PG4200: Algorithms And Data Structures

Lesson 02: Generics, Stacks and Queues

Prof. Andrea Arcuri

Generics

Data Types

- In Java (and other statically typed languages) you need to declare the *type* of the variable
    - eg, “*int x*” or “*String y*”
- In collections (arrays, lists, queues, stacks, etc.) you store data, but of which type?

Example

- *StringContainer*: to store strings
- *IntegerContainer*: to store integers
- *WebSocketContainer*: to store web socket objects
- *SongContainer*: to store song objects
- *ShopCartContainer*: to store items in a shop cart
- etc.
- Do you see the problem here?

Polymorphism?

- Issue: would need a different implementation for each container for each possible type class ever
- What about using a *ObjectContainer* to store *java.lang.Object* instances?
- In Java, all objects have *Object* class as ancestor, so could add any type due to polymorphism
    - e.g., can add *String* and *Song* in same *ObjectContainer*
- Problem: yes, we can insert anything, but what would we read back is *Object*, and not *String* or *Song*

//Add: String “foo”, Integer 5

container.add(“foo”);

container.add(5);

ObjectContainer

add(Object x)

[0]	“foo”
[1]	5

Object x = container.get(0);

//we do not know if String or

//something else

Java Generics <T>

- *List<T>*: define a *generic* type, which can be substituted with any type
    - note: “T” is just a label, could be anything
- Eg. *List<String>, List<Integer>, List<Song>*
- If I am only storing a variable (e.g., in a class field or array), I do need to care of its type, as not going to call any method on it
    - eg, “*T x = input;*” do not need to care of actual type of T, as long as *input* is of that type

<T extends Foo>

- In some cases you need Generics, but still need to call methods on it
- With *<T>* you would only be allowed to call methods from *java.lang.Objects*
- *<T extends Foo>* means any type that extends/implements the class/interface *Foo*
- Note: there is also a *<T super Foo>*, but we will not need it

Primitive Types

- Given a generic **List<T>**, then we cannot instantiate with **int**, eg, **List<int>** does not compile

- **int** is a primitive type, and NOT an object extending *java.lang.Object*
    - others: **double**, **float**, **long**, **char**, **boolean**, etc.

- For each primitive type, Java provides an object wrapper, eg **Integer** for **int**
    - so can have **List<Integer>**

- Being an object, it can be null
    - eg, **Integer i = null;**

Autoboxing and Unboxing

- Integer i = 5;
  - better than writing: Integer i = new Integer(5);
  - Other example: Character c = ‘a’;
- Autoboxing: Java compiler can automatically box a primitive into a wrapper object
  - eg, primitive 5 into object of type Integer
- Unboxing: automatically from wrapper to primitive
  - eg, int k = i;
- It is not for free, so usually better to use primitives in your code, unless dealing with collections or nullable values

Stacks and Queues

Stack

- Type of collection
- Add on top of the stack (**push**)
- Remove from top (**pop**)
- Can only read from top (**peek**)
- *LIFO: Last In, First Out*

Why?

- The type of operations are more restricted compared to other collections we saw so far
- But if you are only interested in the operations of a stack, you can have specialized, *high-performant* implementations for it

Example

- You need to work on some data X, so you push X on stack
- While working with X, you need to work on some other Y (**push** Y), but, once done with it (**pop**), need to go back to X (**peek**)
- While working on Y, might need to work on a Z (**push** Z), which itself might need to push more data on stack, etc.

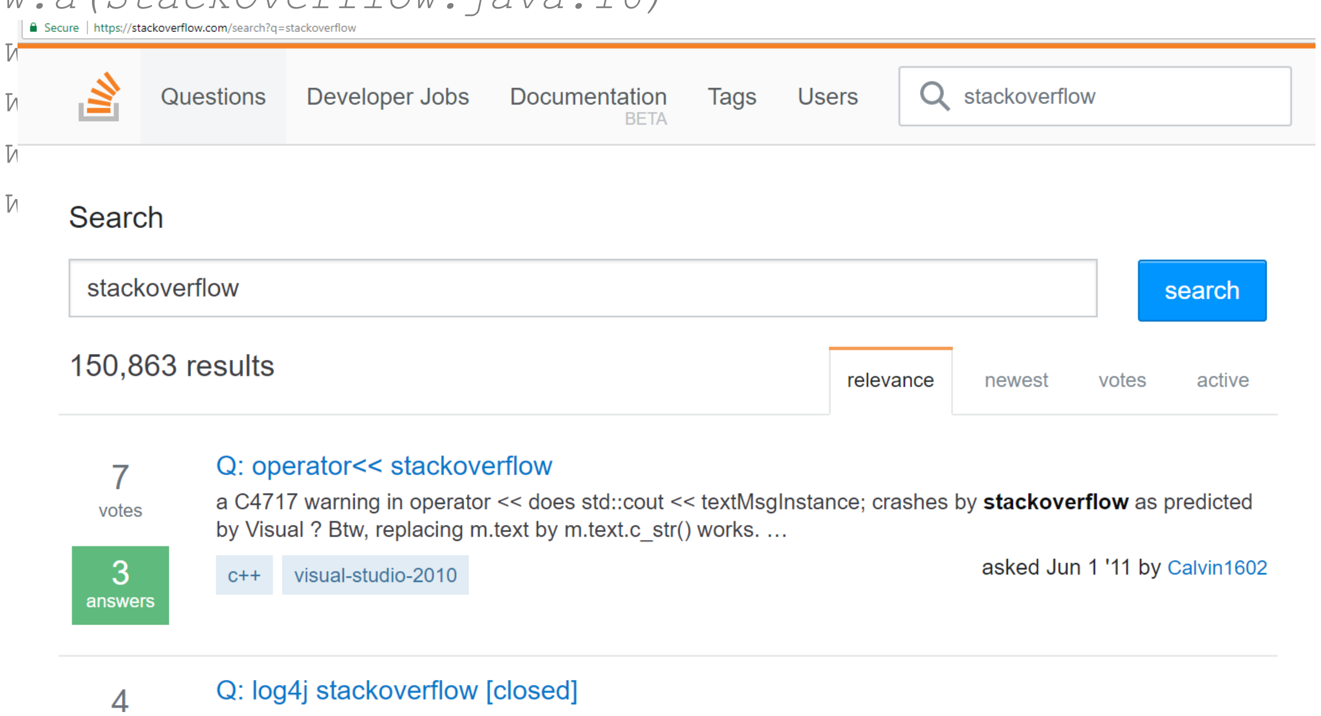
Method Call Stack

- For each method call, there is a frame, eg containing input parameters
- At each call, the JVM needs to push frame, and pop it once method is completed

```
public class StackOverflow {

    public static void main(String[] args){
        a(0);
    }

    public static int a(int x){
        x++;

        x = b(x);

        return x;
    }

    public static int b(int y){
        return a(y);
    }
}
```

