CS151 Intro to Data Structures

LinkedLists

Warmup

Create an array of size 3 and fill it with the following values:

- a String of you choice
- an Integer of your code
- a Double of your choice

Loop over the array and print what type each element is

hint: use instanceof

instanceof

- An operator that tests to see if an object is an instance of a specified type
- Every subclass object is an instance of its super class – not true the other way

```
class A {} class B extends A{} class C extends B{}
A[] as = {new A(), new B(), new C()};
for (int i=0; i<as.length; i++) {
  System.out.print((as[i] instanceof A)+ " ");
  System.out.print((as[i] instanceof B)+ " ");
  System.out.println(as[i] instanceof C);
}
```

Announcements

- HW01 and Lab 2 due tonight
- HW02 and Lab 3 released
  - Lab3 is autograded

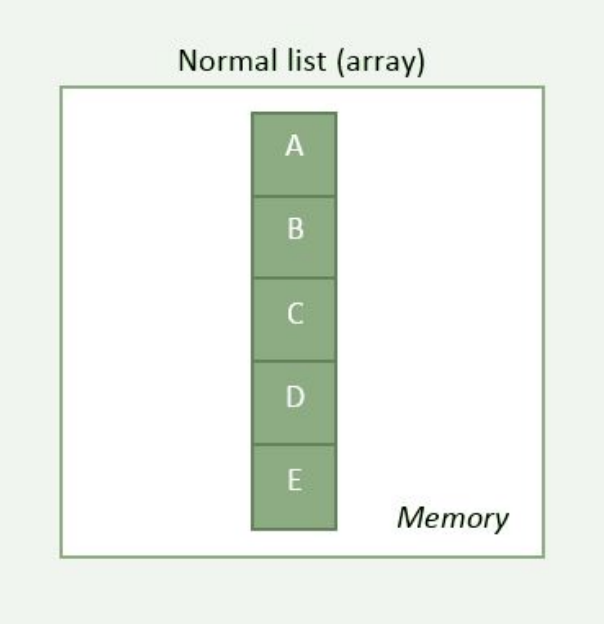
Outline

- LinkedLists
- DoublyLinkedLists

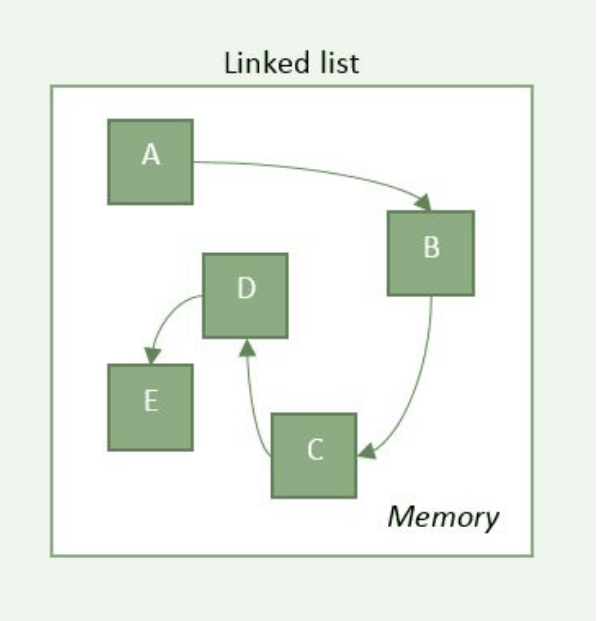
Linked List

List versus Array - memory

An array is a single consecutive piece of memory



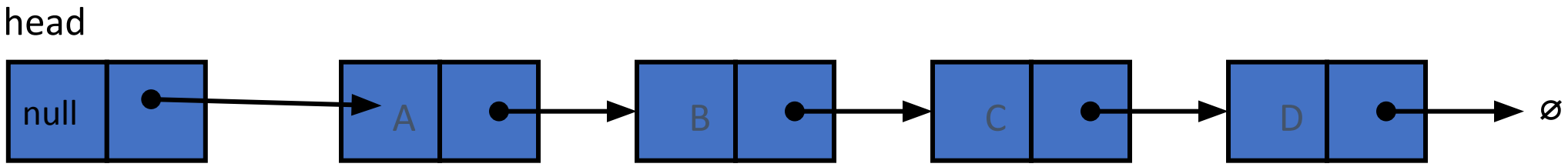
A list can be made of many disjoint pieces



