# Lecture 3

## Part D

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

size of queue vs. size of array

% modulo

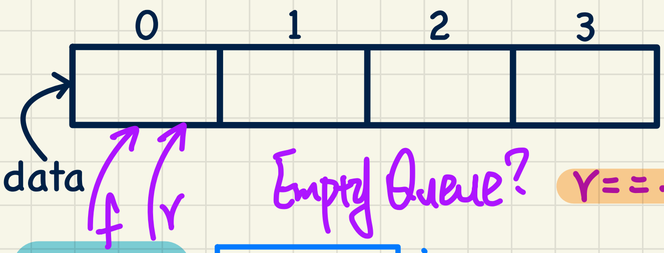
1. fix-sized (no resizing)

# Implementing the Queue ADT using a Circular Array

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

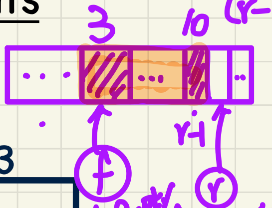
## Phase 1: enqueue 3 elements

data[r] = item;
r++;

f Queue Full? $\overset{f\ 0}{\overset{\shortparallel}{(\underset{3}{r}+1)\%\underset{4}{N}}}$ r.

when r points to is the only empty slot.

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



# empty slots before
# slots before this index

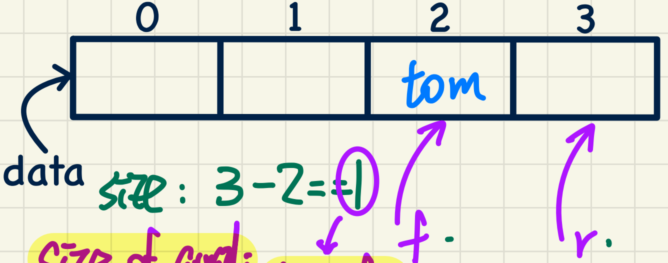
size : r>f
↳ r-f

③
||
1+(4-2)

size : r<f
↳ r+(N-f)

## Phase 2: dequeue 2 times

data[f]=null; f++;



size : 3-2=1

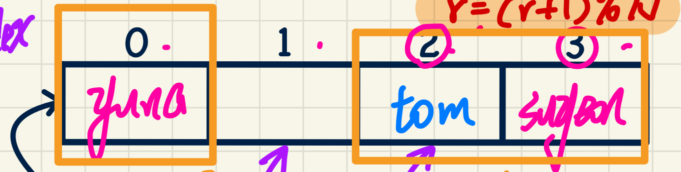
size of array: 4 (fixed)

size of q: 1.

$\frac{(3+1)\%4}{0}$

## Phase 3: enqueue 2 elements

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



[0, r-1]
↓
(r-1)-0+1
= r

r

f

[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

worst-case: $O(n)$

```
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;
  }
}
```

$O(n)$

$O(1)$

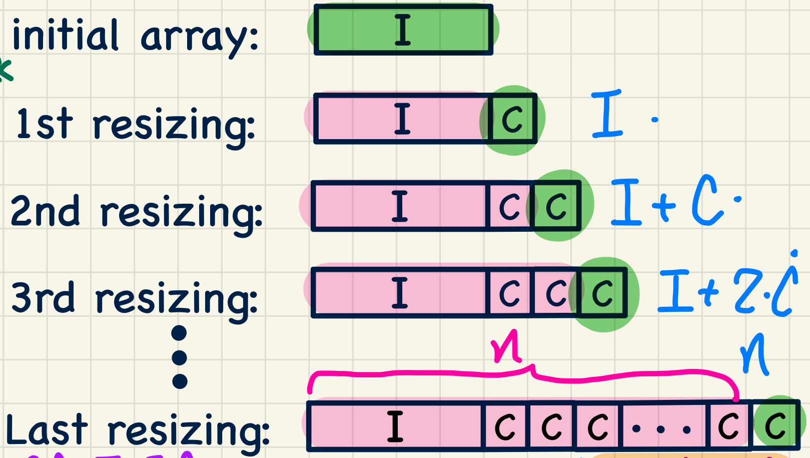
this only occurs once in a while

Sum of Arith. Seq. $a_1 + a_2 + \dots + a_k$

$= \frac{(a_1 + a_k)\cdot k}{2}$

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

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



initial array:

1st resizing:

2nd resizing:

3rd resizing:

Last resizing:

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

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

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

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

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

$O(n^2)$ total RT

**Amortized/ Average RT:**

$O(\frac{n^2}{n}) = \boxed{O(n)}$

**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

$$S_k = \underbrace{\overset{I\cdot r^0}{\boxed{I}}}_{1st} + \underbrace{I\cdot r}_{2nd} + \underbrace{I\cdot r^2}_{3rd} + \underbrace{I\cdot r^3}_{4th} + \cdots + \underbrace{I\cdot r^{k-1}}_{kth}$$

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

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

Avg: total RT / # op.

P1 $\boxed{2^x = y \Rightarrow x = \log y}$ $\boxed{2^3 = 8 \Rightarrow 3 = \log 8}$

# Amortized Analysis: Dynamic Array with Doubling

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

```
public class ArrayStack<E> implements Stack<E> {
  private int I;
  private int capacity;
  private E[] data;
  public ArrayStack() {
    I = 1000; /* arbitrary initial size */
    capacity = I;
    data = (E[]) new Object[capacity];
    t = -1;
  }
  public void push(E e) {
    if (size() == capacity) {
      /* resizing by doubling */
      E[] temp = (E[]) new Object[capacity * 2];
      for(int i = 0; i < capacity; i ++) {
        temp[i] = data[i];
      }
      data = temp;
      capacity = capacity * 2
    }
    t++;
    data[t] = e;
  }
}
```

resizing worst: O(n)

O(n)

O(1)

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

P3 $\boxed{2^{\log x} = x}$

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

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

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

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

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

initial array: I

1st resizing: I I  I

2nd resizing: I I I I  2·I

⋮

Last resizing: $\underbrace{\text{I I} \cdots \text{I I}}_{n}\underbrace{\text{I I} \cdots \text{I I}}_{n}$ n

$$\text{Total RT} = \underset{\text{1st}}{\overset{2^0 \cdot I}{I}} + \underset{\text{2nd}}{2 \cdot I} + \underset{\text{3rd}}{2^2 \cdot I} + \cdots + \underset{\text{kth}}{\overset{2^{k-1} \cdot I}{n}}*$$

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

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

**Amortized/ Average RT:**

$O(\frac{n}{n}) = \boxed{O(1)}$

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

and the last push triggers the last **resizing** routine. O(n)

## push/enqueue

**Exercise** ArrayList

↳ implemented by ?

| | const. increments | doubling |
|---|---|---|
| Worst-case RT | $O(n)$ | $O(n)$ |
| Amortised/avg. RT | $O(n)$ | $O(1)$ |

due to that doubling the size each time makes it substantially less frequent to resize

