PG4200: Algorithms And Data Structures

Lesson 02: Stack and Queue

Dr. Andrea Arcuri

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.

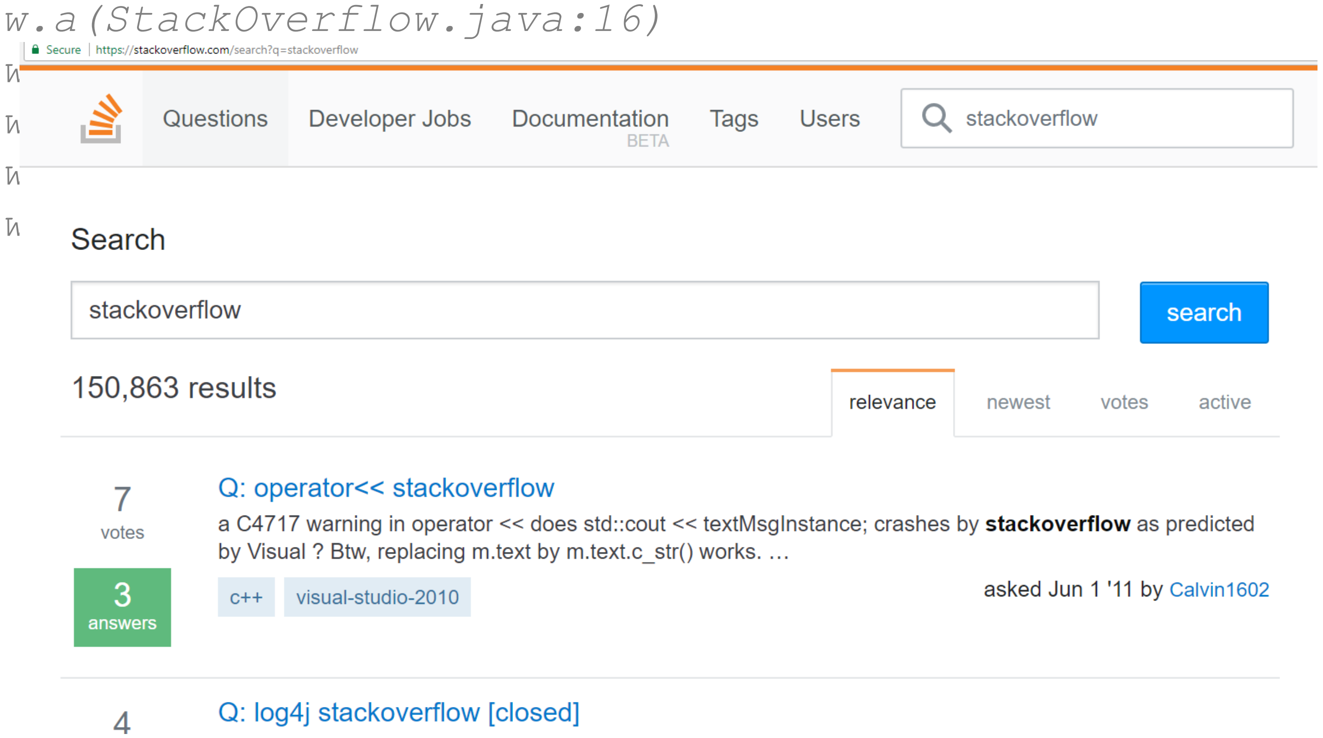
Method Call Stack

- 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

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

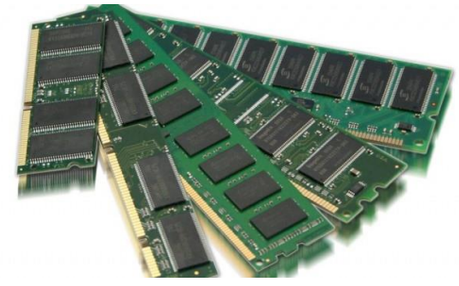
- Before we go into details of how to implement a Stack or a Queue, we need to have clear understanding of how memory is handled in Java

- *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 1st 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