Stack Overflow

```
Exception in thread "main" java.lang.StackOverflowError
    at org.pg4200.datastructure.stack.StackOverflow.b(StackOverflow.java:22)
    at org.pg4200.datastructure.stack.StackOverflow.a(StackOverflow.java:16)
    at org.pg4200.datastructure.stack.StackOverflow.b(StackOverflow.java:22)
    at org.pg4200.datastructure.stack.StackOverflow.a(StackOverflow.java:16)
    at org.pg4200.datastructure.stack.StackOverflow.b(StackOverflow.java:22)
    at org.pg4200.datastructure.stack.StackOverflow.a(StackOverflow.java:16)
    at org.pg4200.datastructure.stack.StackOverflow.b(StackOverflow.java:22)
    at org.pg4200.datastructure.stack.StackOverflow.a(StackOverflow.java:16)
    at org.pg4200.datastructure.stack.StackOverflow.b(StackOverflow.java:22)
    at org.pg4200.datastructure.stack.StackOverflow.a(StackOverflow.java:16)
    at org.pg4200.datastructure.stack.StackOverflow.b(StackOverflow.java:22)
    at org.pg4200.datastructure.stack.StackOverflow.a(StackOverflow.java:16)
    at org.pg4200.datastructure.stack.StackOverflow
    at org.pg4200.datastructure.stack.StackOverflow
    at org.pg4200.datastructure.stack.StackOverflow
    at org.pg4200.datastructure.stack.StackOverflow
    ...
```



Queue

- Type of collection
- Add at the back, *tail* of the queue/line (**enqueue**)
- Remove from the head of the line (**dequeue**)
- *FIFO: First In, First Out*

Example:
Task Scheduler

- Process/thread add *tasks to do* on a queue
- Other process/thread workers read from queue and execute the task
- The *oldest* tasks need to be completed *first*
- While workers are executing tasks, new tasks could be added to the queue

Stack/Queue as List

- Stack
  - push(value) -> add(size(), value)
  - pop() -> delete(size()-1)
  - peek() -> get(size()-1)
- Queue
  - enqueue(value) -> add(size(), value)
  - dequeue() -> delete(0)
- It could be fine to use a list implementation for stacks/queues, but there are cases in which it is *very inefficient*

Memory Model

Questions

- Node bar = new Node();
    - what is the variable “*bar*” concretely?
    - what does “*new*” actually do?
    - what is the difference between “*bar*” variable and the object created by “*new Node()*”?
- bar.next = bar.next.next;
    - what is happening here?
    - are objects created or deleted?

Overview

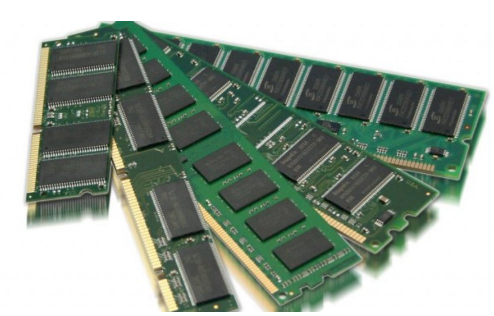
- Before we go into details of how to implement a Stack or a Queue, we need to have clear understanding of how memory is handled in Java

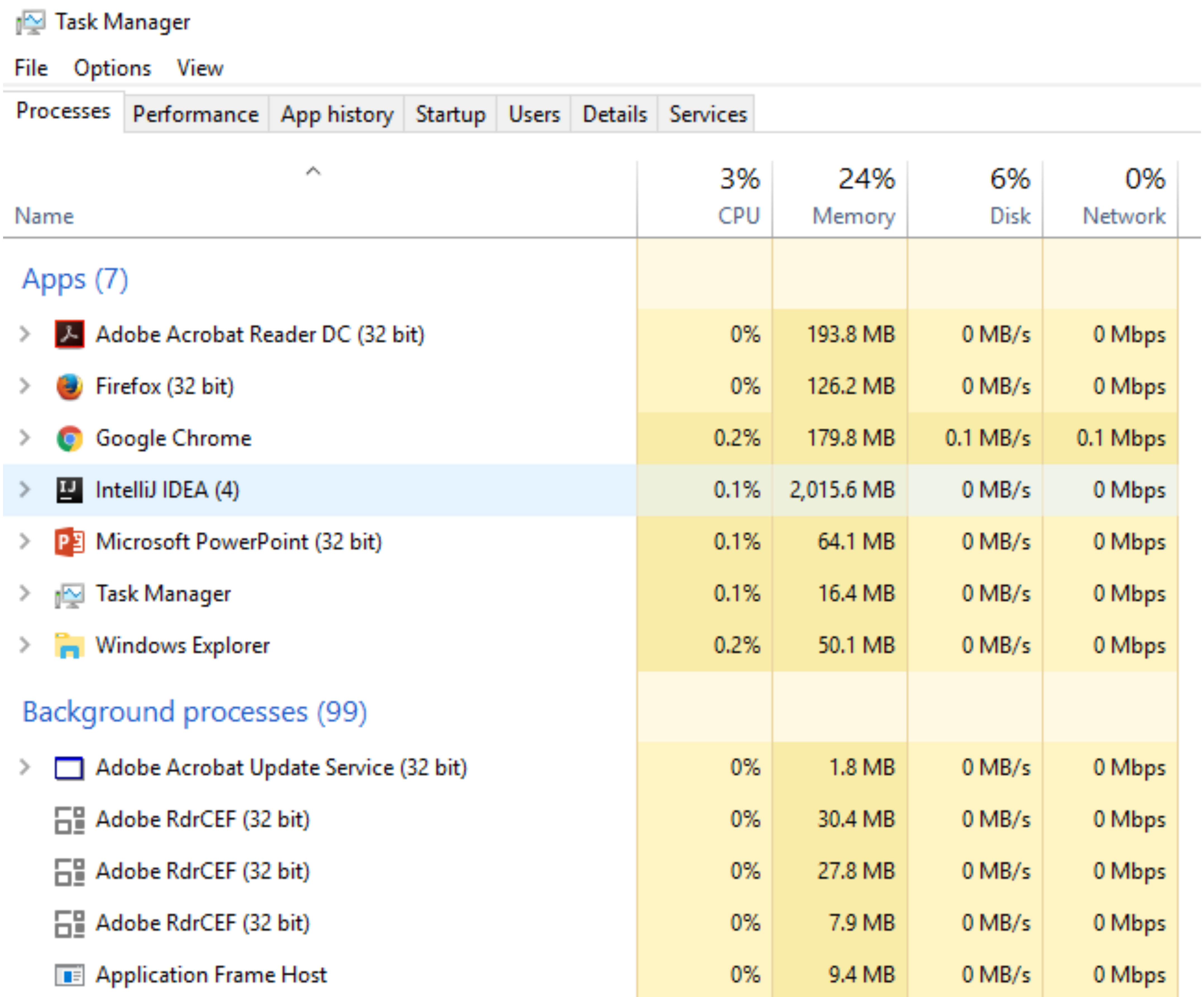
- *Pointers* and *memory* are usually hard to understand… but critical, otherwise it will be nearly impossible to understand the data structures in this course

- Should had been covered in the 1^{st} year
    - so this is just a high level revision…

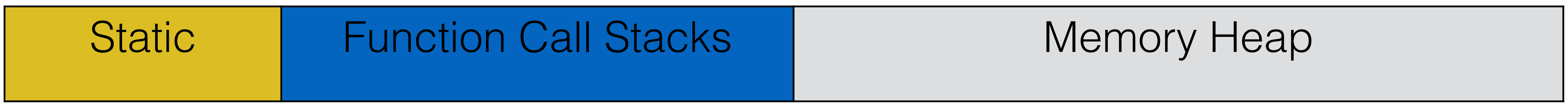
Very Simplified Model

- A process will get allocated a certain amount of space on your RAM by the Operating System (OS)
    - eg, you have 16G on your laptop and process needs 1G
- The process will use such memory to allocate variables and objects
    - How the process handles this memory should be independent from the other processes
- *Think of the memory like a big array*, where process is allowed to write/read within a [i] – [j] range
    - Eg, if process got 1GB, it could use RAM from position 12G till 13G