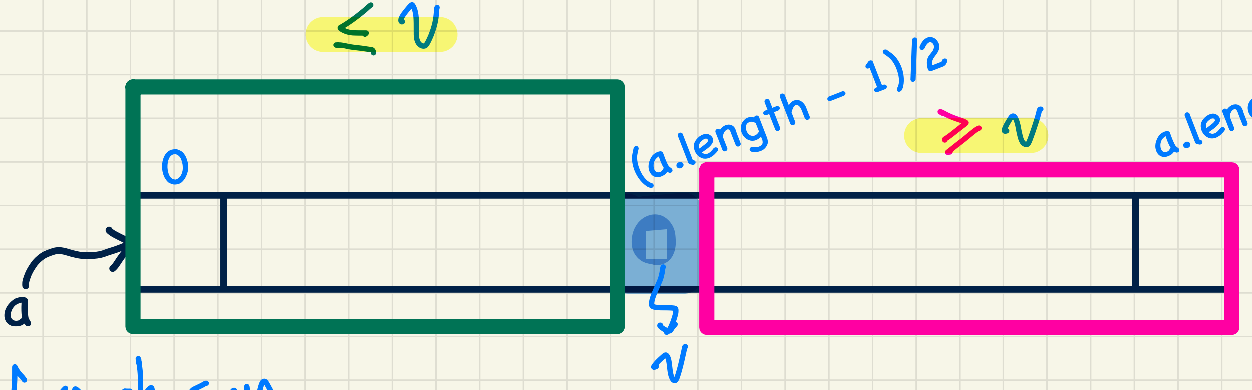
# Lecture 4

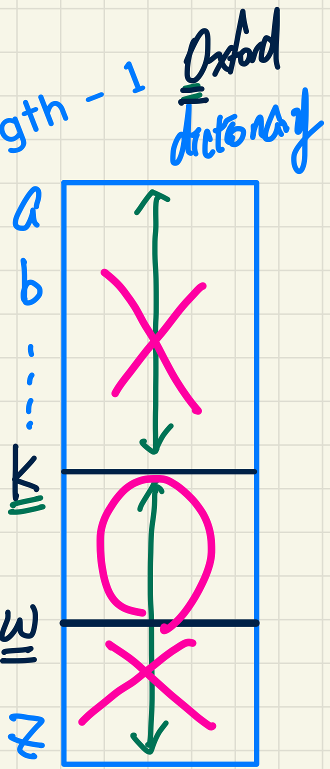
## Part B

## *Examples on Recursion*
## *Binary Search*

# Binary Search: Ideas

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





size of search space

n

$\frac{n}{2}$

$\frac{n}{4}$

⋮

1

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

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

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

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

# Binary Search in Java

```
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

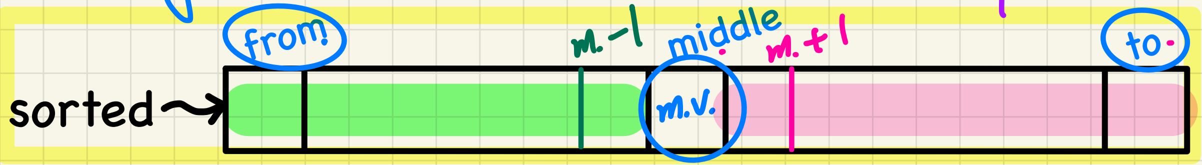
call by value

define the range of indices of the search space.

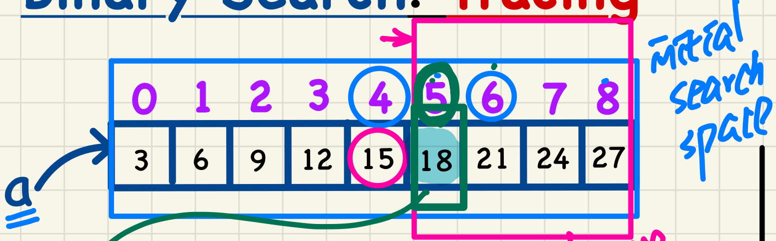
recursive case

key == middleValue

narrowed search spaces represent a strictly smaller problem to solve.



# Binary Search: Tracing



search space of 1st recursion

search space of 2nd recursion

search(a,18) key

binarySearchH(a,0,8,18)

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

m.v. a[4] == 15

7

m. +1

to.

binarySearchH(a,5,8,18)

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

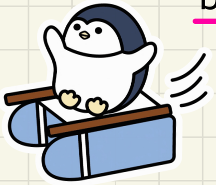
m.v. a[6] = 21

binarySearchH(a,5,5,18)

