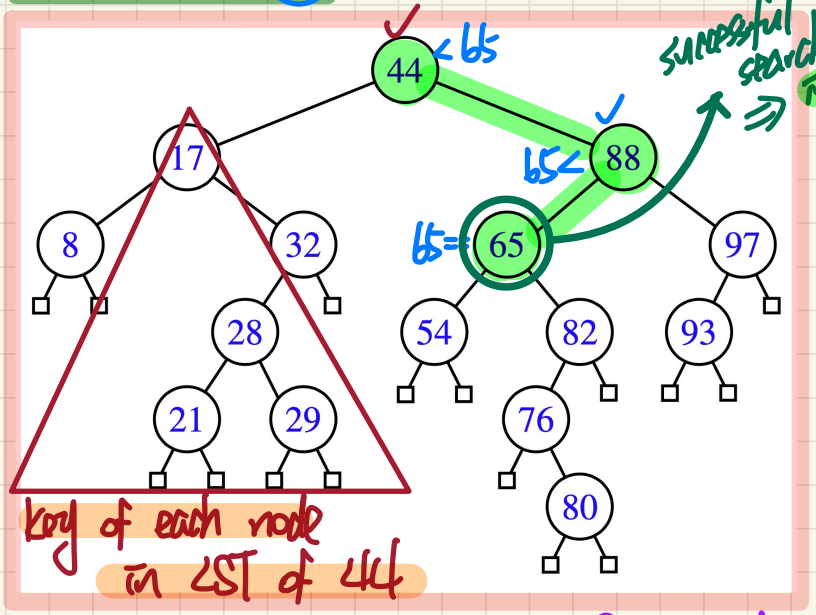
Part C

Binary Search Trees -
Implementing a Generic BST in Java
Searching

BST Operation: Searching a Key



Search key 65

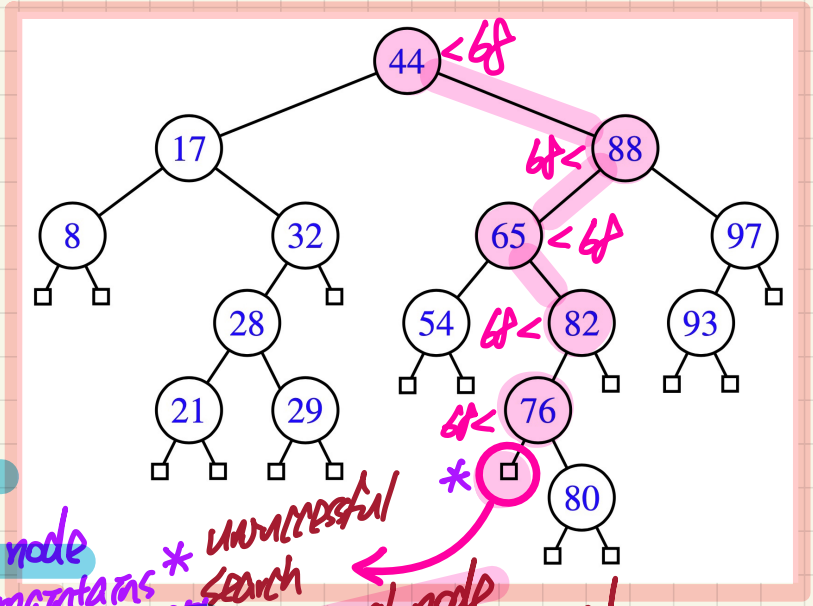


key of each node in LST of 44 must be < 44 (assumed search property)

successful search ⇒ external node storing 65 returned

Assumption: BST ⇒ search property

Search key 68



* Storing this returned ext. node with key 68 maintains the search property ⇒ unsuccessful search external node returned

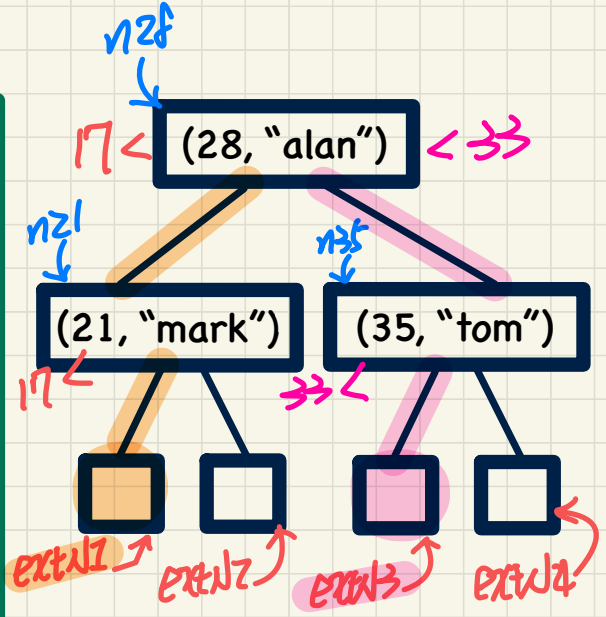
Tracing: Searching through a BST

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

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

internal nodes -> successful search

external nodes -> unsuccessful search.

Exercise



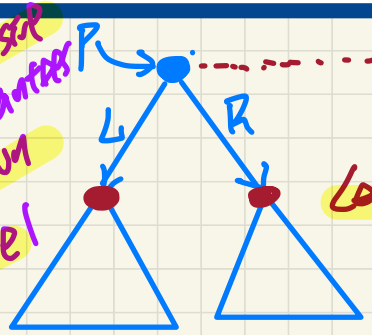
Running Time: Search on a BST

```
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;
}
```

external node

internal node

Each recursive call ~~guarantees~~ moves down by 1 level



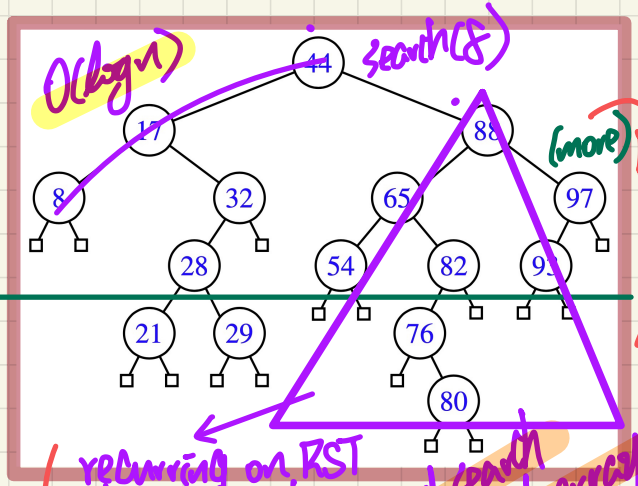
correct: O(h)

RT

O(n) X ∵ RT less if the BST is balanced

O(log n) X ∵ RT more if the BST is ill-balanced

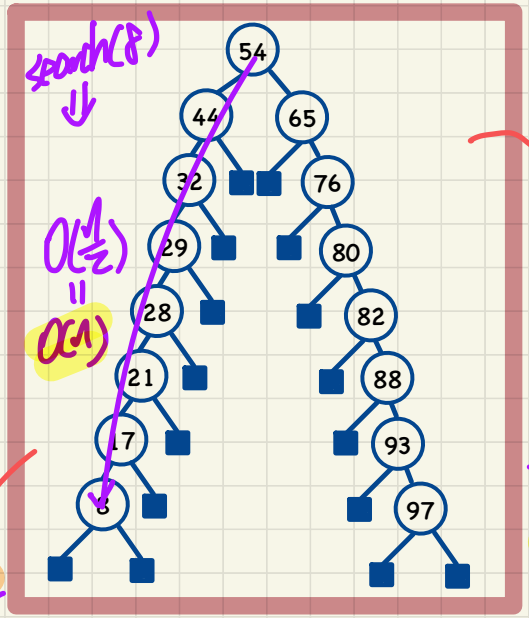
Binary Search: Non-Linear vs. Linear Structures



balanced
$h \approx \log n$
$\log 15 = 3...$

(more)

recurring on RST reduces the search space by half.



→ ill-balanced

O(log n)

half guarantee the search space is reduced by half.

sorted array (in-order traversal of BST) half.

A binary search on a sorted array is equivalent to as far as RT is concerned a search on a balanced BST.

recurring on right

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

REVIEW!

Lecture 5b

Part D

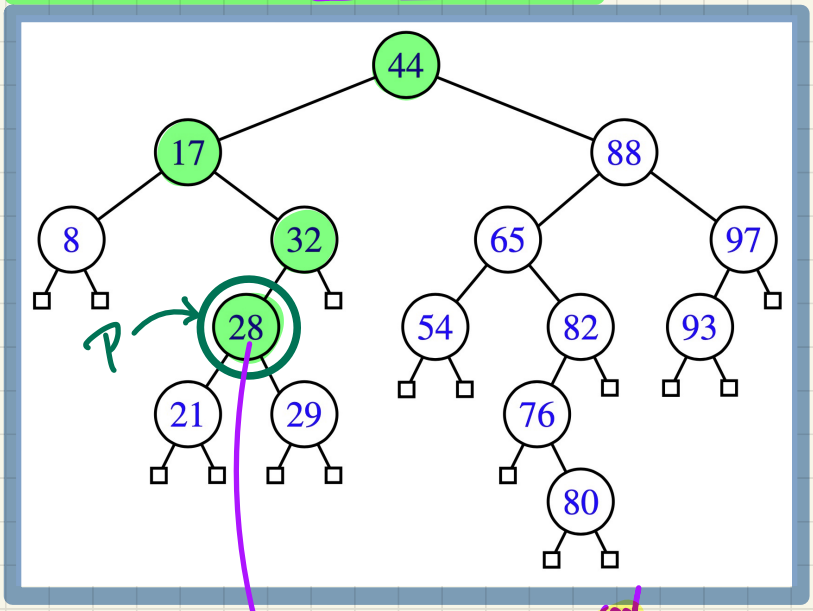
Binary Search Trees -
Implementing a Generic BST in Java
Insertion

Visualizing BST Operation: Insertion

search key

data value (not for searching)

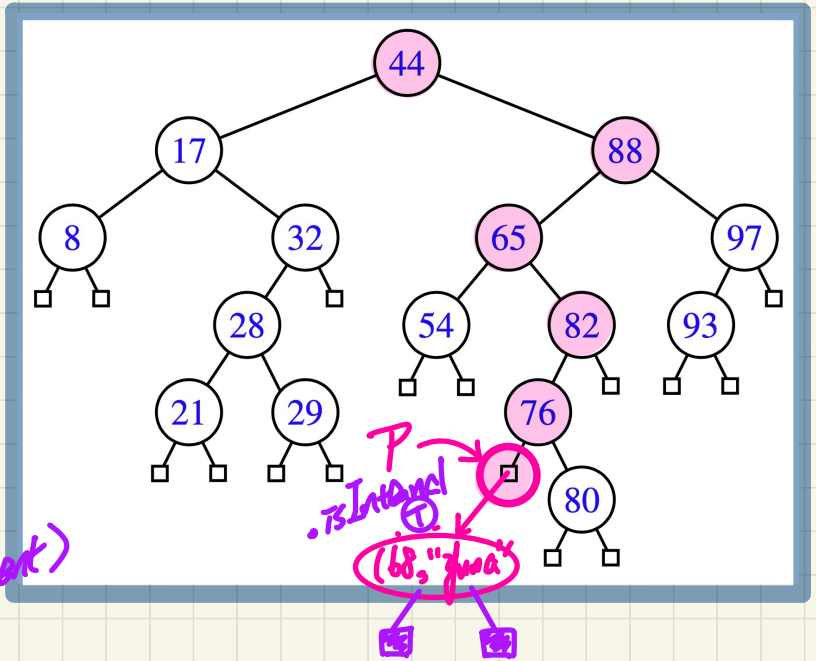
Insert Entry (28, "suyeon")



Replace whatever value that's associated with key 28 by "suyeon" (e.g. setElement)

O(h) ~ height of tree

Insert Entry (68, "yuna")



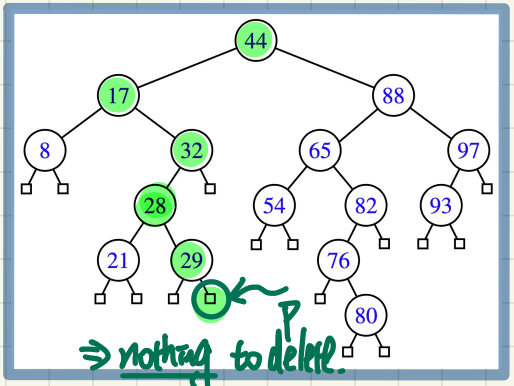
Lecture 5b

Part E

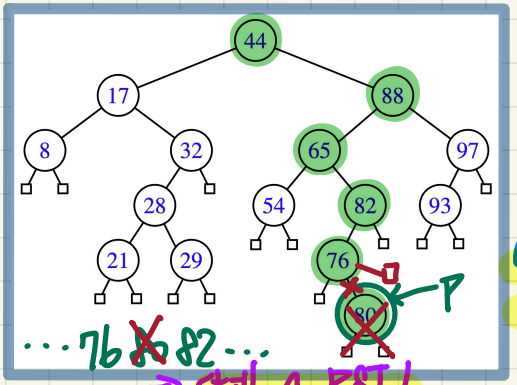
Binary Search Trees -
Implementing a Generic BST in Java
Deletion

