

LEC 05

CSE 123

Linked Nodes

Questions during Class?
Raise hand or send here

sli.do #cse123



BEFORE WE START


Talk to your neighbors:

Did you see / light any fireworks last night? If not, what did you do with your break?

Music: [123 24su Lecture Tunes](#) 

Instructor: Joe Spaniac**TAs:** Andras Eric Sahej Zach
Daniel Nicole Trien


Lecture Outline

- **Announcements** 
- Contiguous vs. Non-contiguous memory
- Reference Semantics
 - Trains cont.
- Linked Nodes
 - `ListNode` class
 - Iterating over `ListNode`s

Announcements

- Resubmission Period 1 due tonight (7/5) at 11:59pm
 - Submit the assignment again, mark your new attempt final, fill out the linked google form
- Creative Project 2 is out, due Wednesday (7/10) at 11:59pm
 - Generally regarded as a fun one!
- Resubmission Period 2 opening tonight, due next Friday (7/12)
 - Assignments available: C1, P1
- Check-in 2 next Thursday (7/11)
 - Taken in quiz section, should help in preparation for the quiz.
 - Reminder: you only *need* to attend 2 of these for the grade, but they should be beneficial regardless.

Lecture Outline

- Announcements
- **Reference Semantics** 
 - Trains cont.
- Contiguous vs. Non-contiguous memory
- Linked Nodes
 - `ListNode` class
 - Iterating over `ListNode`s

Reference Semantics

- In Java, variables are treated two different ways:

Value Semantics	Reference Semantics
Primitive types (int, double, boolean) + Strings	Object types (int[], Scanner, ArrayList)
Values stored locally	Values stored in memory, reference stored locally
Initialization copies value (many copies of value)	Initialization copies reference (only one value)

```
int x = 10;  
int y = x;  
  
y++; // x remains unchanged
```

```
int[] x = new int[10];  
int[] y = x;  
  
y[0]++; // x[0] changed
```

- We often draw “reference diagrams” to keep track of everything



Reference Semantics

- In Java, variables are treated two different ways:

Value Semantics	Reference Semantics
Primitive types (int, double, boolean) + Strings	Object types (int[], Scanner, ArrayList)
Values stored locally	Values stored in memory, reference stored locally
Initialization copies value (many copies of value)	Initialization copies reference (only one value)

```
int x = 10;  
int y = x;  
  
y++; // x remains unchanged
```

```
int[] x = new int[10];  
int[] y = x;  
  
y[0]++; // x[0] changed
```

- We often draw “reference diagrams” to keep track of everything



Reference Semantics

- In Java, variables are treated two different ways:

Value Semantics	Reference Semantics
Primitive types (int, double, boolean) + Strings	Object types (int[], Scanner, ArrayList)
Values stored locally	Values stored in memory, reference stored locally
Initialization copies value (many copies of value)	Initialization copies reference (only one value)

```
int x = 10;  
int y = x;  
  
y++; // x remains unchanged
```

```
int[] x = new int[10];  
int[] y = x;  
  
y[0]++; // x[0] changed
```

- We often draw “reference diagrams” to keep track of everything



Reference Semantics

- In Java, variables are treated two different ways:

Value Semantics	Reference Semantics
Primitive types (int, double, boolean) + Strings	Object types (int[], Scanner, ArrayList)
Values stored locally	Values stored in memory, reference stored locally
Initialization copies value (many copies of value)	Initialization copies reference (only one value)

```
int x = 10;  
int y = x;  
  
y++; // x remains unchanged
```

```
int[] x = new int[10];  
int[] y = x;  
  
y[0]++; // x[0] changed
```

- We often draw “reference diagrams” to keep track of everything



Reference Semantics

- In Java, variables are treated two different ways:

Value Semantics	Reference Semantics
Primitive types (int, double, boolean) + Strings	Object types (int[], Scanner, ArrayList)
Values stored locally	Values stored in memory, reference stored locally
Initialization copies value (many copies of value)	Initialization copies reference (only one value)

```
int x = 10;  
int y = x;  
  
y++; // x remains unchanged
```

```
int[] x = new int[10];  
int[] y = x;  
  
y[0]++; // x[0] changed
```

