ANNOUNCEMENTS
today is **No Laptop Monday**!

WARMUP
"a chain is only as strong as its weakest link"
- what does this expression mean?
- what is a chain?
- what is a link?
- is this true for real-world metal chains? why or why not?

TODAY linked lists

Kahoot review

(map, %,
modifying data while you iterate over it)

```
class Kahoot {
    public static void main(String[] arguments) {
        HashMap<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < 5; ++i) {
            map.put(i, i);
        }
        for (int i = 0; i < 5; ++i) {
            map.put(i, map.get((i + 1) % 5));
        }
        System.out.println(map.get(4));
    }
}
```

linked lists

review: list interface

list interface

```
- // Get the element with this index.
- ElementType get(int index);

- // Add (append) an element to the back of the list.
  void add();
  // Add (insert) an element into the list so it has this index
  void add(int index, ElementType element);

- // Remove (delete) the element in the list at this index.
  void removeByIndex(int index);
  // Remove (delete) the first element with this value.
  void removeByValue(ElementType element);

- // Get the number of elements currently in the list.
  int size();
```

list interface

```
- // Many other functions can be included in the list
  interface.
  - void sort(); // Sort the list.
  - void reverse(); // Reverse the list.

  - List<ElementType> sorted(); // Get sorted copy of the list.
  - List<ElementType> reversed(); // Get reversed copy of list.

  - // Get index of first element with this value.
    int find(ElementType element);

  - ...
```

a few weeks ago, we implemented the list interface using an array

this was the **array list**

now we will implement the list interface using nodes with links to other nodes

this will be the **linked list**

why are we doing this?

it will be cool to see two very different implementations of the same interface ☃️

review: array list

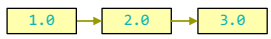
review: accessing an array [list] is O(1)

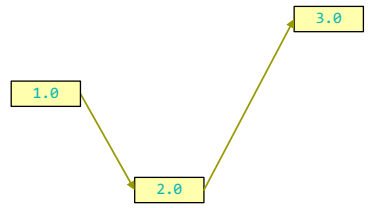
- how Java does `array[4]`, step-by-step:
  - start at the head (0-th element) of the array
    🟦⬛⬛⬛⬛⬛⬛⬛
  - move over 4 slots (using an $\mathcal{O}(1)$ "add")
    ⬛⬛⬛⬛🟦⬛⬛⬛
  - return the value of the element in that slot
    ⬛⬛⬛⬛🟧⬛⬛⬛
- we can't actually see Java do this (it's too "low level")
  - but we *can* see C do it! (stay tuned 😀)

runtime of array list operations

- get() – a single operation that retrieves a value from privateArray
  - O(1)
- set() - a single operation that sets a value in privateArray
  - O(1)
- add() – insert element, shift other elements down to make room in for loop
  - O(1) – if adding to back (and no need to make room)
  - O(n) – if adding to middle
  - O(n) – if adding to any location and we have to grow privateArray (ideally this won't happen often)
- remove() – remove element, shift other elements down to eliminate gap
  - O(n)

linked list





linked list

```
class LinkedList {
    Node head;
}
```

```
class Node {
    ValueType value;
    Node next;
}
```

```
List list = new List();
```

```
list.head = new Node(1.0);
```


```
list.head.next = new Node(2.0);
```


```
list.head.next.next = new Node(3.0);
```

linked list

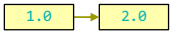
```
class LinkedList {
    Node head;
}
```

```
class Node {
    ValueType value;
    Node next;
}
```

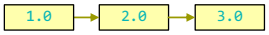
```
List list = new List();
```

```
list.add(1.0);
```

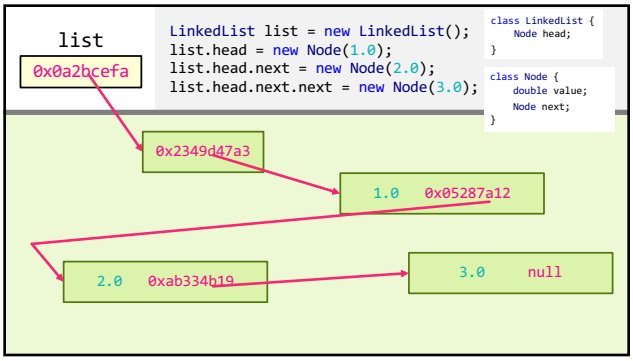

```
list.add(2.0);
```

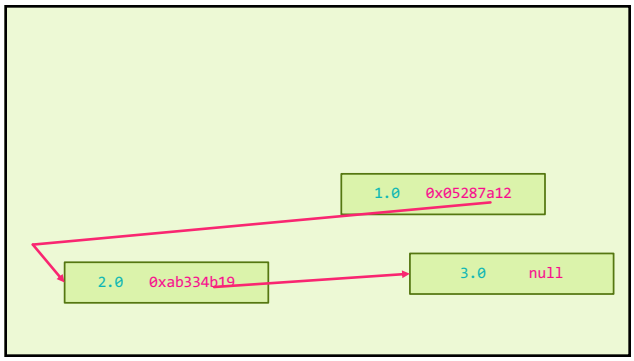



```
list.add(3.0);
```



what does this actually look like in memory?





what does this *mean*?

cons? ☹️

pros? 🙂

linked list implementation

write the usage code first

```
public static void main(String[] arguments) {
    LinkedList list = new LinkedList();
    System.out.println(list.size());
    list.head = new Node("Hans");
    list.head.next = new Node("The");
    list.head.next.next = new Node("Parrot");
    System.out.println(list.size());
}
```

implement `Node` and `LinkedList`

```java
static class LinkedList {
    Node head;
}

static class Node {
    String value;
    Node next;

    Node(String value) {
        this.value = value;
        this.next = null;
    }
}
```

implement `size()`

```java
static class LinkedList {
    Node head;

    int size() {
        int result = 0;
        Node curr = head;
        while (curr != null) {
            ++result;
            curr = curr.next;
        }
        return result;
    }
}
```

all code

```java
class LinkedListExample {

    static class LinkedList {
        Node head;
    }

    static class Node {
        String value;
        Node next;

        Node(String value) {
            this.value = value;
            this.next = null;
        }
    }

    public static void main(String[] arguments) {
        LinkedList list = new LinkedList();
        System.out.println(list.size());
        list.head = new Node("Hans");
        list.head.next = new Node("The");
        list.head.next.next = new Node("Parrot");
        System.out.println(list.size());
    }

}
```

