LINKED LISTS

cs2420 | Introduction to Algorithms and Data Structures | Spring 2015

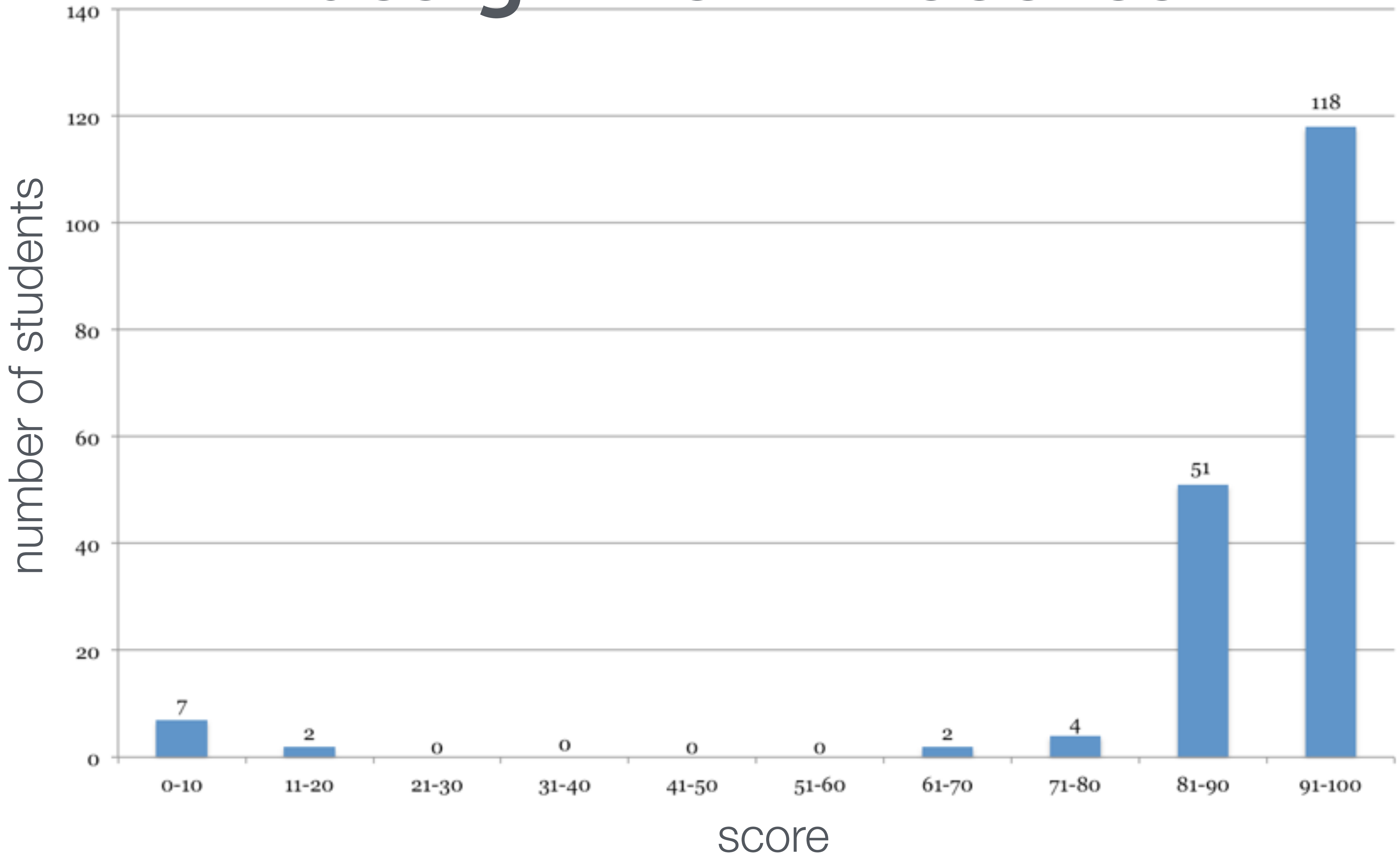
administrivia…

-assignment 5 due tonight at midnight

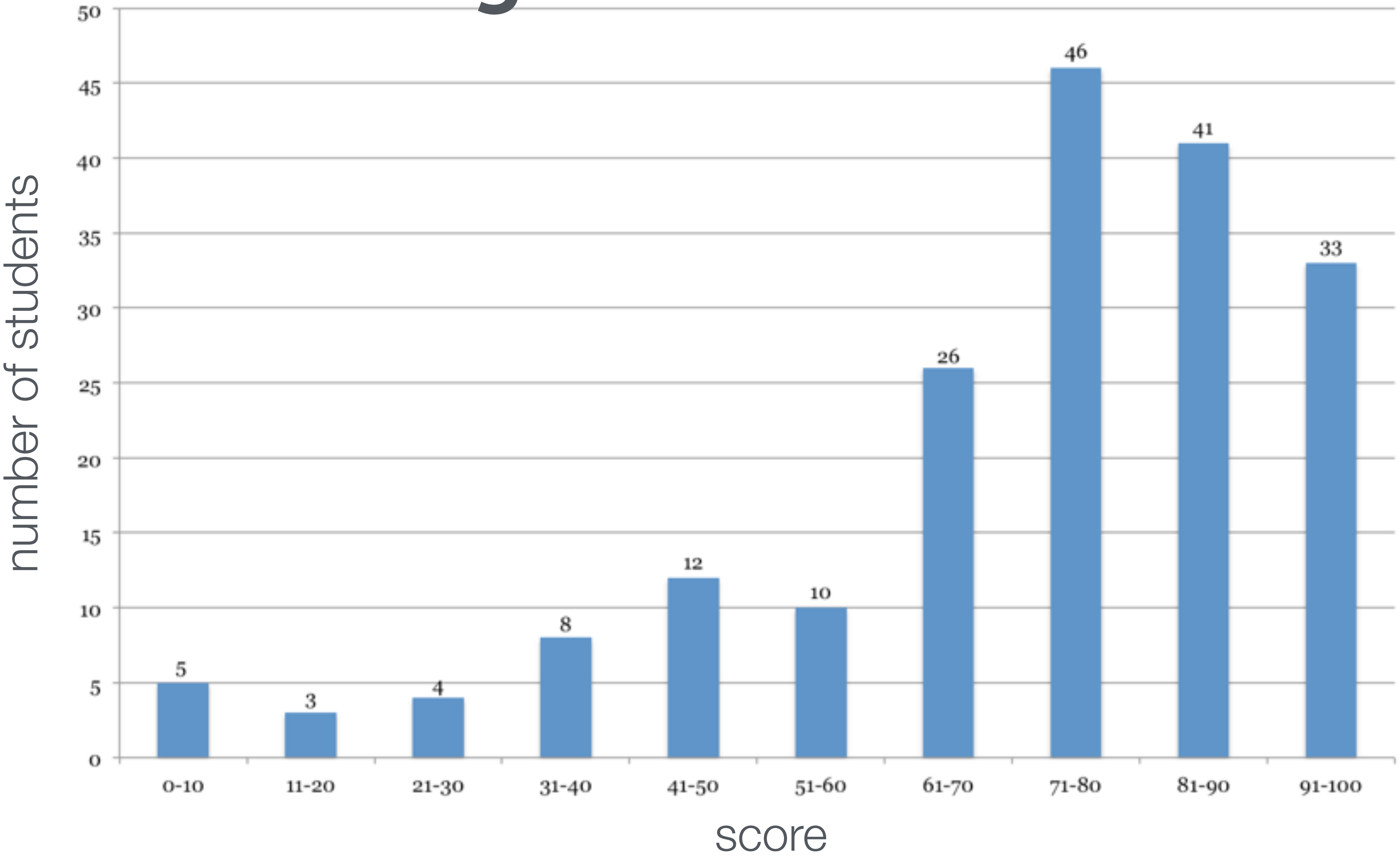
-assignment 6 is out

- -YOU WILL BE SWITCHING PARTNERS!

assignment 2 scores



assignment 3 scores



last time…

mergesort

divide and conquer

quicksort

another divide and conquer

mergesort

1) divide the array in half

~~2) sort the left half~~

~~3) sort the right half~~

2) take the left half, and go back to step 1 UNTIL???

3) take the right half, and go back to step 1 UNTIL???

4) merge the two halves together

WHAT DOES THIS LOOK LIKE?

watch a video online…

```
void mergesort(int[] arr, int left, int right)
{
  // arrays of size 1 are already sorted
  if(start >= end)
    return;

  int mid = (left + right) / 2;
  mergesort(arr, left, mid);
  mergesort(arr, mid+1, right);
  merge(arr, left, mid+1, right);
}
```

DIVIDE

CONQUER

```
void merge(int[] arr, start, mid, end)
{
  // create temp array for holding merged arr
  int[] temp = new int[end - start + 1];

  int i1 = 0, i2 = mid;
  while(i1 < mid && i2 < end)
  {
    put smaller of arr[i1], arr[i2] into temp;
  }

  copy anything left over from larger half to temp;
  copy temp over to arr;
}
```

notes on merging

-the major disadvantage of mergesort is that the merging of two arrays requires an extra, temporary array

-this means that mergesort requires 2x as much space as the array itself

- -can be an issue if space is limited!
- -an *in-place* mergesort exists, but is complicated and has worse performance

-to achieve the overall running time of **O(N log N)** it is critical that the running time of the merge phase be linear