- We often draw “reference diagrams” to keep track of everything



Reference Semantics

- In Java, variables are treated two different ways:

Value Semantics	Reference Semantics
Primitive types (int, double, boolean) + Strings	Object types (int[], Scanner, ArrayList)
Values stored locally	Values stored in memory, reference stored locally
Initialization copies value (many copies of value)	Initialization copies reference (only one value)


```
int x = 10;  
int y = x;  
  
y++; // x remains unchanged
```

```
int[] x = new int[10];  
int[] y = x;  
  
y[0]++; // x[0] changed
```


- We often draw “reference diagrams” to keep track of everything



Lecture Outline

- Announcements
- Reference Semantics
 - **Trains cont.** 
- Contiguous vs. Non-contiguous memory
- Linked Nodes
 - `ListNode` class
 - Iterating over `ListNodes`

Lecture Outline

- Announcements
- Reference Semantics
 - Trains cont.
- **Contiguous vs. Non-contiguous memory** 
- Linked Nodes
 - ListNode class
 - Iterating over ListNode

Contiguous vs. Non-contiguous

- Computer memory = one really, *really* big array.

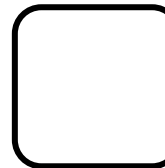
Memory

85	47	-51	44	-38	35	-58	79	27	-14	-9	-36	11	5
-24	-38	-66	-27	36	-1	23	20	31	-40	-97	-59	-4	62
-34	38	37	-52	-15	99	6	68	-67	-58	-85	-15	62	-29
13	-17	-85	-99	-20	-33	54	38	-66	8	9	53	71	39
36	24	27	90	-32	72	-73	11	-85	29	40	80	-77	-79
-90	-64	29	-27	91	64	28	-97	44	59	26	-35	34	21
-68	76	-1	-6	-52	77	21	37	80	69	-34	8	-79	-77
1	-46	-26	99	-24	-98	25	-79	92	-18	14	57	22	20
-76	-5	-86	-64	66	-78	47	-66	69	18	-74	-53	41	-86
-31	-9	90	-53	46	55	85	37	52	58	70	-13	59	79
17	20	91	-55	-74	0	-96	-69	-36	90	45	-60	-95	21

Contiguous vs. Non-contiguous

- Computer memory = one really, *really* big array.
 - `int[] arr = new int[10];`

arr



Memory

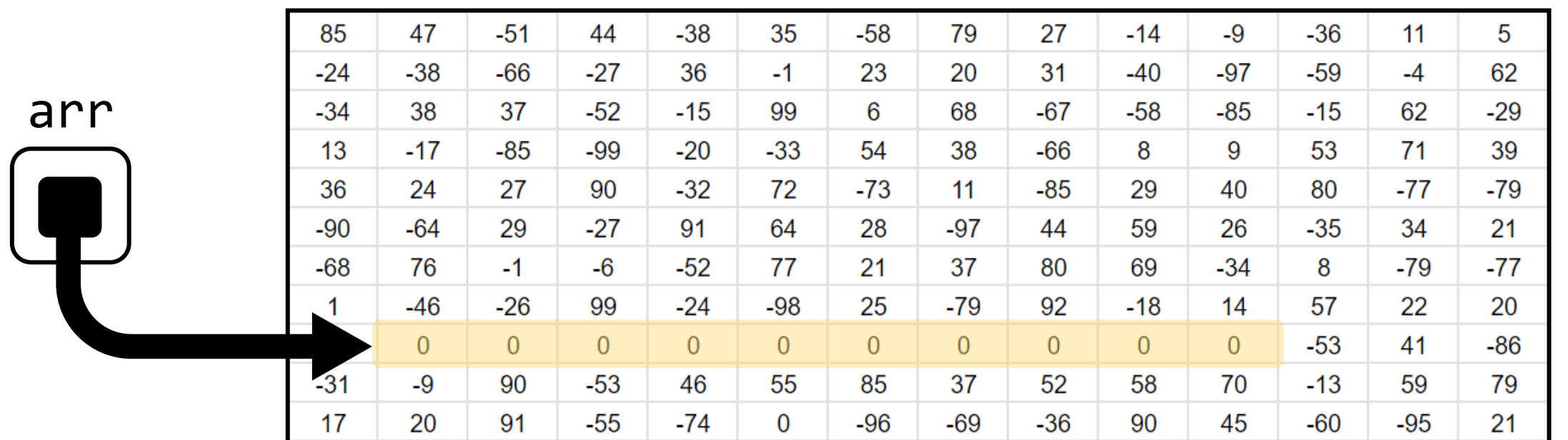
85	47	-51	44	-38	35	-58	79	27	-14	-9	-36	11	5
-24	-38	-66	-27	36	-1	23	20	31	-40	-97	-59	-4	62
-34	38	37	-52	-15	99	6	68	-67	-58	-85	-15	62	-29
13	-17	-85	-99	-20	-33	54	38	-66	8	9	53	71	39
36	24	27	90	-32	72	-73	11	-85	29	40	80	-77	-79
-90	-64	29	-27	91	64	28	-97	44	59	26	-35	34	21
-68	76	-1	-6	-52	77	21	37	80	69	-34	8	-79	-77
1	-46	-26	99	-24	-98	25	-79	92	-18	14	57	22	20
-76	-5	-86	-64	66	-78	47	-66	69	18	-74	-53	41	-86
-31	-9	90	-53	46	55	85	37	52	58	70	-13	59	79
17	20	91	-55	-74	0	-96	-69	-36	90	45	-60	-95	21

