Carleton University

COMP 2402

Linked Lists

Alina Shaikhet

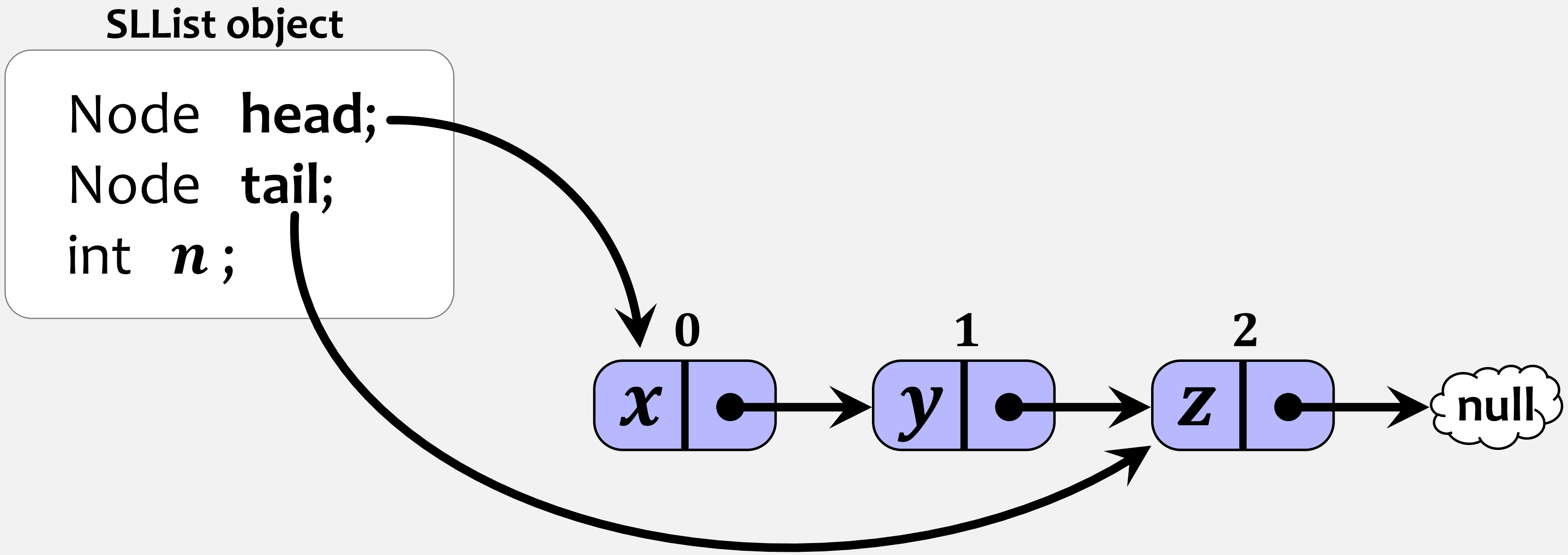
List implementations

<table>
<tr><th></th><th></th><th>get(i) / set(i,x)</th><th>add(i,x) / remove(i)</th></tr>
<tr><td rowspan="3">fast at one end</td><td>ArrayList (JCF)</td><td rowspan="3">$O(1)$</td><td rowspan="3">$O(1+n-i)$</td></tr>
<tr><td>ArrayStack (ODS)</td></tr>
<tr><td>RootishArrayStack (ODS)</td></tr>
<tr><td rowspan="2">fast at both ends</td><td>ArrayDeque (ODS)</td><td rowspan="2">$O(1)$</td><td rowspan="2">$O(1+\min\{i,n-i\})$</td></tr>
<tr><td>DualArrayDeque (ODS)</td></tr>
<tr><td rowspan="2">dynamic</td><td>LinkedList (JCF)</td><td colspan="2" rowspan="2">$O(1+\min\{i,n-i\})$</td></tr>
<tr><td>DLList (ODS)</td></tr>
</table>

SLList – Singly-Linked List

```
class Node {
    T x;
    Node next;
}
```

reference to another Node



Linked List is a data structure consisting of a sequence of data elements (nodes).

Each node contains a piece of data (an element) and a pointer/reference to the next node.

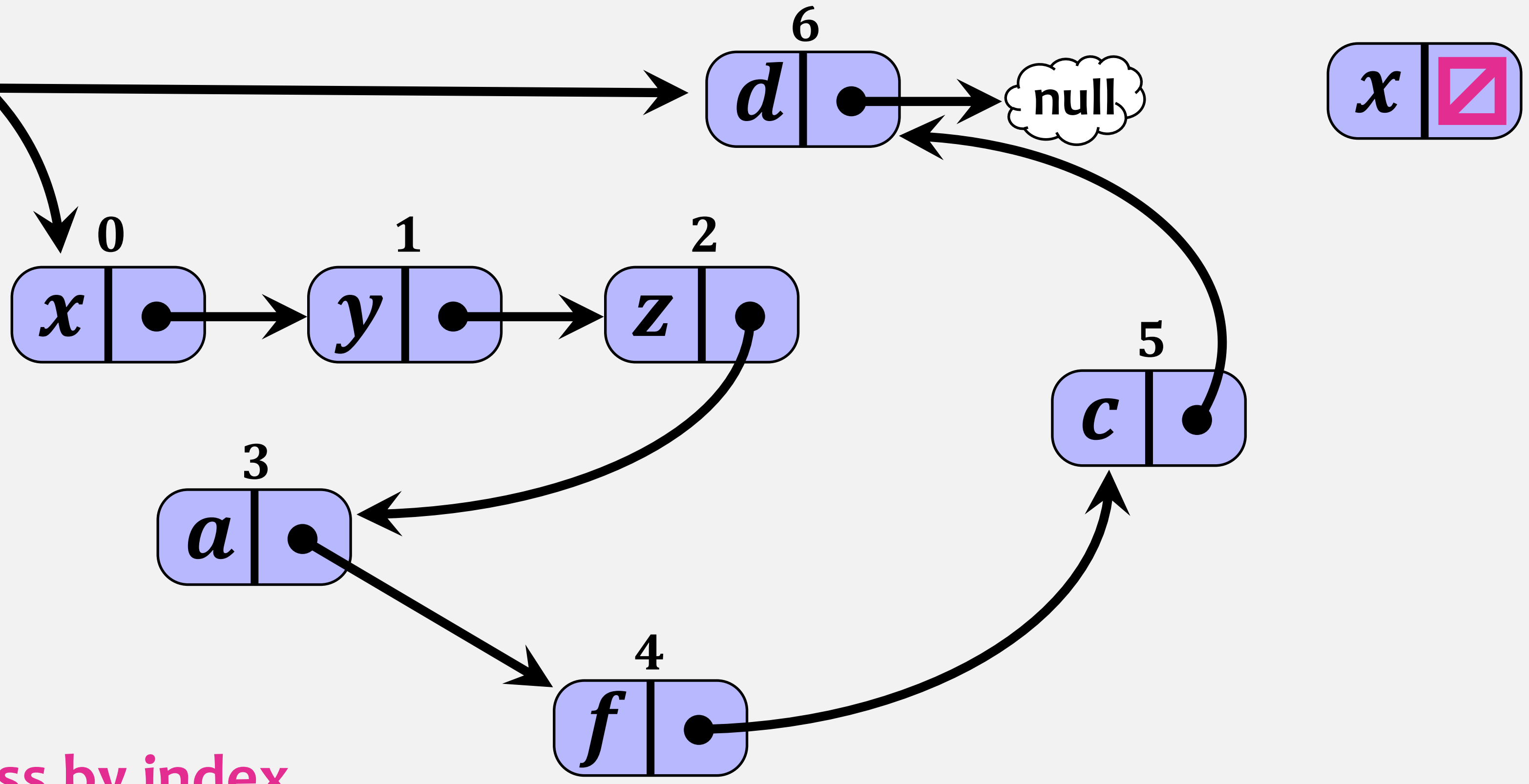
SLList – Singly-Linked List

```
class Node {
    T  x;
    Node  next;
}
```

reference to another Node

SLList object

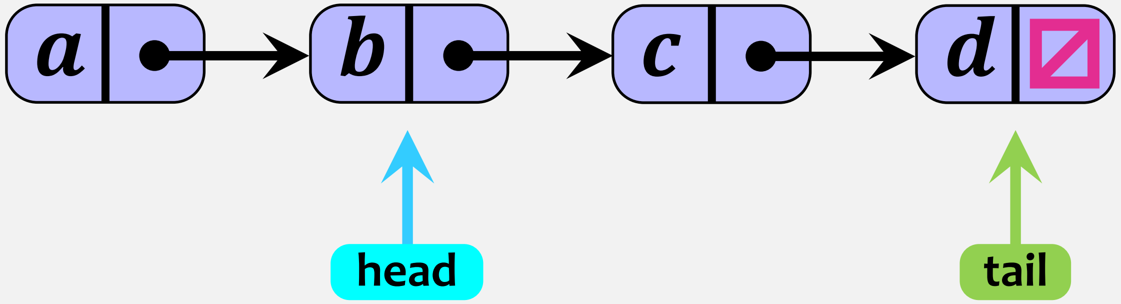
```
Node  head;
Node  tail;
int  n;
```

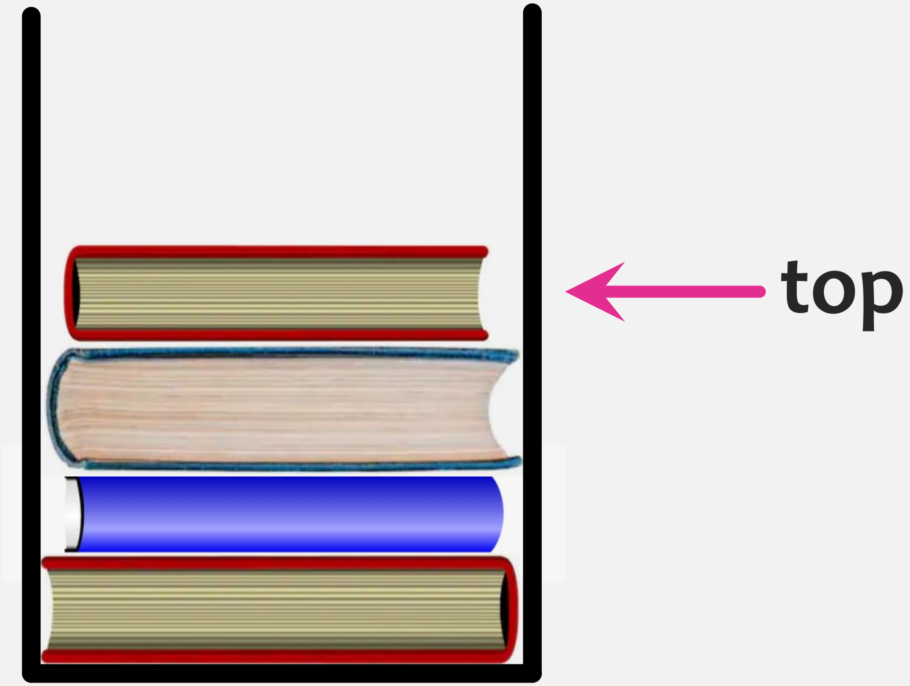


There is no direct access by index

Stack Implementation – push()