Task Manager
File Options View
Processes Performance App history Startup Users Details Services
3% CPU
24% Memory
6% Disk
0% Network
Name
Apps (7)
Adobe Acrobat Reader DC (32 bit) 0% 193.8 MB 0 MB/s 0 Mbps
Firefox (32 bit) 0% 126.2 MB 0 MB/s 0 Mbps
Google Chrome 0.2% 179.8 MB 0.1 MB/s 0.1 Mbps
IntelliJ IDEA (4) 0.1% 2,015.6 MB 0 MB/s 0 Mbps
Microsoft PowerPoint (32 bit) 0.1% 64.1 MB 0 MB/s 0 Mbps
Task Manager 0.1% 16.4 MB 0 MB/s 0 Mbps
Windows Explorer 0.2% 50.1 MB 0 MB/s 0 Mbps
Background processes (99)
Adobe Acrobat Update Service (32 bit) 0% 1.8 MB 0 MB/s 0 Mbps
Adobe RdrCEF (32 bit) 0% 30.4 MB 0 MB/s 0 Mbps
Adobe RdrCEF (32 bit) 0% 27.8 MB 0 MB/s 0 Mbps
Adobe RdrCEF (32 bit) 0% 7.9 MB 0 MB/s 0 Mbps
Application Frame Host 0% 9.4 MB 0 MB/s 0 Mbps

Java Memory



- At a very, very high level, the JVM divides its allocated memory in 3 main parts
- **Static**: containing for example the bytecode to run
- **FCS**: one stack per thread for the function calls
- **Heap**: where objects are stored

Function Call Stack

```
public void foo(){
      int x = 0;
      int k = bar(x);
      print(k);
}

private void bar(int y){
      int z = y * y;
      return z;
}
```

- When *foo()* is called, we need to store *x* and *k* somewhere in memory
- When *bar()* is called, we need to store *y* and *z*, plus we should not lose *x* from *foo()*
- Once *bar()* is terminated, we do not need *y* and *z* any more

Function Call Frame

- Create a *frame* for each function call
- A frame stores all the input and all the local variables, eg., *x*, *k*, *y* and *z*
- When we start a function call, we *push* its frame to the stack
- Once function call ends, we *pop* its frame

Before *bar()* Is Called

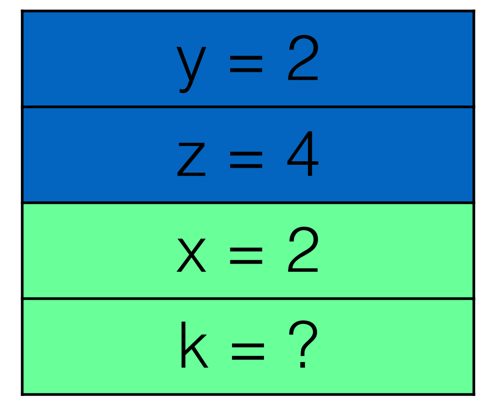
```
public void foo(){
      int x = 2;
      int k = bar(x);
      print(k);
}

private void bar(int y){
      int z = y * y;
      return z;
}
```

x = 2
k = ?

One frame on stack for the *foo()* call

Inside *bar()*

```
public void foo(){
      int x = 2;
      int k = bar(x);
      print(k);
}

private void bar(int y){
      int z = y * y;
      return z;
}
```



Push new frame for *bar(y)*

Note that *y* is initialized with same value of *x*. Changing *y* does not affect *x*, as in different frames

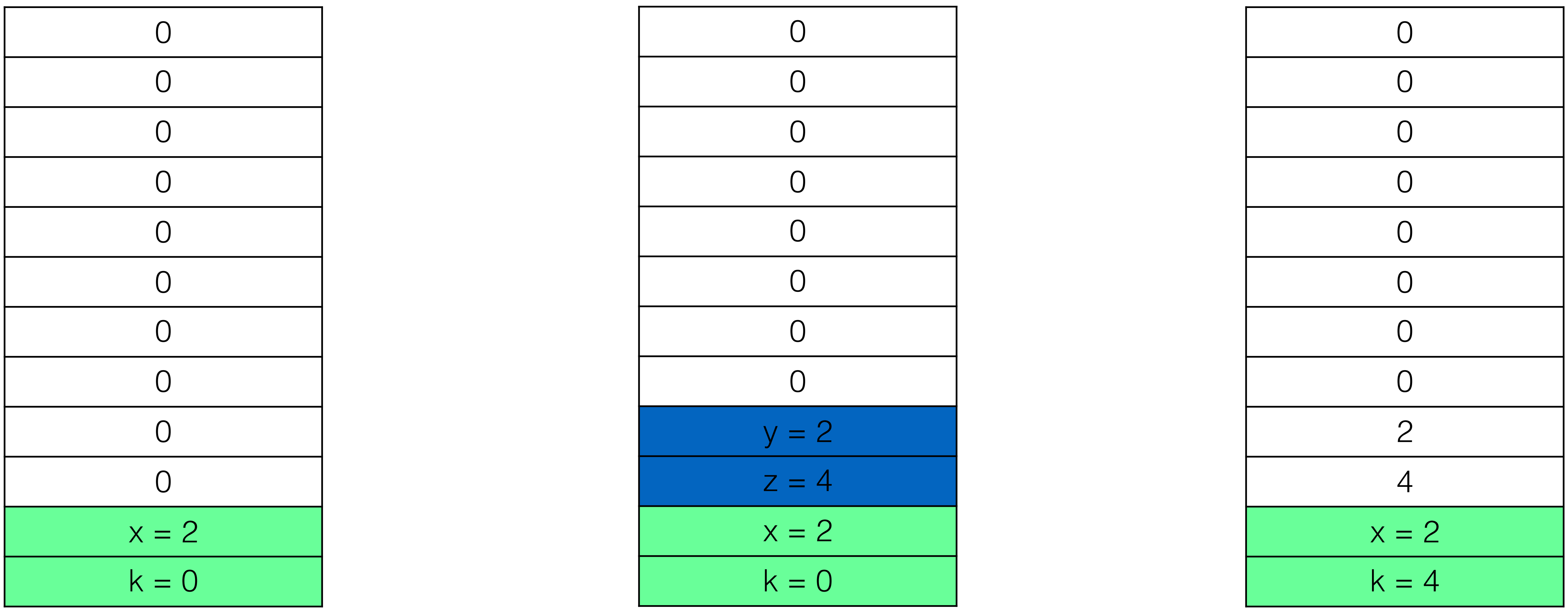
Once *bar()* Is Completed

```
public void foo(){
     int x = 2;
     int k = bar(x);
     print(k);
}

private void bar(int y){
     int z = y * y;
     return z;
}
```

x = 2
k = 4

Pop stack of bar(y), as no needed any more.