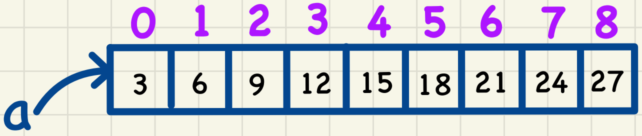
from

m.-1

↳ true



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

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

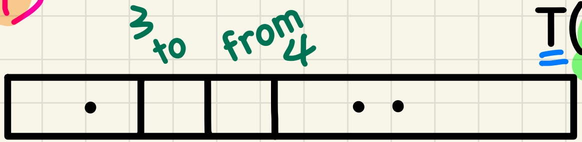
$O(1)$

$O(1)$

$O(1)$

subproblem of size $n-1$

**Base Case:**
**Empty Array**

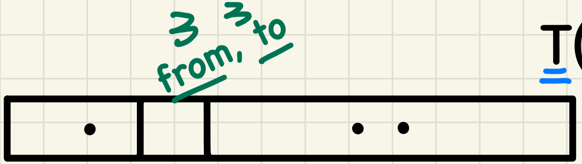


$T(0) = 1$

$[4, 3]$ has 0 numbers

---

**Base Case:**
**Array of Size 1**

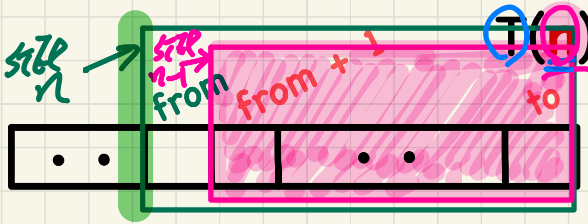


$T(1) = 1$

$[3, 3]$ has 1 number

---

**Recursive Case:**
**Array of size > 1**



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

strictly smaller problem

# Running Time: Unfolding Recurrence Relation

$T(0) = 1$

$T(1) = 1$

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

→ recurrence relation derived from Java imp. of recursive algorithm.

$$T(n) = T(\underline{n-1}) + 1$$

$T(n-1)$

$$= \boxed{T(\underline{(n-1)-1}) + 1} + 1$$

$n-2$

$T(n-2)$

$$= \boxed{T(\underline{(n-2)-1}) + 1} + 1 + 1$$

$n-3$

$$= \ldots \quad n-(n-1)$$

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

$1$

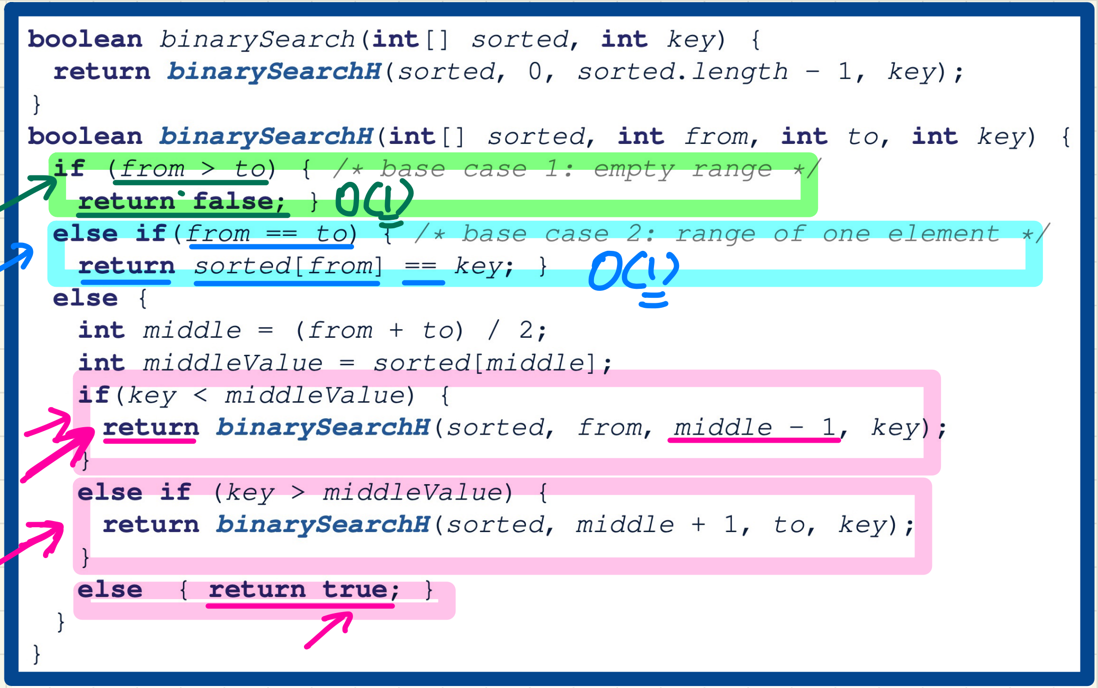
How many?

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

$$= n$$

$O(n)$

# Binary Search: Running Time

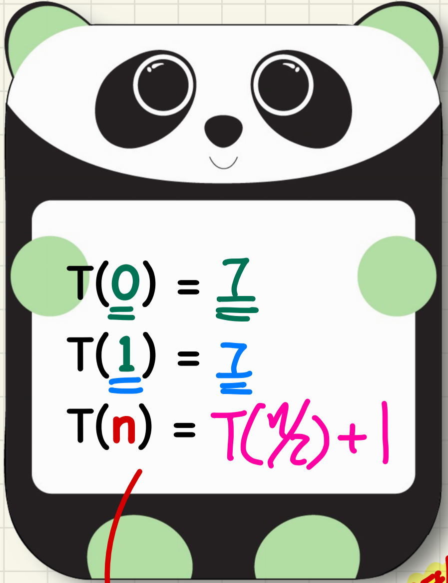