quicksort

1) select an item in the array to be the ***pivot***

2) ***partition*** the array so that all items less than the pivot are to the left of the pivot, and all the items greater than the pivot are to the right

3) ~~sort the left half~~

4) ~~sort the right half~~

3) take the left half, and go back to step 1 UNTIL???

4) take the right half, and go back to step 1 UNTIL???

WHAT DOES THIS LOOK LIKE?

watch a video online…

```
void quicksort(int[] arr, int left, int right)
{
  // arrays of size 1 are already sorted
  if(start >= end)
    return;

  int pivot_index = partition(arr, left, right);
  quicksort(arr, left, pivot_index-1);
  quicksort(arr, pivot_index+1, right);
}
```

WHAT IS THE DIVIDE STEP?

WHAT IS THE CONQUER STEP?

in-place partitioning

1) select an item in the array to be the ***pivot***

2) swap the pivot with the last item in the array *(just get it out of the way)*

3) step from left to right until we find an item > pivot
   *-this item needs to be on the **right** of the partition*

4) step from right to left until we find an item < pivot
   *-this item needs to be on the **left** of the partition*

5) swap items

6) continue until left and right stepping cross

7) swap pivot with left stepping item

choosing a pivot

-the median of all array items is the best possible choice… why?
    -is time-consuming to compute
    -finding true median is **O(N)**

-it is important that we avoid the worst case
    -what IS the worst case(s)?

-middle array item is a safe choice… why?

-*median-of-three:* pick a few random items and take median
    -why not the first, middle, and last items?