Name	3% CPU	24% Memory	6% Disk	0% Network
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;
}
```

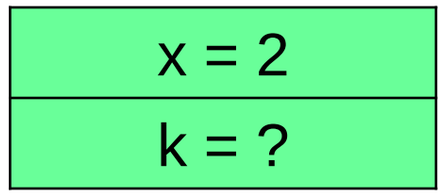
- When *foo()* is called, we need to store *x* and *k* somewhere in memory
- When *bar()* is called, we need to store *y* and *z*, plus we should not lose *x* from *foo()*
- Once *bar()* is terminated, we do not need *y* and *z* any more

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

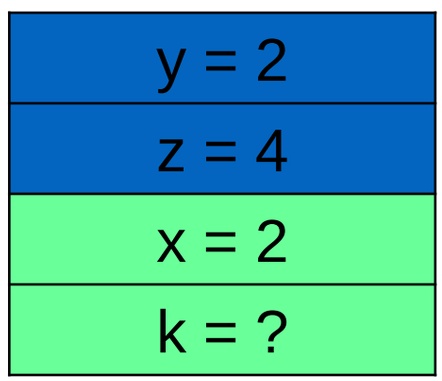
```
public void foo(){
      int x = 2;
      int k = bar(x);
      print(k);
}

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



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

0
0
0
0
0
0
0
0
0
x = 2
k = 0

0
0
0
0
0
0
0
0
y = 2
z = 4
x = 2
k = 0

0
0
0
0
0
0
0
0
2
4
x = 2
k = 4

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

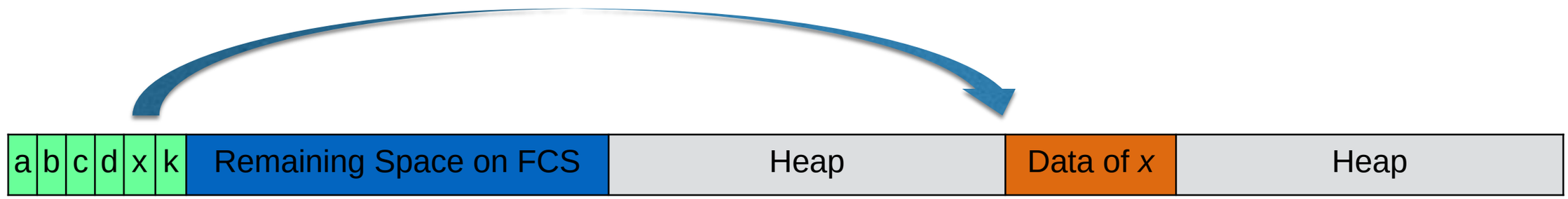
```
public void foo(
        int a, boolean b,
        char c, double d){
      X x = new X(a,b,c,d);
      int k = bar(x);
      print(k);
}

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

```
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

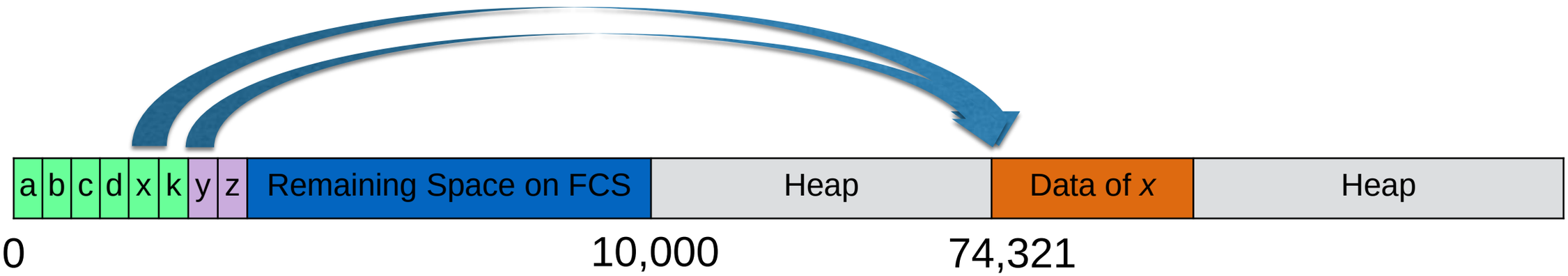
```
public void foo(
        int a, boolean b,
        char c, double d){
      X x = new X(a,b,c,d);
      int k = bar(x);
      print(k);
}
```