Contiguous vs. Non-contiguous

- Computer memory = one really, *really* big array.

- `int[] arr = new int[10];`

Memory



85	47	-51	44	-38	35	-58	79	27	-14	-9	-36	11	5
-24	-38	-66	-27	36	-1	23	20	31	-40	-97	-59	-4	62
-34	38	37	-52	-15	99	6	68	-67	-58	-85	-15	62	-29
13	-17	-85	-99	-20	-33	54	38	-66	8	9	53	71	39
36	24	27	90	-32	72	-73	11	-85	29	40	80	-77	-79
-90	-64	29	-27	91	64	28	-97	44	59	26	-35	34	21
-68	76	-1	-6	-52	77	21	37	80	69	-34	8	-79	-77
1	-46	-26	99	-24	-98	25	-79	92	-18	14	57	22	20
-31	-9	90	-53	46	55	85	37	52	58	70	-13	59	79
17	20	91	-55	-74	0	-96	-69	-36	90	45	-60	-95	21

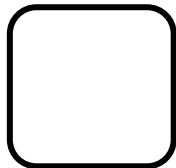
We call this “contiguous” memory

Contiguous vs. Non-contiguous

- Computer memory = one really, *really* big array.
 - EngineCar engine = new EngineCar("Empire Builder", 10, new SleeperCar(10));

Memory

engine



85	47	-51	44	-38	35	-58	79	27	-14	-9	-36	11	5
-24	-38	-66	-27	36	-1	23	20	31	-40	-97	-59	-4	62
-34	38	37	-52	-15	99	6	68	-67	-58	-85	-15	62	-29
13	-17	-85	-99	-20	-33	54	38	-66	8	9	53	71	39
36	24	27	90	-32	72	-73	11	-85	29	40	80	-77	-79
-90	-64	29	-27	91	64	28	-97	44	59	26	-35	34	21
-68	76	-1	-6	-52	77	21	37	80	69	-34	8	-79	-77
1	-46	-26	99	-24	-98	25	-79	92	-18	14	57	22	20
-76	-5	-86	-64	66	-78	47	-66	69	18	-74	-53	41	-86
-31	-9	90	-53	46	55	85	37	52	58	70	-13	59	79
17	20	91	-55	-74	0	-96	-69	-36	90	45	-60	-95	21

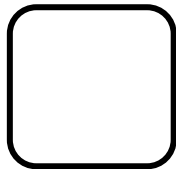
Contiguous vs. Non-contiguous

- Computer memory = one really, *really* big array.

```
- EngineCar engine = new EngineCar("Empire Builder", 10,
                                new SleeperCar(10));
```