-*random pivot:* faster than median-of-three, but lower quality

quicksort vs **mergesort**

-both are **O(N log N)** in the average case

-mergesort is also **O(N log N)** in the *worst* case

- -so, why not always use mergesort?

-mergesort requires **2N** space

- -and, copying everything from the merged array back to the original takes time

-quicksort requires no extra space

- -thus, no copying overhead!
- -but, in **$O(N^2)$** worst case <wha wha>

-both are divide and conquer algorithms (recursive)

-mergesort sorts “on the way up”

-after the base case is reached, sorting is done as the calls return and merge

-quicksort sorts “on the way down”

-once the base case is reached, that part of the array is sorted

-though quicksort is more popular, it is not always the right choice!

today…

-memory allocation

-linked structures

-linked lists

-insertion & deletion

-implementation details

-doubly linked lists

-`LinkedList` vs `ArrayList`

memory primer

-all data in your program resides in memory at some point during its life

-think of memory as giant blocks of bytes

-each byte has its own memory address

-addresses are just numbers [0 — *num_bytes*]

-byte n is next to byte n-1 and n+1
    -ie. memory is ordered

memory in Java

-what actually happens when you use the `new` keyword?

-`new` instructs the system to find a contiguous block of bytes big enough to hold whatever you are creating

- `-int arr[] = new int[10];`
  - *-finds a block of memory big enough to hold 10 ints*
- -`arr[0]` is right next to `arr[1]` in memory
  - *-the addresses of these two numbers are contiguous!*

- -arrays are a **random access** data structure
  - -any item in the array can be accessed instantly

-EXAMPLE

- -to access item `23` in an array, simply take the address of the beginning of the array and add `23` times the size of each item
- -address of `arr[23]` is address of `arr[0]+(23*4)`

-no matter the size of the array, accessing item `i` can be done in **O(c)**

- -ie. one addition and one multiplication

-each time you call new, the allocated block can be anywhere in memory

```
Circle c1 = new Circle();
Circle c2 = new Circle();
```

-`c1` may be at location 2048, and `c2` may be at location 640

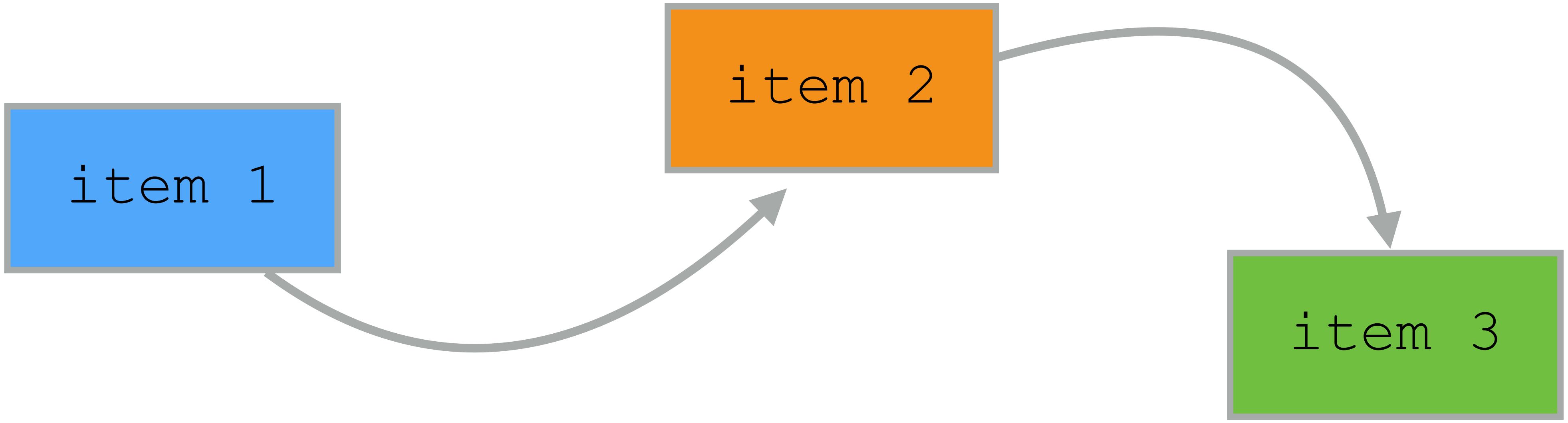
-you have no control over this!

linked structures

-**linked structures** are data storage in which individual items have *links* (references) to other items

-items don't reside in a single contiguous block of memory

-items can be *dynamically* added or removed from the structure, simply by creating or destroying links



HOW IS THIS DIFFERENT THAN AN ARRAY?

-linked structures have a reference to another instance of the structure

-looks a bit like a recursive class definition

```
class LinkedNode {
  //each node stores some data
  int ID;
  String name;

  LinkedNode next; //and one of itself!
}
```