ANNOUNCEMENTS
it's snowing! ❄️

WARMUP
for a linked list, recalculating size() is O(n)
- why?
- how could you make size() (or perhaps... size) O(1)?
-- is this *spooky* in any way? 👻

TODAY linked list big O, garbage collection preview 🚚

size()

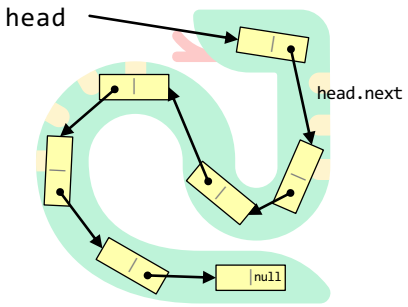
- how would implement this method?
  - start out with...
    - `curr = head`
    - `result = 0`
  - while `curr` not `null`... `++result` and set `curr = curr.next`
- what is the big O runtime of this method?
  - O(n) 😩
- this seems like a pretty steep cost to pay just to know the list's size... what would be a more efficient approach?
  - store size as an instance variable
    - update it every time you change the number of elements in the list (inside of `add`, `remove`, etc.)
- what is the runtime of this approach?
  - O(1) 😀

while way more efficient, this approach is perhaps a bit spooky 👻

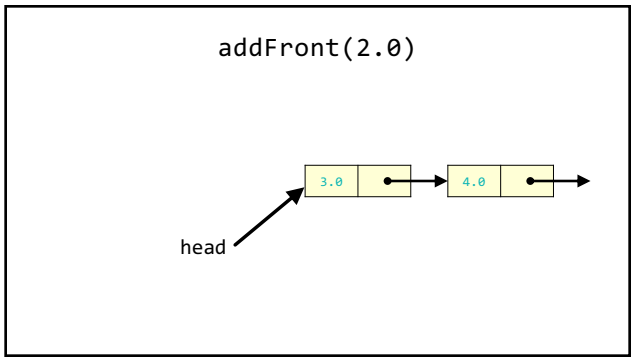
a whole bunch of different functions now have to modify the instance variable size
(mess up, and functions that use size will be very weirdly broken)

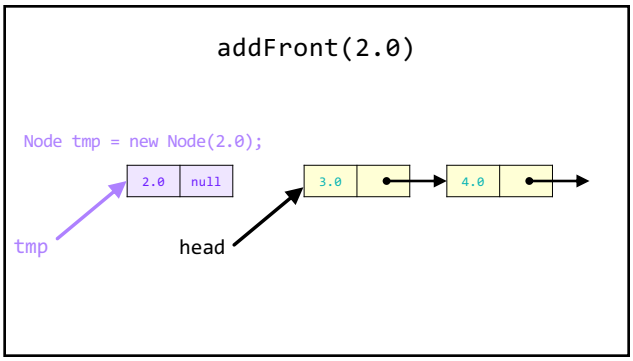
maybe... start out with size() as a function, get everything working perfectly, and then (very carefully) rewrite size as an instance variable
note: none of the homework problems require you to know the size of the list