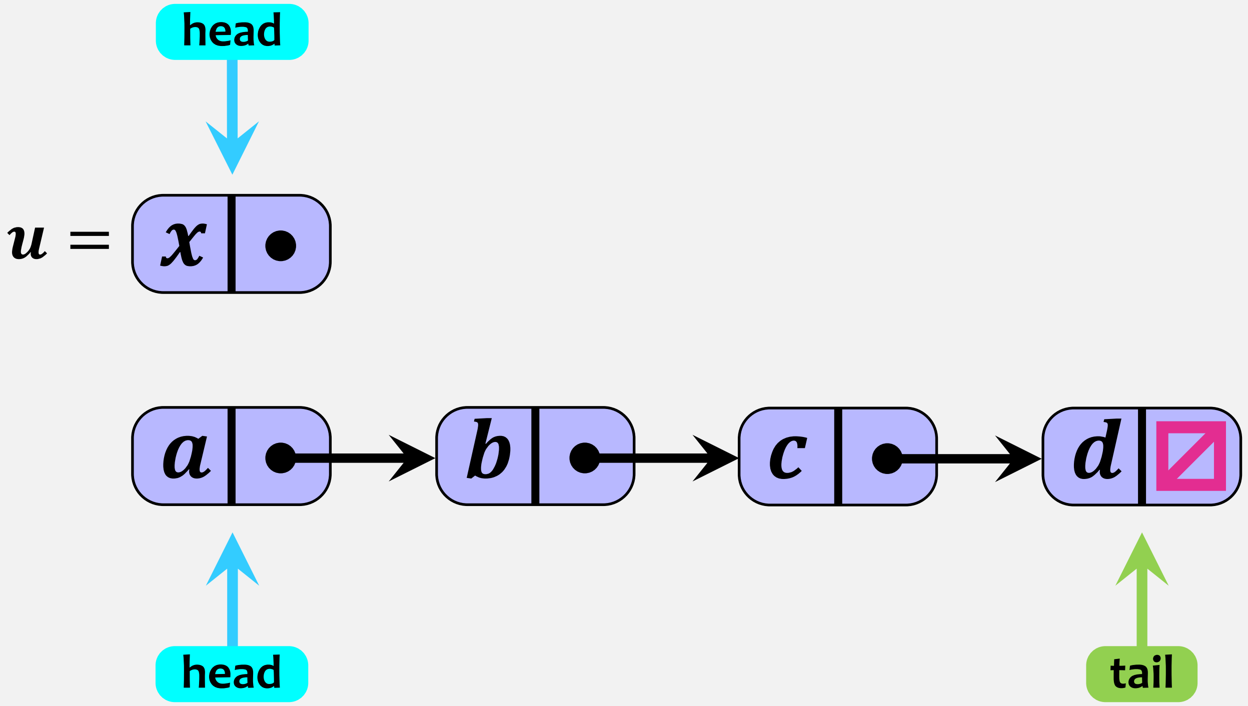
An SLList can efficiently implement the Stack operations push(x) and pop() by adding and removing elements at the **head** of the sequence.





Stack Implementation – push()

An SLList can efficiently implement the Stack operations push($\boldsymbol{x}$) and pop() by adding and removing elements at the **head** of the sequence.



T push($\boldsymbol{x}$):

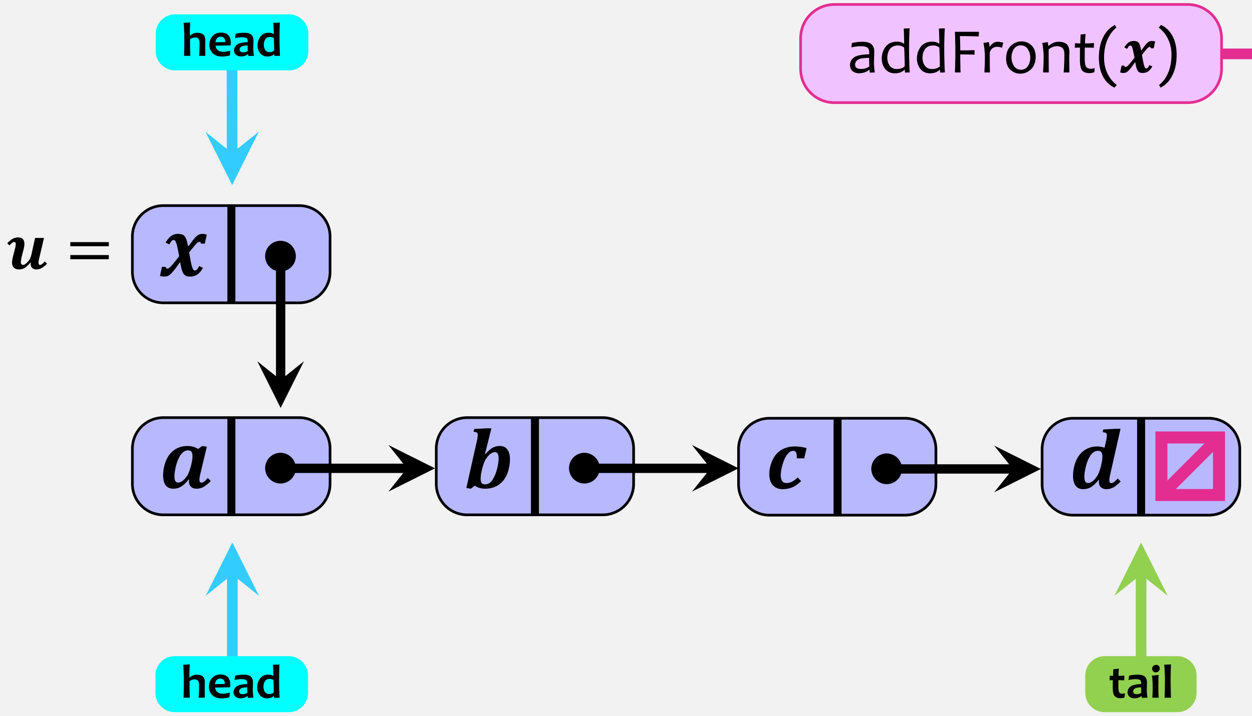
Node $\boldsymbol{u}$ = new Node();

$\boldsymbol{u}.\boldsymbol{x} = \boldsymbol{x}$;

head = $\boldsymbol{u}$;

Stack Implementation – push()

An SLList can efficiently implement the Stack operations push(x) and pop() by adding and removing elements at the **head** of the sequence.



addFront(x)

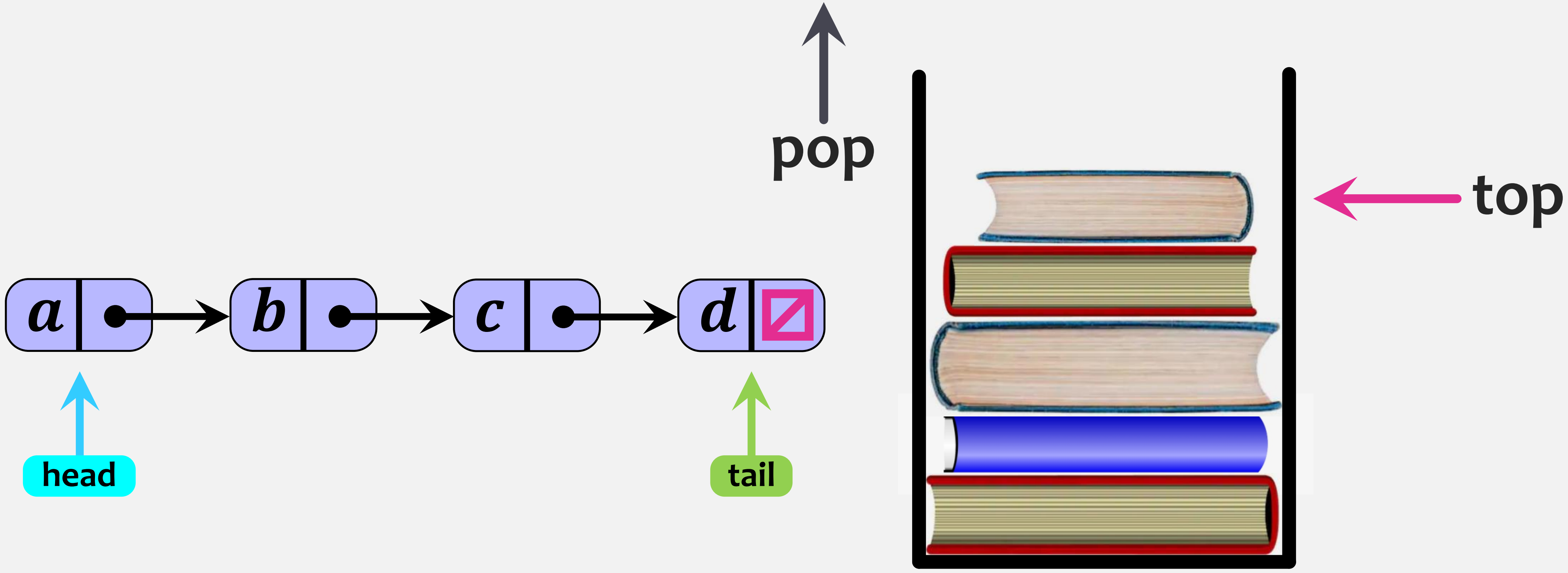
T push(x):

  Node u = new Node();
  $u.x = x$;
  u.next = head;
  head = u;
  if $(n == 0)$ then
    tail = u;
  $n++$;
  return x;

$O(1)$

Stack Implementation – pop()

An SLList can efficiently implement the Stack operations push($\boldsymbol{x}$) and pop() by adding and removing elements at the **head** of the sequence.



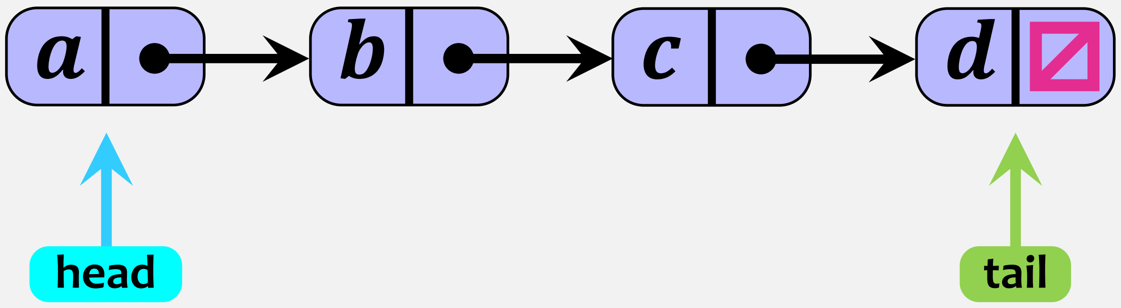
Stack Implementation – pop()

An SLList can efficiently implement the Stack operations push($\boldsymbol{x}$) and pop() by adding and removing elements at the **head** of the sequence.

removeFront()

```
T pop():
    if (n == 0) then
        return null;
    T x = head.x;
    head = head.next;
    n − −;
    if (n == 0) then
        tail = null;
    return x;
```

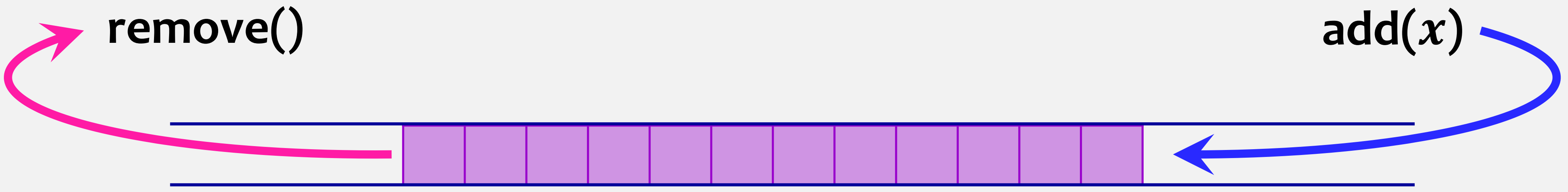
$\boldsymbol{O(1)}$



Can we implement pop() as removeBack()?