-nodes could also have multiple links

  -think of a family tree, or airports

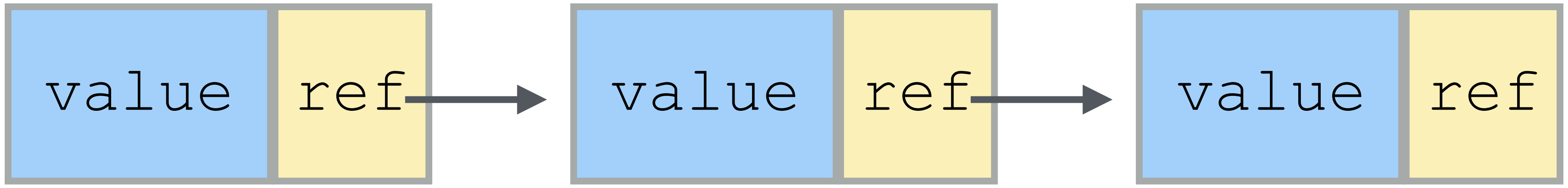
```
class LinkedNode {
  //each node stores some data
  int ID;
  String name;

  ArrayList<LinkedNode> neighbors;
}
```

linked lists

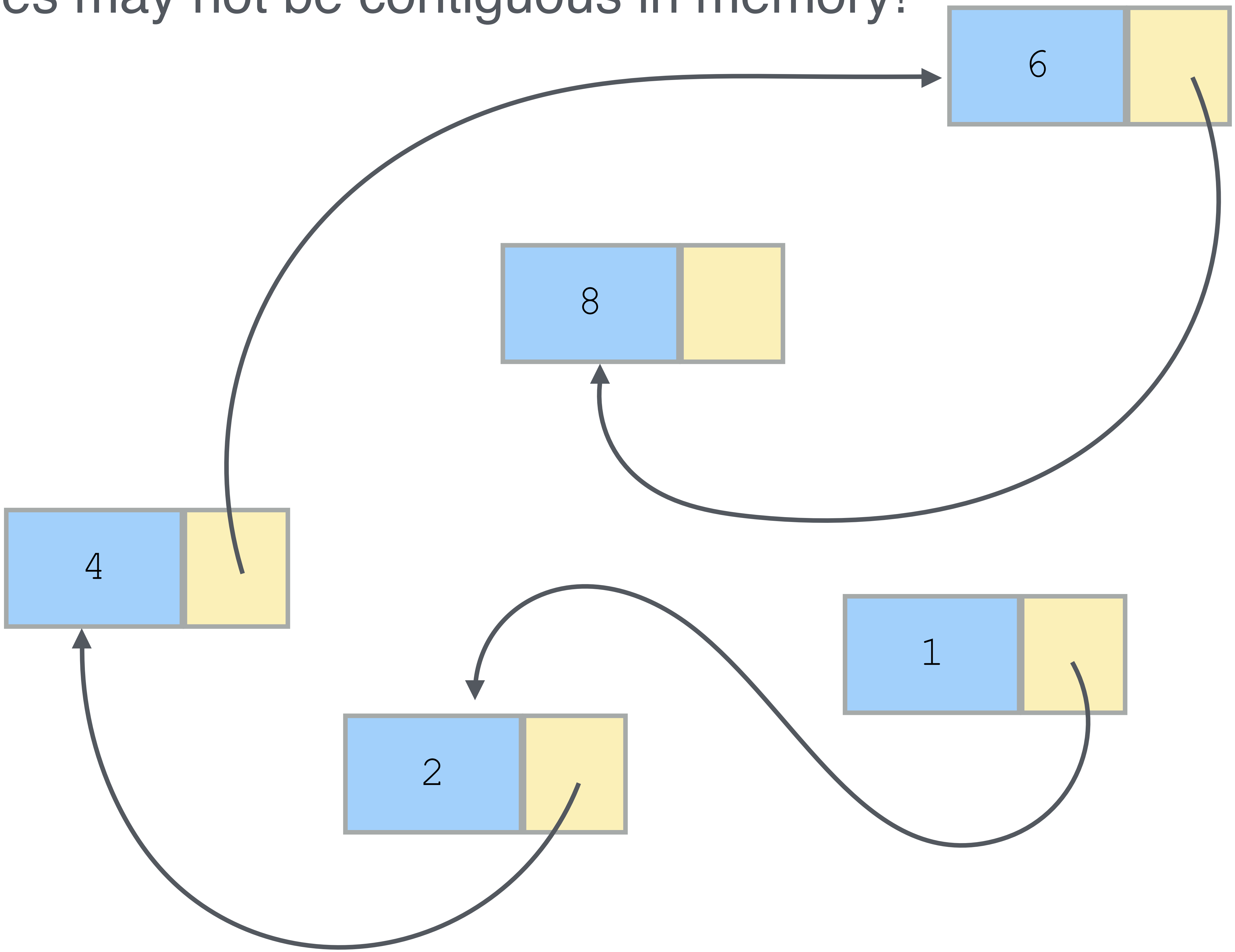
-we’ve seen a *list* implemented with an array in `ArrayList<>`

-a *linked list* is another way to implement a list

-each **node**, or item in the list, has a link to the next item in the list



-a single node consists of some **data** and a **reference** to another node nodes may not be contiguous in memory!