Visualizing BST Operation: Deletion

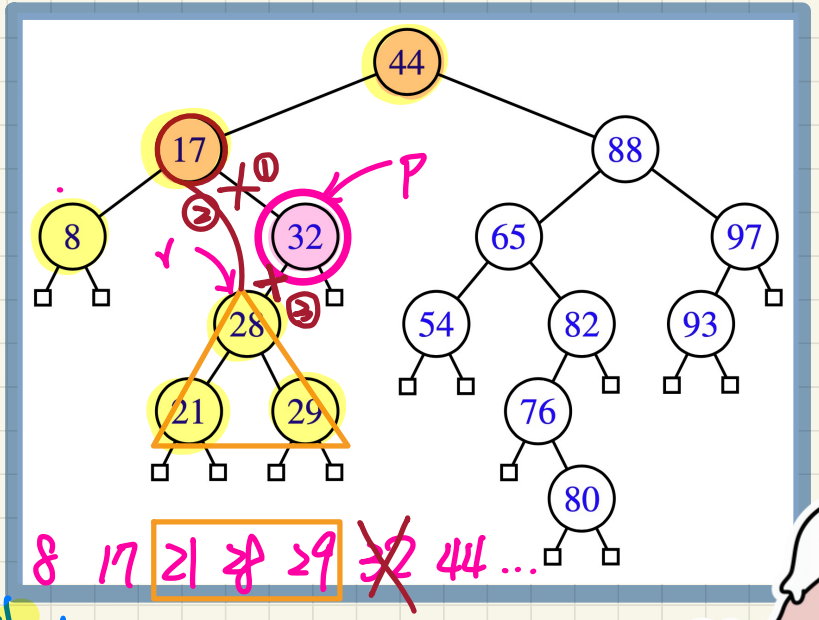
Case 1: Delete Entry with Key 31



Case 2: Delete Entry with Key 80



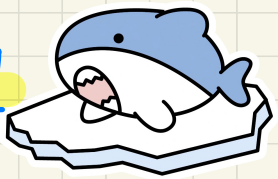
Case 3: Delete Entry with Key 32

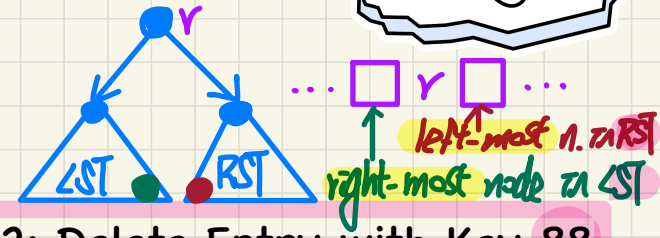


Orphan subtree: a subtree rooted at an internal child node

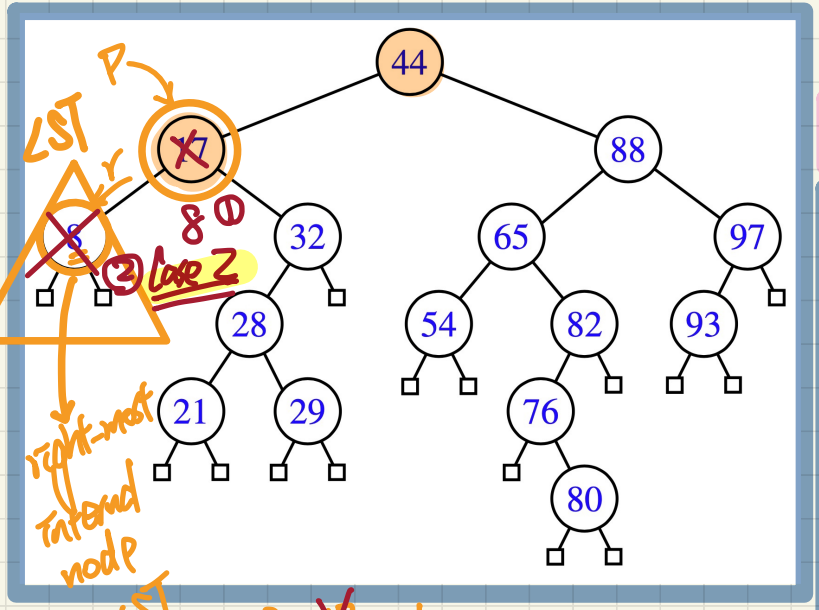
Visualizing BST Operation: Deletion

In-Order Traversal

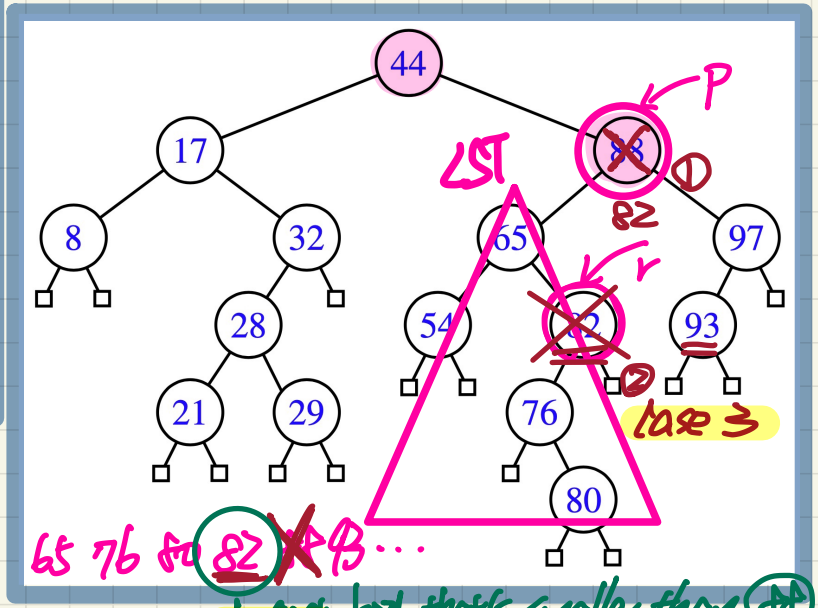




Case 4.1: Delete Entry with Key 17



Case 4.2: Delete Entry with Key 88

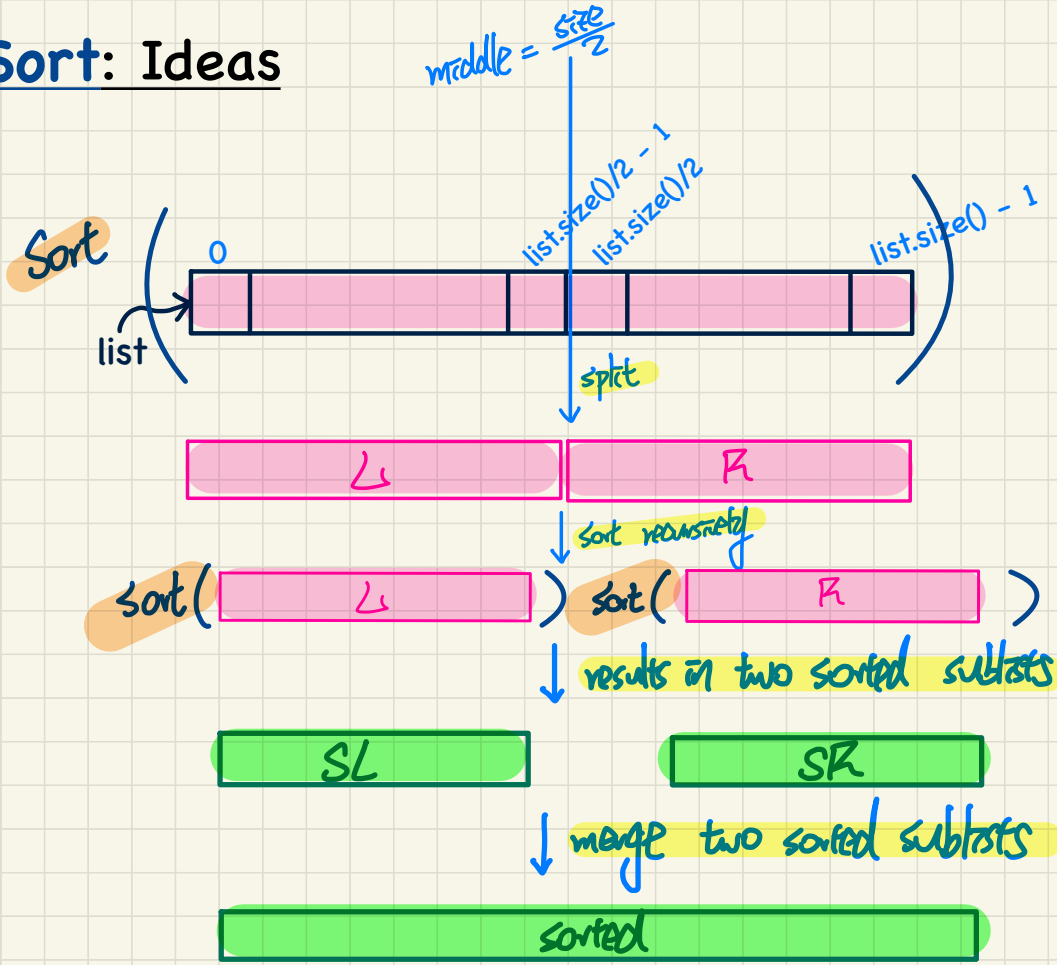


Lecture 4

Part C

Examples on Recursion
Merge Sort

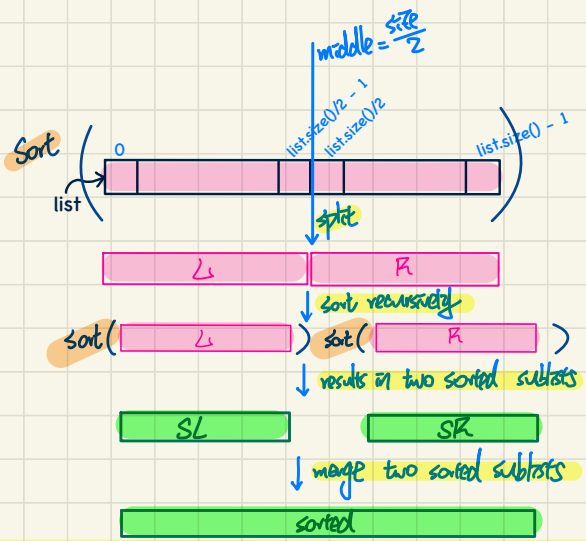
Merge Sort: Ideas

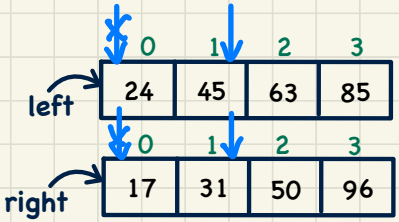


Merge Sort in Java

```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

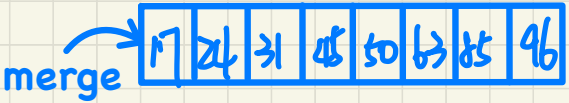




Precondition

L and R sorted

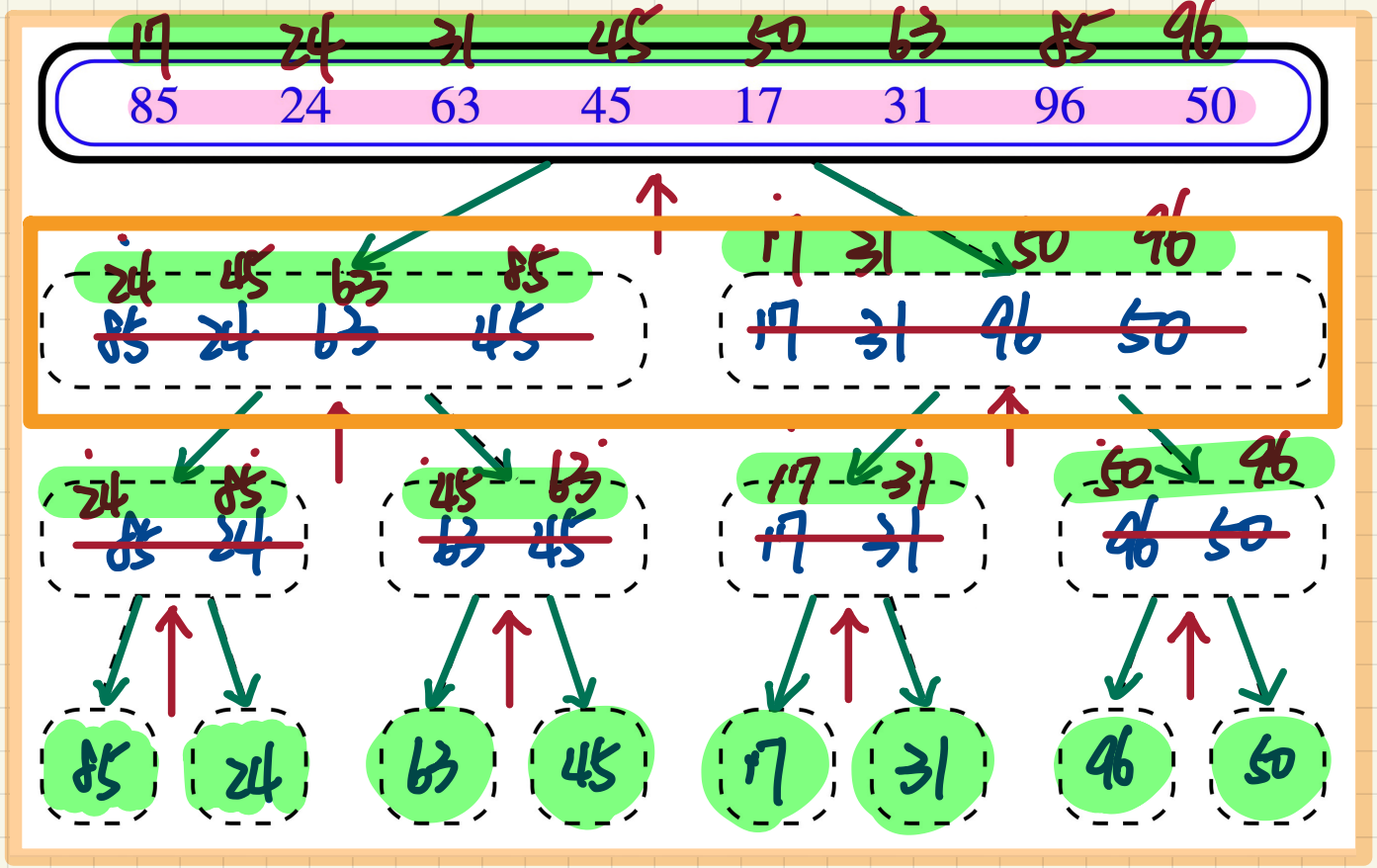
```java
/* Assumption:  L and R are both already sorted. */
private List<Integer> merge(List<Integer> L, List<Integer> R) {
  List<Integer> merge = new ArrayList<>();
  if(L.isEmpty()||R.isEmpty()) { merge.addAll(L); merge.addAll(R); }
  else {
    int i = 0;
    int j = 0;
    while(i < L.size() && 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;
}
```



Merge Sort: Tracing

split

merge



Lecture 5c

Part A

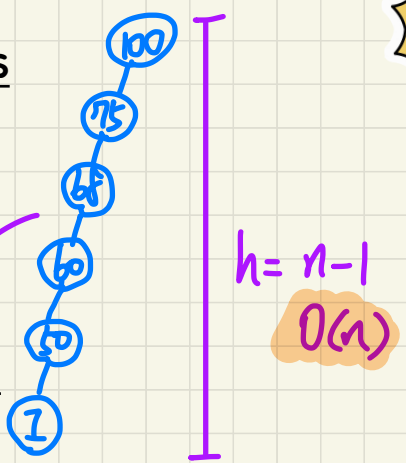
Balanced Binary Search Tree - Motivation and Property

Worst-Case RT: BST with Linear Height

Example 1: Inserted Entries with Decreasing Keys

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

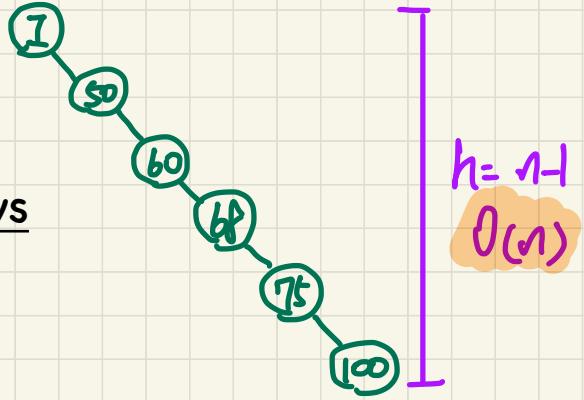
key



Example 2: Inserted Entries with Increasing Keys

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

searching worst-case RT: O(1)



Example 3: Inserted Entries with In-Between Keys