- 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

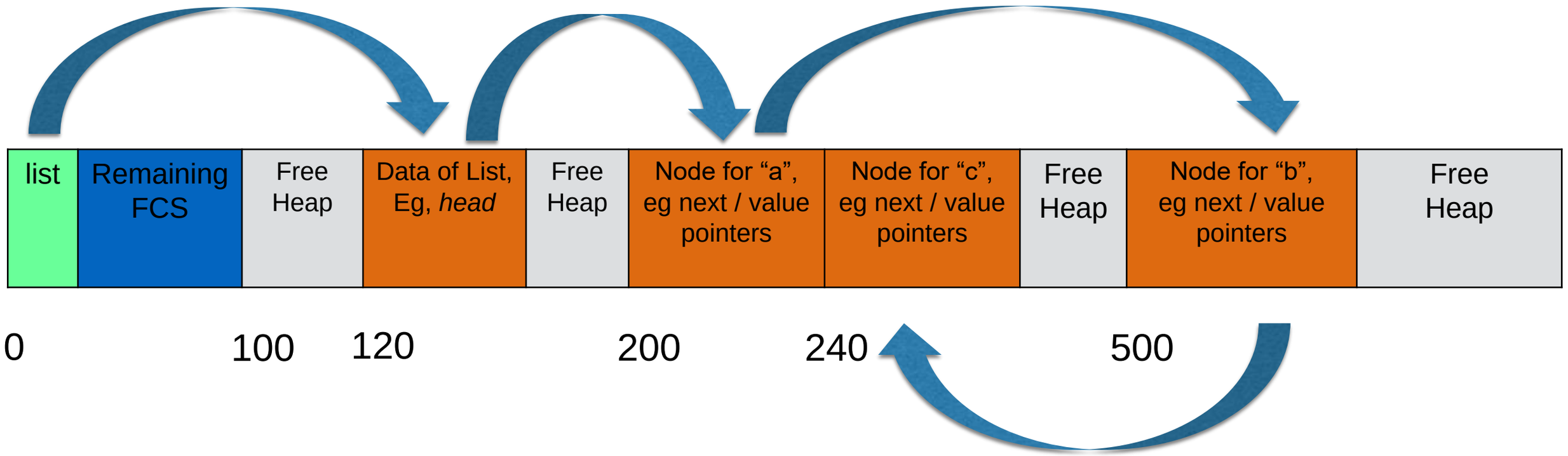
Linked List Example

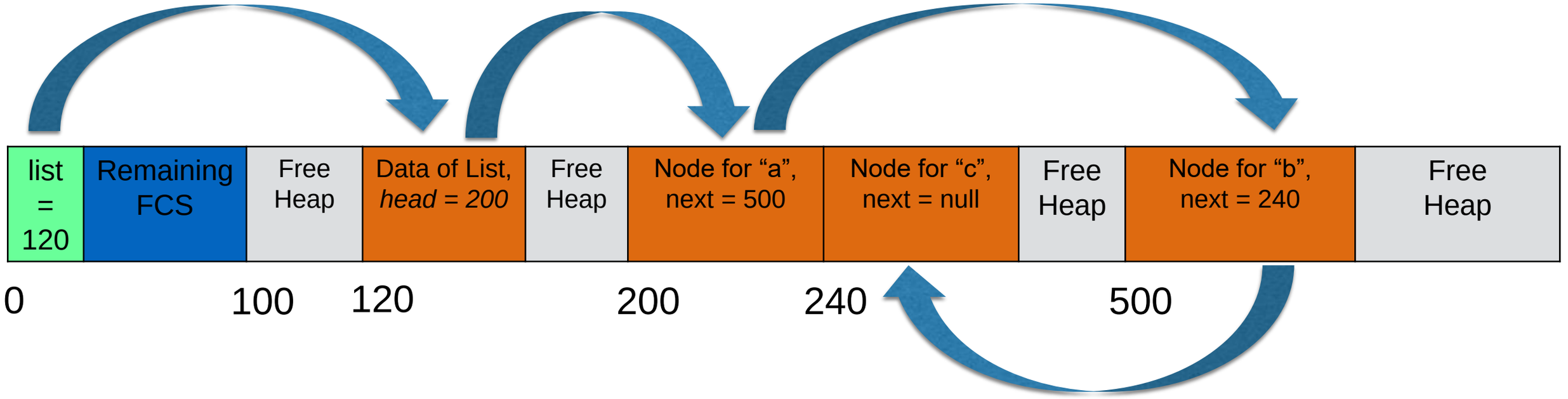
```
public void foo(){
   List list =
         new LinkedList();
   list.add(“a”);
   list.add(“b”);
   list.add(“c”);
}
```