Linked List

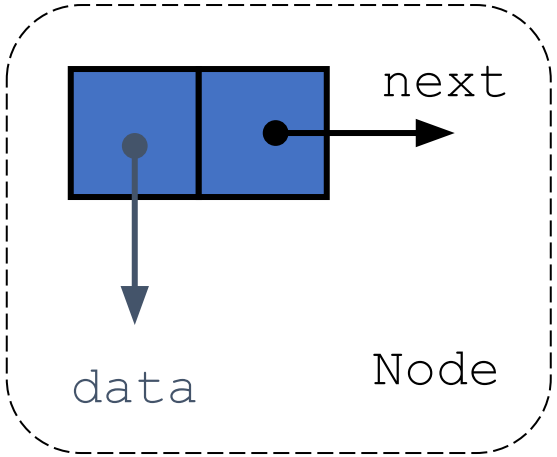
- A linked list is a lists of objects (**nodes**)
- The **nodes** form a linear sequence
- Linked lists are typically unbounded, that is, they can grow infinitely.

node: basic unit that contains data and one or more references or *links* to other nodes.



A node

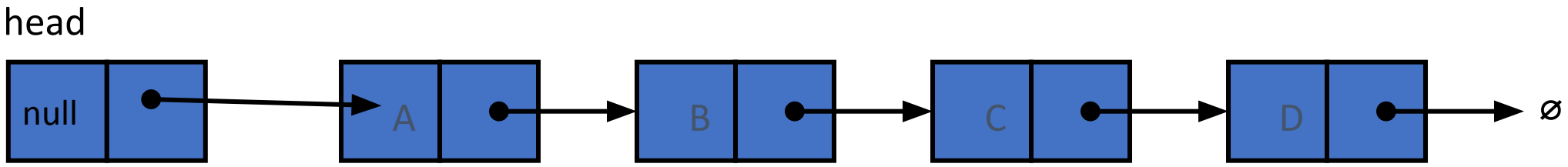
```
public class Node<T> {
   private T data;
   private Node next;
}
```



Linked List

How might we loop over all of the elements of a linked list?

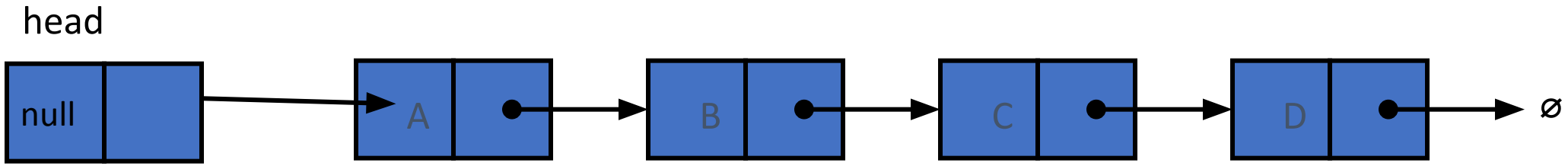
```
public class Node<T> {
  private T data;
  private Node next;
}
```



Linked List Operations

- Access
- Insertion
- Removal

Access Operation



- Check if the head node is what you are looking for
- Iterate through nodes:
  - Stop when found
  - Otherwise return null

Access Operation

Let’s code it

- Computational Complexity?
  - O(n)

Insert Operation

Let's code it

- Computational complexity?
  - Insert at head?
    - O(1)
  - Insert at tail?
    - O(n)
  - Insert at arbitrary location? (middle of list)
    - O(n)

Insert Operation

What if we keep a pointer to the tail?

```
private Node tail;
```

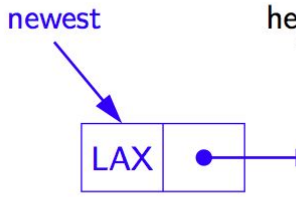
How does this change our insertTail method?

Computational complexity?

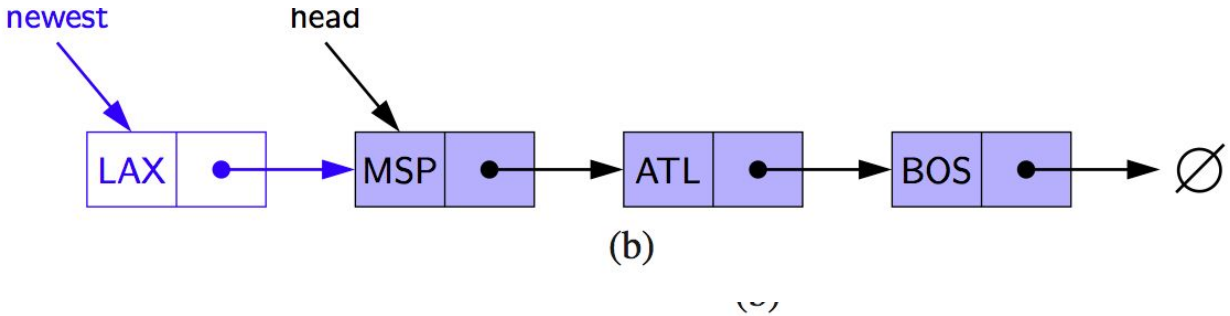
O(1)