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

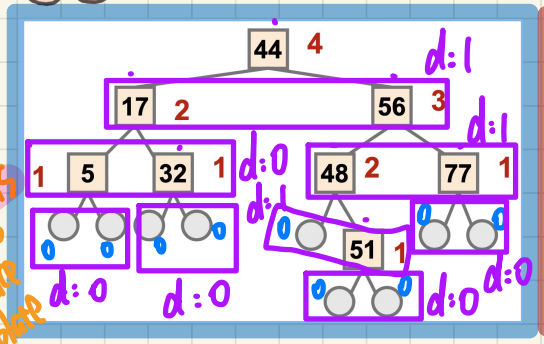
Exercise

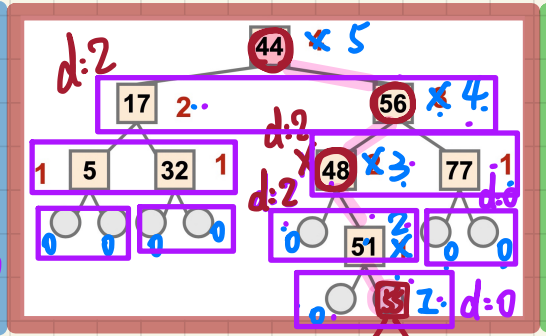
Balanced BST: Definition

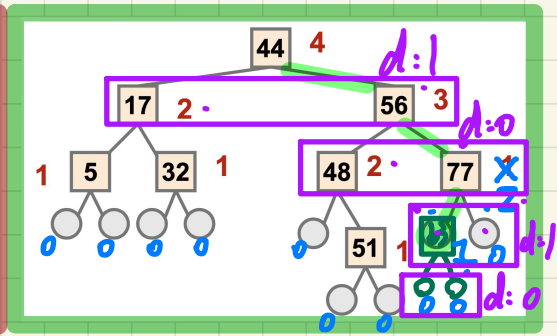
- internal node
- height
- height balance

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

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







when rotations stop to take place

ancestor path of 55: 55, 51, 48, 56, 44

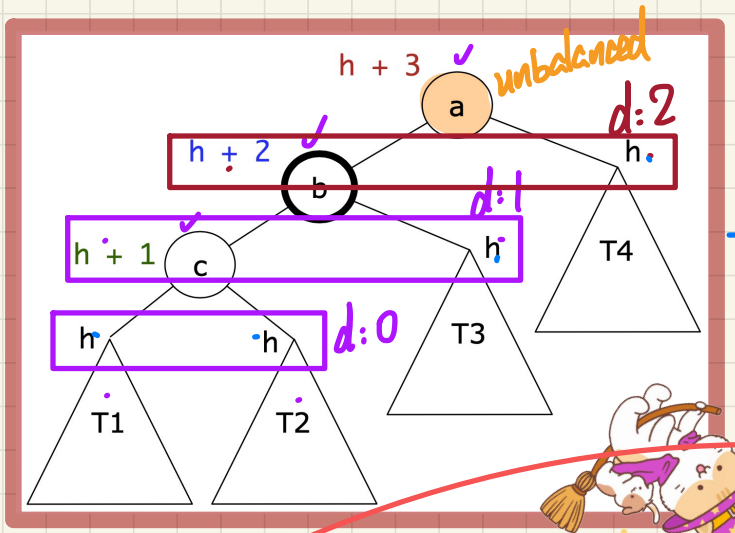
unbalanced after insertion.

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

Q. Still a balanced BST after inserting 55? NO.

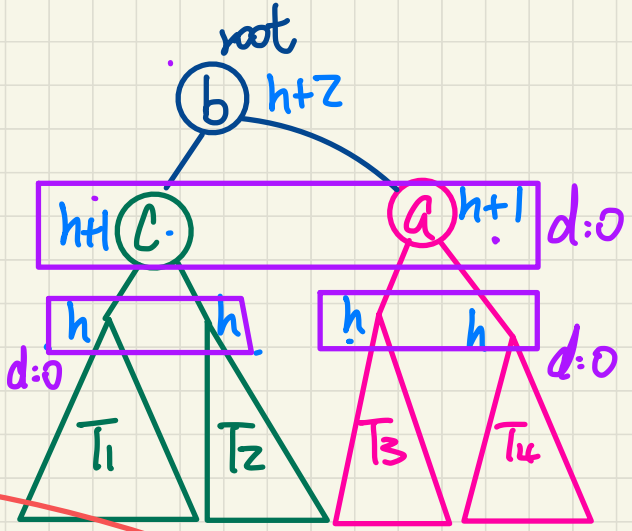
Q. Still a balanced BST after inserting 63? YES.

Restoring Balance via Rotations



Rotate middle node b to the right

same



Before Rotation, In-Order Traversal:

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

① balanced? ✓

② same content? ✓

③ BST? ✓

After Rotation, In-Order:

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

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

Q. After a **right-rotation** on node b, is the resulting tree still a **BST**?

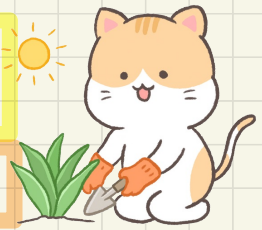
Lecture 5c

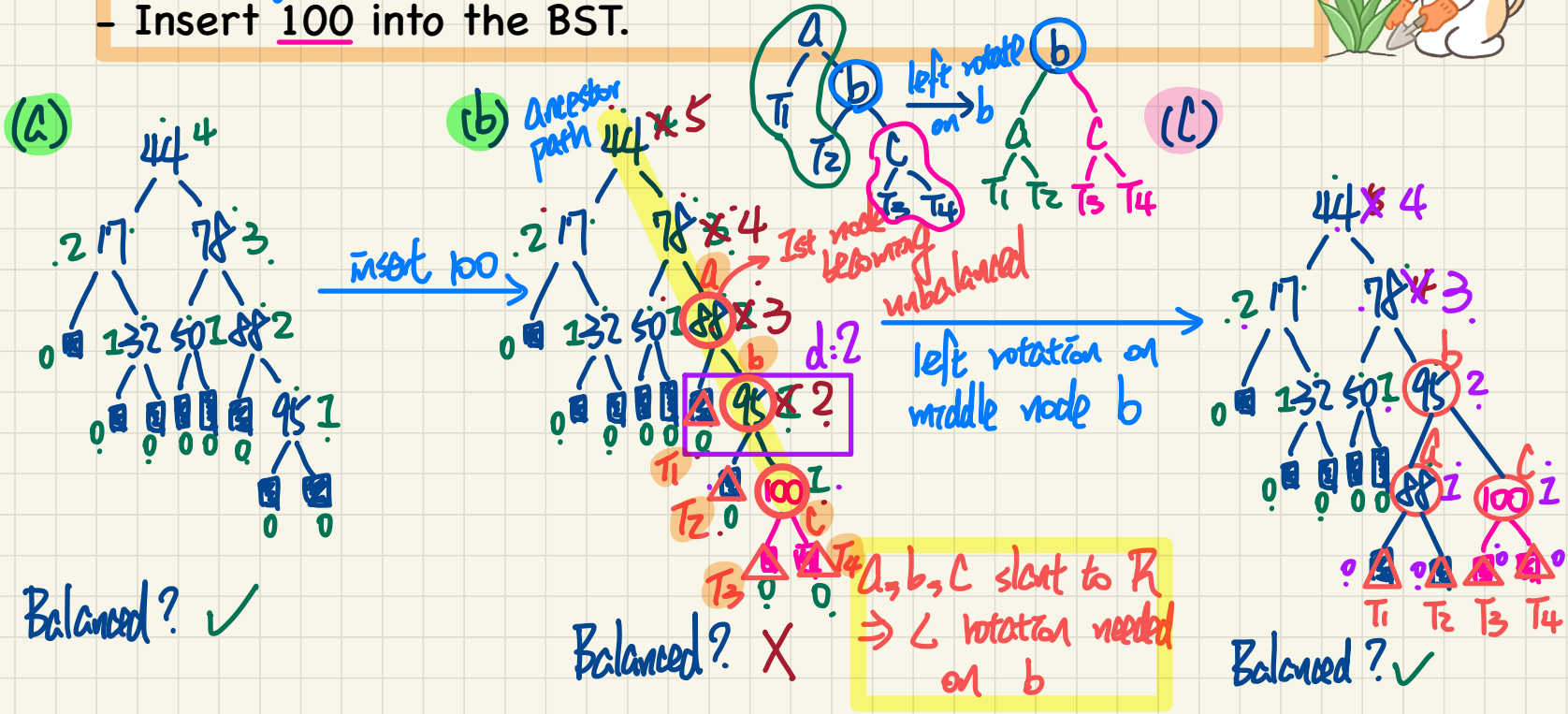
Part B

Balanced Binary Search Tree - Trinode Restructuring after Insertion

Trinode Restructuring after Insertion: Left Rotation

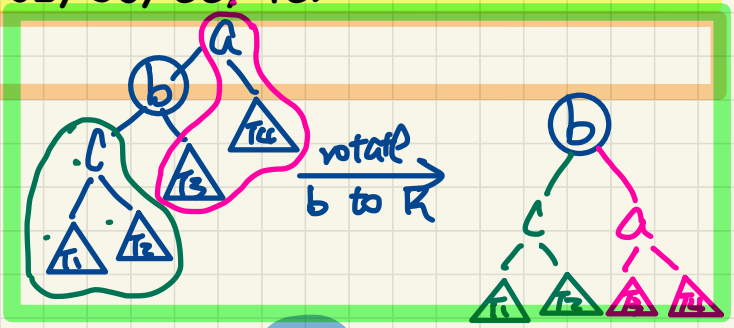
- Insert the following sequence of keys into an empty BST:
  <44, 17, 78, 32, 50, 88, 95>
- Insert 100 into the BST.



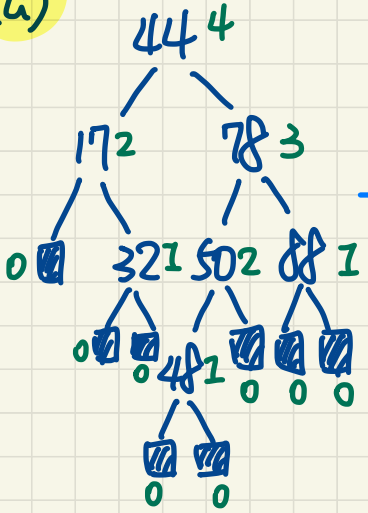


Trinode Restructuring after Insertion: Right Rotation

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

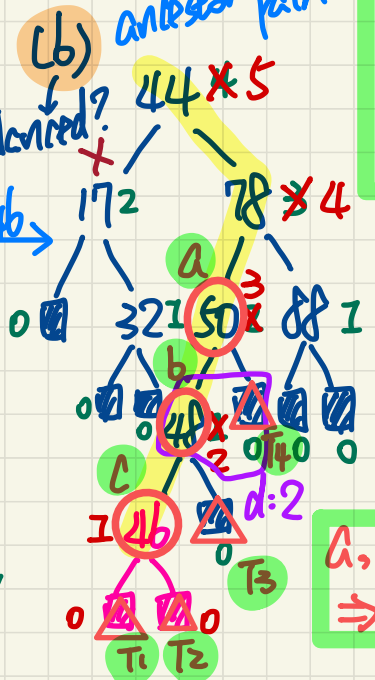


(a)



Balanced? ✓

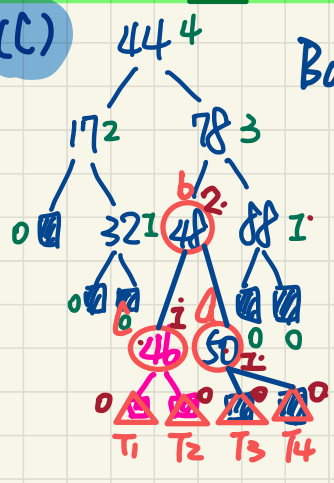
(b)



a, b, c slant to L
⇒ R rotation

rotate the middle node b to R