Actual Bytes In Memory



Consider each cell as contiguous 32 bits

When we pop frame, data is still actually there. Will be overwritten at next frame push

Performance Issue

```
public void foo(){
     int x = 2;
     int k = bar(x);
     print(k);
}

private void bar(int y){
     int z = y * y;
     return z;
}
```

- When we call *bar(x)*, the 32 bits of *x* are copied from current frame to the frame of *bar()* in the *y* variable
- 32 bits are OK, but what if we have large objects???
- *Passing by value* is inefficient

Pointers/References

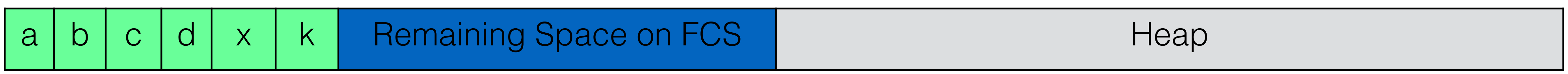
- Java does not allow you (yet) to have objects on the FCS
  - Only allowed primitive values (eg, int, double, boolean) and pointers
  - Note: other languages allows you objects on FCS, eg C++
- To have objects, those will be allocated on the **heap**
- The FCS will have **pointers** to the **heap**

Allocation on Heap

```
public void foo(
       int a, boolean b,
       char c, double d){
     X x = new X(a,b,c,d);
     int k = bar(x);
     print(k);
}

private void bar(X y){
     int z = y.compute();
     return z;
}
```

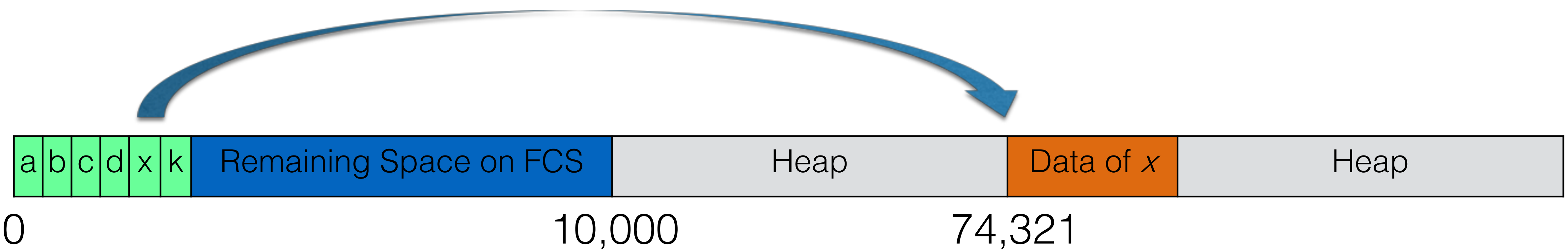
- The *x* variable is not going to contain the 4 inputs
- These are stored in the *heap*
- *x* is just a **pointer** to the location on the heap
- Assume *X* has 4 private fields, initialized in constructor

```java
public void foo(
       int a, boolean b,
       char c, double d){
     X x = new X(a,b,c,d);
     int k = bar(x);
     print(k);
}
```



- FCS growing from left to right
- Frame contains data for 4 inputs and 2 local variables
- *X* is a 64 bit address in the memory, ie it is a number, like an index in an array

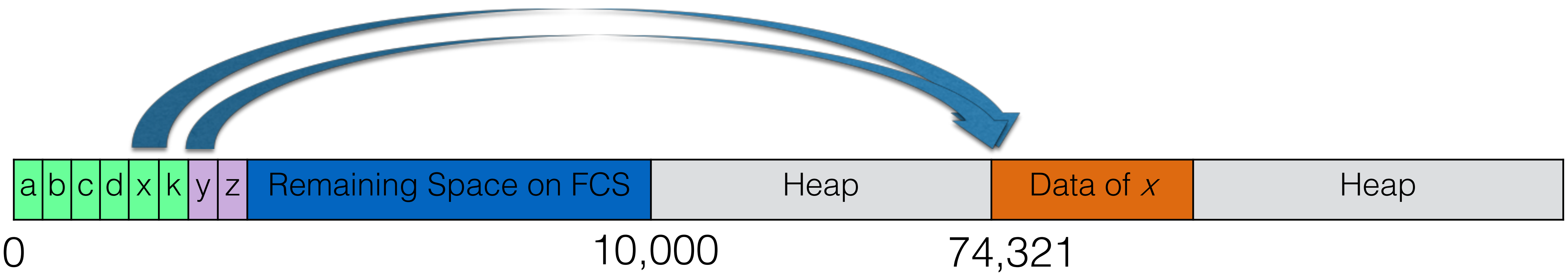
```java
public void foo(
        int a, boolean b,
        char c, double d){
    X x = new X(a,b,c,d);
    int k = bar(x);
    print(k);
}
```



- The *new* keyword allocates memory in the heap for storing all the data of *x*
- Can't control where in heap data of *x* is allocated, but will be at some known position, eg 74321
- When JVM calls *new*, it will choose a *free* area in the heap
- The variable *x* in the FCS will contain the numeric address, eg 74321

```
public void foo(
        int a, boolean b,
        char c, double d){
    X x = new X(a,b,c,d);
    int k = bar(x);
    print(k);
}
```

```
private void bar(X y){
        int z = y.compute();
        return z;
}
```



- The frame pushed for *bar(x)* contains data for *y* and *z*
- *x* in the frame of *foo()* has same value of *y* in frame of *bar()*, ie 74321
- The “Data of x” has not be copied when calling *bar(x)*, we just copied the *reference*, ie the address 74321