-with an array, we have a single variable that can access any item with `[ ]`

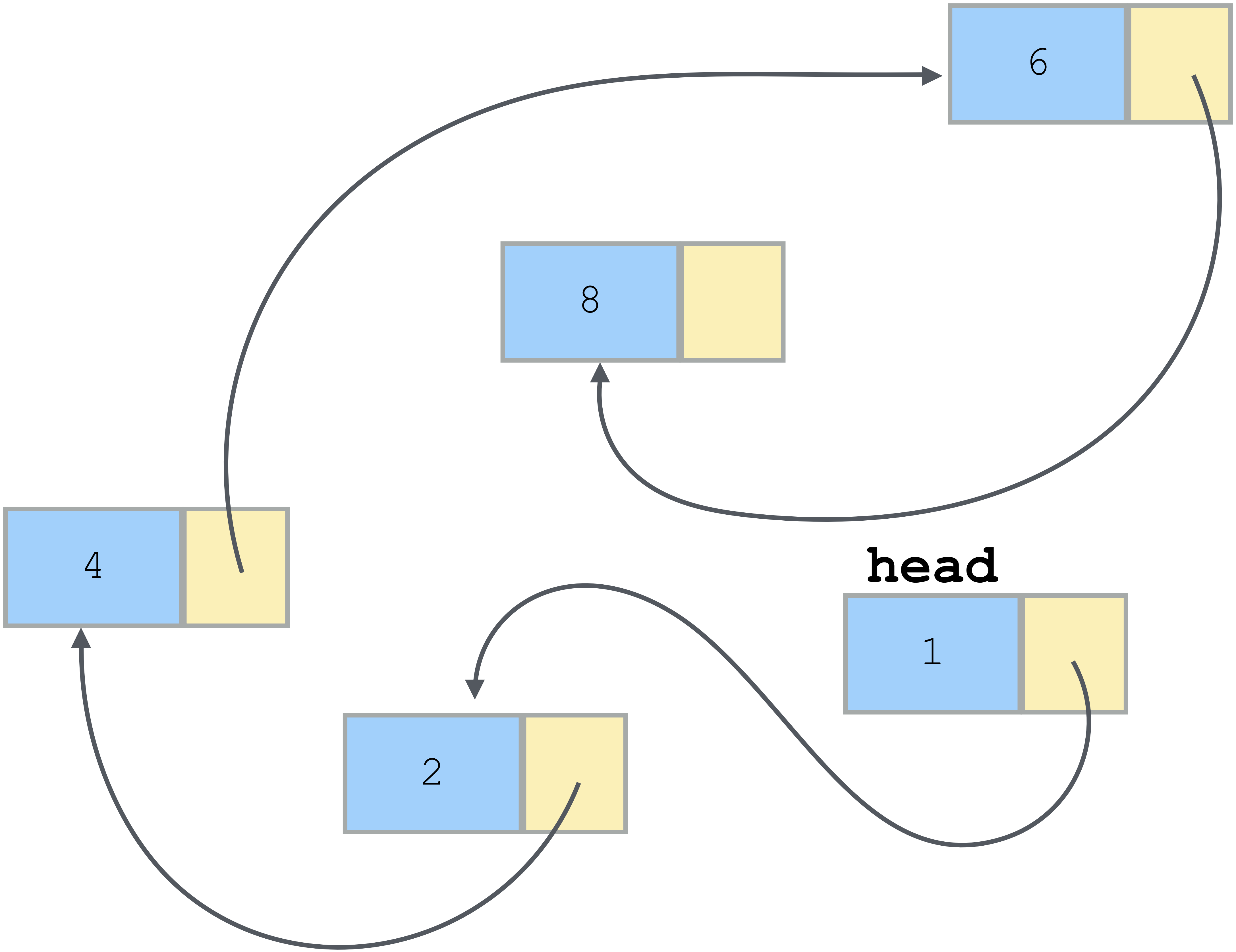
-with a linked list, how do we access individual elements?

  -HINT: we need somewhere to start

-always keep track of the first node

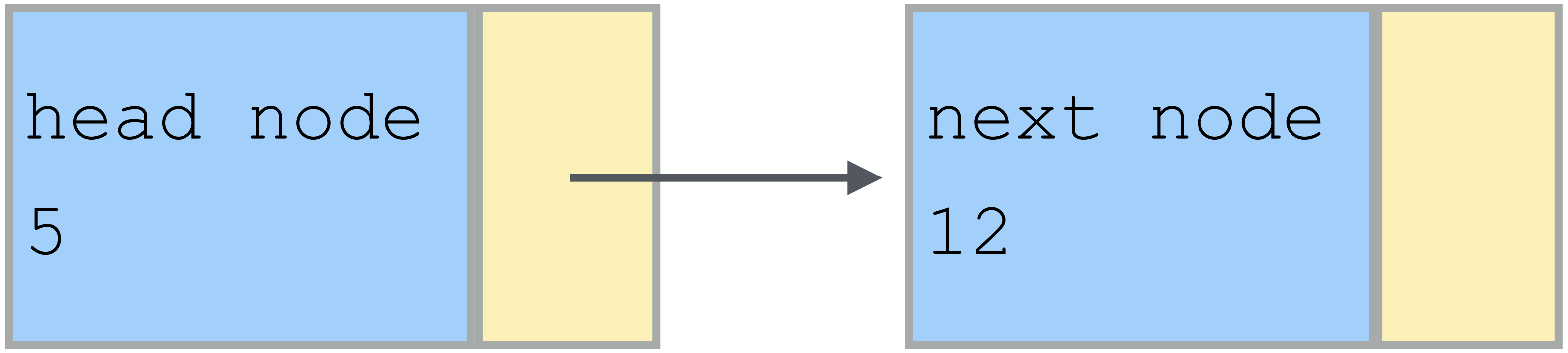
  -called the **head**

-from the head node we can access any other node by following the links



6
8
4
head
1
2

```
LinkedNode head = new LinkedNode();
head.ID = 5;
head.name = “head node”;

head.next = new LinkedNode();

LinkedNode temp = head.next;
temp.ID = 12;
temp.name = “next node”;
```



linked list vs **array**

-cost of accessing a random item at location `i`?