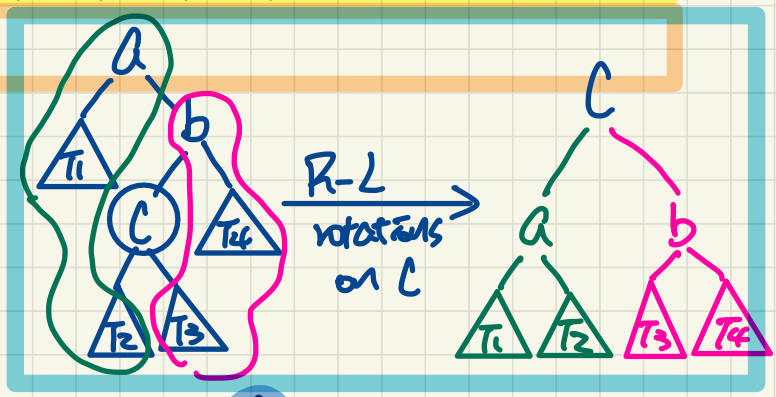
(c)



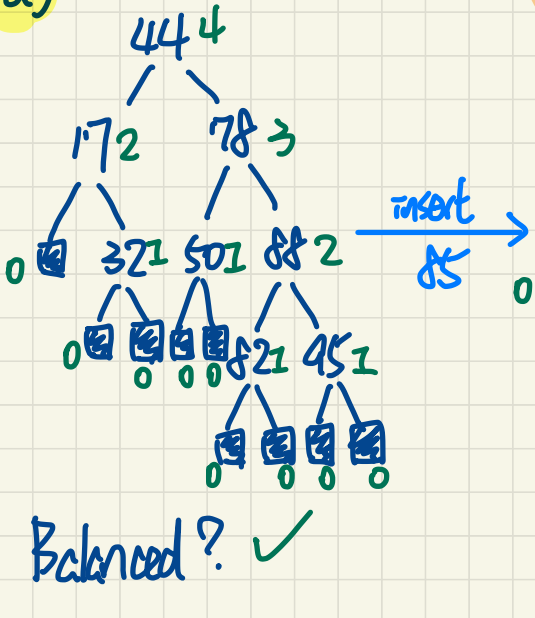
Balanced? ✓

Trinode Restructuring after Insertion: R-L Rotations

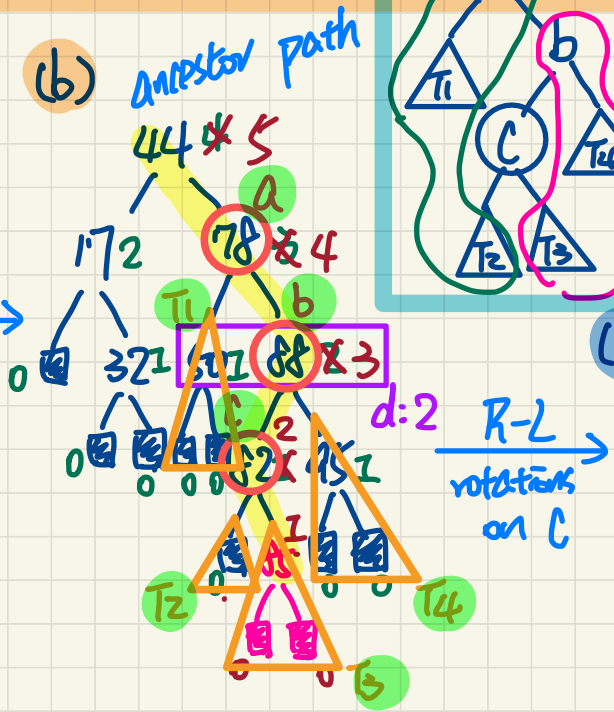
- Insert the following sequence of keys into an empty BST:
  <44, 17, 78, 32, 50, 88, 82, 95>
- Insert 85 into the BST.



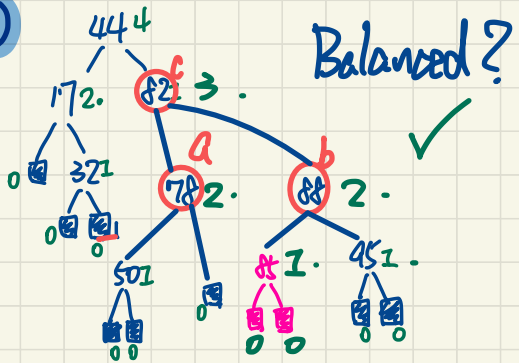
(a)



(b)



(c)



Trinode Restructuring after Insertion: L-R Rotations

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

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

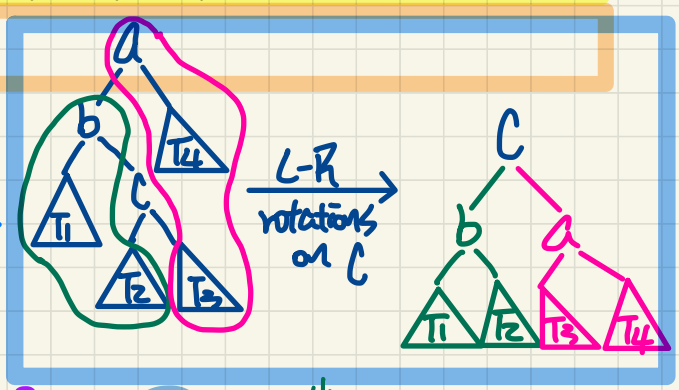
- Insert 54 into the BST.

Exercise

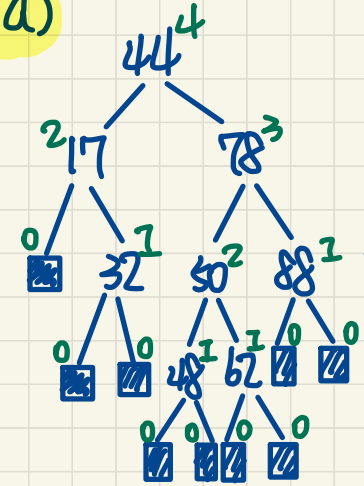
Trinode Restructuring after Insertion: L-R Rotations

Solution

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



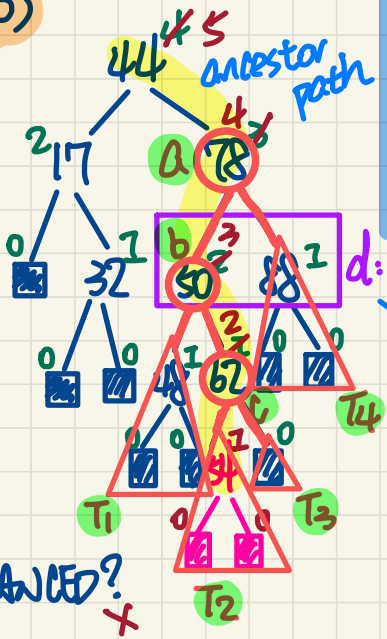
(a)



BALANCED? ✓

insert 54

(b)

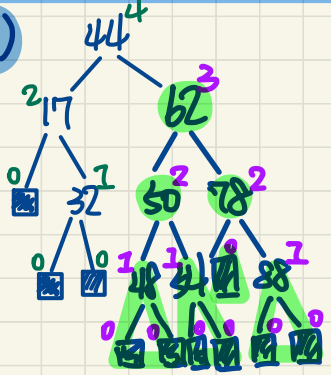


BALANCED? ✗

d:2

L-R rotations on C (62)

(c)



BALANCED? ✓

Lecture 5c

Part C

Balanced Binary Search Tree – Trinode Restructuring after Deletion

Trinode Restructuring after Deletion: Single Rotation

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

  <44, 17, 62, 32, 50, 78, 48, 54, 88>

- Delete 32 from the BST.

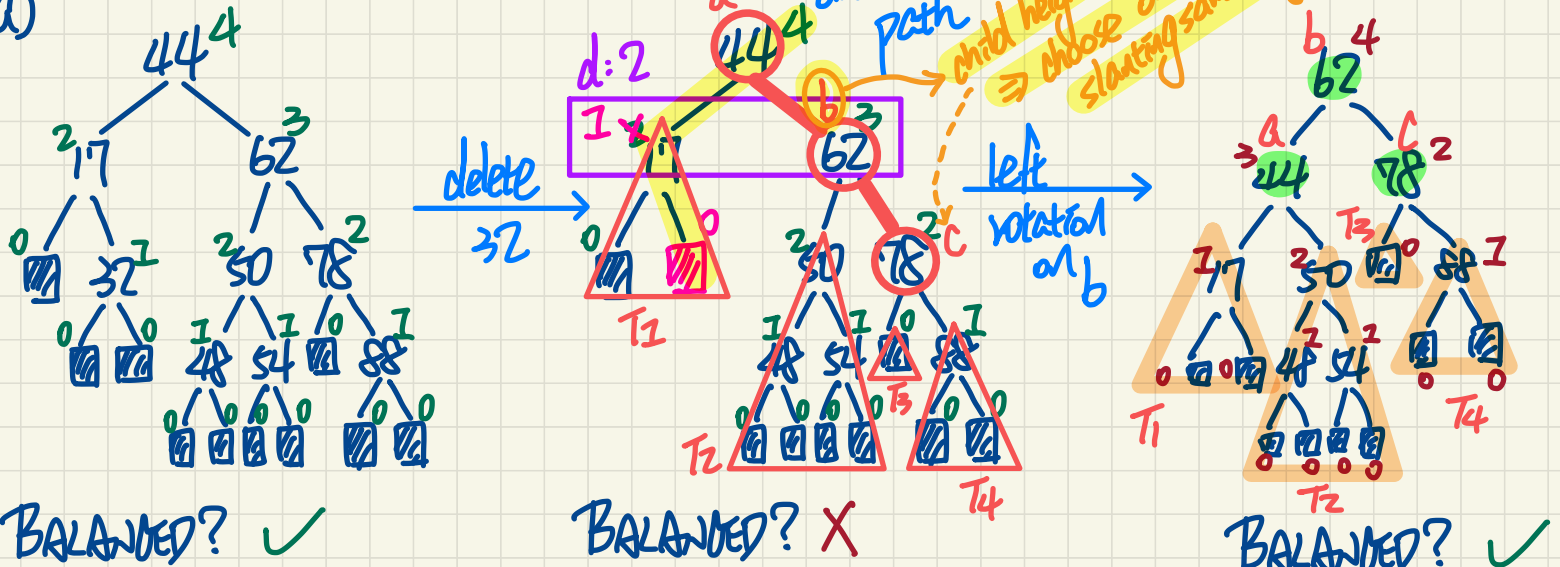
Exercise

Trinode Restructuring after Deletion: Single Rotation

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

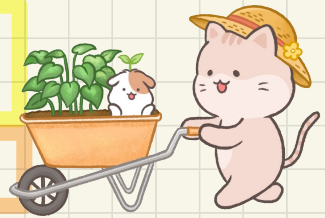
Solution

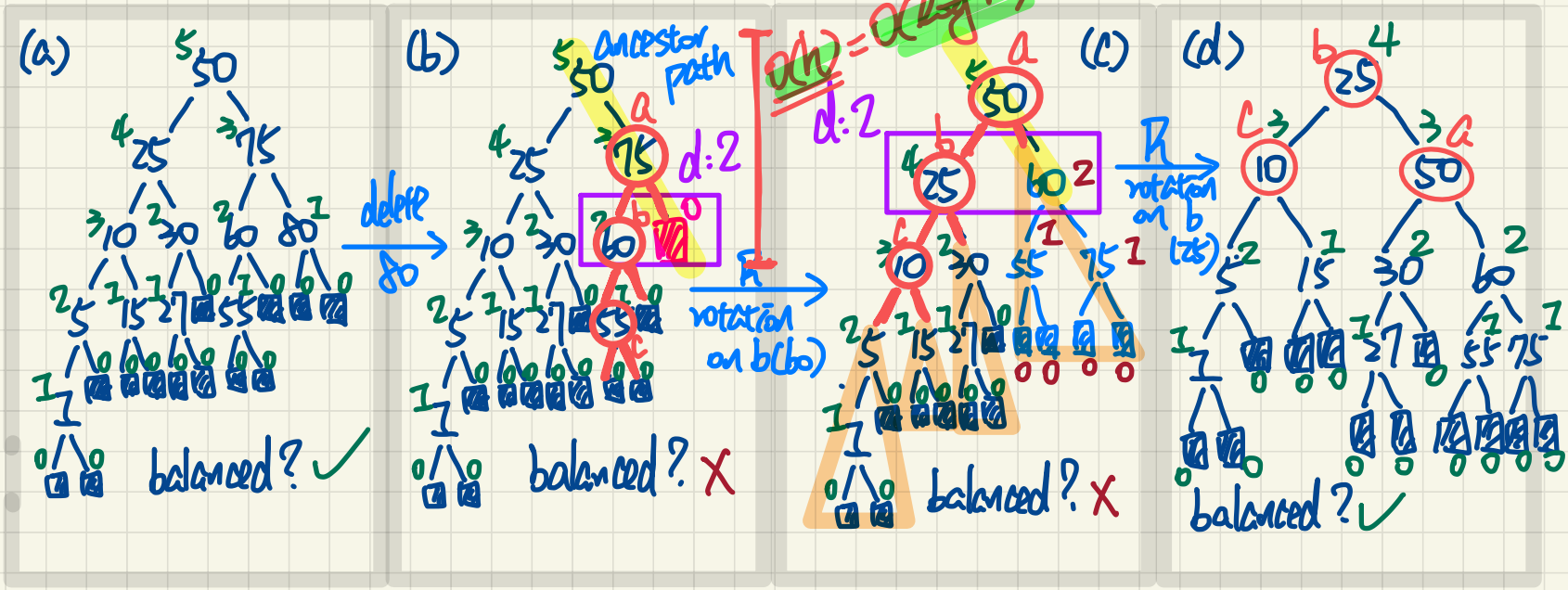
(a)



Trinode Restructuring after Deletion: Multiple Rotations

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





Lecture 4

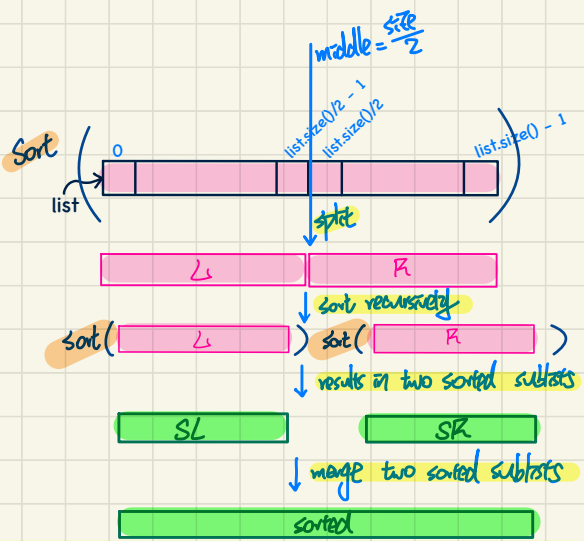
Part C

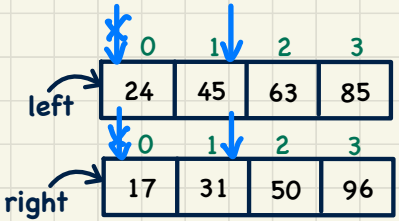
Examples on Recursion
Merge Sort
(continued)

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





Precondition

L and R sorted