Yes, but runtime will be $\boldsymbol{O(n)}$

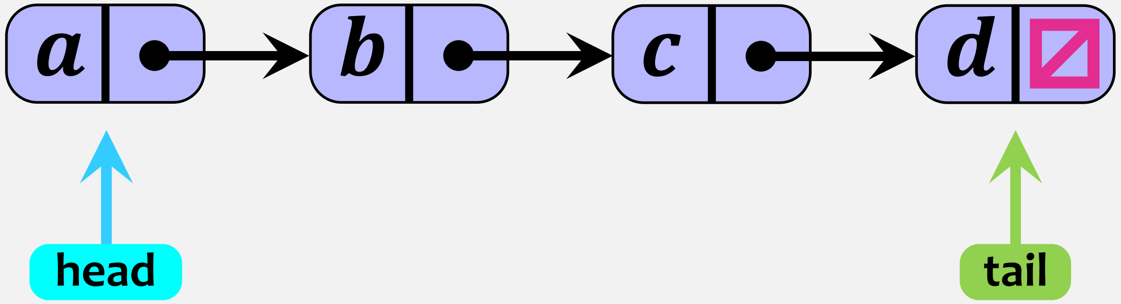
FIFO Queue Implementation – remove()



removeFront() →

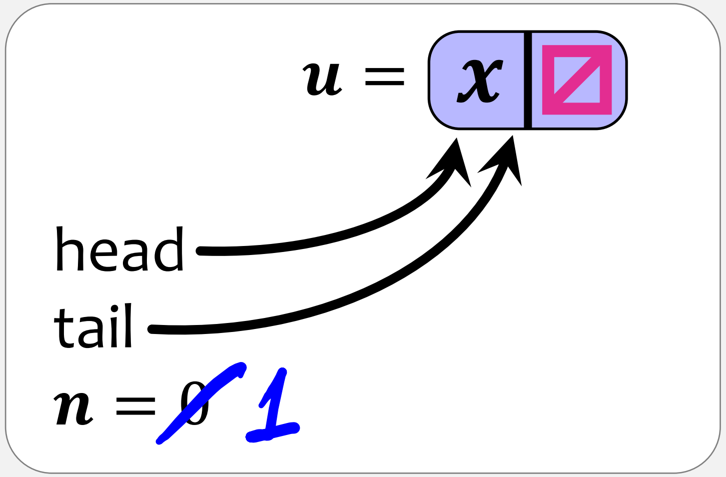
```
T remove():
    if (n == 0) then
        return null;
    T x = head.x;
    head = head.next;
    n − −;
    if (n == 0) then
        tail = null;
    return x;
```

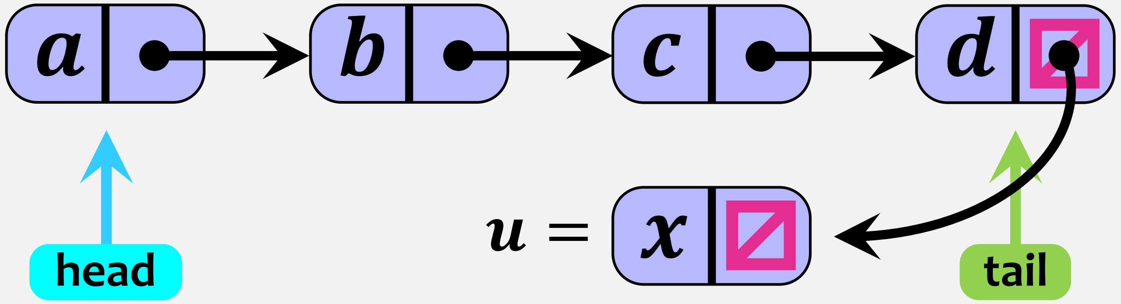
$O(1)$



removals are done from the **head** of the list,
additions are done at the **tail** of the list.

FIFO Queue Implementation – add(x)





addBack(x) →

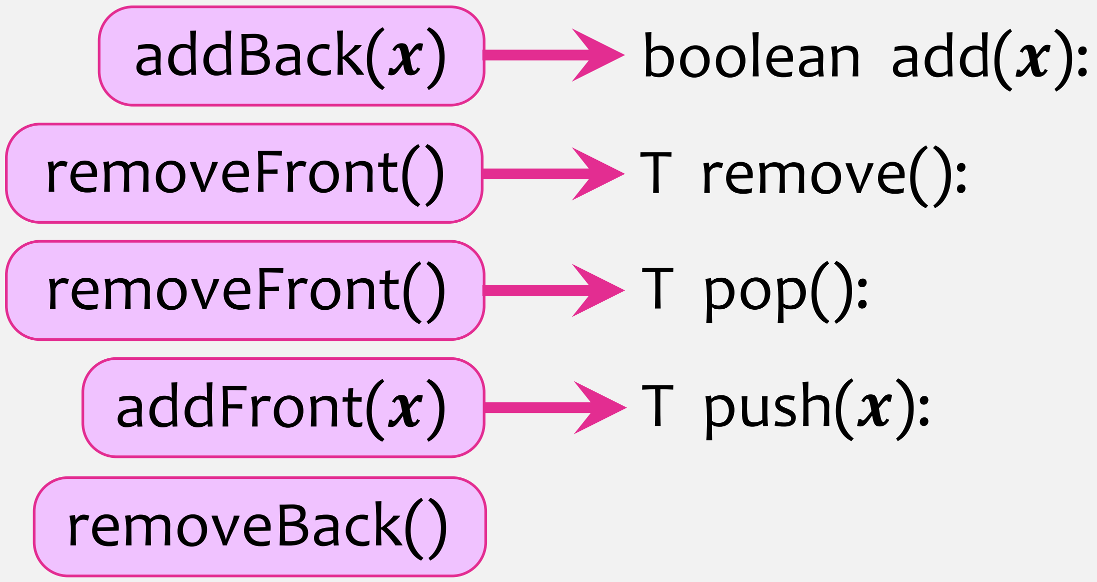
```
boolean add(x):
    Node u = new Node();
    u.x = x;
    if (n == 0) then
        head = u;
    else
        tail.next = u;
    tail = u;
    n++;
    return true;
```

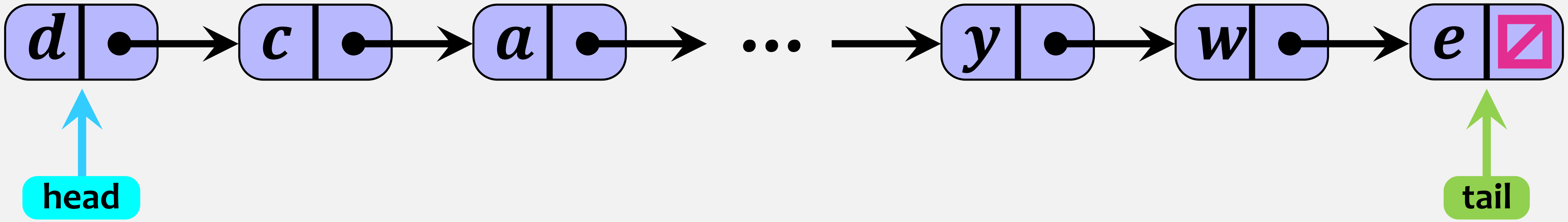
$O(1)$

Deque Implementation

removeBack() operation runs in $\boldsymbol{O(n)}$ time.

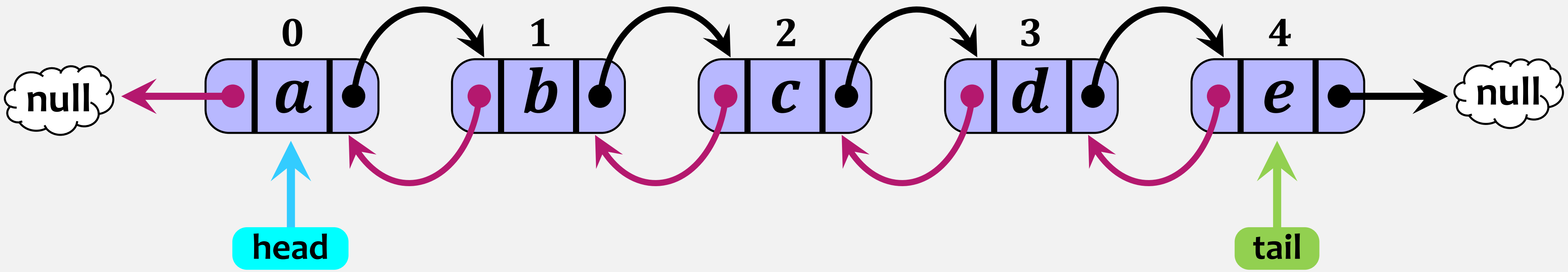
Therefore, with a SLList you cannot make an efficient implementation of a Deque Interface.