Memory

engine



85	47	-51	44	-38	35	-58	79	27	-14	-9	-36	11	5
-24	-38	-66	-27	36	-1	23	20	31	-40	-97	-59	-4	62
-34	38	10	<i>Empire Builder</i>	<i>10</i>			68	-67	-58	-85	-15	62	-29
13	-17	-85	-99	-20	-33	54	38	-66	8	9	53	71	39
36	24	27	90	-32	72	-73	11	-85	29	40	80	-77	-79
-90	-64	29	-27	91	64	28	-97	44	59	26	-35	34	21
-68	76	-1	-6	-52	77	21	37	80	69	-34	8	-79	-77
1	-46	-26	99	-24	-98	25	-79	92	-18	14	57	22	20
-76	-5	-86	-64	66	-78	47	-66	69	18	-74	-53	41	-86
-31	-9	90	-53	46	55	85	37	52	58	70	-13	59	79
17	20	91	-55	-74	0	-96	-69	-36	90	45	-60	-95	21

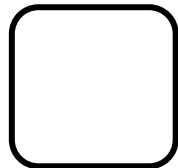
Contiguous vs. Non-contiguous

- Computer memory = one really, *really* big array.

```
- EngineCar engine = new EngineCar("Empire Builder", 10,
                                new SleeperCar(10));
```