```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;
}
```

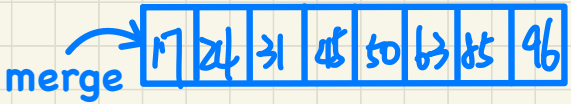
(a) # iterations: min(L.size(), R.size())

O(1)

O(1)

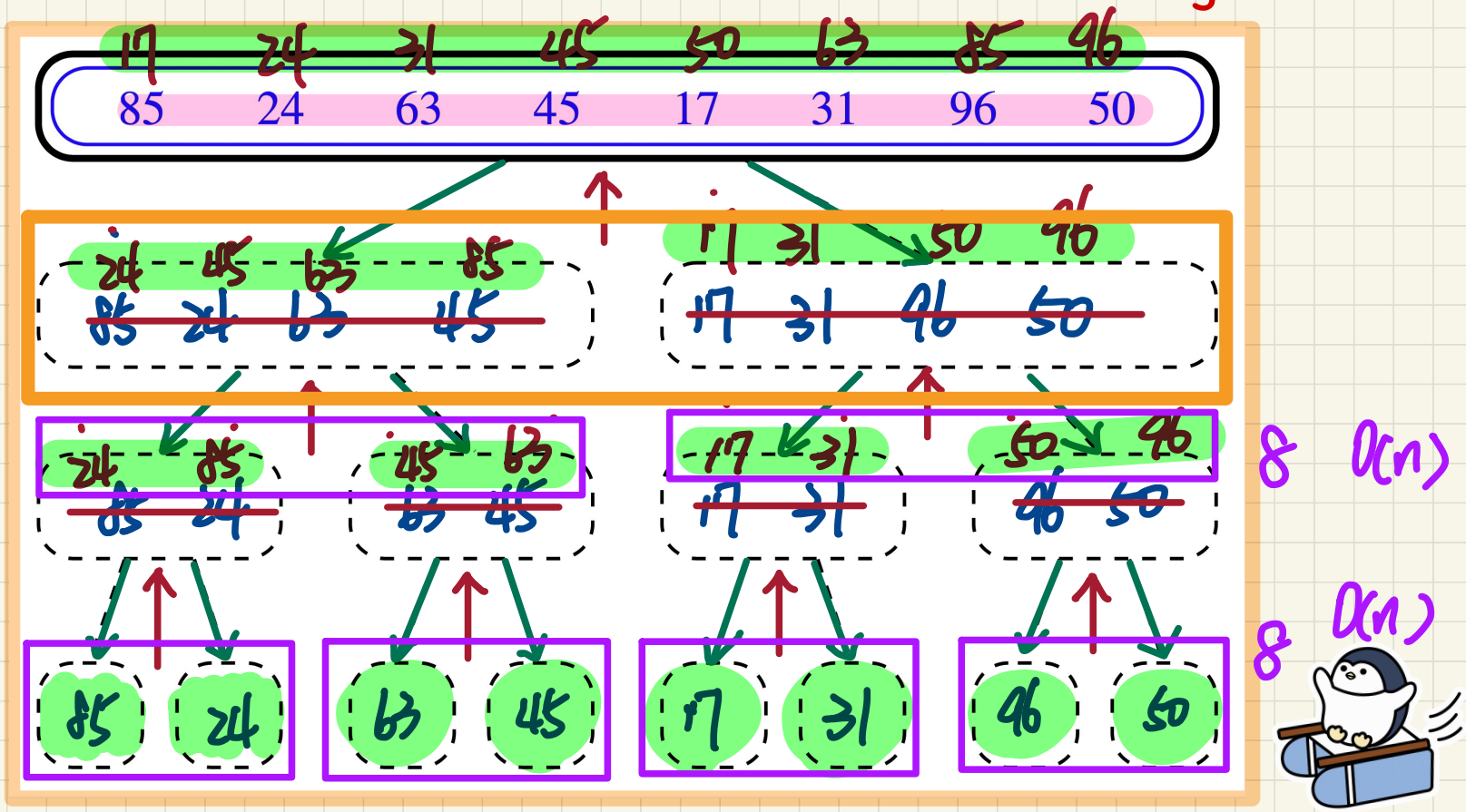
(b) # iterations: remaining # of items to loop over in the longer list

(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;
}
```

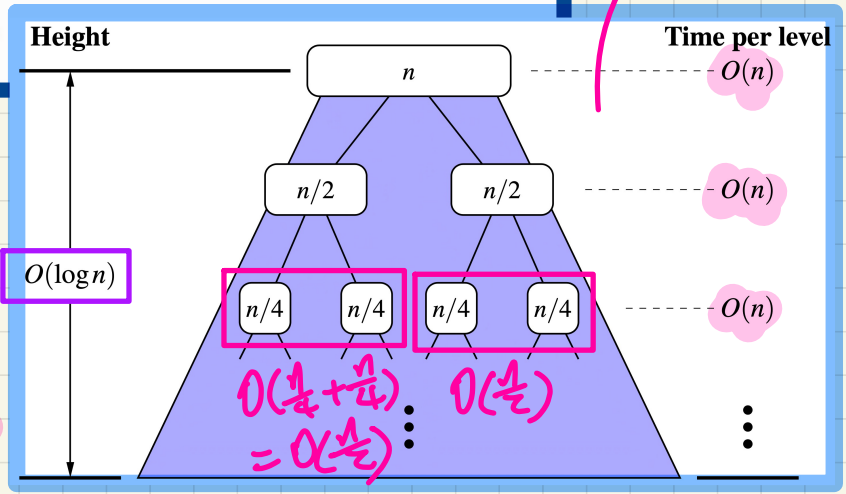
size = $\frac{n}{2}$

sort(left)
sort(right)

O(1)

O(n)

recursion tree is a full BT



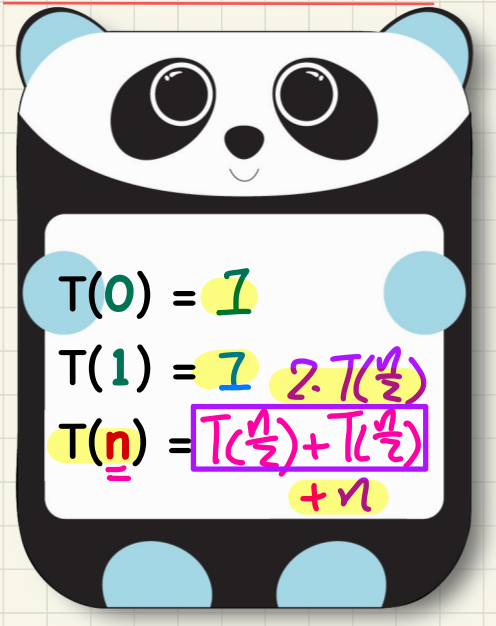
Total RT:

$O(\log n \times n)$

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

height of balanced BST

Running Time as a Recurrence Relation



Running Time: Unfolding Recurrence Relation

$T(0) = 1$

$T(1) = 1$

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

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

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

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

$\vdots$

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

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

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

$n = 8$

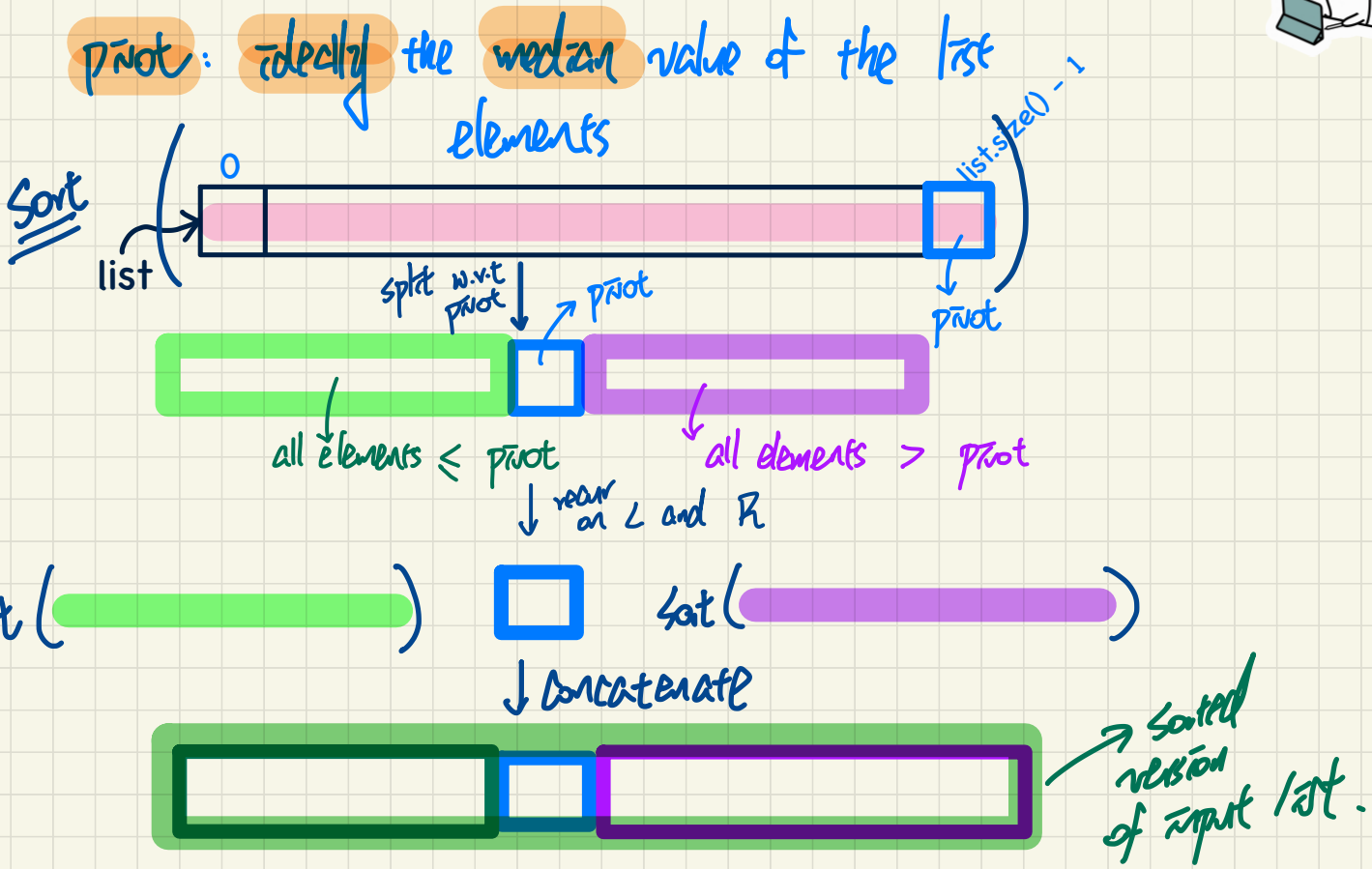
$2^{\log 8} = 8$

Lecture 4

Part D

Examples on Recursion
Quick Sort

Quick Sort: Ideas



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)

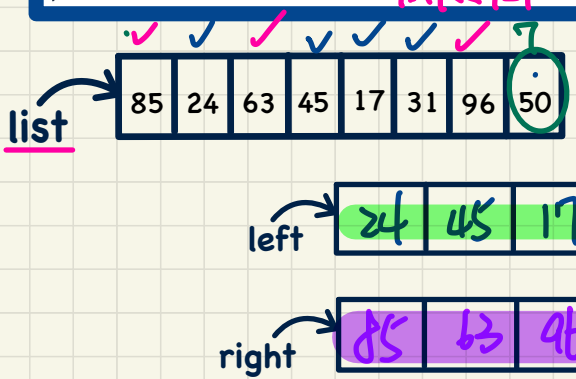
O(n)

O(n)

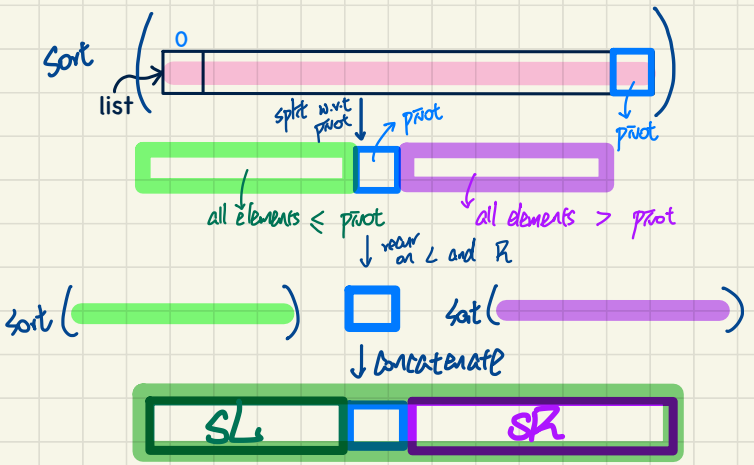
1. Best case: pivot is s.t. $|L| \approx |R|$

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

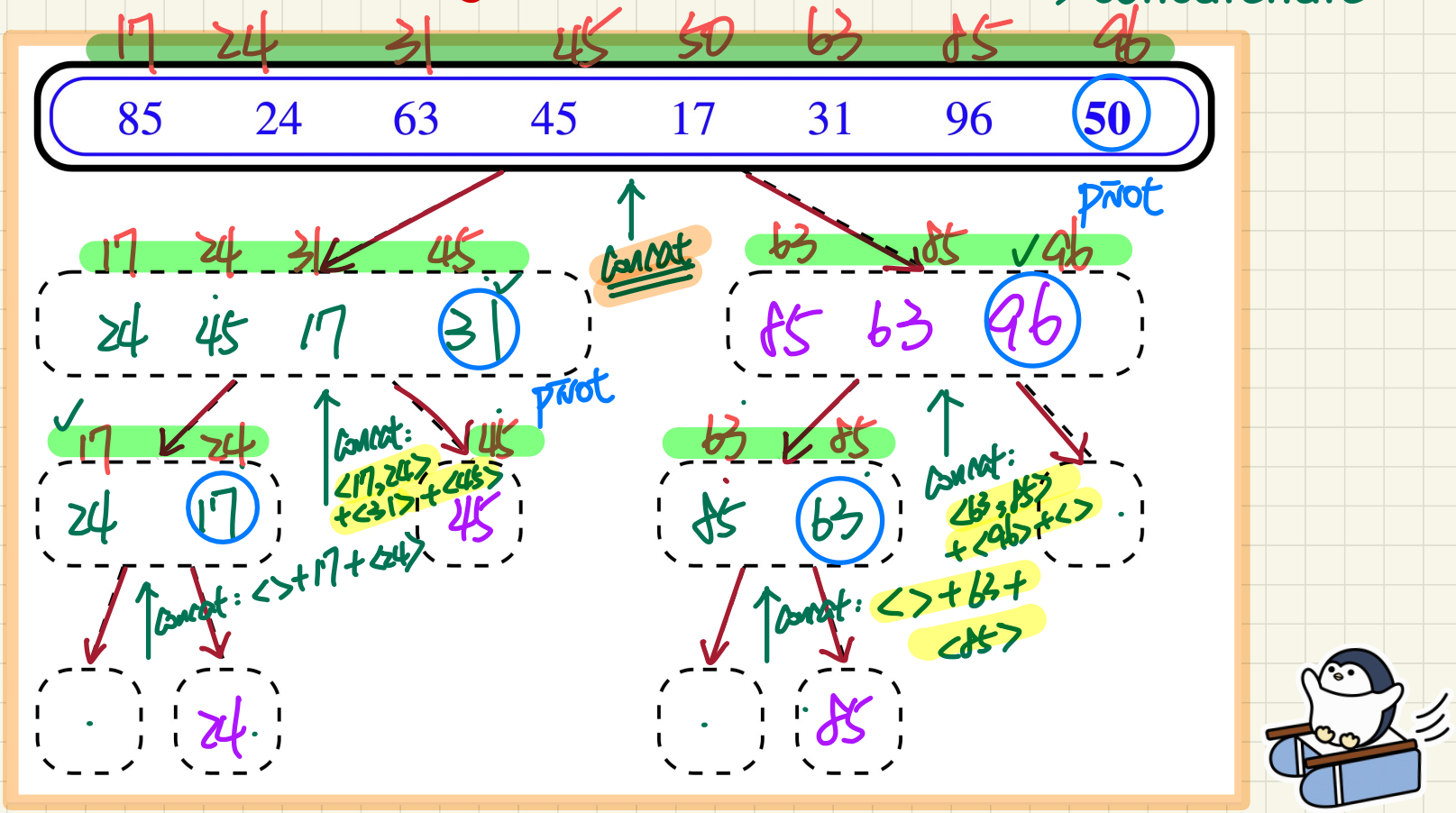
O(n)




```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;
}
```



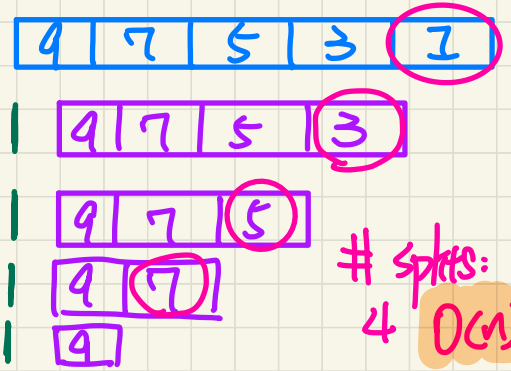
Quick Sort: Tracing

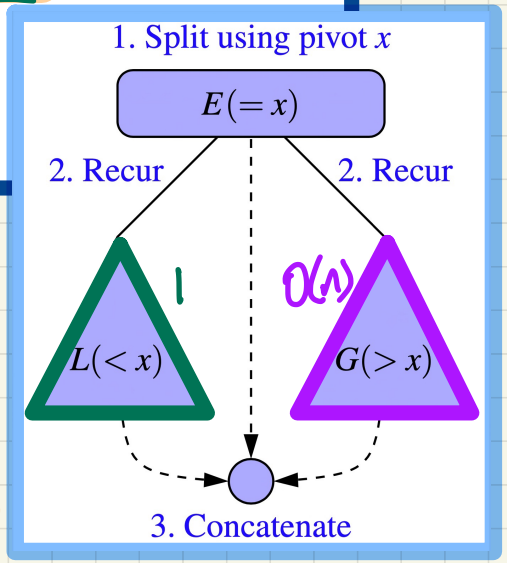


Quick Sort: Worst-Case Running Time

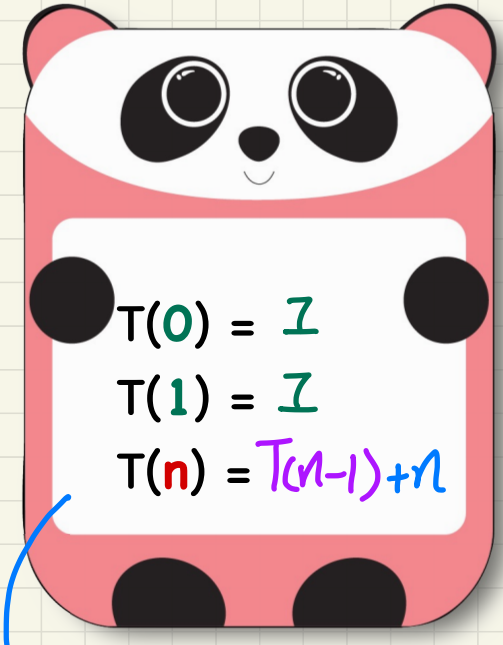
```
public List<Integer> sort(List<Integer> list) {
  List<Integer> sortedList;
  if(list.size() == 0) { sortedList = new ArrayList<>(); }
  else if(list.size() == 1) {
    sortedList = new ArrayList<>(); sortedList.add(list.get(0)); }
  else {
    int pivotIndex = list.size() - 1;
    int pivotValue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

O(n)





Running Time as a Recurrence Relation



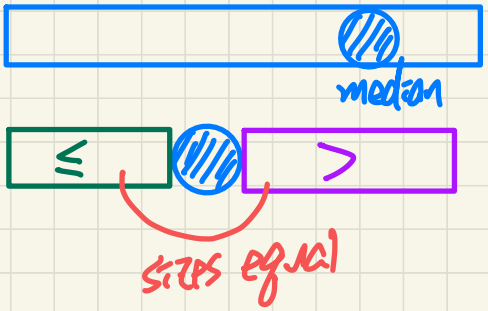
Exercise: Solve by unfolding

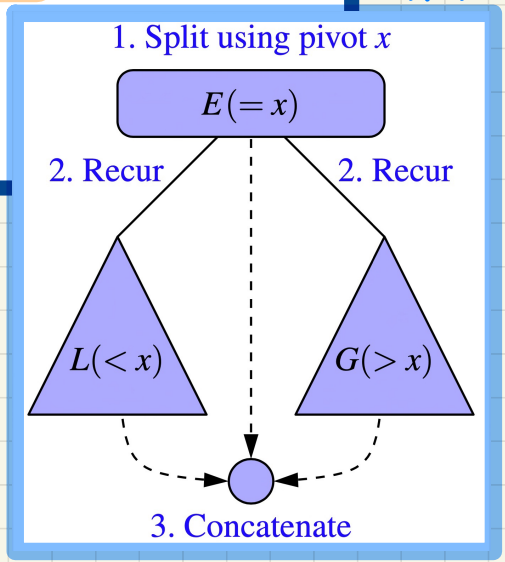
Quick Sort: Best-Case Running Time

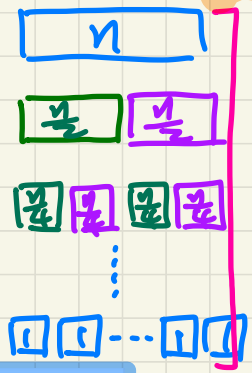
```
public List<Integer> sort(List<Integer> list) {
  List<Integer> sortedList;
  if(list.size() == 0) { sortedList = new ArrayList<>(); }
  else if(list.size() == 1) {
    sortedList = new ArrayList<>(); sortedList.add(list.get(0)); }
  else {
    int pivotIndex = list.size() - 1;
    int pivotValue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

O(1)

O(n)

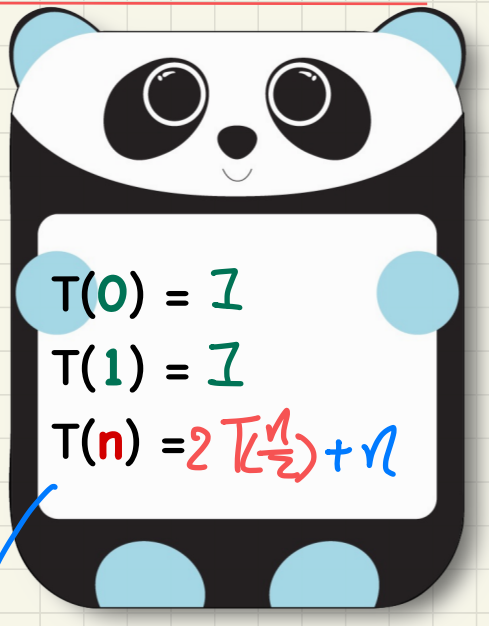






log₂n

Running Time as a Recurrence Relation



Ex.2

Exercise1: Solve by Unfolding.