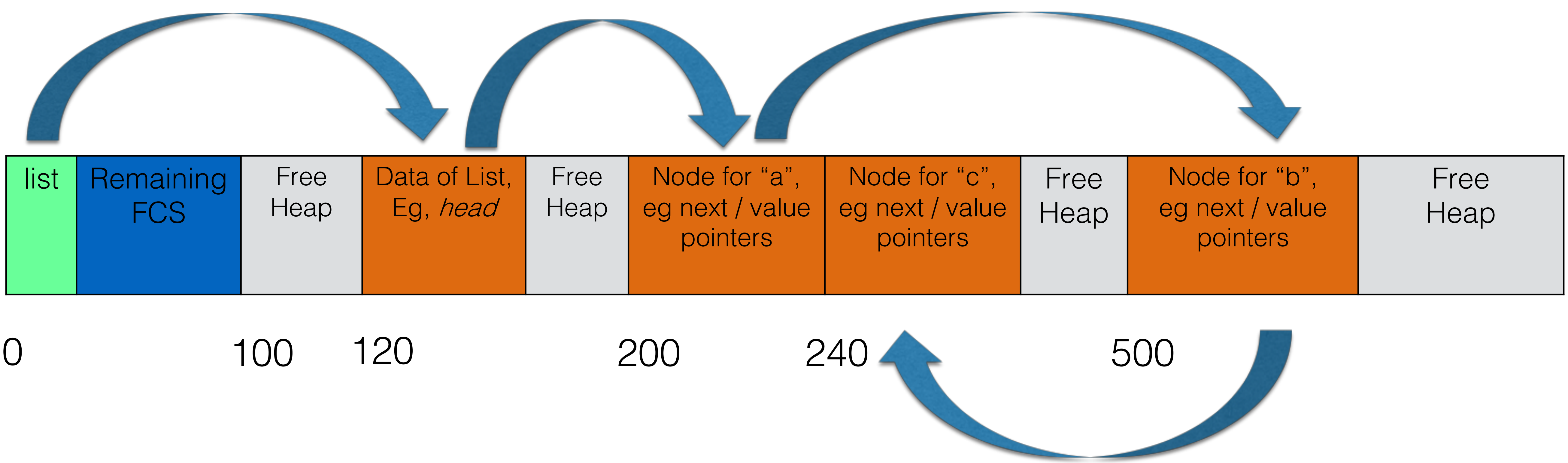
LinkedList Example

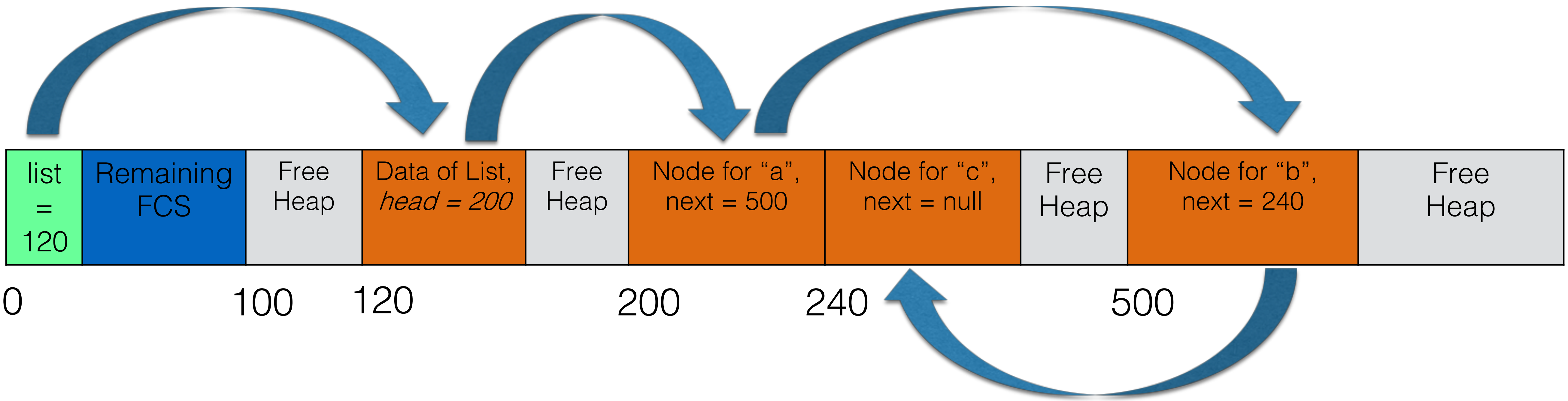
```
public void foo(){
    List list =
        new LinkedList();
   list.add("a");
   list.add("b");
   list.add("c");
}
```

- Assume LinkedList based on nodes
- List has an *head*
- Each node has a *next* reference

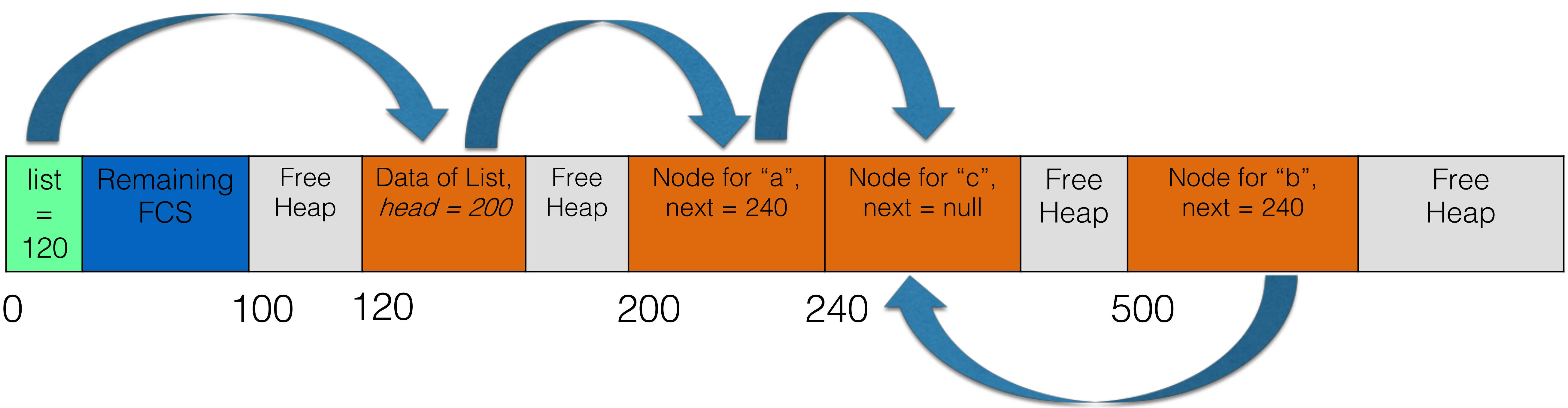
```java
public void foo(){
   List list =
       new LinkedList();
  list.add("a");
  list.add("b");
  list.add("c");
}
```

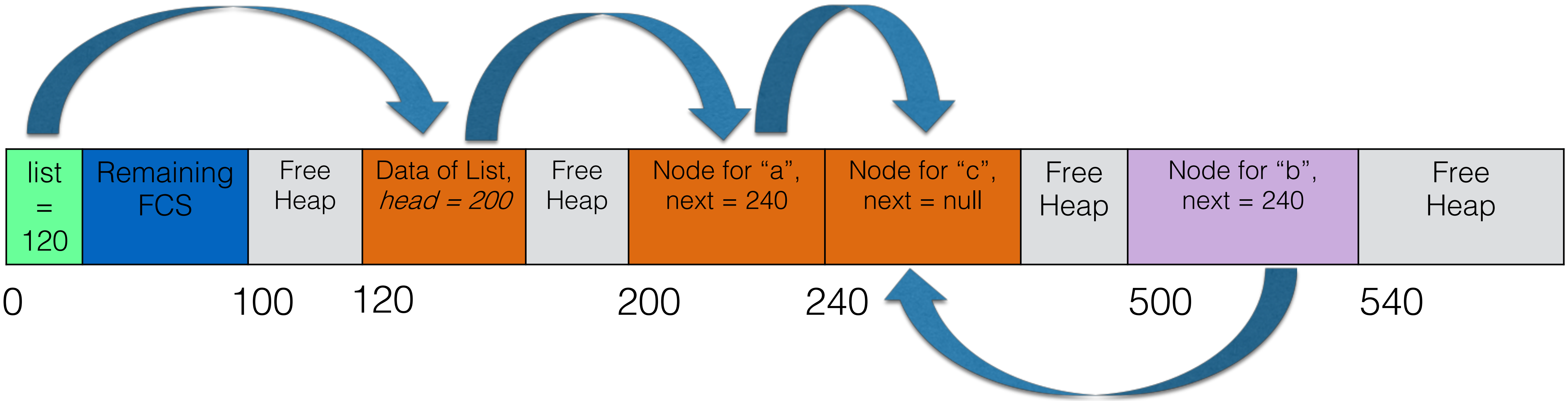
- The *list* reference on FCS will point to position where list object is, ie 120
- The *head* in such data will contain the value 200, ie address of first element
- The *next* fields contains address of next elements





- Delete node for "b" with: **current.next = current.next.next**
- Where **current** is the node for "a"





- Deleting "b" means it is not accessible any more starting from *list* pointer in the FCS
  - but it is still there in memory!!!
- When calling *new* many times, might *run out of free space*
- At that point, somehow we need to be able to reuse the space occupied by the "b" node, ie location 500-540

Garbage Collector (GC)