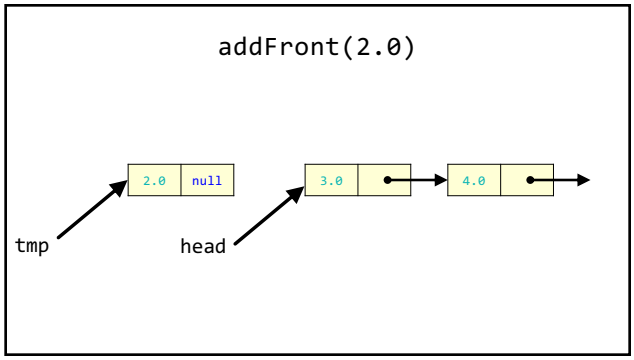
link lsssssst

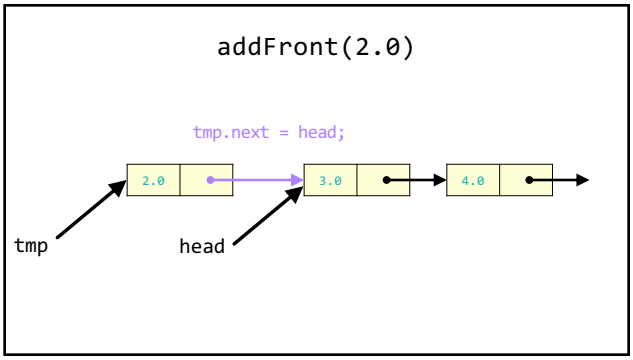


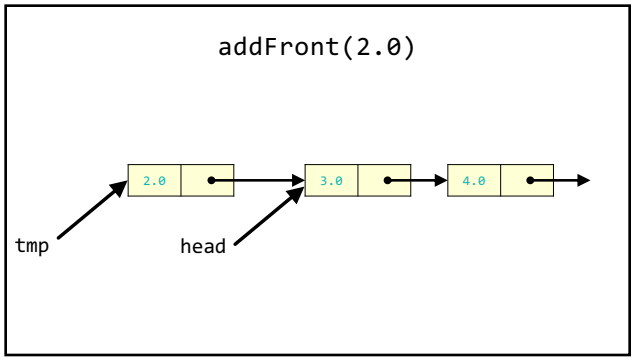
`addFront(value) // add(0, value)`

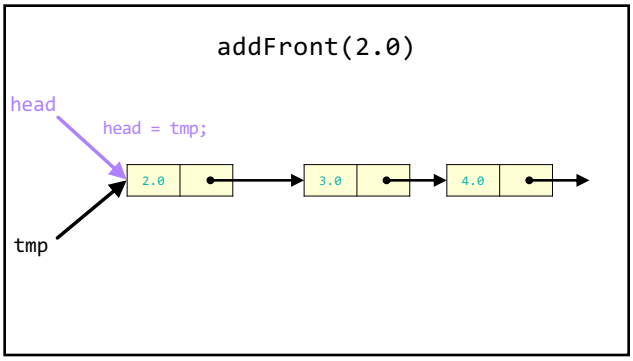

addFront(2.0)
3.0
4.0
head


addFront(2.0)
Node tmp = new Node(2.0);
2.0
null
3.0
4.0
tmp
head

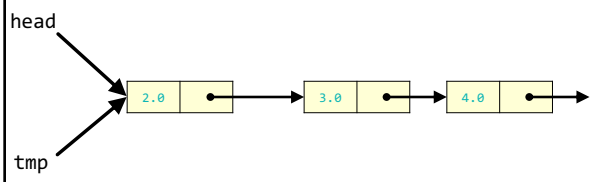

addFront(2.0)
2.0
null
3.0
4.0
tmp
head


addFront(2.0)
tmp.next = head;
2.0
3.0
4.0
tmp
head

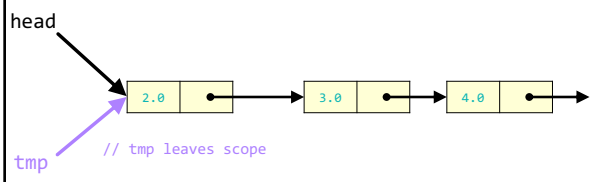

addFront(2.0)
2.0
3.0
4.0
tmp
head


addFront(2.0)
head
head = tmp;
2.0
3.0
4.0
tmp

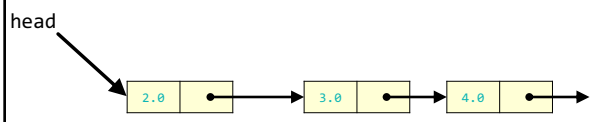
addFront(2.0)



addFront(2.0)



addFront(2.0)

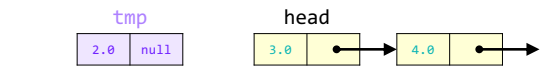


same thing but with
labels instead of arrows
for head and tmp

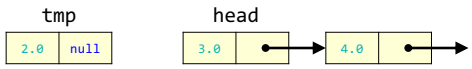
addFront(2.0)

addFront(2.0)

```
Node tmp = new Node(2.0);
```

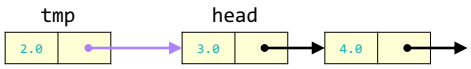


addFront(2.0)

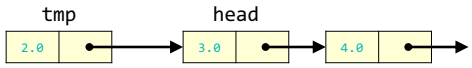


addFront(2.0)

tmp.next = head;

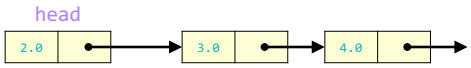


addFront(2.0)

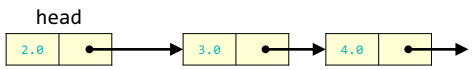


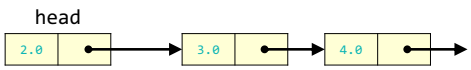
addFront(2.0)

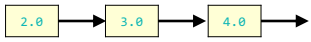
head = tmp;



addFront(2.0)





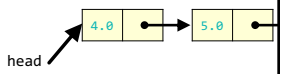


(even) more abstract diagram

example to try on paper

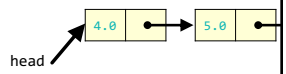
example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



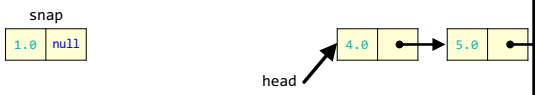
example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



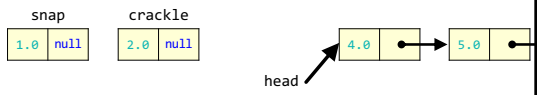
example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



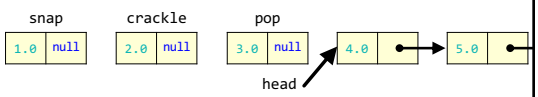
example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



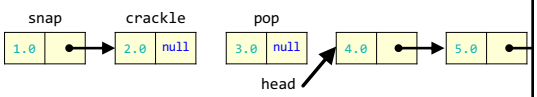
example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



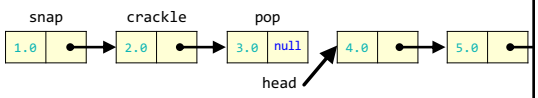
example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



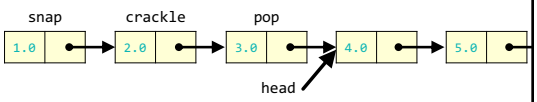
example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



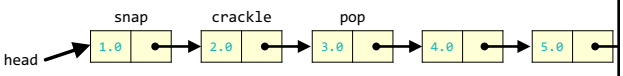
example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



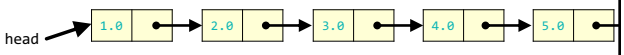
example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



was that faster than calling addFront() over and over?

same big O or different big O?

was that faster than calling addFront() over and over?

yes

(only updated head once)

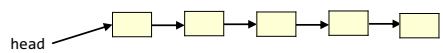
same big O or different big O?

same

(still have to "hook up" O(n) references)

doubly-linked list

singly-linked list



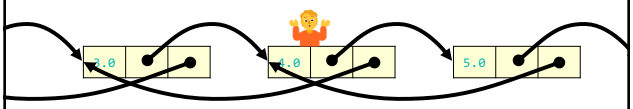

```
class LinkedList {
    Node head;
}
```

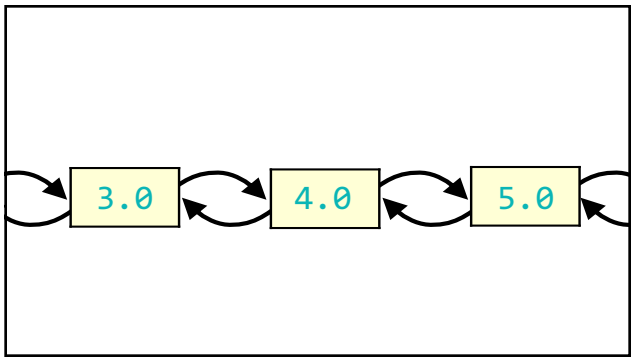
```
class Node {
    double value;
    Node next;
}
```

doubly-linked list


```
class LinkedList2 {
    Node2 head;
    Node2 tail;
}
```

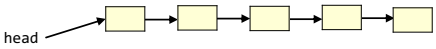
```
class Node2 {
    double value;
    Node2 next;
    Node2 prev;
}
```