$T(0) = 1$

$T(1) = 1$

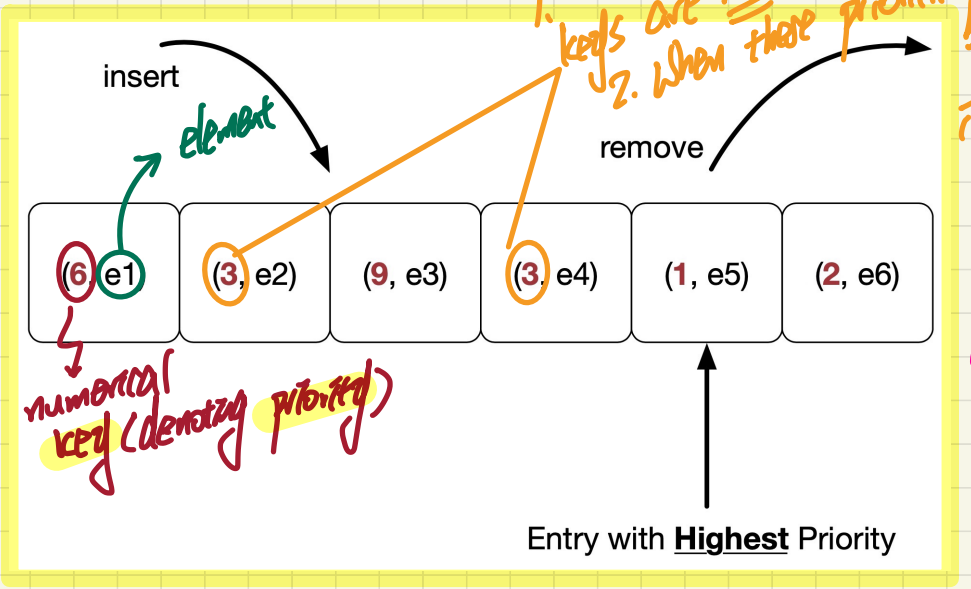
$T(n) = T(\frac{n}{10}) + T(\frac{9n}{10}) + n$

Lecture 5d

Part A

Priority Queue - Intro & List-Based Implementations

What is a Priority Queue (PQ)



1. keys are not for searching purpose because the highest, 2. when these priorities are chosen. it does not matter which entry is

Compare: 2. PQ.
1. FIFO q. remove element based on chronological order of insertion.
remove element based on priorities

1. In PQ, no notions of "front" or "end" of the q.
2. The lower the key value of an entry, the higher its priority.
   ↳ the entry with the minimum key value has the highest priority.

List-Based Implementations of Priority Queue (PQ)

PQ Method	List Method A1: Sorted List		List Method A2: 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)*

≈ insertion sort

≈ selection sort

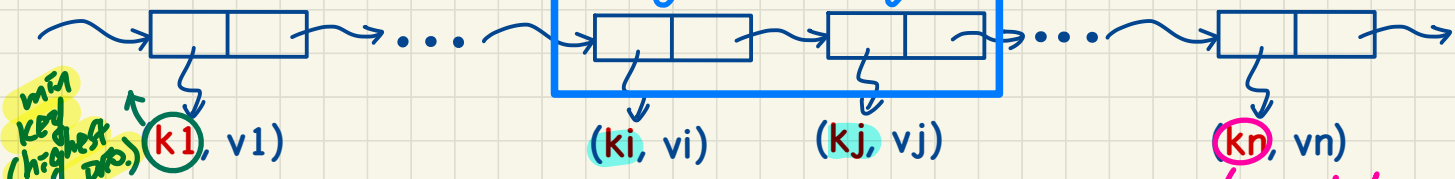
e.g. if app has more frequent calls to "min" ⇒ choose A1

Exercise

Use DLL for A1 & A2. Does it make a difference to RT?

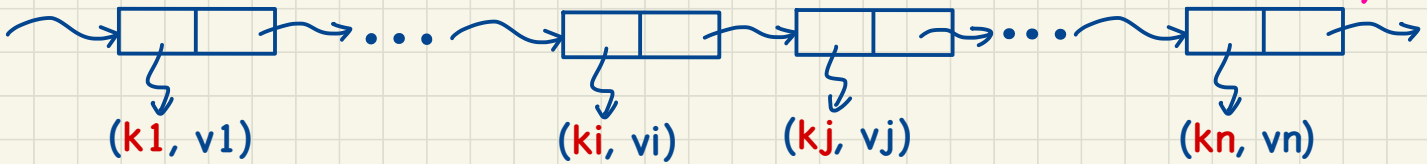
Approach 1: Sorted List

$k_i \le k_j$ (entry i "more important" than entry j)



Approach 2: Unsorted List

max key (lowest priority)



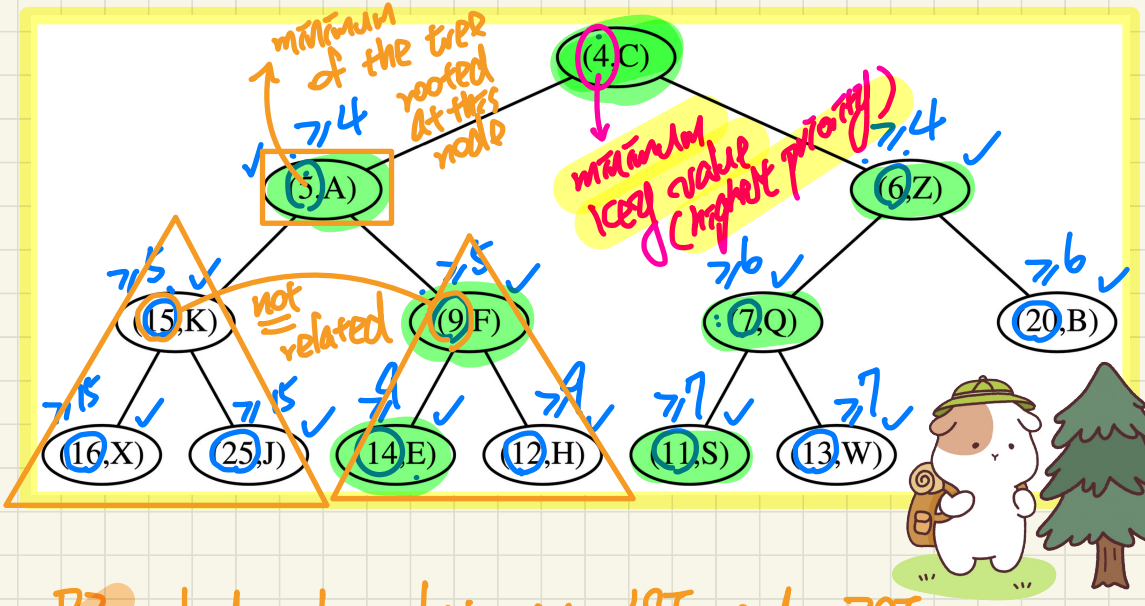
Lecture 5d

Part B

Heaps - Examples and Properties

Heaps: Relational Properties of Keys

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



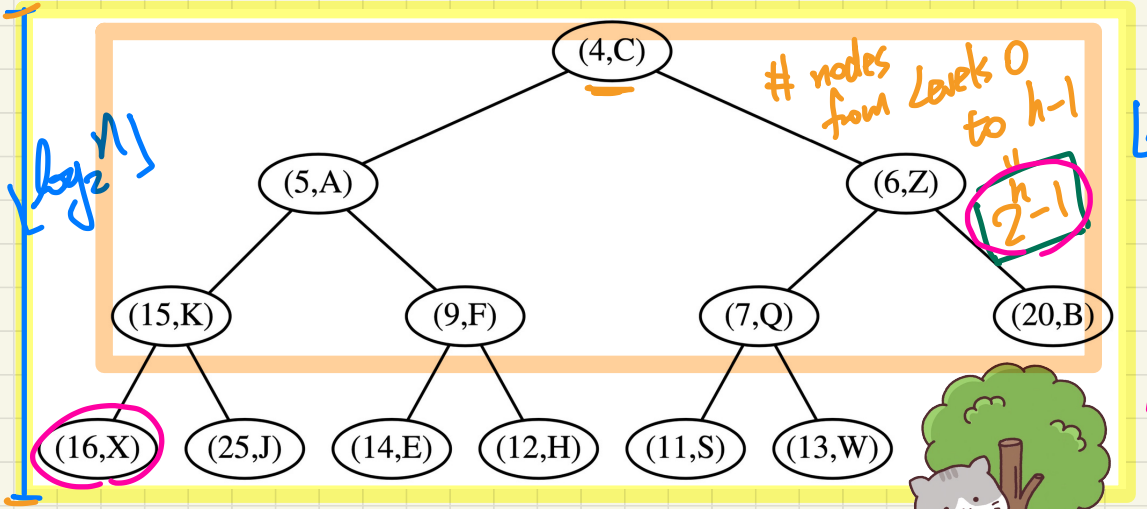
P1. Any leaf-to-root path has a sorted seq of keys.

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



$n = 13$

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

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

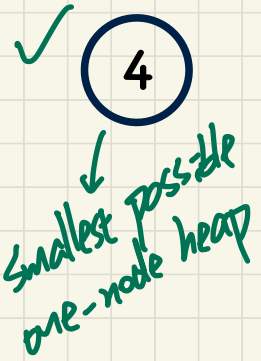
Max # of nodes: $(2^h-1)+2^h = 2^{h+1}-1$