Inserting at the Head

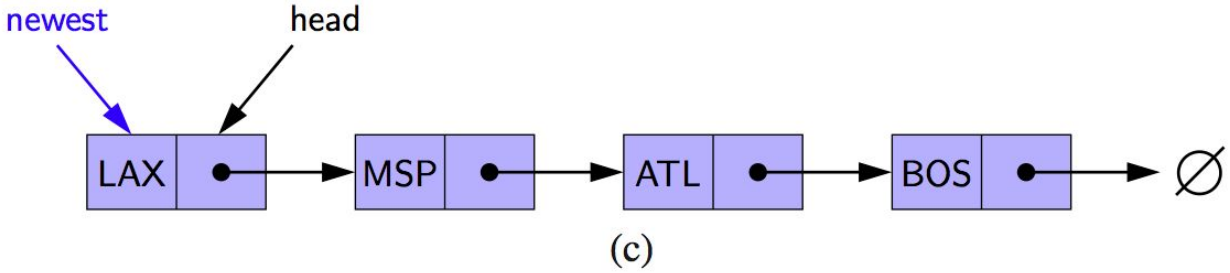
1. create a new node

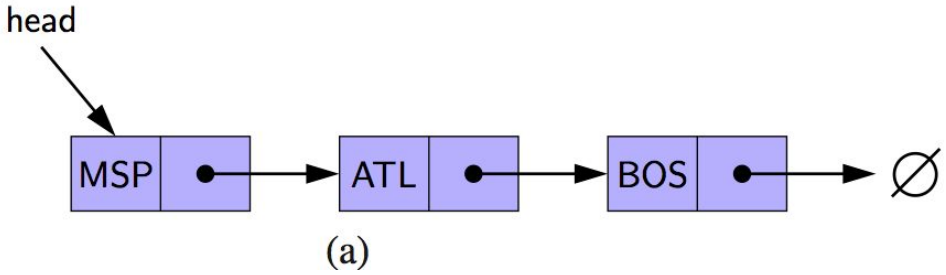


1. have new node point to old head



1. update head to point to new node

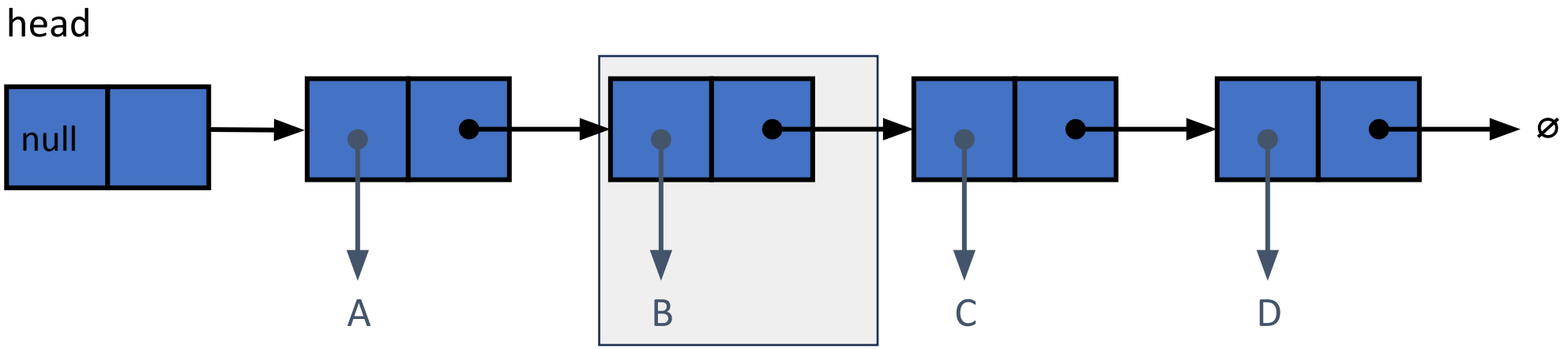


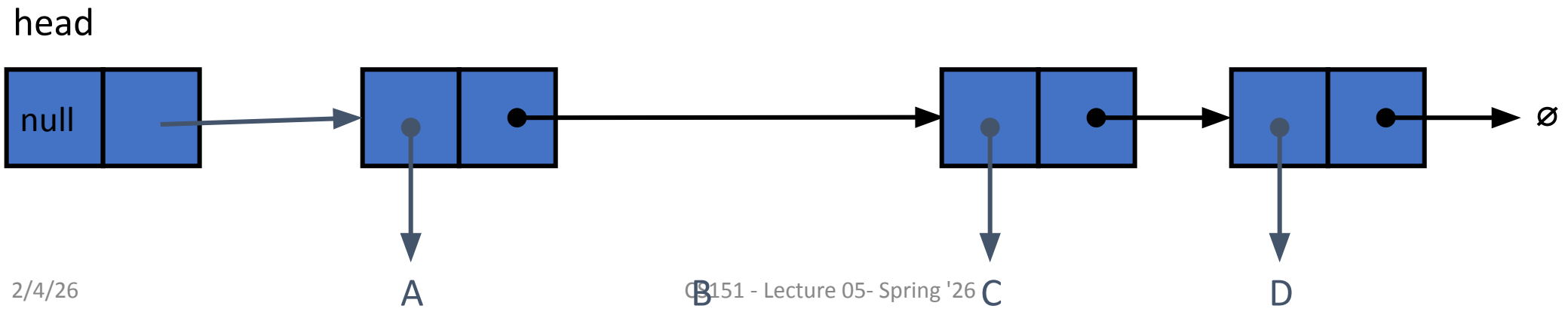


Remove Operation

- Let’s write it on the board quickly