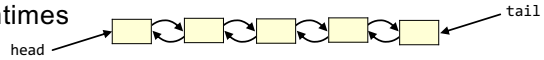
runtimes

singly-linked list runtimes



- `list.add(index, value)`
  - `// O(n)`
- `list.removeByIndex(index)`
  - `// O(n)`
- `list.size()`
  - `// O(n)`
- `list.addFront() // list.add(0, value)`
  - `// O(1)`
- `list.removeFront() // list.removeByIndex(0)`
  - `// O(1)`
- `list.addBack()`
  - `// O(n)`
- `list.removeBack()`
  - `// O(n)`

doubly-linked list runtimes



- `list.add(index, value)`
  - `// O(n)`
- `list.removeByIndex(index)`
  - `// O(n)`
- `list.size()`
  - `// O(n)`
- `list.addFront()`
  - `// O(1)`
- `list.removeFront()`
  - `// O(1)`
- `list.addBack()`
  - `// O(1)`
- `list.removeBack()`
  - `// O(1)`

a doubly-linked list is a great way to implement a **deque** (**double-ended queue**)

- `O(1) addFront()`
  `O(1) removeFront()`
  `O(1) addBack()`
  `O(1) removeBack()`

- could you pull this off with an **array list**?
  - no.
    - addFront() is O(n)

- could you pull this off with an **array**?
  - actually **yes**! (at least if you mean amortized runtime – resizing is O(n))

array deque

(implementing a double-ended queue using an array)

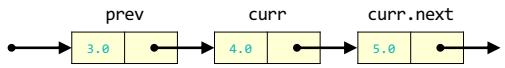
- **just one idea:** store list in the *middle* of the array

```
[ null, null, null, null, null, null, null, null, null, null ]
                                                   // addFront(1.0)
[ null, null, null, null,  1.0, null, null, null, null, null ]
                                                   // addFront(2.0)
[ null, null, null,  2.0,  1.0, null, null, null, null, null ]
                                                   // addFront(3.0)
[ null, null,  3.0,  2.0,  1.0, null, null, null, null, null ]
                                                   // addBack(4.0)
[ null, null,  3.0,  2.0,  1.0,  4.0, null, null, null, null ]
                                                   // removeFront()
[ null, null, null,  2.0,  1.0,  4.0, null, null, null, null ]
```

- could also treat the array as circular (using % or `floorMod`)

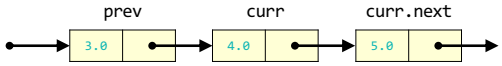
remove

remove

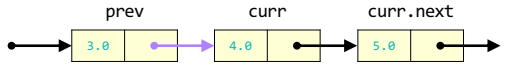



```
Node prev = null; // the previous node
Node curr = head; // the current node
while (...) {
    ...
    prev = curr;
    curr = curr.next;
}
```

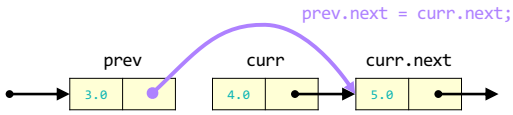
remove



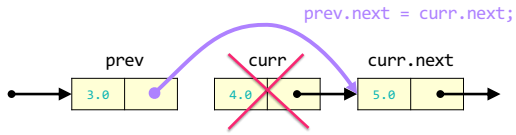
remove



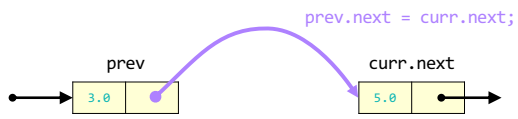
remove



remove

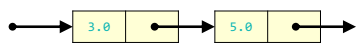


remove



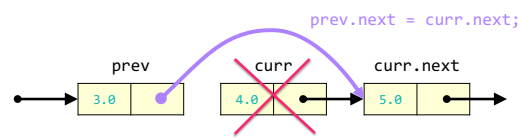
remove

remove



where did the middle node go?

- way back when...we created it using `new`
- then we just like..."stopped referring to it"
- is it gone now? 🤔



answer: yes / sorta-kinda
(at least in Java)

switch to Windows Laptop

- garbage collection motivation
- extra time:
  MD5 hash (review from last week)

ANNOUNCEMENTS ☃️

colloquium (alumni panel) today!
juggling club after colloquium!

WARMUP

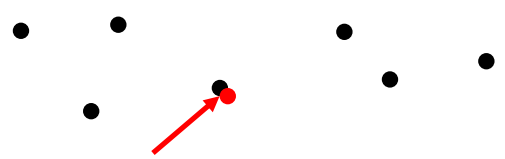
imagine you have n circles
how do you answer the querstion "are there any collisions?"
what is the worst case big O runtime

TODAY review, memory 🚚, memory in C

quadtrees preview

problem: collision detection

consider n dots (bullets, orcs, etc.) that can all collide with each other...

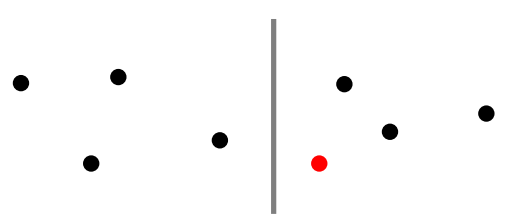


naive collision detection is $\mathcal{O}(n^2)$ ☹️

can we do better?

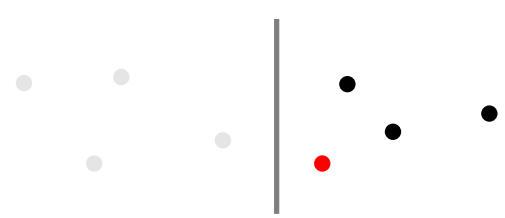
problem: collision detection

consider n dots (bullets, orcs, etc.) that can all collide with each other...

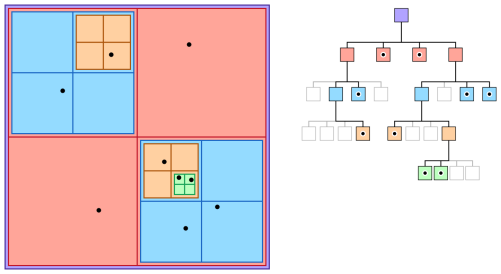


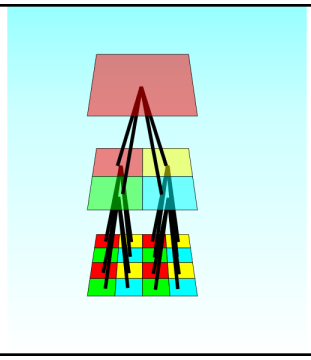
problem: collision detection

consider n dots (bullets, orcs, etc.) that can all collide with each other...



yes 🙂 this is called a quadtree





james-bern.github.io

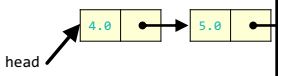
- open the Best Stack Overflow Answer of All Time
- **answer...**
  - what does user4842163 do for work?
  - what is their test case?
  - how fast is their fastest algorithm?
  - does their algorithm have any interesting features?