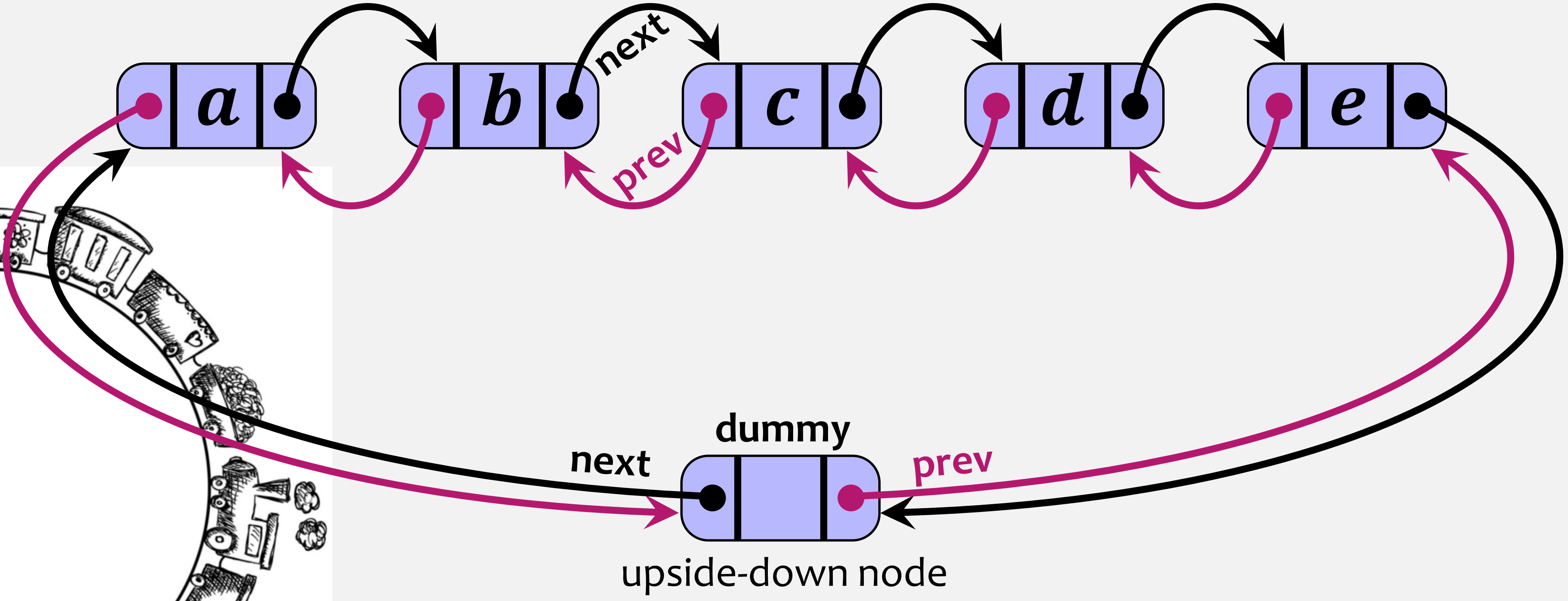
DLList – Doubly-Linked List

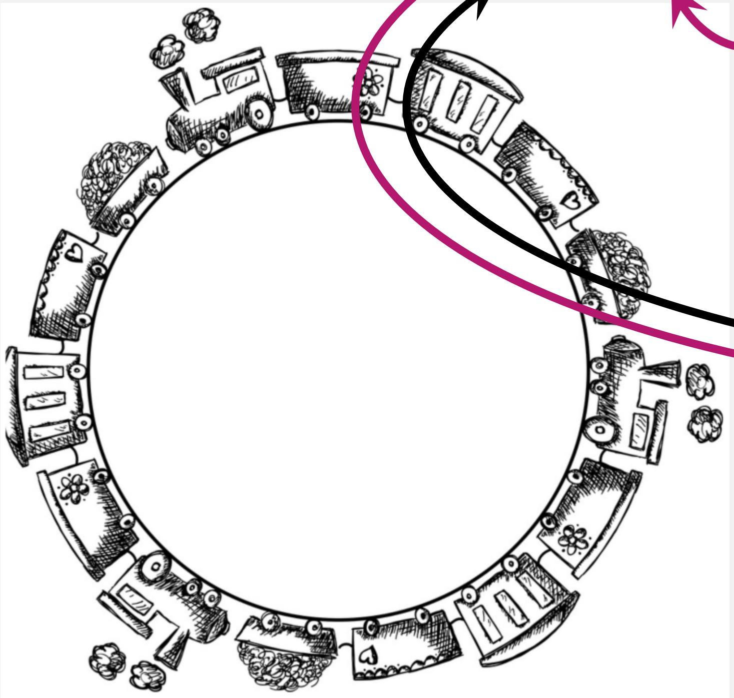
Without “helper” node:



DLList – Doubly-Linked List

Each node contains a piece of data (an element) and two pointers/references: one to the next node (**next**) and one to the previous (**prev**).

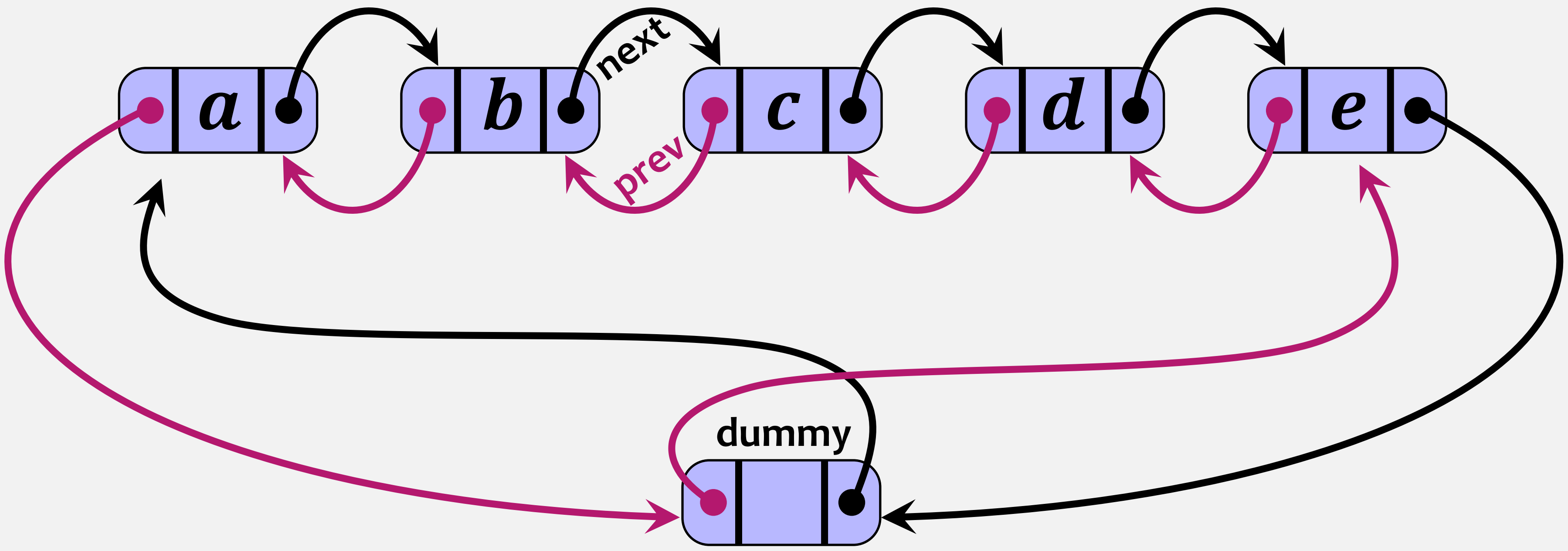




helper (dummy node) does not contain any data, but acts as a placeholder. In this way, the nodes of the list are (doubly-)linked into a cycle.

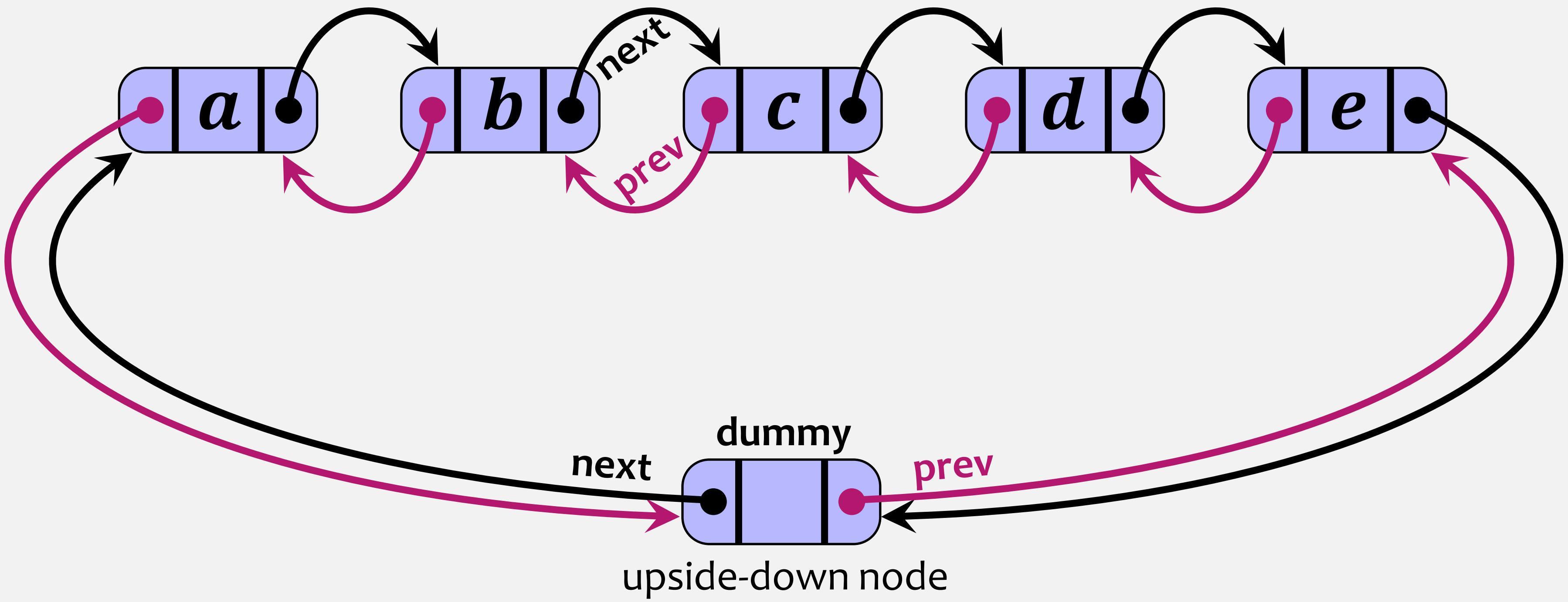
DLList – Doubly-Linked List

Each node contains a piece of data (an element) and two pointers/references: one to the next node (**next**) and one to the previous (**prev**).



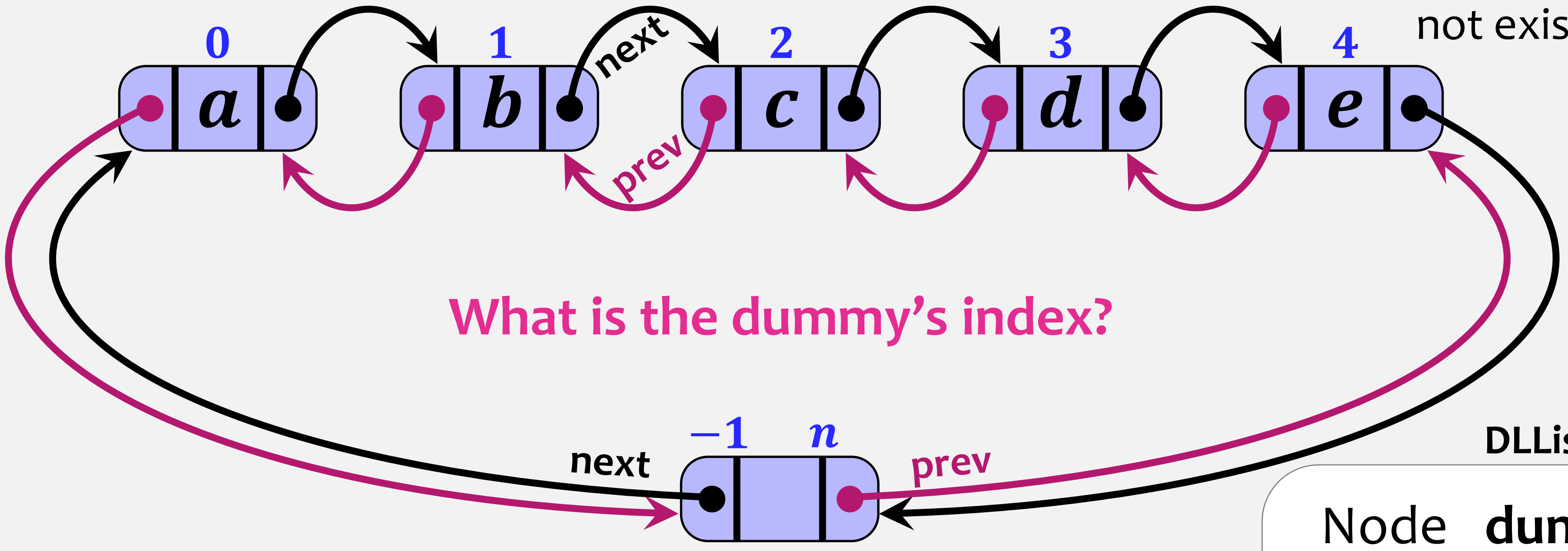
DLList – Doubly-Linked List

Each node contains a piece of data (an element) and two pointers/references: one to the next node (**next**) and one to the previous (**prev**).