Remove Operation `remove("B")`





Properties in LinkedList

What do we need to keep track of?

- Head
- Number of elements (optional)

Quiz

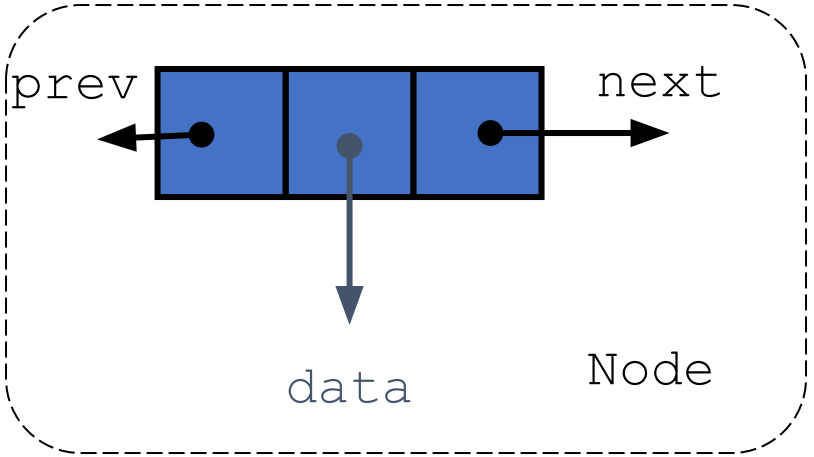
Rank from most efficient to least efficient:

- LinkedList find
- ExpandableArray find
- LinkedList insert at beginning
- ExpandableArray insert at beginning