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

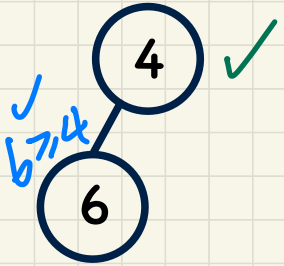
Example Heaps

< relational / structural

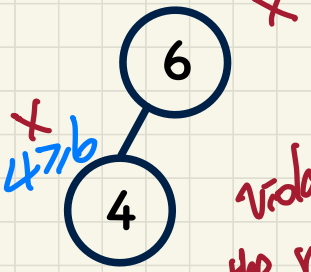
Example 1



Example 2

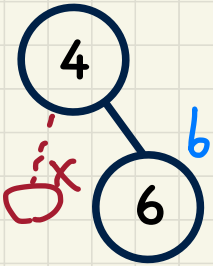


Example 3



Violating the relational property

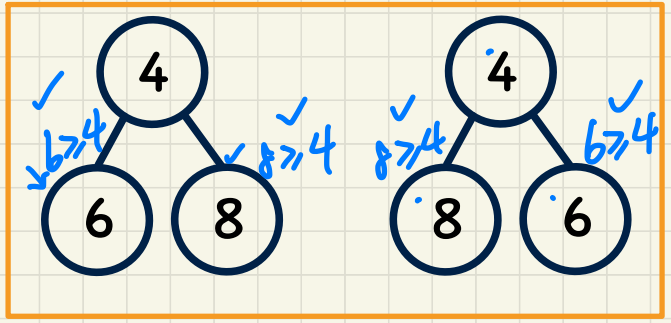
Example 4



Violating structural property

Example 5

full BTs ⇒ complete BTs



Example 6

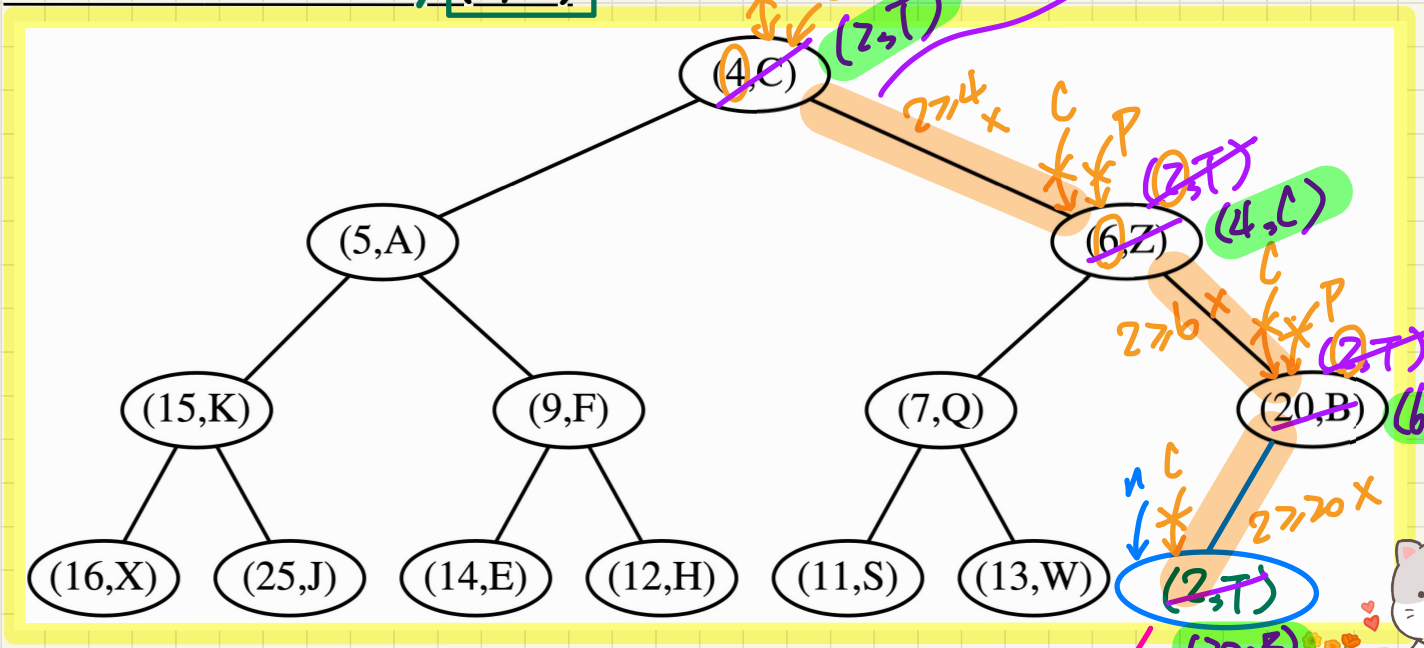
Lecture 5d

Part C

Heaps - Insertions

Heap Operations: Insertion

Insert a new entry (2, T)



Lecture 5d

Part D

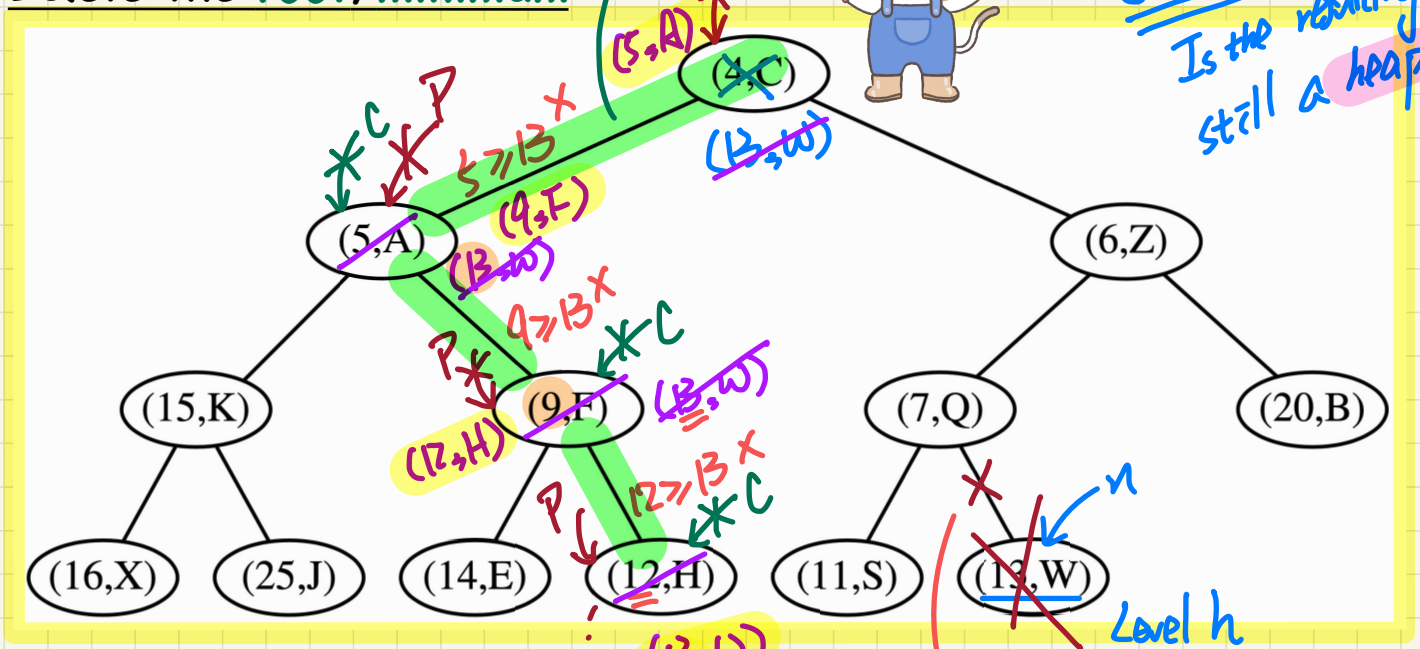
Heaps - Deletions

Heap Operations: Deletion

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

Delete the root/minimum

Exercise
Is the resulting tree still a heap?



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

Lecture 5d

Part E

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***.

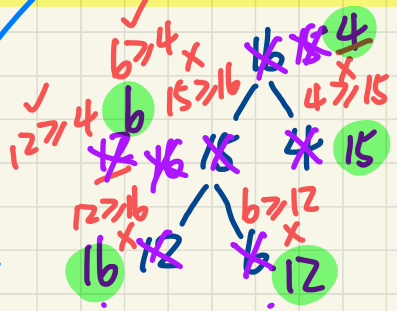
Exercise: Build a heap out of the following 15 keys:

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

first inserted to level 2

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

first inserted to level 1



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

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

$O(n \cdot \log n)$

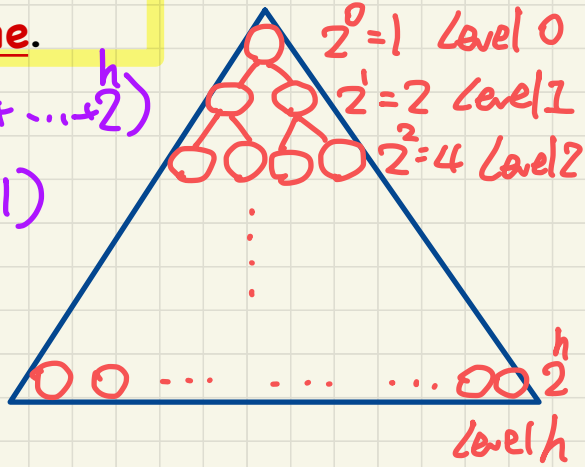
RI: # nodes at level i × # up-heap bubbling steps

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

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

$+ \dots$

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



Exercise: Complete inserting the remaining keys to the heap.

Lecture 5d

Part F

Heaps - Bottom-Up Heap Construction

Bottom-Up Heap Construction

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

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

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

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

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

  ...

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

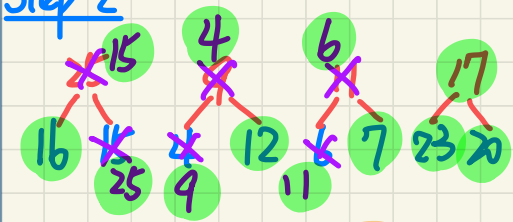
  $\therefore \frac{N+1}{2^{h+1}} = 1$ heap, each with height h and size $2^{h+1} - 1$ constructed.

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

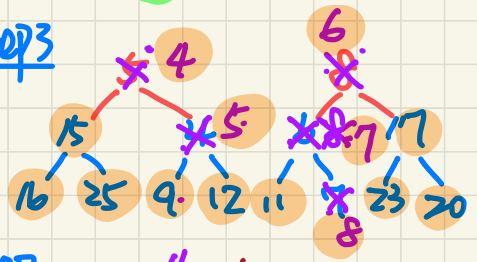
Assumption: Key values supplied **all at once**.

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

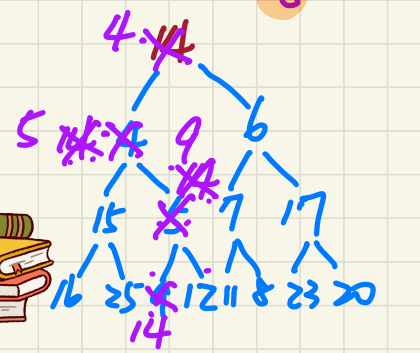
Step 2
25%



Step 3
12.5%



Step



Lecture 5d

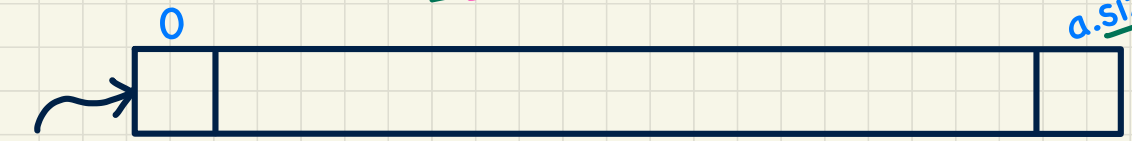
Part G

Heaps - Heap Sort Algorithm

Heap Sort: Ideas

$O(N \cdot \log N)$

N entries



Construct a heap out of N entries

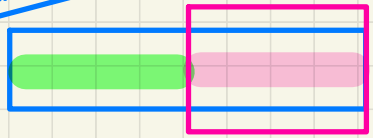
(A) Top-Down

$O(N \cdot \log N)$

(B) Bottom-Up

$O(N)$

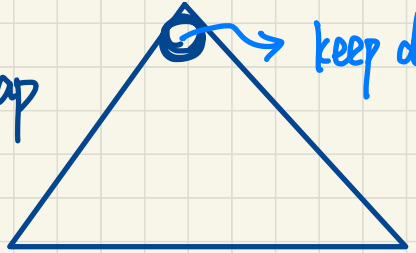
SelectionSort



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

$\approx$ Selection Sort

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



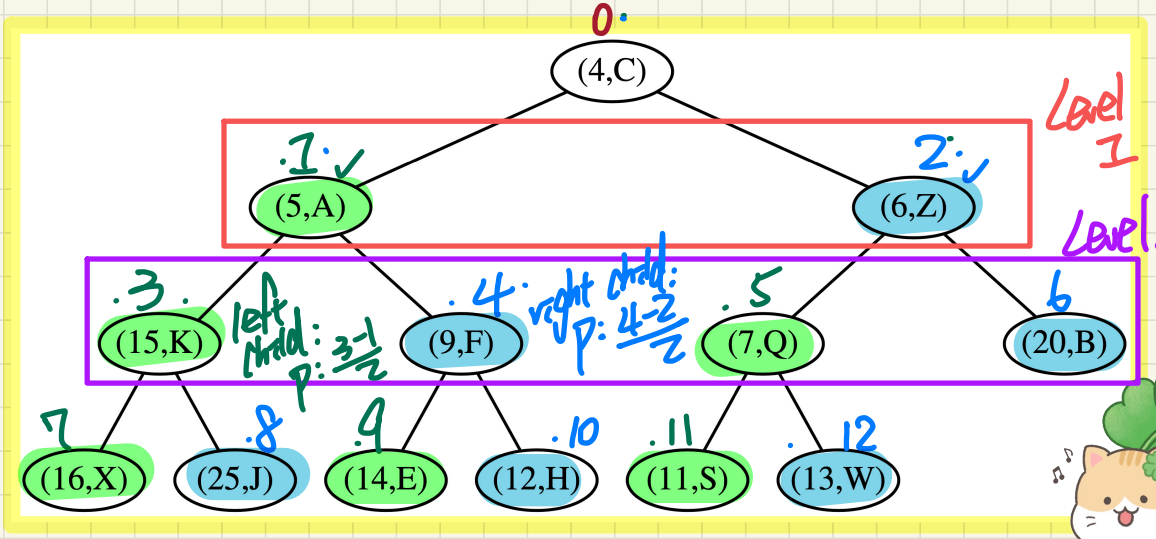
keep deleting the root until the heap is empty. N deletions, each $O(\log N) \Rightarrow O(N \cdot \log N)$

Lecture 5d

Part H

Heaps - Array-Based Implementation

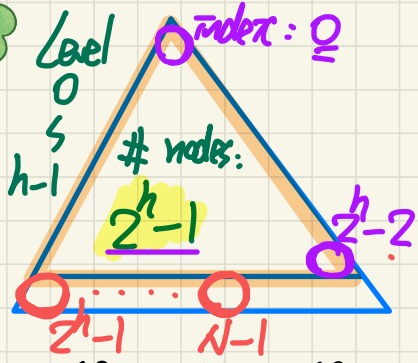
Array-Based Representation of a Complete BT



Exercise

What if the BT is not complete?

(bad for space util.)



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

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 !