Memory

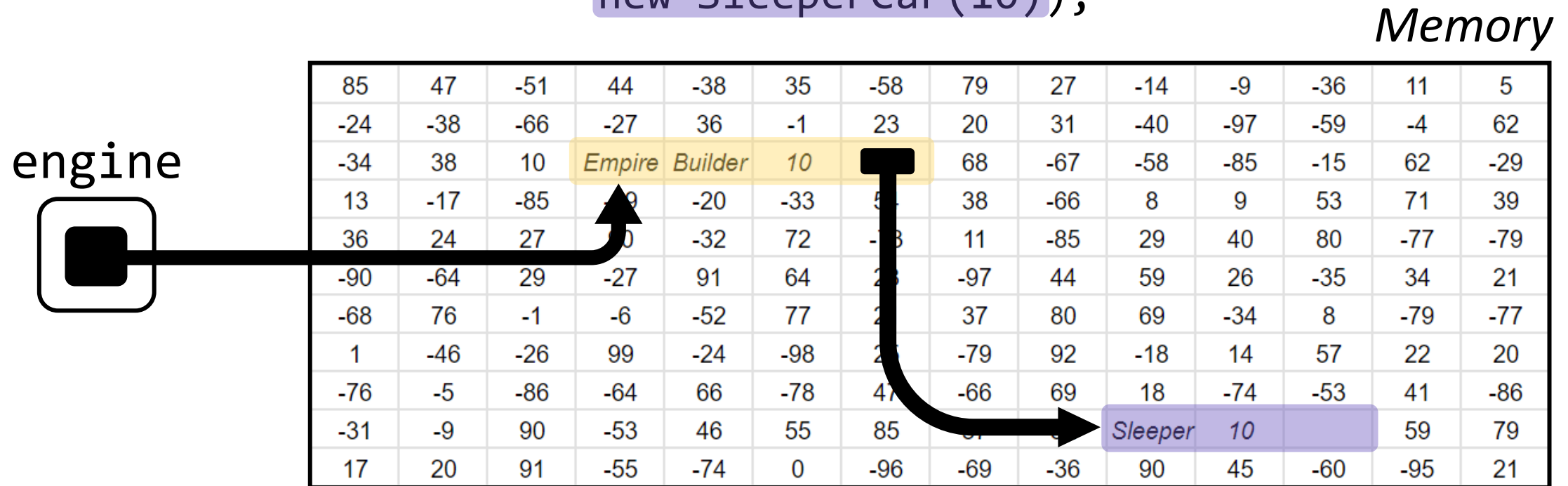
engine



85	47	-51	44	-38	35	-58	79	27	-14	-9	-36	11	5
-24	-38	-66	-27	36	-1	23	20	31	-40	-97	-59	-4	62
-34	38	10	<i>Empire Builder</i>	<i>10</i>			68	-67	-58	-85	-15	62	-29
13	-17	-85	-99	-20	-33	54	38	-66	8	9	53	71	39
36	24	27	90	-32	72	-73	11	-85	29	40	80	-77	-79
-90	-64	29	-27	91	64	28	-97	44	59	26	-35	34	21
-68	76	-1	-6	-52	77	21	37	80	69	-34	8	-79	-77
1	-46	-26	99	-24	-98	25	-79	92	-18	14	57	22	20
-76	-5	-86	-64	66	-78	47	-66	69	18	-74	-53	41	-86
-31	-9	90	-53	46	55	85	37	52	<i>Sleeper</i>	<i>10</i>		59	79
17	20	91	-55	-74	0	-96	-69	-36	90	45	-60	-95	21

Contiguous vs. Non-contiguous

- Computer memory = one really, *really* big array.
 - EngineCar engine = `new EngineCar("Empire Builder", 10, new SleeperCar(10));`




We call this “non-contiguous” memory

Contiguous vs. Non-contiguous

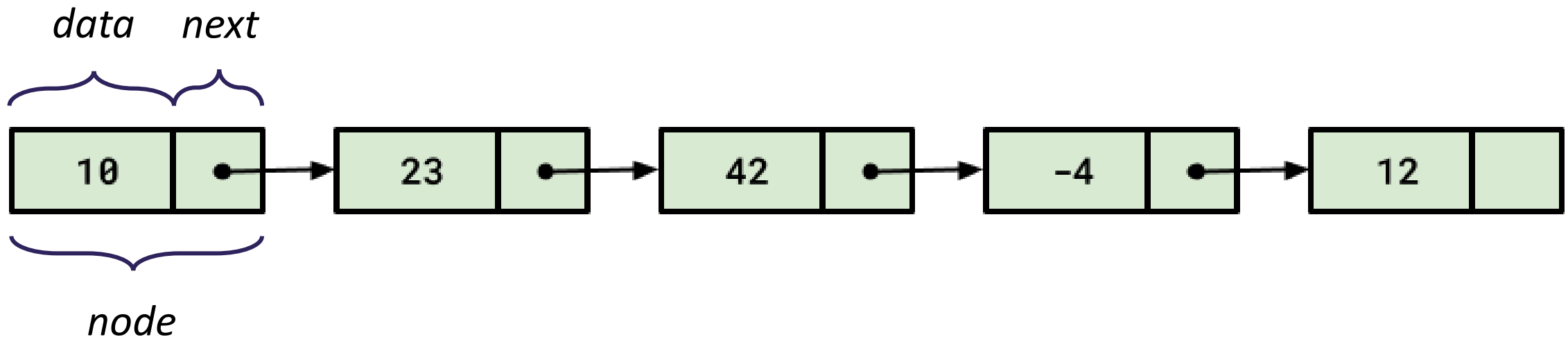
- Computer memory = one really, *really* big array.
- Contiguous memory = impossible to resize directly
 - Surrounding stuff in memory (we can't just overwrite)
 - Best we can manage is get more space and copy
- Non-contiguous memory = easy to resize
 - Just get some more memory and link it to the rest
- Is it possible to create a non-contiguous List implementation?
 - Could make the resizing / shifting problems easier...

Lecture Outline

- Announcements
- Reference Semantics
 - Trains cont.
- Contiguous vs. Non-contiguous memory
- **Linked Nodes** 
 - `ListNode` class
 - Iterating over `ListNode`s

Linked Nodes

- We want to chain together ints “**non-contiguously**”
 - A bunch of train cars where each is responsible for a single integer
- Accomplish this with nodes we link together
 - Each node stores an `int` (*data*) and an reference to the next node (*next*)



ListNode

- Java class representing a “**node**”
- Two fields to store discussed state:
 - Fields are public?! We’ll come back to this
- Why can `ListNode` be a field in the `ListNode` class?

```
public class ListNode {  
    public int data;  
    public ListNode next;  
}
```

Iterating over ListNodes

- General pattern iteration code will follow:

```
ListNode curr = front;
while (curr != null) {
    // Do something

    curr = curr.next;
}
```