```
boolean binarySearch(int[] sorted, int key) {
  return binarySearchH(sorted, 0, sorted.length - 1, key);
}
boolean binarySearchH(int[] sorted, int from, int to, int key) {
  if (from > to) { /* base case 1: empty range */
    return false; }
  else if(from == to) { /* base case 2: range of one element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

$O(1)$

$O(1)$

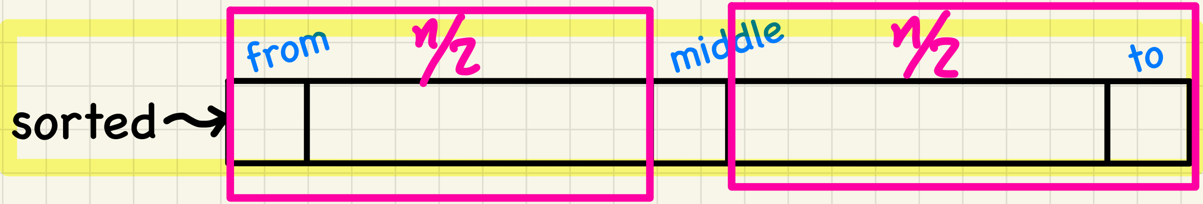
## Running Time as a Recurrence Relation



$T(0) = 1$

$T(1) = 1$

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



wrong:

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

X !! either L or R but not both

# Running Time: Unfolding Recurrence Relation

$T(0) = 1$

$T(1) = 1$

once reaching here, no more unfoldings

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

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

↳ without loss of generality.

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

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

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

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

$$\frac{n}{2^1} \quad = \left(T\left(\boxed{\frac{n}{4}}\right) + 1\right) + 1$$

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

$$\frac{n}{2^2} \quad = \left(T\left(\boxed{\frac{n}{8}}\right) + 1\right) + 1 + 1$$

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

$$\frac{n}{2^3} \quad = \left(T\left(\frac{n}{16}\right) + 1\right) + 1 + 1 + 1$$

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

$\vdots$

$$= T(\boxed{1}) \ \underline{+1+1 \cdots +1}$$

How many? $\log n$

$O(\log n)$