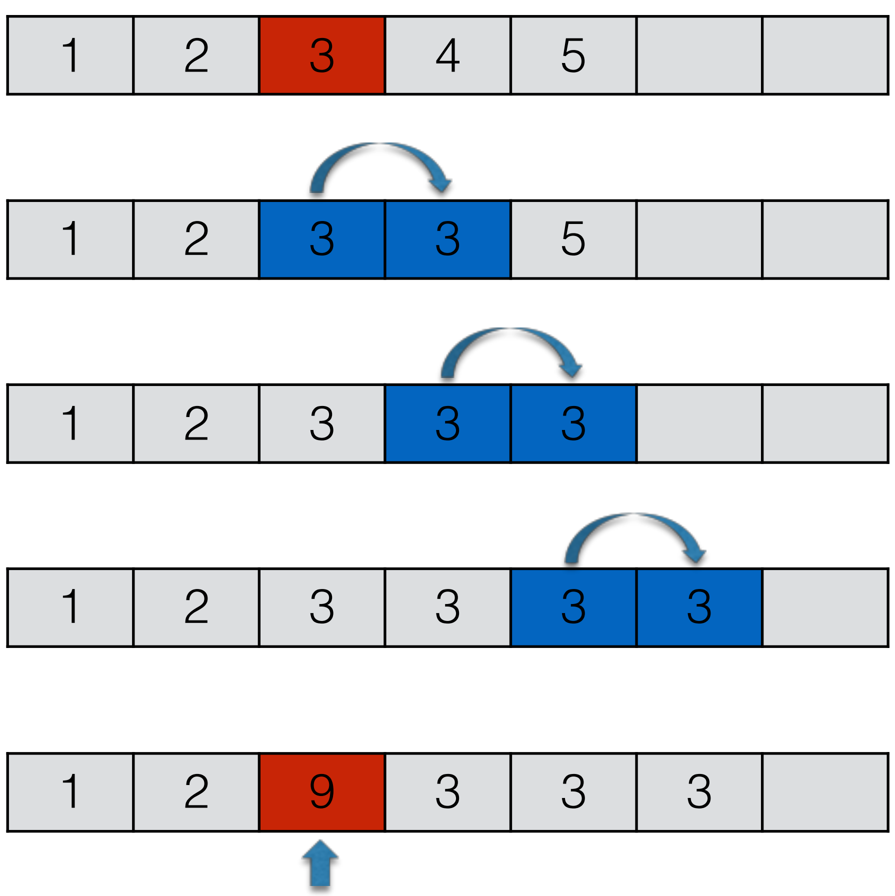
- Called by JVM when run out of space on heap
- Starting from the pointers on FCS, recursively find all reachable objects
- Non-reachable objects (eg "*b*" node) will be marked as "Free Heap", and their space can be reused by **new** operator when new instances are created
- GC are quite complex, as need to be very efficient, because they **block** the entire code execution

ArrayList

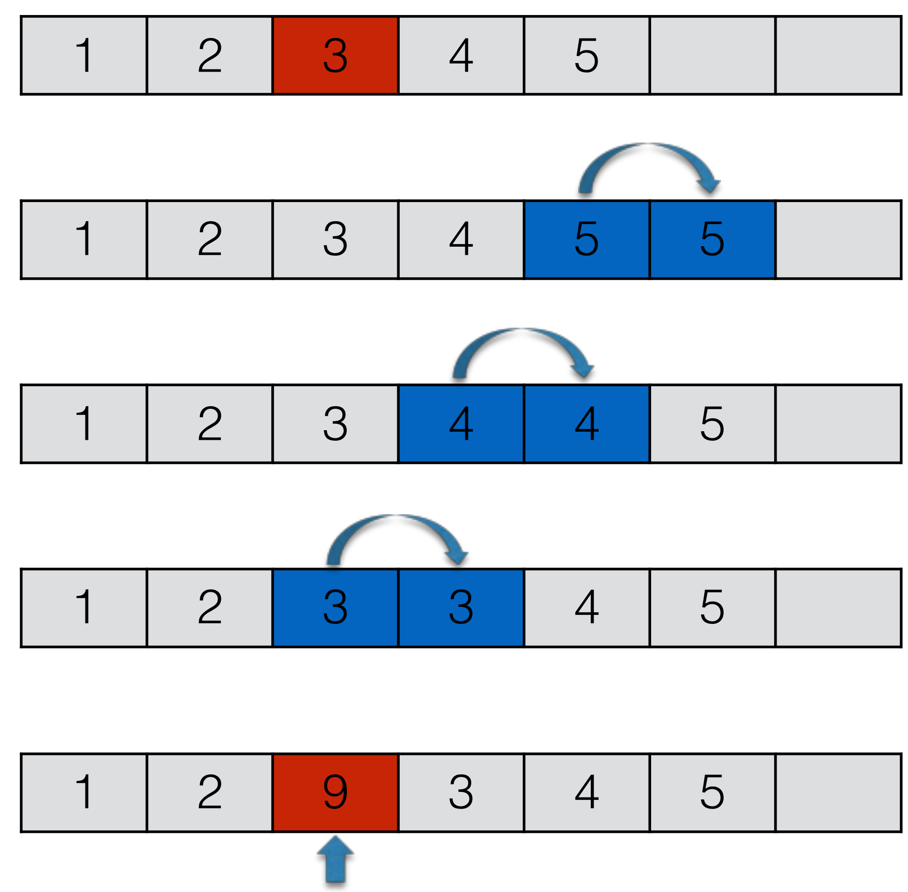
Insertion

- Need to right-shift all values from *index* before inserting new value
- On an array, we set 1 value at a time, ie **a[i+1] = a[i],** possibly in a loop
- Loop must start from end to avoid overwrite
- Assume adding a **9** at position 2 (currently occupied by value **3**)
- *But what if array is full???*

WRONG

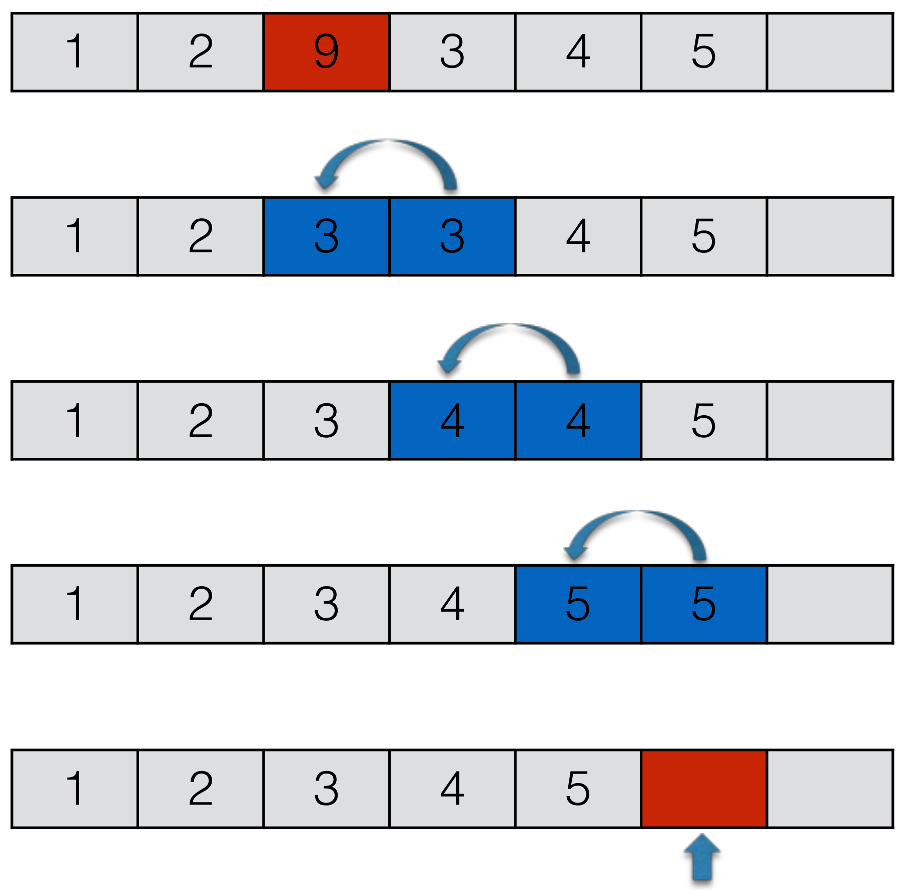


CORRECT



Deletion

- Similar to addition, here we need to shift left, and then remove last element
- **a[i] = a[i+1]**, in a loop
- Assume *delete(2)*, ie remove the value **9** at index position 2