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

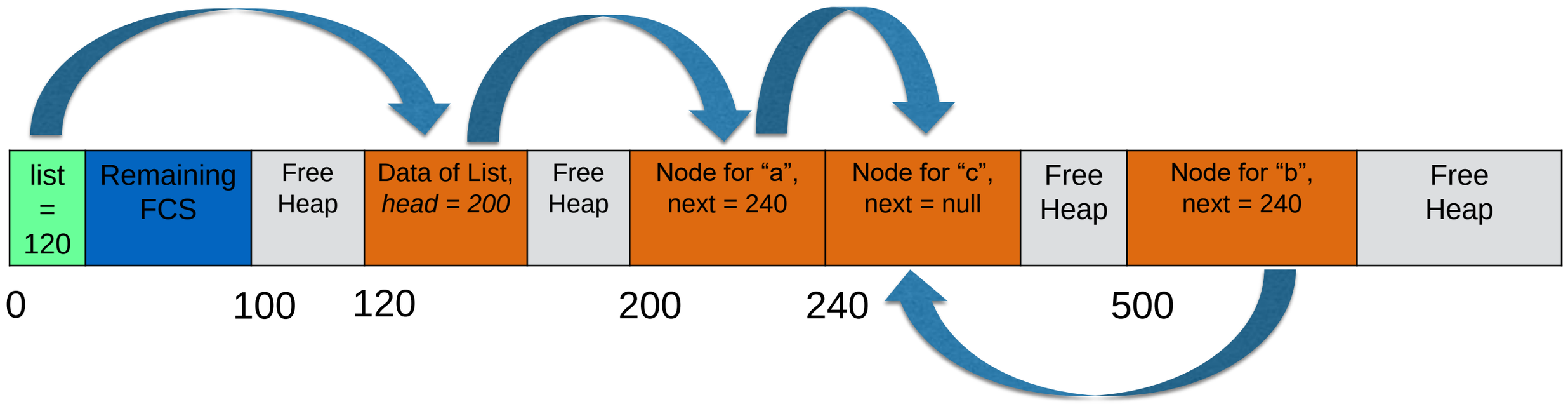
```
public void foo(){
    List list =
          new LinkedList();
   list.add("a");
   list.add("b");
   list.add("c");
}
```

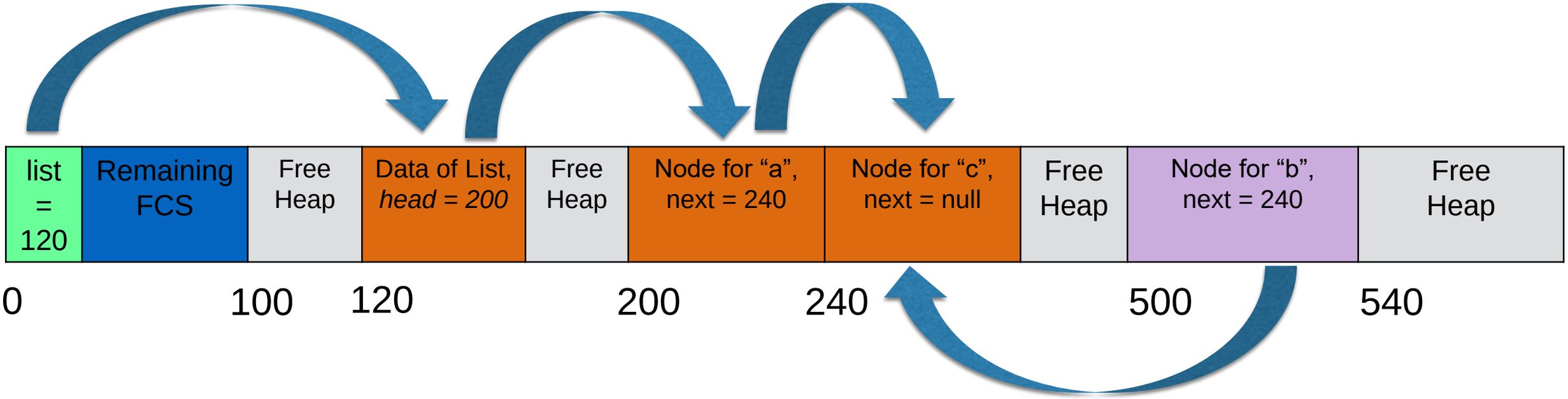
- The *list* reference on FCS will point to position where list object is, ie 120
- The *head* in such data will contain the value 200, ie address of first element
- The *next* fields contains address of next elements





- 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

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