review
prepend example
& array deque

review: prepending a list

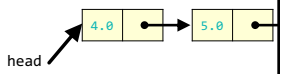
example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



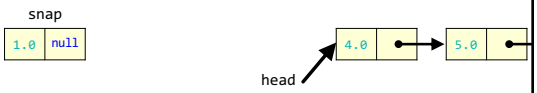
example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



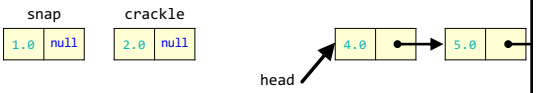
example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



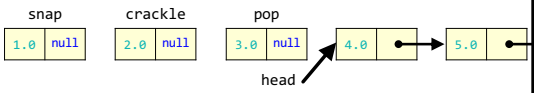
example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



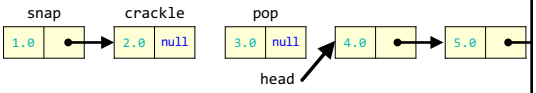
example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



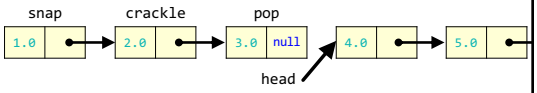
example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



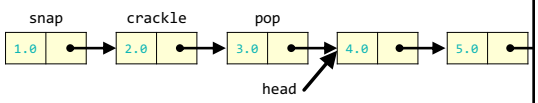
example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



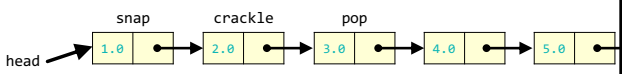
example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



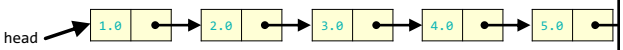
example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



example

```
Node snap = new Node(1.0);
Node crackle = new Node(2.0);
Node pop = new Node(3.0);
snap.next = crackle;
crackle.next = pop;
pop.next = head;
head = snap;
```



review:
double-ended queue implementations

a doubly-linked list is a great way to implement a **deque** (**double-ended queue**)

- O(1) addFront()
  O(1) removeFront()
  O(1) addBack()
  O(1) removeBack()

- could you pull this off easily with Java's **ArrayList**?
  - no.
    - addFront() and removeFront() are both O(n)
      (we have to "move everything else over")

- could you pull this off with just an **array** (or your own special array list)?
  - actually **yes**! (at least if you mean amortized runtime – resizing is O(n))

array deque

(implementing a double-ended queue u TODO: replace line with nextFront and nextBack

- **just one idea:** store list in the *middle* of the array

```
[ null, null, null, null, null, null, null, null, null, null ]
                                                    // addBack(1.0)
[ null, null, null, null,  1.0, null, null, null, null, null ]
                                                    // addBack(2.0)
[ null, null, null,  2.0,  1.0, null, null, null, null, null ]
                                                    // addBack(3.0)
[ null, null,  3.0,  2.0,  1.0, null, null, null, null, null ]
                                                    // addFront(4.0)
[ null, null,  3.0,  2.0,  1.0,  4.0, null, null, null, null ]
                                                    // removeBack()
[ null, null, null,  2.0,  1.0,  4.0, null, null, null, null ]
```

- could also treat the array as circular (using % or `floorMod`)

review: type

variables in Java have a specified type

```
# okay in Python
foo = 7     # foo is an int
foo = False # now, foo is a boolean
```

```
// NOT okay in Java
int foo = 7; // foo is an int
foo = false; // Error: incompatible types:
             // boolean cannot be converted to int
```

in Java, declaring and initializing variables are separate things

```
- // Option A: two lines
  int foo; // declare a variable foo of type int
  foo = 7; // initialize foo to 7

- // Option B: one line
  int foo = 7; // declare int foo and initialize it to 7
```

memory

note: i am intentionally breaking these examples into many steps to teach you about memory

don't actually code this way!

```
int foo = 7; // <-- Good!
```

"the stack"

local variable primitives & **references** to Objects

undefined *by default (will NOT compile if used)*

"the heap"

the actual **Objects** (arrays and String's count as Objects)

cleared to 0 or null by default

basic example

int a is a primitive

int[] b is a reference to an array

"the stack"

local variable primitives & **references** to Objects

undefined *by default (will NOT compile if used)*

"the heap"

the actual **Objects** (arrays and String's count as Objects)

cleared to 0 or null by default

```
{
    int a = 7;

    int[] b = new int[5];
    b[0] = 3;
}
```

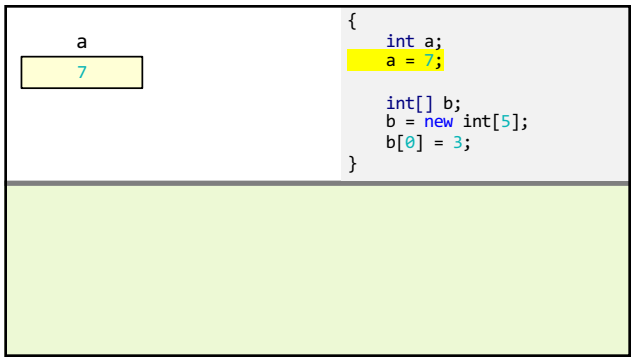
```
{
    int a;
    a = 7;

    int[] b;
    b = new int[5];
    b[0] = 3;
}
```

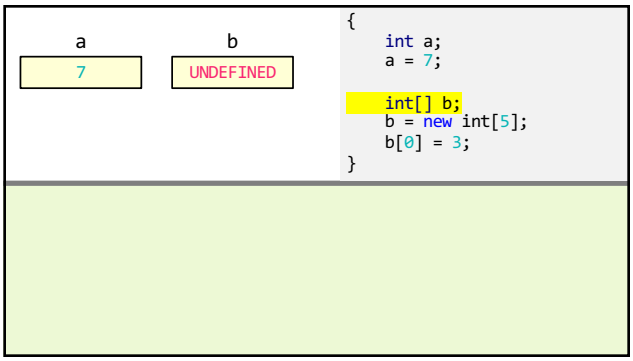
```
{
    int a;
    a = 7;

    int[] b;
    b = new int[5];
    b[0] = 3;
}
```