LinkedList

Implementation

- Each element T contained in a *node* instance

- Each node contains a value, and a **next** field to the following node in the sequence

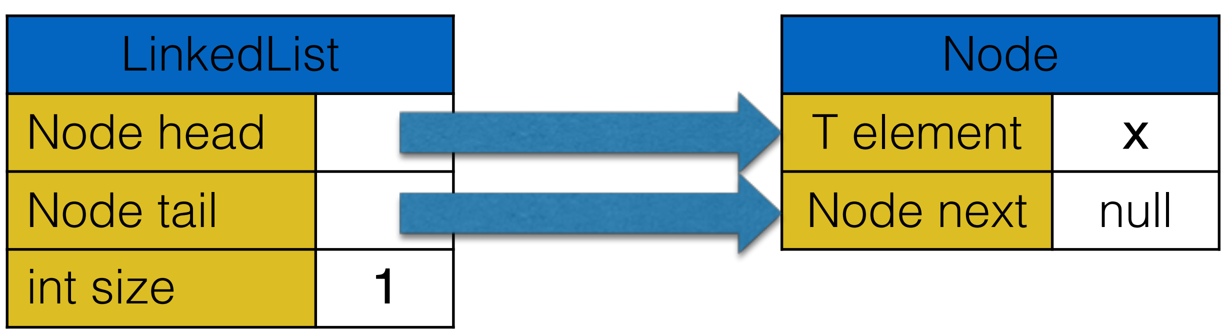
- *LinkedList* object has pointers to the **head** and tail **nodes** of the list
    - could also keep track of the **size** in a variable, to avoid compute it when queried (which would be expensive)

LinkedList	
Node head	null
Node tail	null
int size	0

Node	
T element	null
Node next	null

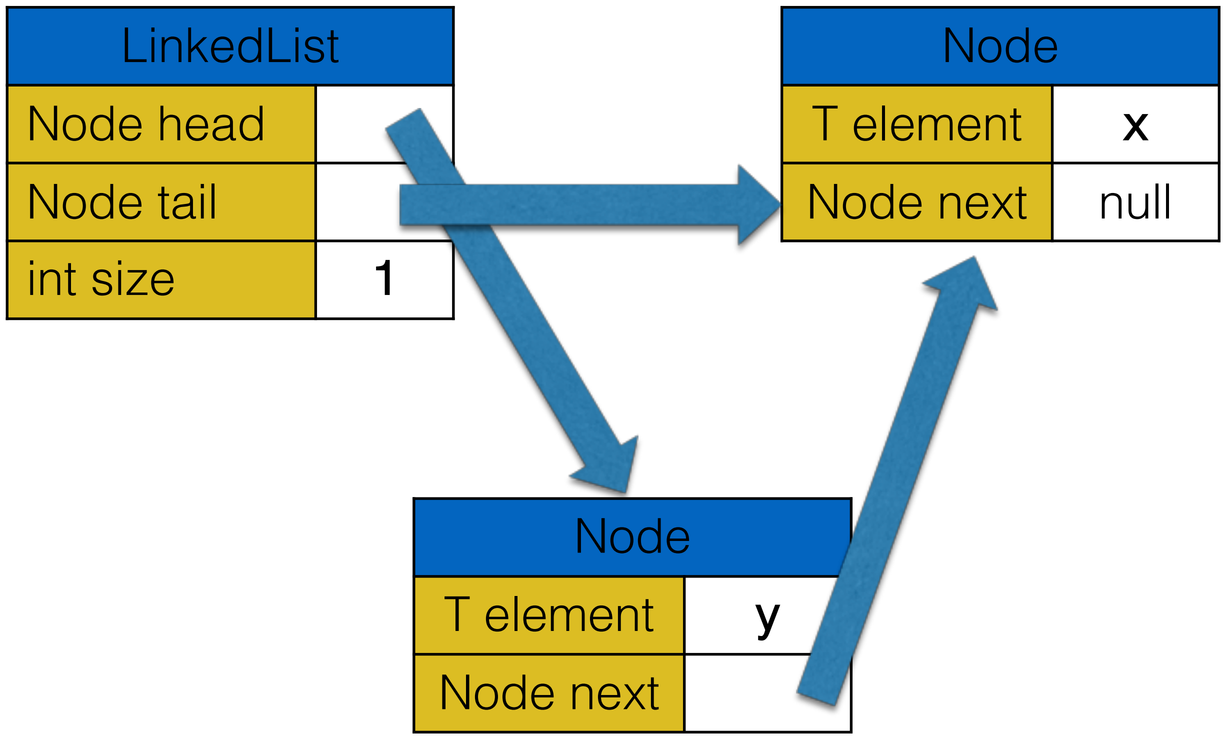
Insertion When Empty

- Create new node for the element **x**
- Update both **head** and **tail** to point to such node



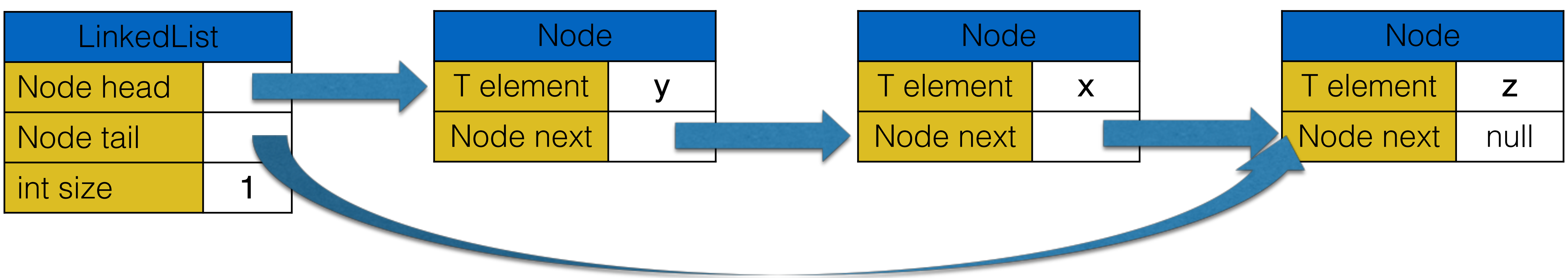
Insertion at 0

- Insert y in a non-empty LinkedList
- Besides creating new node, need to update the **head**
- New node will have to point to the previous head