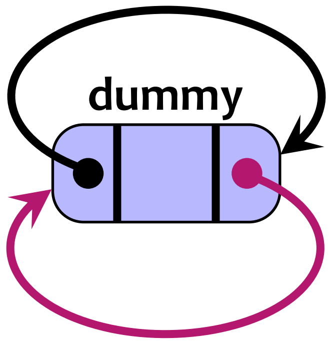


DLList – Doubly-Linked List

Thanks to the dummy node, there is no need to worry about **prev** or **next** pointers not existing.



What is the dummy’s index?



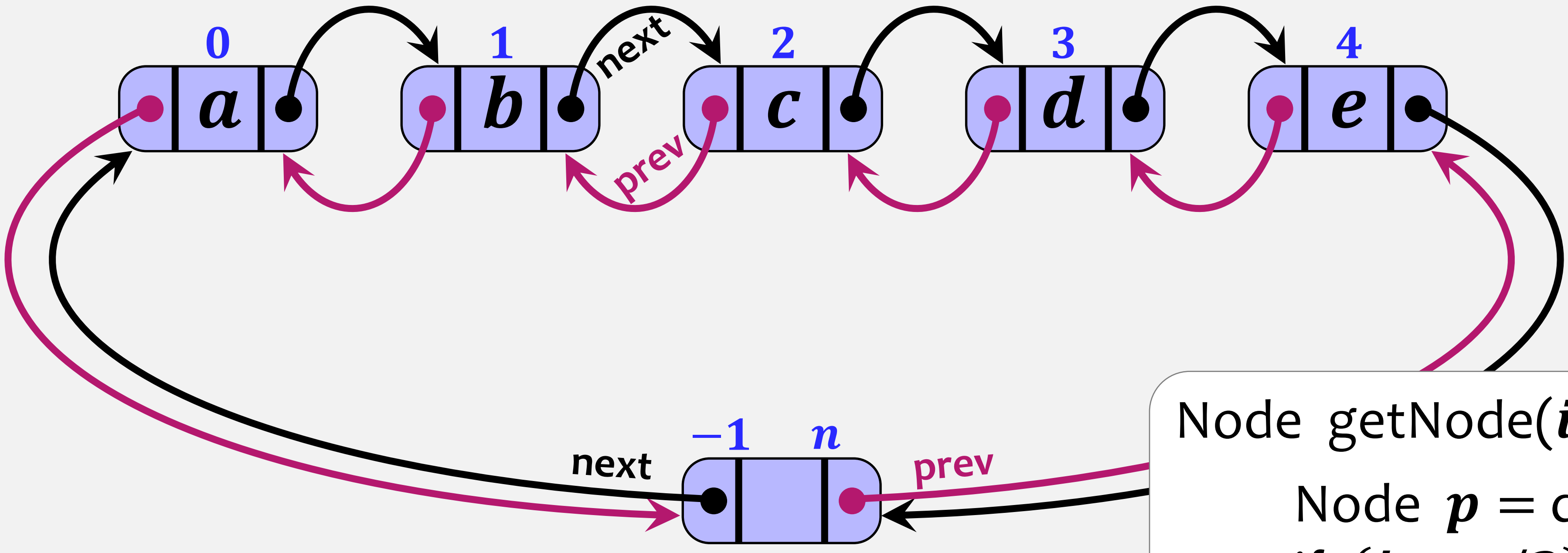
Empty DLList

```
class Node {
    T  x;
    Node  prev;
    Node  next;
}
```

DLList constructor

```
Node  dummy;
int  n;
DLList() {
    dummy = new Node();
    dummy.next = dummy;
    dummy.prev = dummy;
    n = 0;
}
```

DLList – finding a node at position i



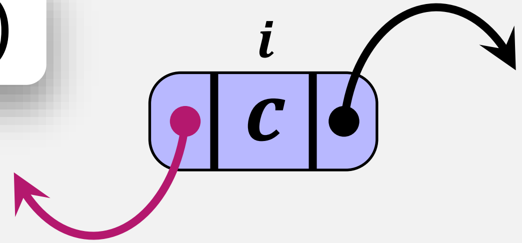
Node getNode(i): // $0 \le i \le n-1$

```
Node p = dummy;
if (i < n/2) then
    for (j = -1; j < i; j++)
        p = p.next;
else
    for (j = n; j > i; j--)
        p = p.prev;
return p;
```

$$O(1 + \min\{i, n-i\})$$

DLList – get($\boldsymbol{i}$), set($\boldsymbol{i}, \boldsymbol{x}$)

$$\boldsymbol{O(1 + \min\{i, n - i\})}$$

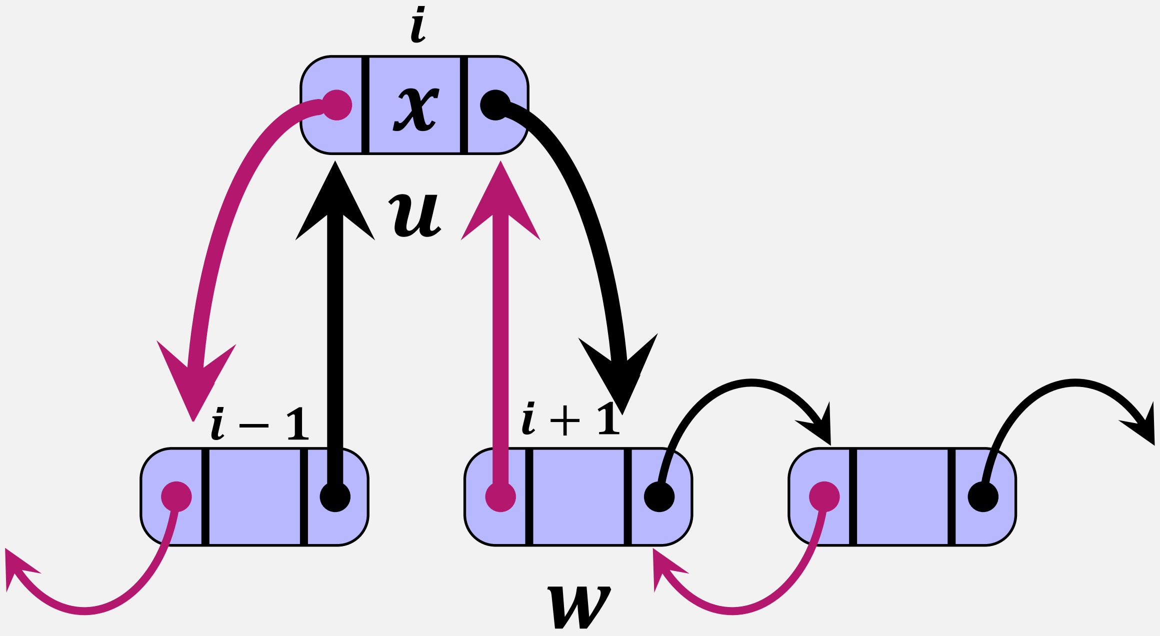


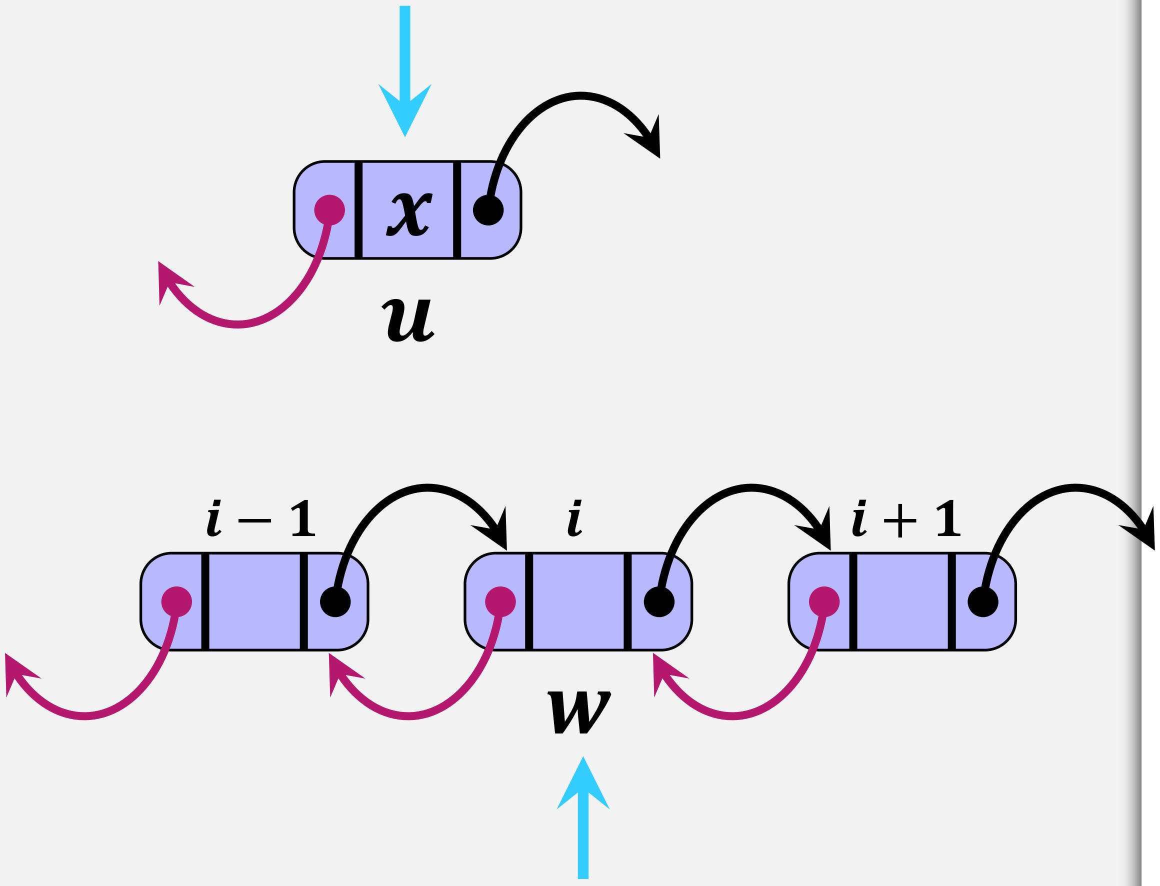

```
T get(i):
    check bounds;
    return getNode(i).x ;
```

```
T set(i, x):
    check bounds;
    Node u = getNode(i);
    T y = u.x;
    u.x = x;
    return y;
```

```
Node getNode(i): // 0 ≤ i ≤ n − 1
    Node p = dummy;
    if (i < n/2) then
        for (j = −1; j < i; j++)
            p = p.next;
    else
        for (j = n; j > i; j−−)
            p = p.prev;
    return p;
```