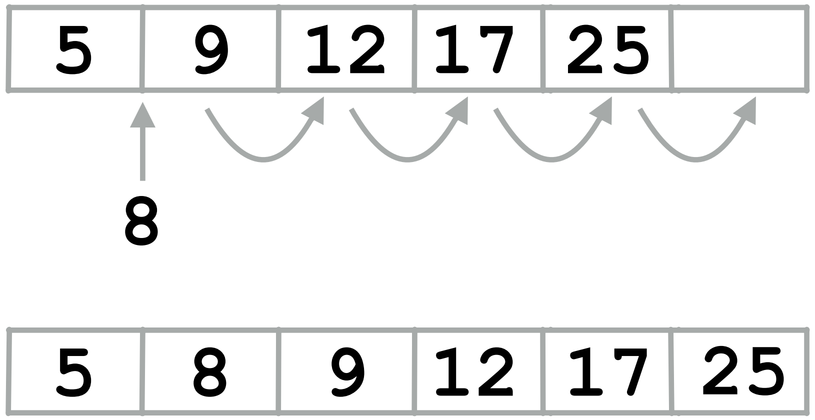
-cost of `removeFirst()`?

-cost of `addFirst()`?

A) **c**
B) **log N**
C) **N**
D) **N log N**
E) **N^2**
F) **N^3**

insertion & deletion

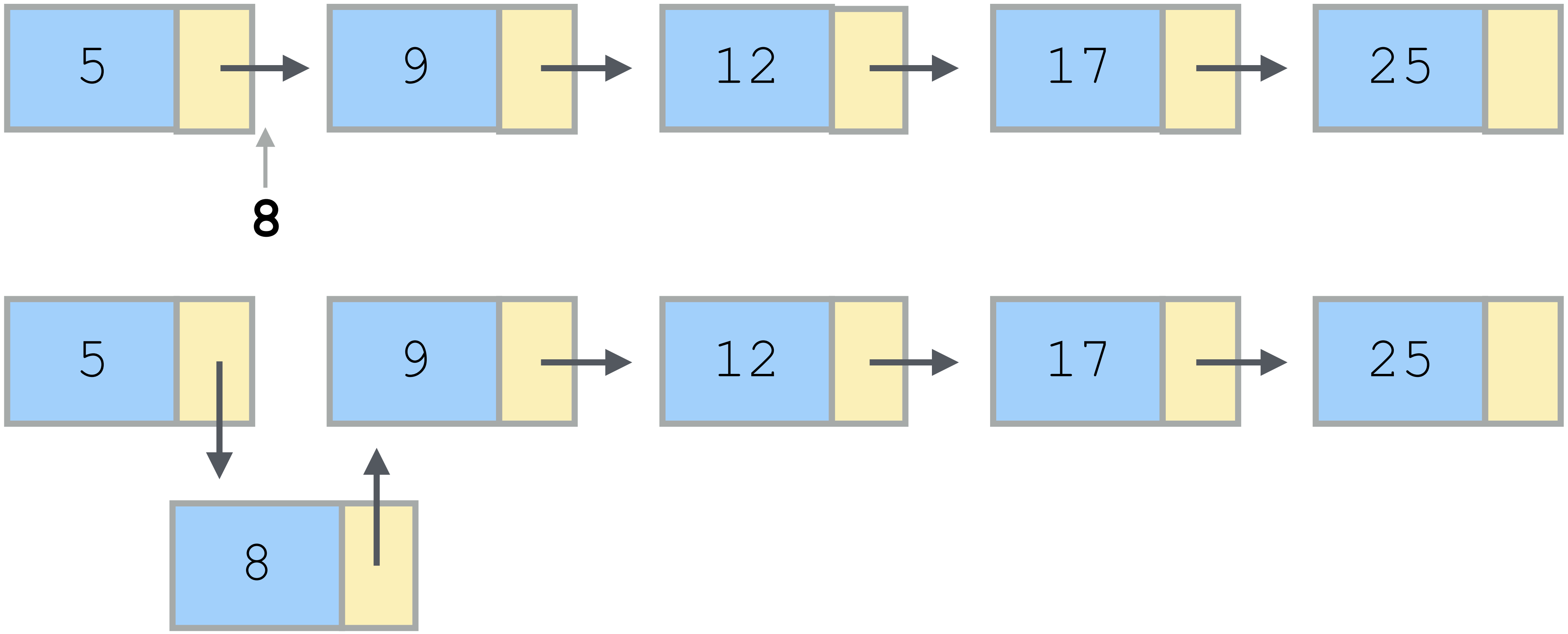
inserting into an array:



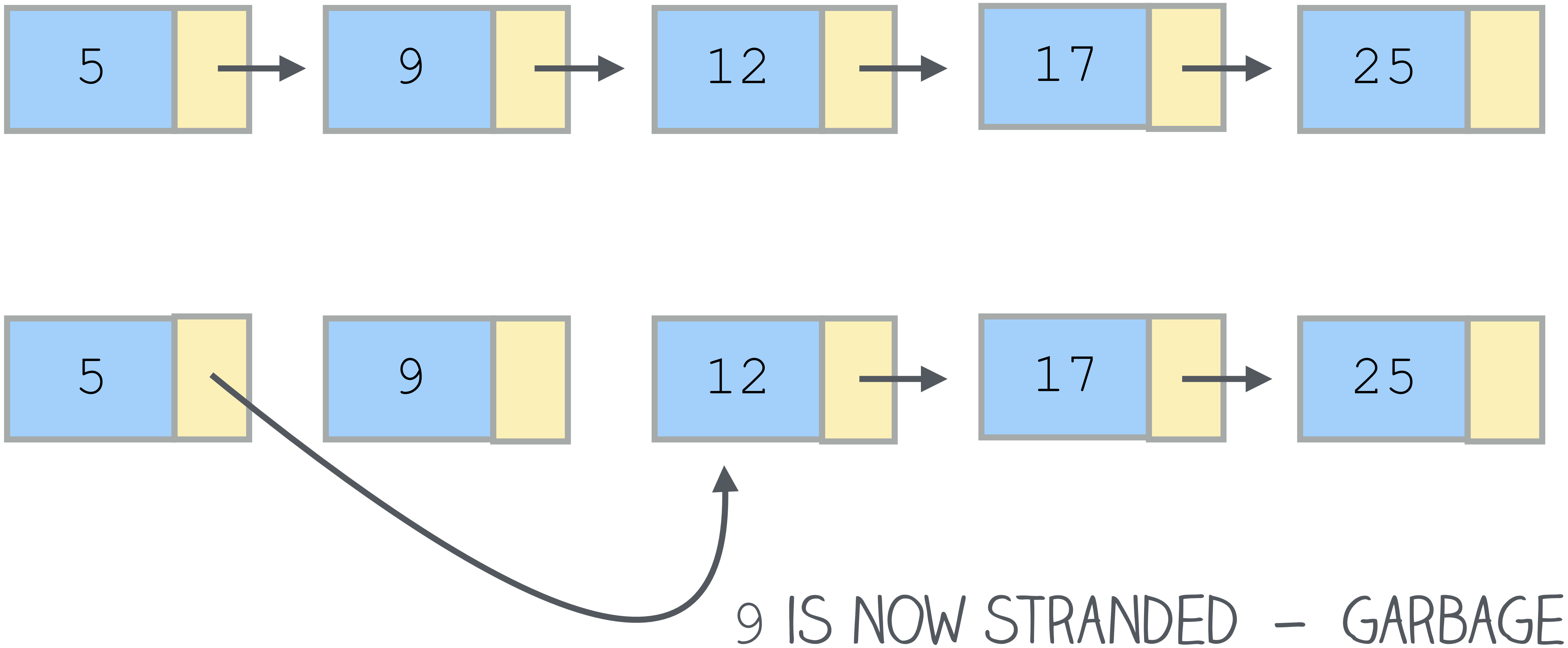
WHAT IS THE COST OF INSERTION?

A) **c**
B) **log N**
C) **N**
D) **N log N**
E) **N^2**
F) **N^3**

inserting into a linked list:



deletion from a linked list:



9 IS NOW STRANDED – GARBAGE COLLECTOR WILL CLEAN IT UP

implementation details

- linked lists have some methods, a `size`, etc.
  - but, it doesn't make sense for every node to store the `size`!

- out class `LinkedList` keeps track of the `size`, head node, and defines all methods

- `Node` should be a simple, inner class (private)
  - with a `data` field, and one or more `Nodes` (links)

nongeneric implementation (only stores `ints`):

```
class LinkedList
{
  private Node head;
  private int size;

  private class Node
  {
    private int data;
    private Node next;
    …
  }
  …
}
```

things to consider…

-what should `next` be for the last item in the list?

-don't let a call to `new LinkedNode()` cause an infinite loop

  -ie. creating a new `LinkedNode`, which creates a new `LinkedNode`, and so on…

-constructor should set `next` to `null`

traversing a linked list:

```
public boolean contains(int item)
{
  Node temp = head;

  while(temp != null)
  {
    if(temp.data == item)
      return true;

    temp = temp.next;
  }

  return false;
}
```

exercise…

- what is the implementation of `get()`?

```
public int get(int i) { … }
```

- NOTES
  - throws `NoSuchElementException` if `i` is out of range
  - move the next node with the `.next` reference

- what is the equivalent method for an `ArrayList`?

doubly-linked lists

-nodes have a link to `next` *and* `previous` node

-allows for traversal in either forward or reverse order

-maintains a `tail` node as well as a `head` node
  -why?

-how can we use a doubly-linked list to optimize `get(i)`?

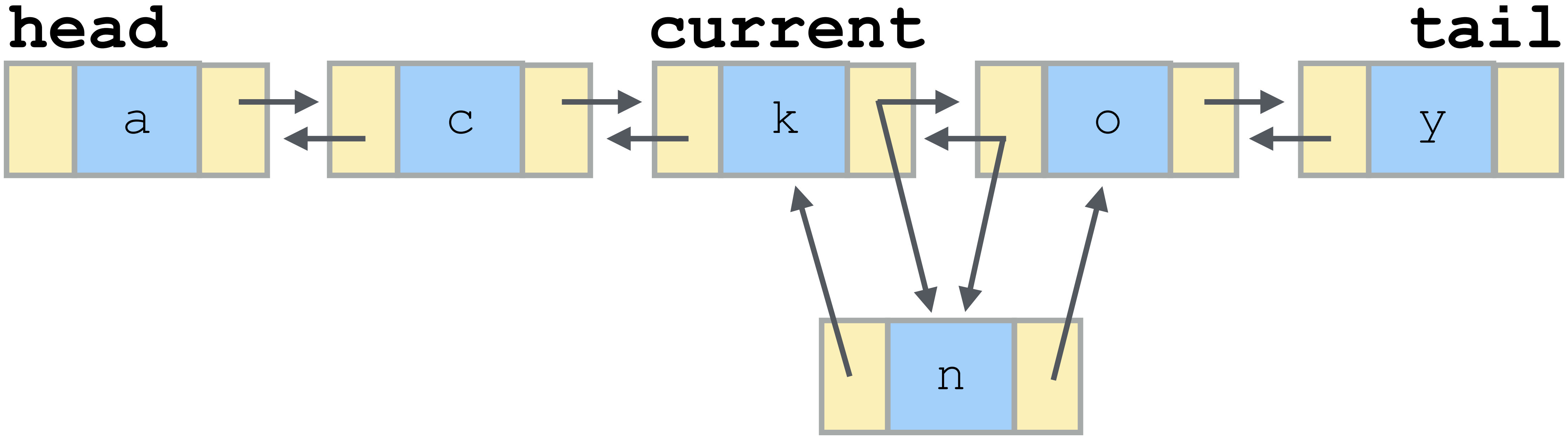
-special cases (empty or single-item lists) are more tricky due to managing `tail` as well as `head`