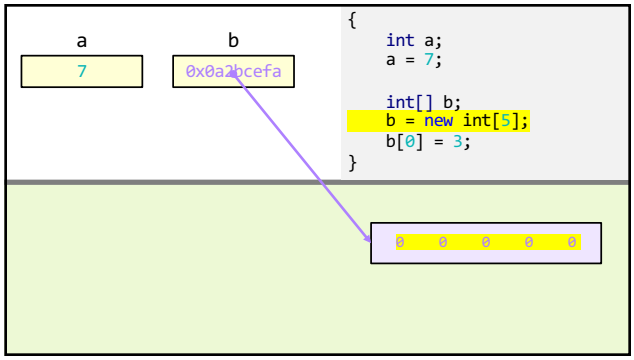
a

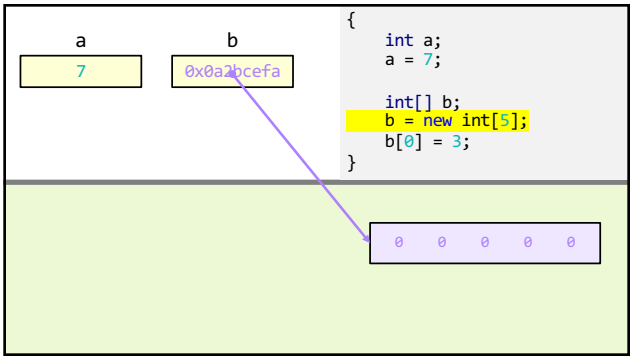
UNDEFINED

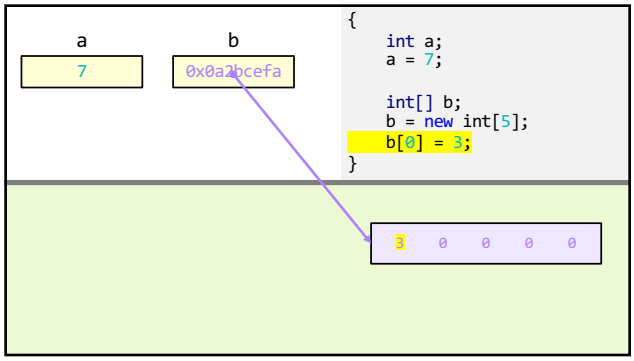
```
{
    int a;
    a = 7;

    int[] b;
    b = new int[5];
    b[0] = 3;
}
```

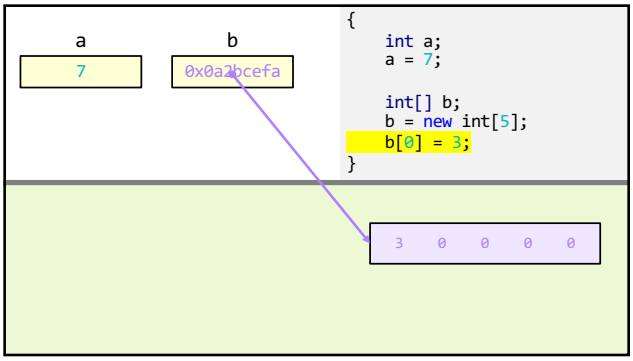

a
7
{
int a;
a = 7;
int[] b;
b = new int[5];
b[0] = 3;
}


a
7
b
UNDEFINED
{
int a;
a = 7;
int[] b;
b = new int[5];
b[0] = 3;
}


a
7
b
0x0a2bcefa
{
int a;
a = 7;
int[] b;
b = new int[5];
b[0] = 3;
}
0 0 0 0 0


a
7
b
0x0a2bcefa
{
int a;
a = 7;
int[] b;
b = new int[5];
b[0] = 3;
}
0 0 0 0 0


a
7
b
0x0a2bcefa
{
int a;
a = 7;
int[] b;
b = new int[5];
b[0] = 3;
}
3 0 0 0 0


a
7
b
0x0a2bcefa
{
int a;
a = 7;
int[] b;
b = new int[5];
b[0] = 3;
}
3 0 0 0 0

```
{
    int a;
    a = 7;

    int[] b;
    b = new int[5];
    b[0] = 3;
}
```

3	0	0	0	0

```
{
    int a;
    a = 7;

    int[] b;
    b = new int[5];
    b[0] = 3;
}
```

3	0	0	0	0

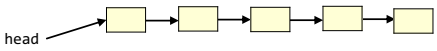
```
{
    int a;
    a = 7;

    int[] b;
    b = new int[5];
    b[0] = 3;
}
```

3	0	0	0	0

```
{
    int a;
    a = 7;

    int[] b;
    b = new int[5];
    b[0] = 3;
}
```

linked list (from HW-08) example

singly-linked list




```
class LinkedList {
    Node head;
}
```

```
class Node {
    String value;
    Node next;
}
```

"the stack"

local variable primitives & **references** to Objects

undefined *by default (will NOT compile if used)*

"the heap"

the actual **Objects** (arrays and String's count as Objects)

cleared to 0 or null by default

```
LinkedList list =
    new LinkedList("Hans -> Parrot");
```

```
LinkedList list;
{
    Node node;
    String string;

    list = new LinkedList();

    node = new Node();
    string = "Hans";
    node.value = string;
    list.head = node;

    node = new Node();
    string = "Parrot";
    node.value = string;
    list.head.next = node;
}
```

list

UNDEFINED

```
LinkedList list;
{
    Node node;
    String string;

    list = new LinkedList();

    node = new Node();
    string = "Hans";
    node.value = string;
    list.head = node;

    node = new Node();
    string = "Parrot";
    node.value = string;
    list.head.next = node;
}
```