DLList – add($\boldsymbol{i}$, $\boldsymbol{x}$)

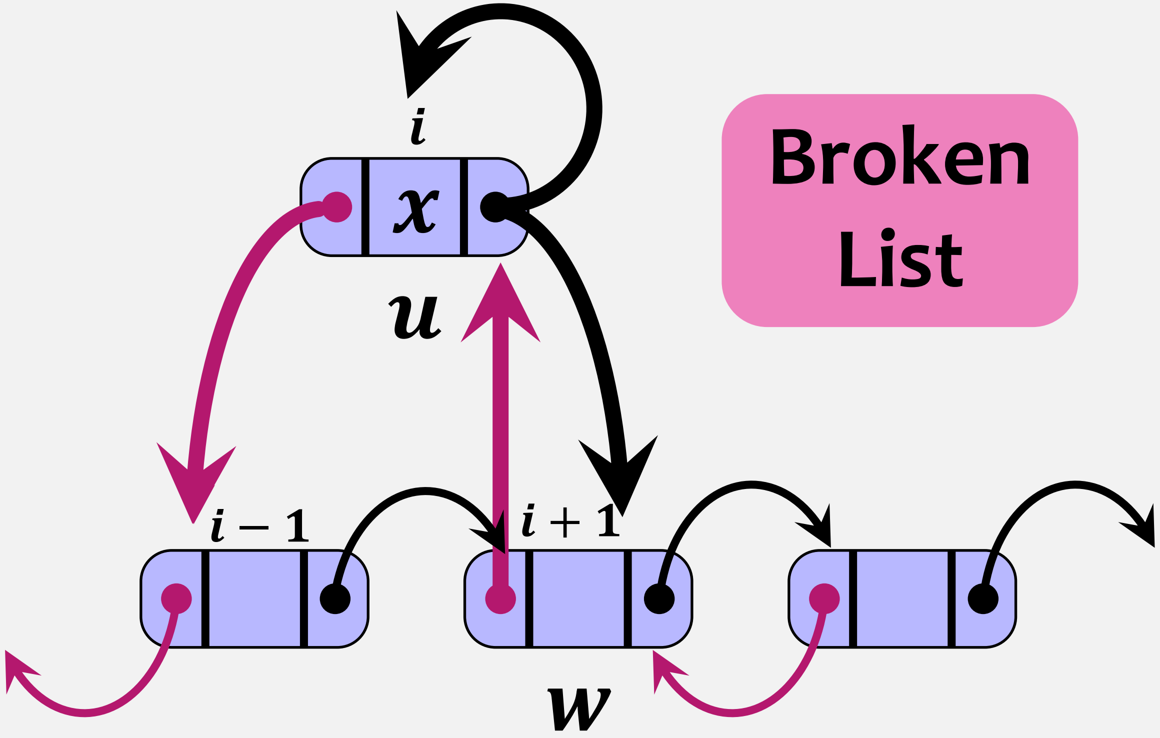




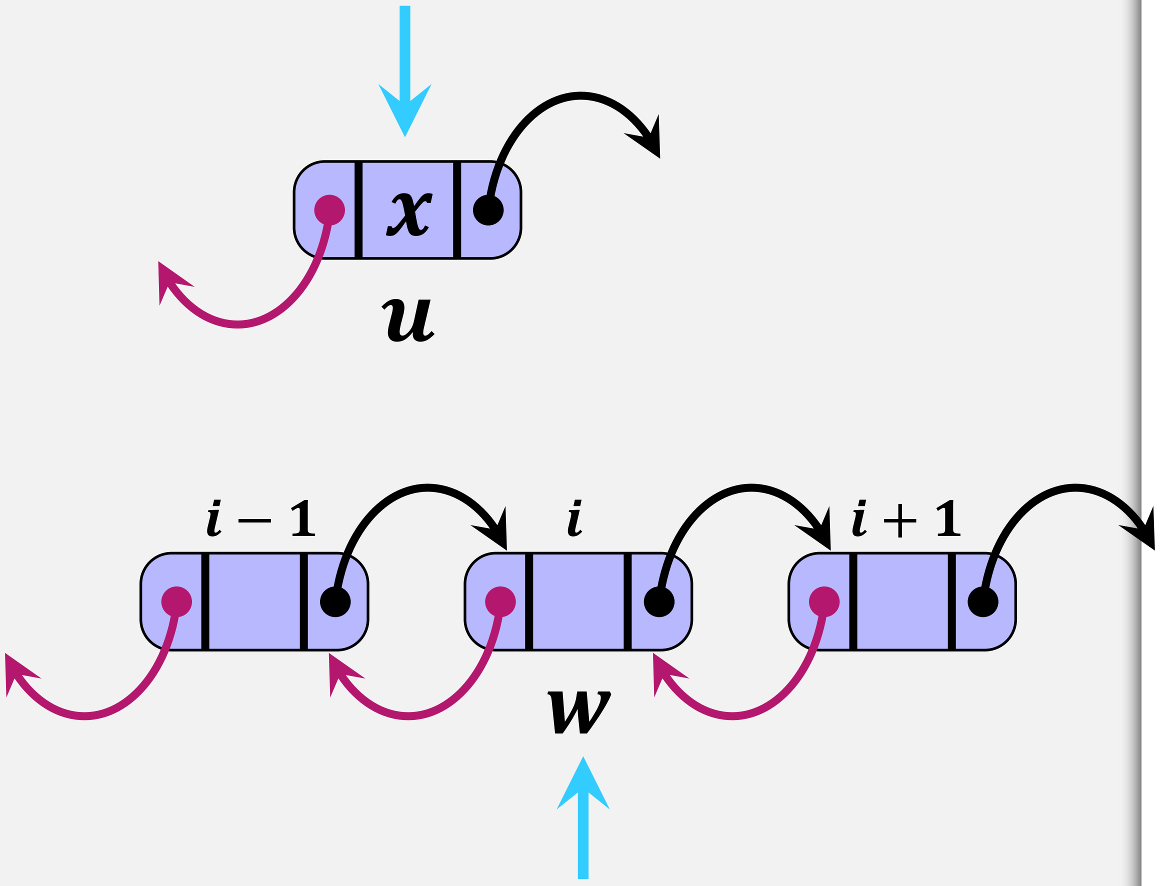
$\boldsymbol{u}$.prev $=$ $\boldsymbol{w}$.prev ;
$\boldsymbol{u}$.next $=$ $\boldsymbol{w}$;
$\boldsymbol{w}$.prev.next $=$ $\boldsymbol{u}$;
$\boldsymbol{w}$.prev $=$ $\boldsymbol{u}$;

If you change the order of the last two lines of code – the code will not work as expected.

DLList – add(i, x)

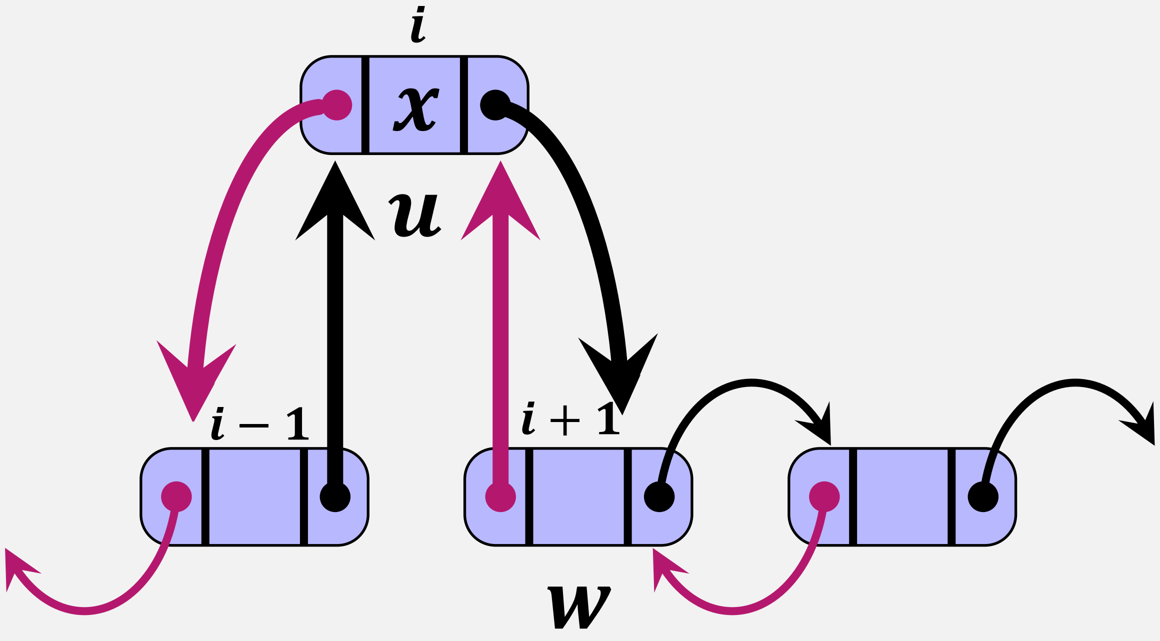


u.prev = w.prev ;
u.next = w ;
w.prev.next = u ;
w.prev = u ;

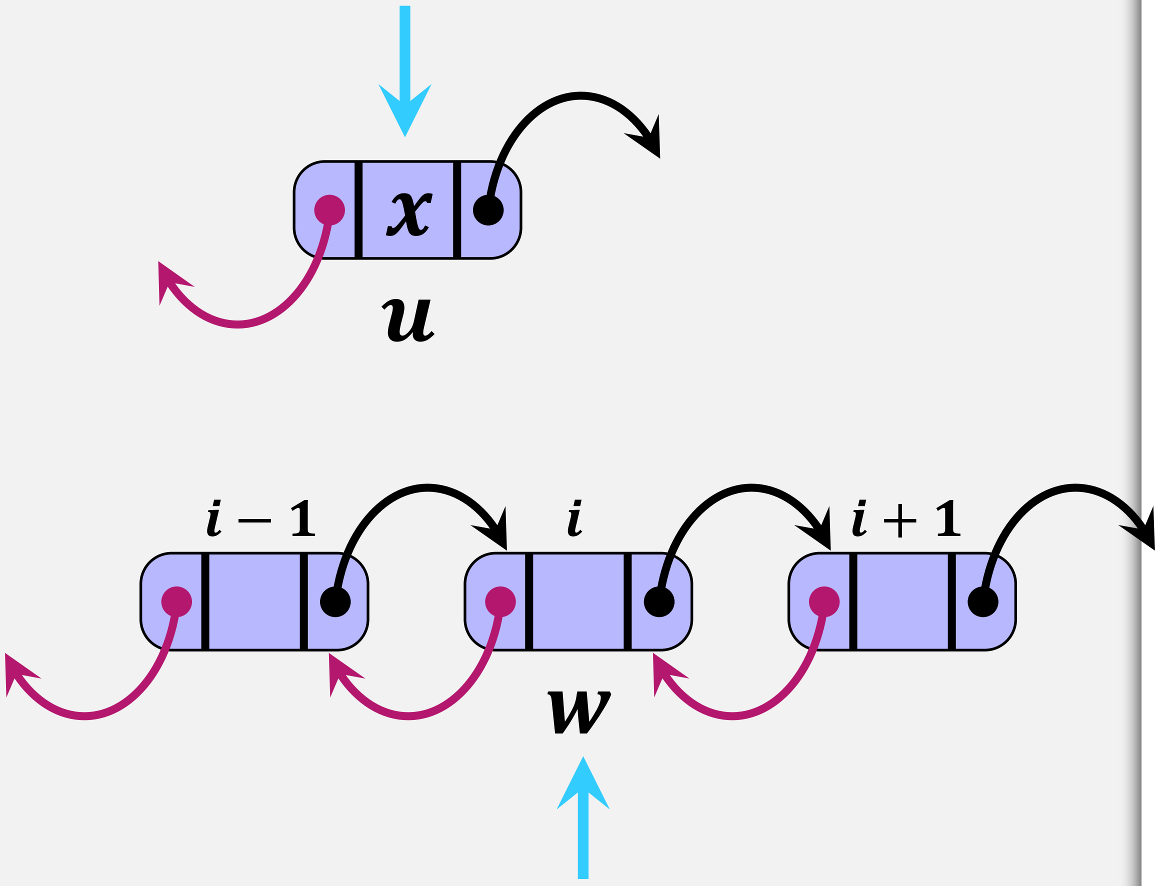


If you change the order of the last two lines of code – the code will not work as expected.