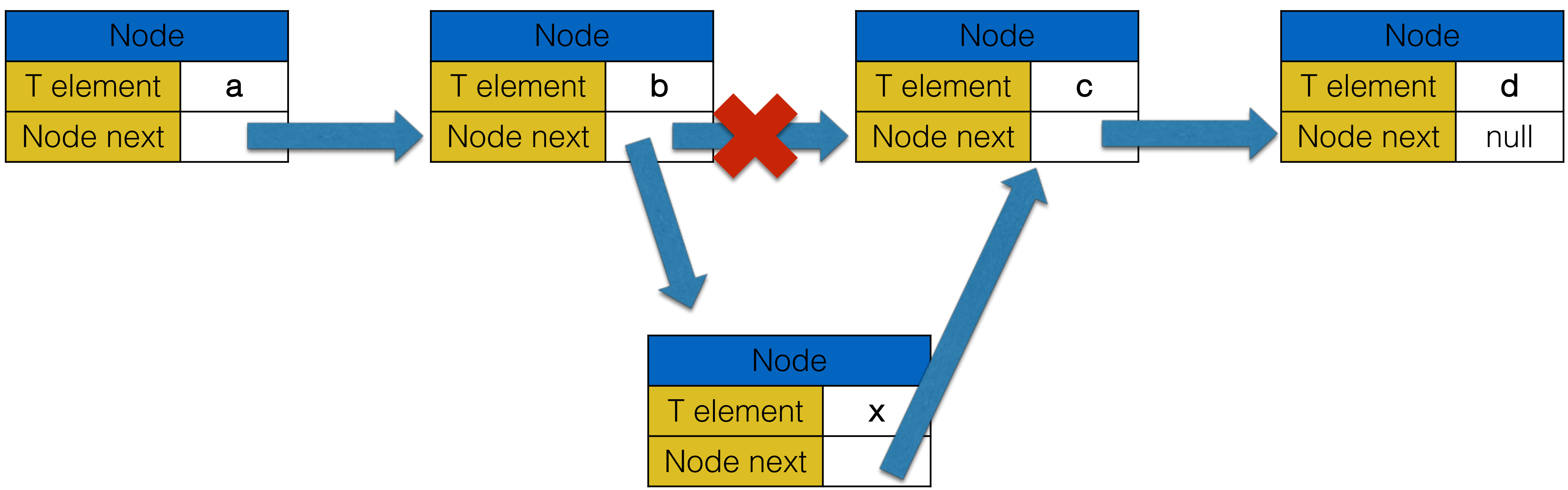
Insertion At The End

- Create new node for **z**
- The **next** of current **tail** should point to this new node
- The **tail** should then be updated to point to it



Insertion In The Middle

- Bit more complex, as no direct access to position **index** from the *LinkedList*
  - we only have **head** and **tail**
- We need to navigate from **head**, following the nexts of **next**
  - eg, **head.next.next.next.next**…
  - usually in a loop **current = current.next**, starting from **current = head**



ArrayList or LinkedList?

- For most cases, *ArrayLists* are better
- Creating node objects for each element in a *LinkedList* is expensive, plus overhead for GC
- *ArrayLists* have the issue of *resize*
  - although it can be automated, it is still expensive when it happens
  - but, if you have an idea of how many elements at most you will store, you can create a buffer array larger than that
- Still very important to understand *LinkedList*, as foundation for *Tree* data-structures

Stack as List

- Fine for both *ArrayList* and *LinkedList* implementations
  - operations at the end of the list are efficient in both implementations
- In the case of *LinkedList*, code can be simplified
  - No need for **head**, as only working at the end of the list with **tail**
  - In the nodes, instead of pointer to **next** node, have pointer to **previous** node
    - ★ otherwise could not delete

Queue

Queue As List

- Fine for *LinkedList*
  - in a queue, we only operate on **head** and **tail**, no need to navigate whole list with **head.next.next…** (which would be inefficient)
- VERY INEFFICIENT for *ArrayList*
  - each *dequeue()* call would force a left-shift of the whole list
- If want to use an array as internal data-structure, we need a better implementation than an *ArrayList*

Implementation

- Besides an internal array, need to have 2 indices, representing position of the **head** and **tail** of the queue on such array
- When empty, **head = tail = -1** (ie an invalid index)
- We do not care what the array contains in the positions outside the head-tail range

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
?	?	1	4	5	3	?	?	?	?

head = 2
tail = 5

Dequeue

- Get the value at position **head**
- Then increase **head** by 1
- We could ignore the value that was stored at the head, but better to put it to **null** for GC

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
?	?	1	4	5	3	?	?	?	?

head = 3
tail = 5

Enqueue

- Increase **tail** by 1
- Add at position given by the **tail** index
- Assuming adding a **4**

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
?	?	1	4	5	3	4	?	?	?

head = 2
tail = 6

End of the Array

- What happens when **tail** reaches the end of the array?
- Several options
  - Left-shift
  - Resize into longer array
  - Ring access (we will see this one in the exercises)

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
?	?	?	?	?	?	?	3	1	5

head = 7
tail = 9

Left-Shift

- Only possible if **head>0**, otherwise no space
- Good if only few elements

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
?	?	?	?	?	?	?	3	1	5

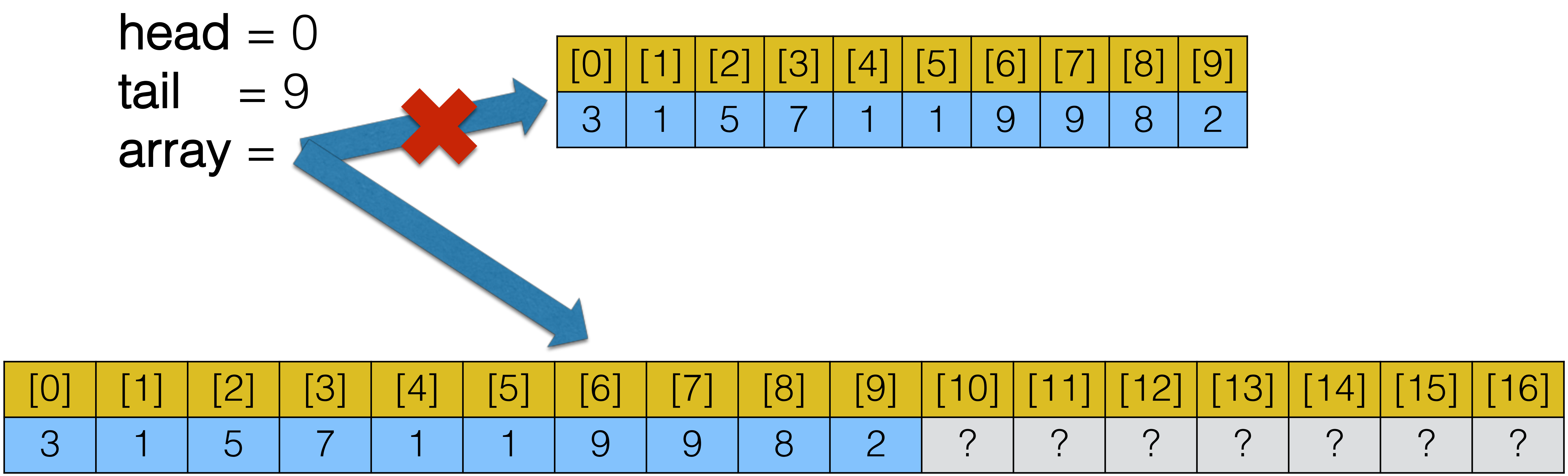
head = 7
tail = 9

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
3	1	5	?	?	?	?	?	?	?

head = 0
tail = 2

Resize

- Create new, larger array
- Copy over all elements
- Use new array as current internal buffer
- Only real option when **head=0**, but can also do it for **head>0** and **size()>k** to avoid too many left-shits



Homework

- Study Book Chapter 1.3
- Study code in the *org.pg4200.les02* package
- Do exercises in *exercises/ex02*
- Extra: do exercises in the book