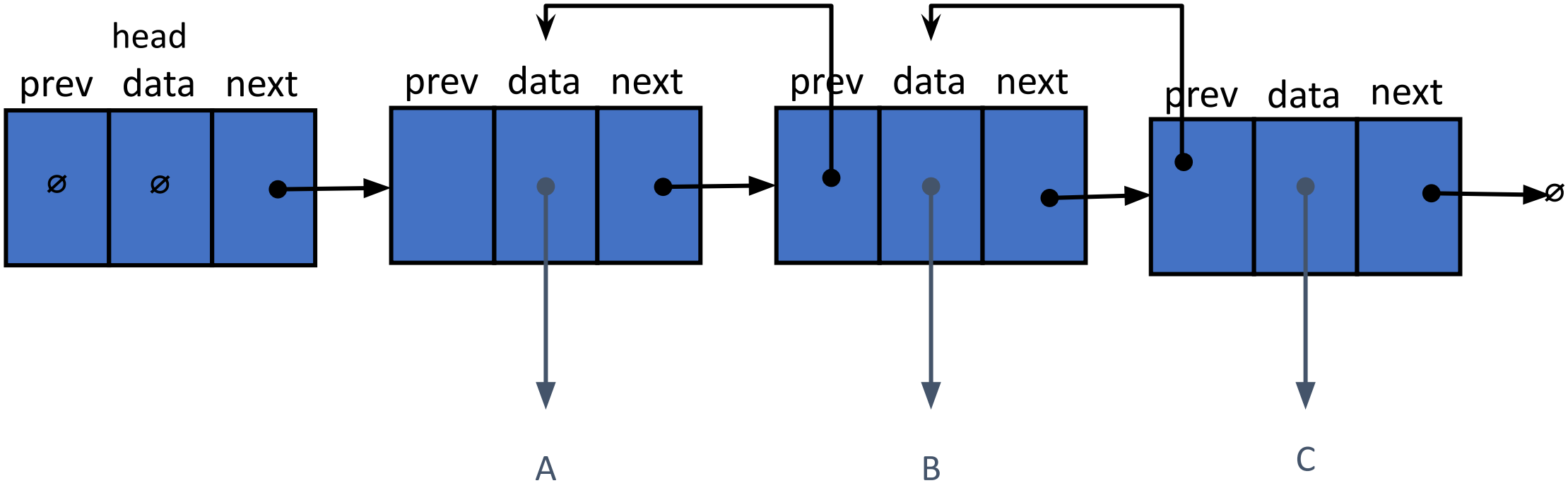
Doubly Linked Lists

A node

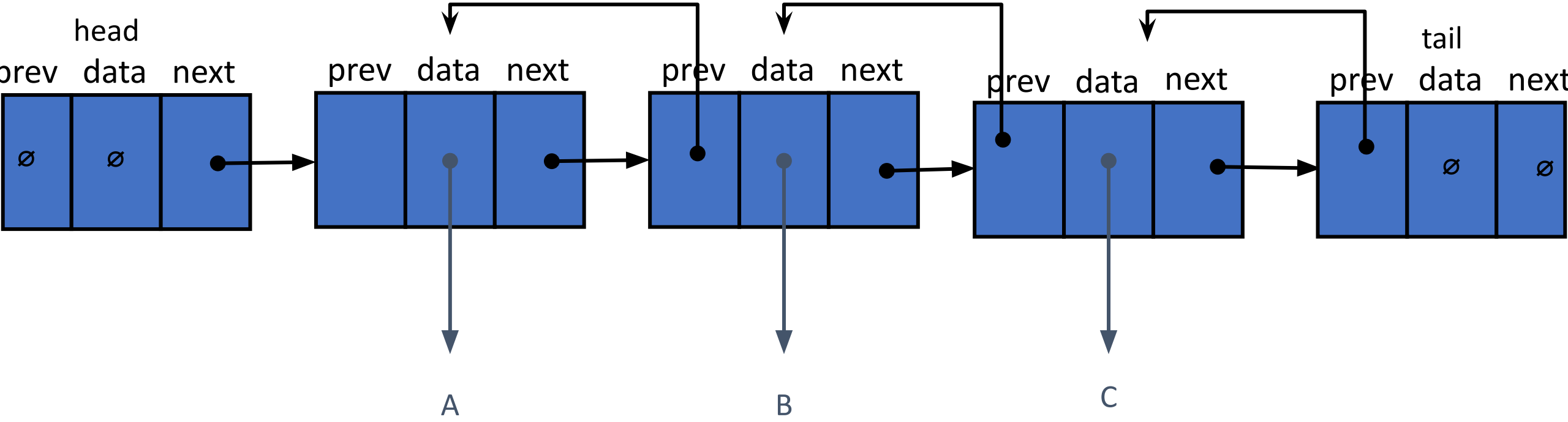
```
public class Node<T> {
   private T data;
   private Node next;
   private Node prev;
}
```



Doubly Linked List



Doubly Linked List



Lab today: Doubly Linked Lists

You'll be implementing a SLL and DLL

`insertSortedAlpha: inserted into the list in alphabetically sorted order`

How would you implement insertSorted with numeric values?

Comparing Strings in Java

`compareTo()` method is used to compare two strings lexicographically.

```
int result = str1.compareTo(str2);
```

It compares two strings character by character based on their Unicode values.
The method returns:

- `0` if `str1` is equal to `str2`
- A **negative value** if `str1` is lexicographically smaller than `str2`
- A **positive value** if `str1` is lexicographically greater than `str2`

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

Case Sensitivity

compareTo() is case-sensitive

```
"Apple".compareTo("apple")
```

returns a negative value since 'A' has a lower Unicode value than 'a'.

Summary

- Linked Lists are data structures with disjoint memory
- not fixed size! Grows as elements are added
- O(n) access
- Insert at beginning is fast O(1)
- General insert is slow
  - in the worst case O(n)
- Removal is also slow O(n)