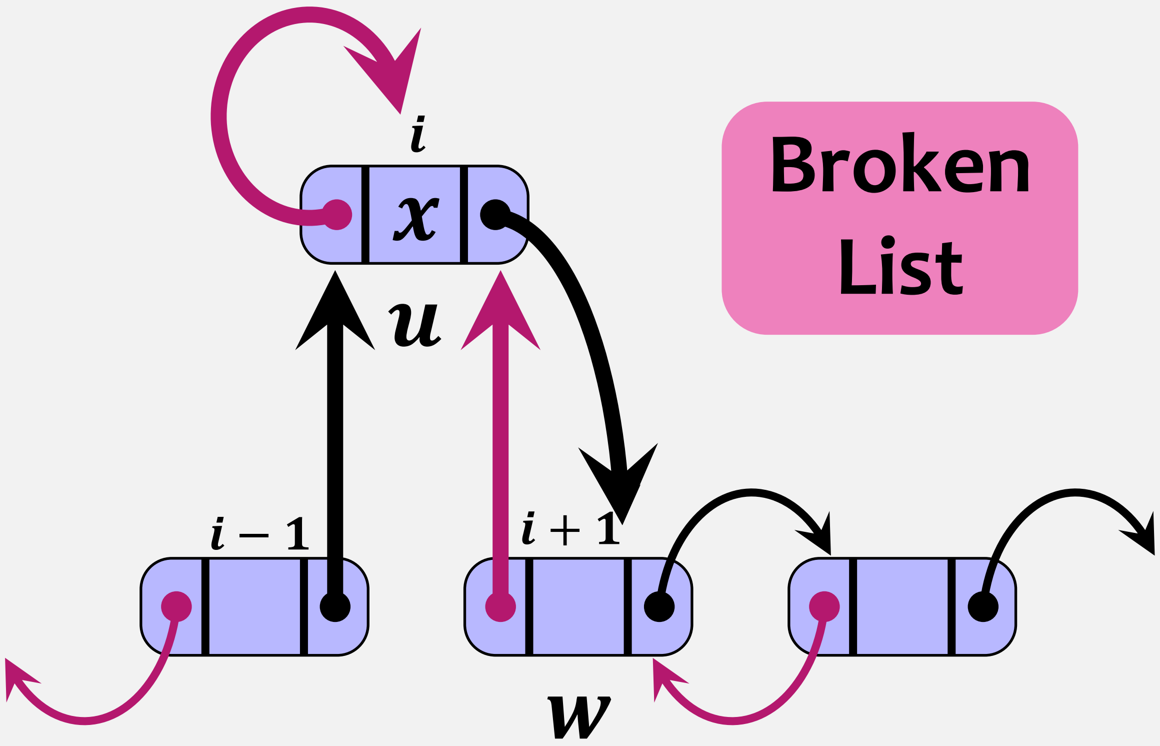
DLList – add($\boldsymbol{i}, \boldsymbol{x}$)



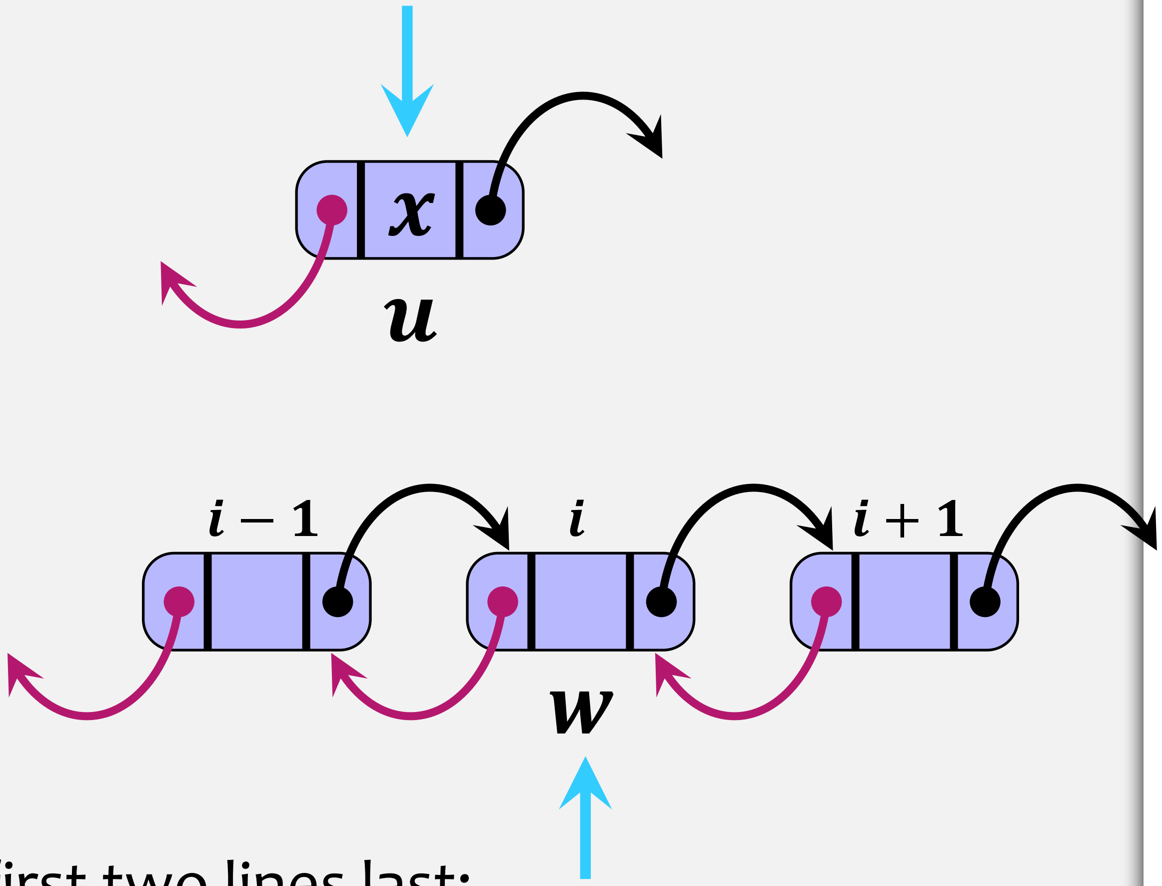
$\boldsymbol{u}$.prev = $\boldsymbol{w}$.prev ;
$\boldsymbol{u}$.next = $\boldsymbol{w}$;
$\boldsymbol{w}$.prev.next = $\boldsymbol{u}$;
$\boldsymbol{w}$.prev = $\boldsymbol{u}$;



DLList – add(i, x)



Broken List



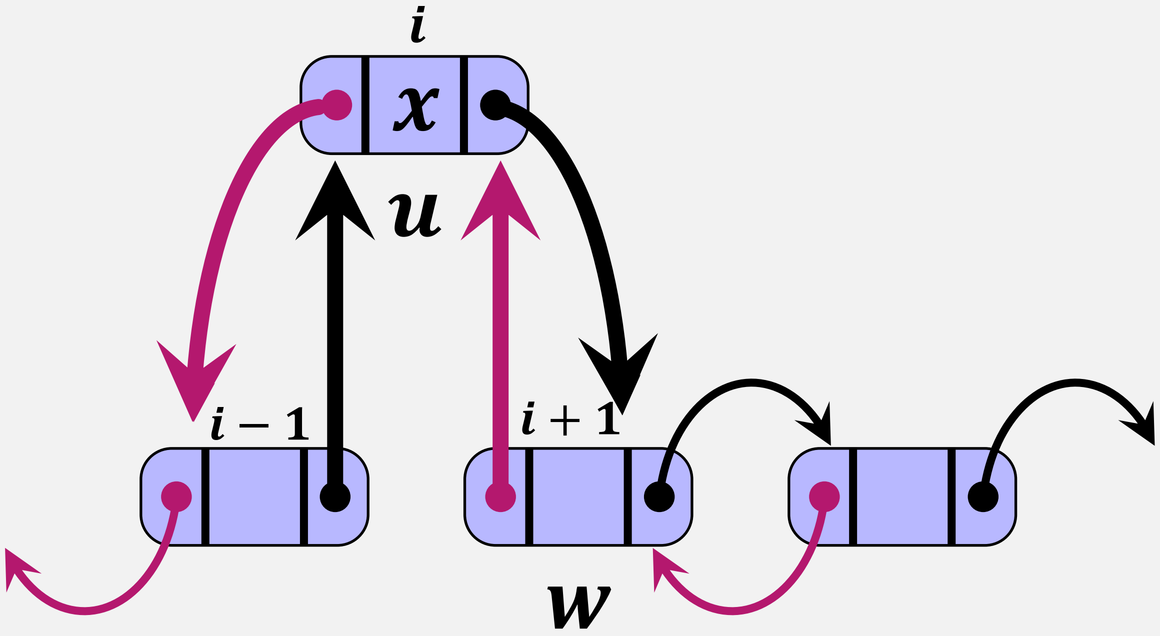
u.prev = w.prev ;
u.next = w ;
w.prev.next = u ;
w.prev = u ;

Or execute the first two lines last:

u.prev = w.prev ;
u.next = w ;

DLList – add($\boldsymbol{i}, \boldsymbol{x}$)

$$O(1 + \min\{i, n - i\})$$



u.prev = w.prev ;
u.next = w ;
w.prev.next = u ;
w.prev = u ;

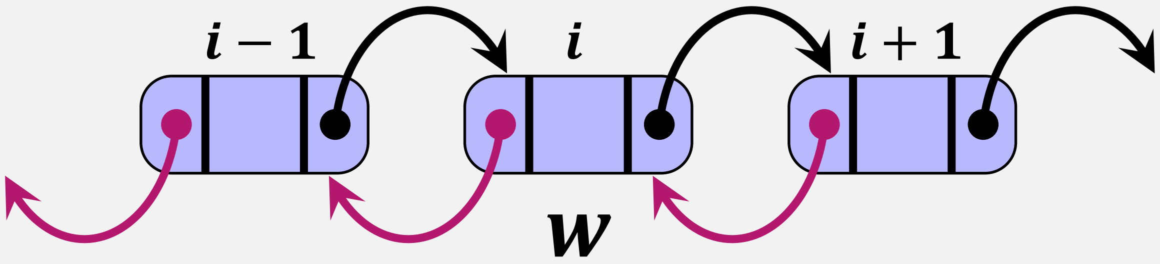
void add(i, x):

    check bounds;
    Node w = getNode(i);
    Node u = new Node();
    $u.x = x$;

    u.prev = w.prev ;
    u.next = w ;
    w.prev.next = u ;
    w.prev = u ;

    n + + ;