-what are the values of `head` and `tail` for any empty list?

-what about for a single-item list?

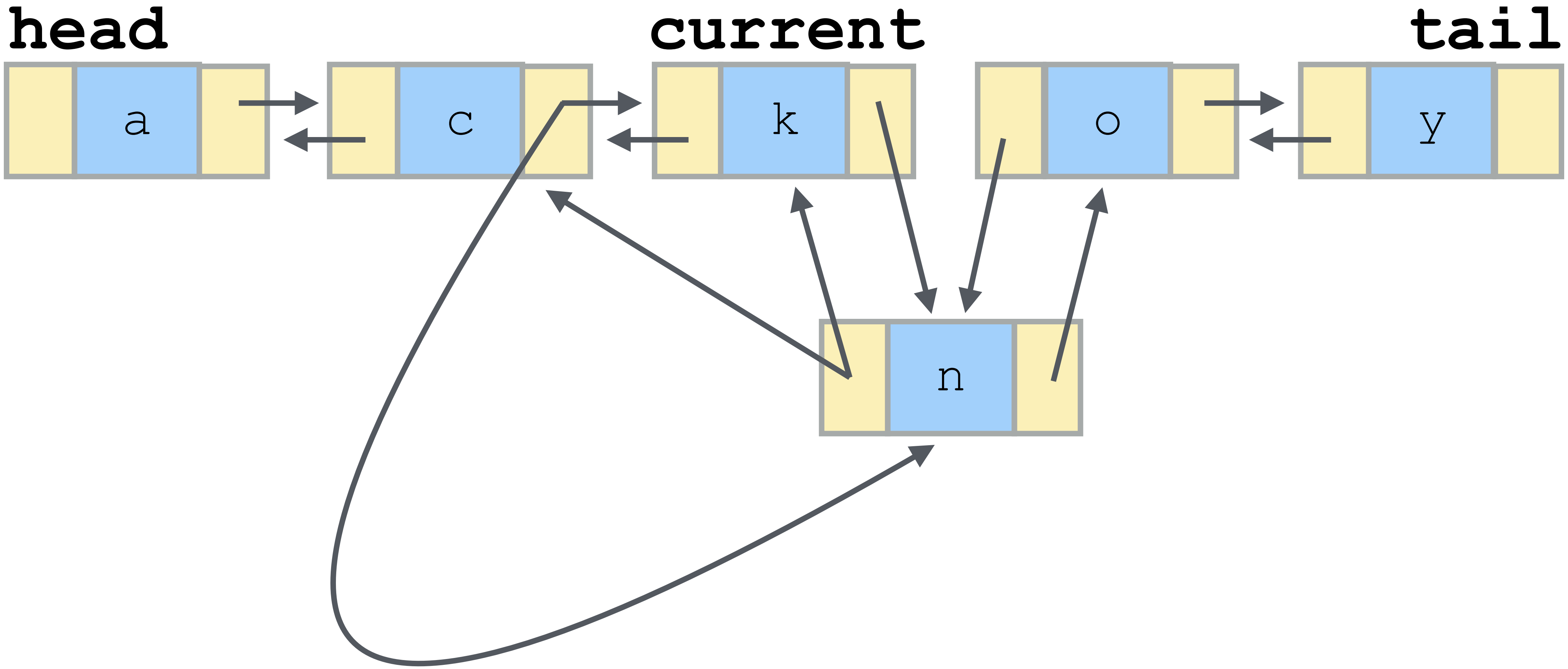
doubly-linked list insertion:

```
newNode = new Node<Character>();
newNode.data = ’n’;

newNode.prev = current;
newNode.next = current.next;
newNode.prev.next = newNode;
newNode.next.prev = newNode;
```



doubly-linked list deletion:

```
current.prev.next = current.next;
current.next.prev = current.prev;
```



generic implementation:

```
class LinkedList<E>
{
  private Node head;
  private Node tail;
  private int size;

  private class Node
  {
    private E data;
    private Node next;
    private Node prev;
    …
  }
  …
}
```

things to consider…

- adding to the front or end of a linked list is a little different than adding somewhere in the middle
    - why?

- removing from a list with 1 node
    - what happens to `head/tail`?

- adding to an empty list
    - what is the current value of `head/tail`?

`LinkedList` VS **`ArrayList`**

`LinkedList` vs `ArrayList`

insertion & deletion: (assuming position is known)	**O(c)**	**O(N)**
accessing a random item:	**O(N)**	**O(c)**

-choose the structure based on the expected use

-what is the common case?

-what if insertion / deletion is always from the front / end?

next time…

- reading
    - chapter 16
    - chapter 2
        - *http://opendatastructures.org/ods-java/*

- homework
    - assignment 5 due tonight
    - assignment 6 is out