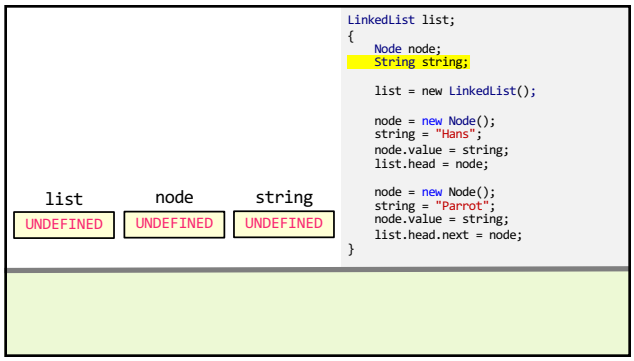
list

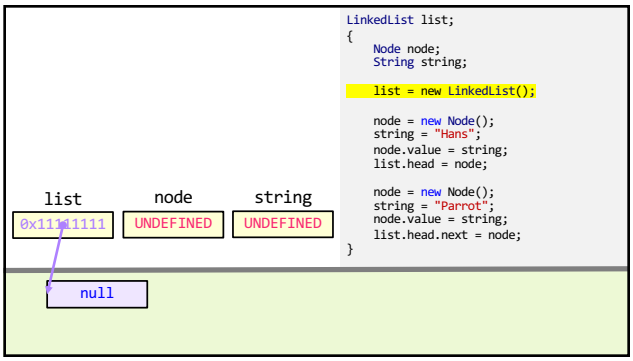
UNDEFINED

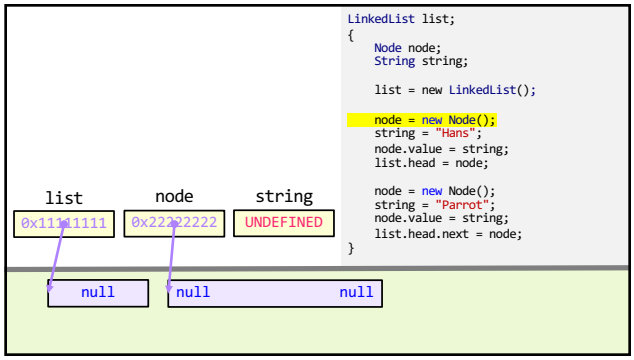
node

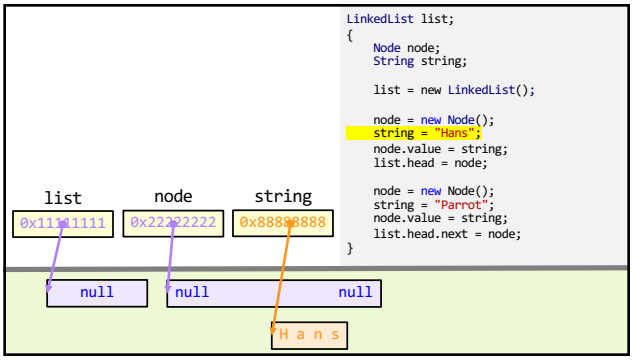
UNDEFINED

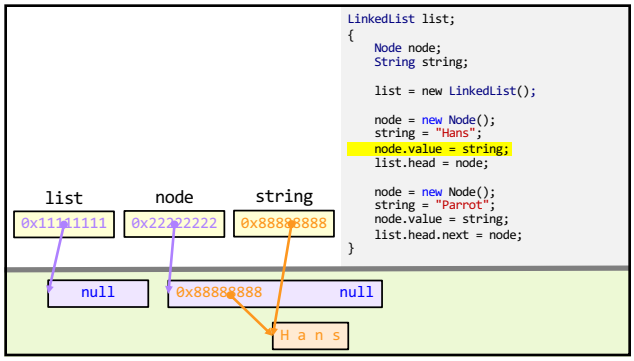
```
LinkedList list;
{
    Node node;
    String string;

    list = new LinkedList();

    node = new Node();
    string = "Hans";
    node.value = string;
    list.head = node;

    node = new Node();
    string = "Parrot";
    node.value = string;
    list.head.next = node;
}
```

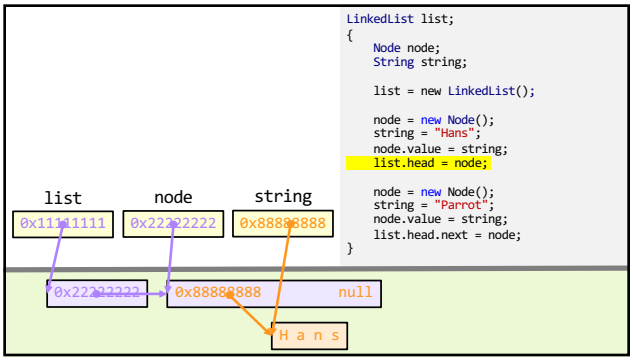

list
node
string
UNDEFINED
UNDEFINED
UNDEFINED
LinkedList list;
{
Node node;
String string;
list = new LinkedList();
node = new Node();
string = "Hans";
node.value = string;
list.head = node;
node = new Node();
string = "Parrot";
node.value = string;
list.head.next = node;
}

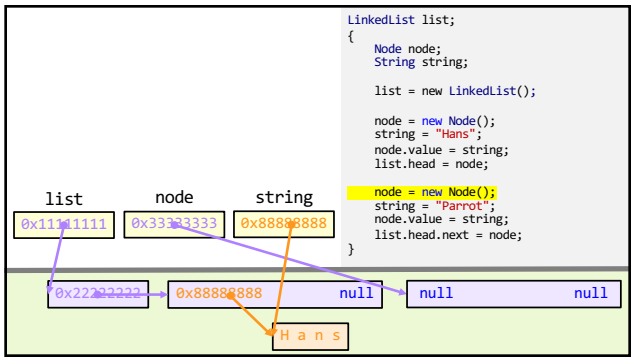

list
node
string
0x11111111
UNDEFINED
UNDEFINED
null
LinkedList list;
{
Node node;
String string;
list = new LinkedList();
node = new Node();
string = "Hans";
node.value = string;
list.head = node;
node = new Node();
string = "Parrot";
node.value = string;
list.head.next = node;
}

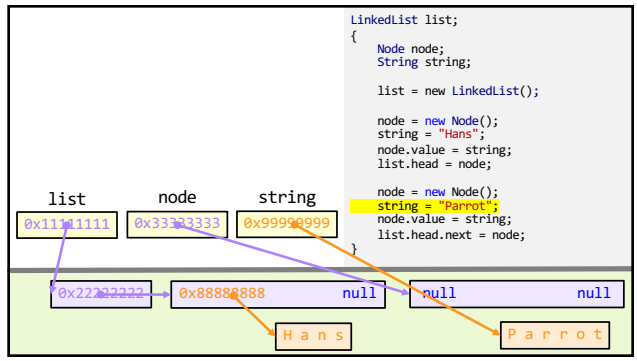

list
node
string
0x11111111
0x22222222
UNDEFINED
null
null
null
LinkedList list;
{
Node node;
String string;
list = new LinkedList();
node = new Node();
string = "Hans";
node.value = string;
list.head = node;
node = new Node();
string = "Parrot";
node.value = string;
list.head.next = node;
}