DLList – remove($\boldsymbol{i}$)

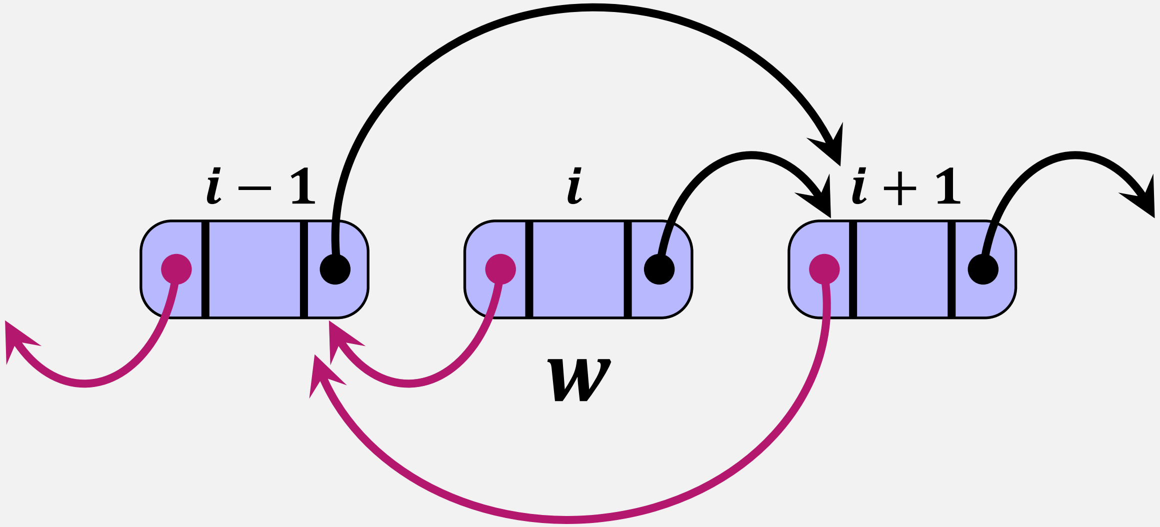


T remove($\boldsymbol{i}$):

  check bounds;
  Node $\boldsymbol{w}$ = getNode($\boldsymbol{i}$);

DLList – remove($\boldsymbol{i}$)

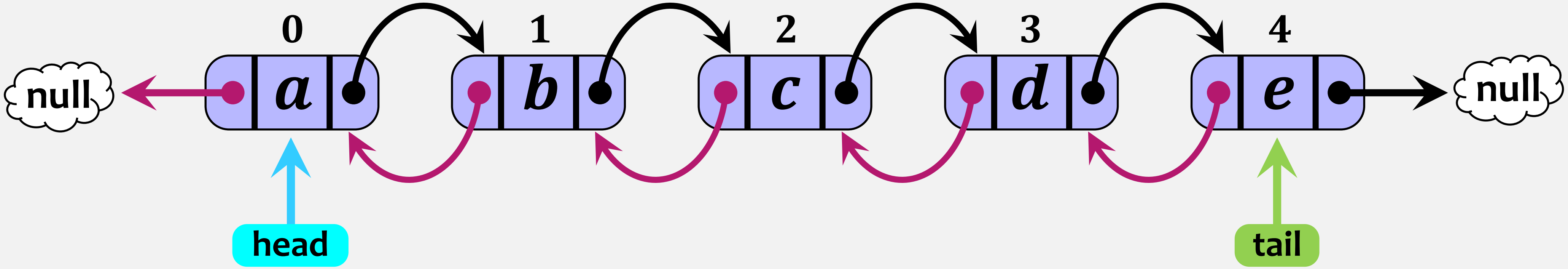
$$\boldsymbol{O(1+\min\{i, n-i\})}$$



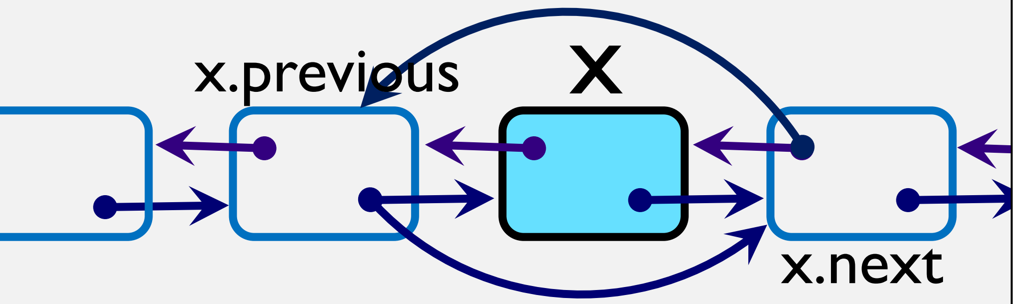

```
T remove(i):
    check bounds;
    Node w = getNode(i);

    w.prev.next = w.next ;
    w.next.prev = w.prev ;

    n − − ;
    return w.x ;
```

DLList – Doubly-Linked List

Without "helper" node:

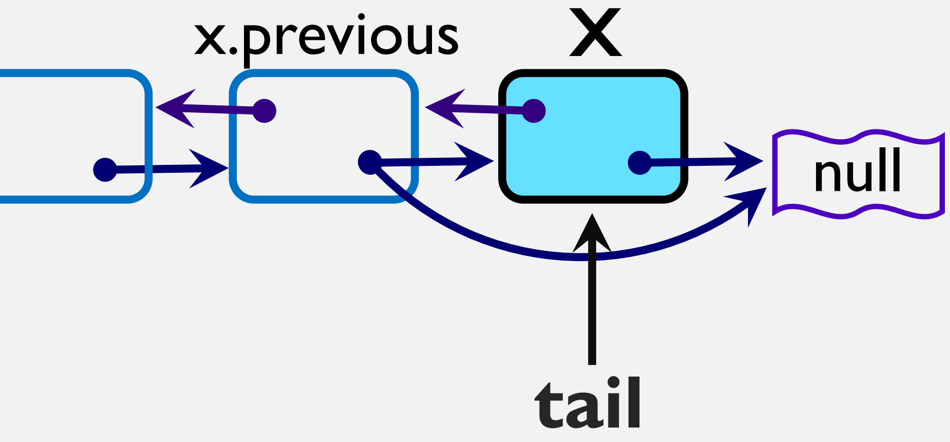


Remove the given node from the list:



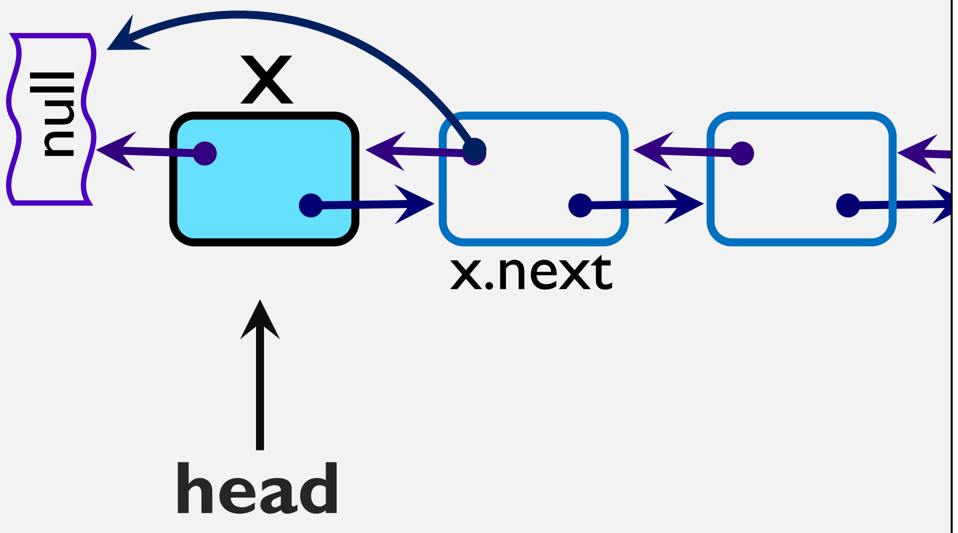

```java
public void remove(Node x) {

}
```

Remove the given node from the list:



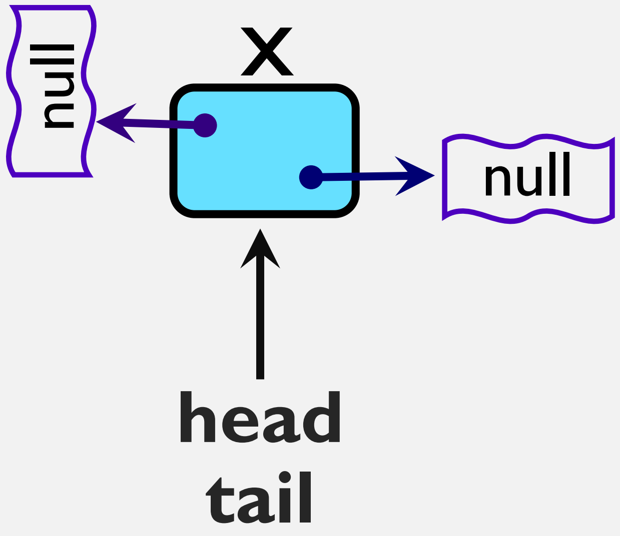

```
public void remove(Node x) {






        if (x == tail) {


        }
        else {
            x.previous.next = x.next;
            x.next.previous = x.previous;
        }

}
```

Remove the given node from the list:




```
public void remove(Node x) {
    if (x == head) {








    }
    else {
        if (x == tail) {
            tail = x.previous;
            tail.next = null;
        }
        else {
            x.previous.next = x.next;
            x.next.previous = x.previous;
        }
    }
}
```

Remove the given node from the list:




```
public void remove(Node x) {
    if (x == head) {
        if (x == tail) {

        }
        else {
            head = x.next;
            head.previous = null;
        }
    }
    else {
        if (x == tail) {
            tail = x.previous;
            tail.next = null;
        }
        else {
            x.previous.next = x.next;
            x.next.previous = x.previous;
        }
    }
}
```

ListSpeed.java

Adds n elements to the **end/front** of the list and then removes all the elements from the **end/front**. Performs n random **get** operations.

Data Structures tested:

- ArrayList
- ArrayStack
- FastArrayStack
- RootishArrayStack
- ArrayDeque
- DualArrayDeque
- LinkedList
- DLList

ods

JCF

```
javac ListSpeed.java
java ListSpeed 200000
```

Theorem 3.2

A **DLList** implements the **List** interface. In this implementation, the get($\boldsymbol{i}$), set($\boldsymbol{i}, \boldsymbol{x}$), add($\boldsymbol{i}, \boldsymbol{x}$), and remove($\boldsymbol{i}$) operations run in $\boldsymbol{O}(\mathbf{1} + \mathbf{min}\{\boldsymbol{i}, \boldsymbol{n} - \boldsymbol{i}\})$ time per operation.