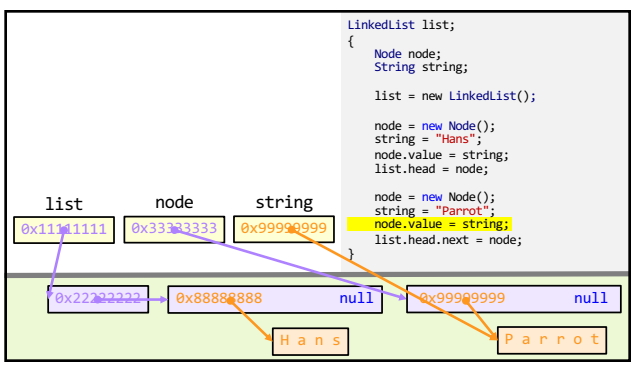

list
node
string
0x11111111
0x22222222
0x88888888
null
null
null
H a n s
LinkedList list;
{
Node node;
String string;
list = new LinkedList();
node = new Node();
string = "Hans";
node.value = string;
list.head = node;
node = new Node();
string = "Parrot";
node.value = string;
list.head.next = node;
}

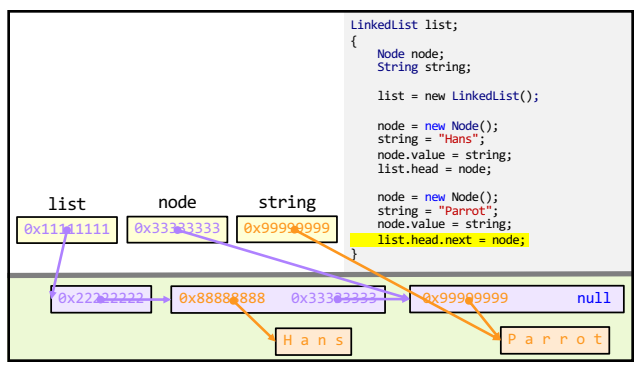

list
node
string
0x11111111
0x22222222
0x88888888
null
0x88888888
null
H a n s
LinkedList list;
{
Node node;
String string;
list = new LinkedList();
node = new Node();
string = "Hans";
node.value = string;
list.head = node;
node = new Node();
string = "Parrot";
node.value = string;
list.head.next = node;
}

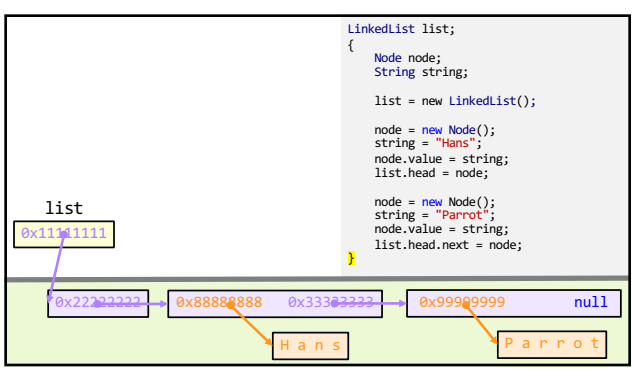

list
node
string
0x11111111
0x22222222
0x88888888
0x22222222
0x88888888
null
H a n s
LinkedList list;
{
Node node;
String string;
list = new LinkedList();
node = new Node();
string = "Hans";
node.value = string;
list.head = node;
node = new Node();
string = "Parrot";
node.value = string;
list.head.next = node;
}

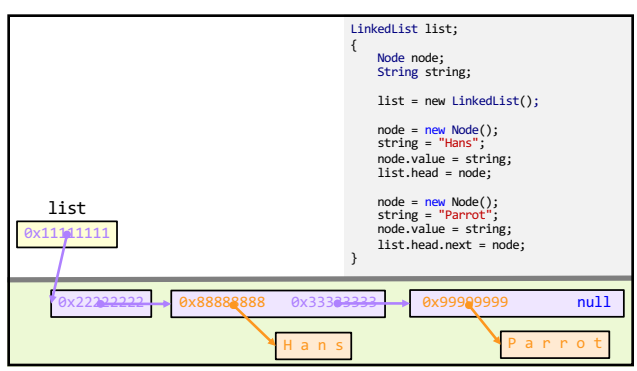

LinkedList list;
{
Node node;
String string;
list = new LinkedList();
node = new Node();
string = "Hans";
node.value = string;
list.head = node;
node = new Node();
string = "Parrot";
node.value = string;
list.head.next = node;
}
list
node
string
0x11111111
0x33333333
0x88888888
0x22222222
0x88888888
null
null
null
Hans


LinkedList list;
{
Node node;
String string;
list = new LinkedList();
node = new Node();
string = "Hans";
node.value = string;
list.head = node;
node = new Node();
string = "Parrot";
node.value = string;
list.head.next = node;
}
list
node
string
0x11111111
0x33333333
0x99999999
0x22222222
0x88888888
null
null
null
Hans
Parrot


LinkedList list;
{
Node node;
String string;
list = new LinkedList();
node = new Node();
string = "Hans";
node.value = string;
list.head = node;
node = new Node();
string = "Parrot";
node.value = string;
list.head.next = node;
}
list
node
string
0x11111111
0x33333333
0x99999999
0x22222222
0x88888888
null
0x99999999
null
Hans
Parrot


LinkedList list;
{
Node node;
String string;
list = new LinkedList();
node = new Node();
string = "Hans";
node.value = string;
list.head = node;
node = new Node();
string = "Parrot";
node.value = string;
list.head.next = node;
}
list
node
string
0x11111111
0x33333333
0x99999999
0x22222222
0x88888888
0x33333333
0x99999999
null
Hans
Parrot


LinkedList list;
{
Node node;
String string;
list = new LinkedList();
node = new Node();
string = "Hans";
node.value = string;
list.head = node;
node = new Node();
string = "Parrot";
node.value = string;
list.head.next = node;
}
list
0x11111111
0x22222222
0x88888888
0x33333333
0x99999999
null
Hans
Parrot


LinkedList list;
{
Node node;
String string;
list = new LinkedList();
node = new Node();
string = "Hans";
node.value = string;
list.head = node;
node = new Node();
string = "Parrot";
node.value = string;
list.head.next = node;
}
list
0x11111111
0x22222222
0x88888888
0x33333333
0x99999999
null
Hans
Parrot

memory in C

in C, we often get to choose whether to allocate memory on the stack or the heap

memory allocation

```
// Java
int[] heapAllocatedArray = new int[10]
heapAllocatedArray[0] = 3;
```

```
// C
int stackAllocatedArray[10];
int *heapAllocatedArray = malloc(10000000 * sizeof(int));

stackAllocatedArray[0] = 3;
heapAllocatedArray[0] = 3;

free(heapAllocatedArray);
```