Algorithms & Data Structures I
CSC 225

Ali Mashreghi

Fall 2018

Department of Computer Science, University of Victoria

Motivation

- The use of array in the implementation of stacks and queues forces the user to use only a fixed amount of memory.
- If a good estimate is not known about how much the stack or queue might grow, either

(1) We have to allocate a lot of memory to make sure that we never run out.

(2) Or, risk getting a full stack/queue error if we don't want to waste memory.

Motivation

- In reality, very often we deal with a **dynamic set** of elements which can **grow** or **shrink** in size

- In this lecture, we will introduce an ADT called **List**, which is very flexible for storing a dynamic set of elements.

Motivation

- We will also discuss two data structures **linked lists** and **resizable arrays**.

- These data structures can also be used for implementing stacks and queues.

List

- A list is an ADT that stores the elements in the **sequential order**.

- A list establishes a **before-after** relationship between the elements.

- Unlike arrays, stacks, and queues, the list ADT allows us to do add and remove operations **on any position at any time** as long as enough memory is available.

List

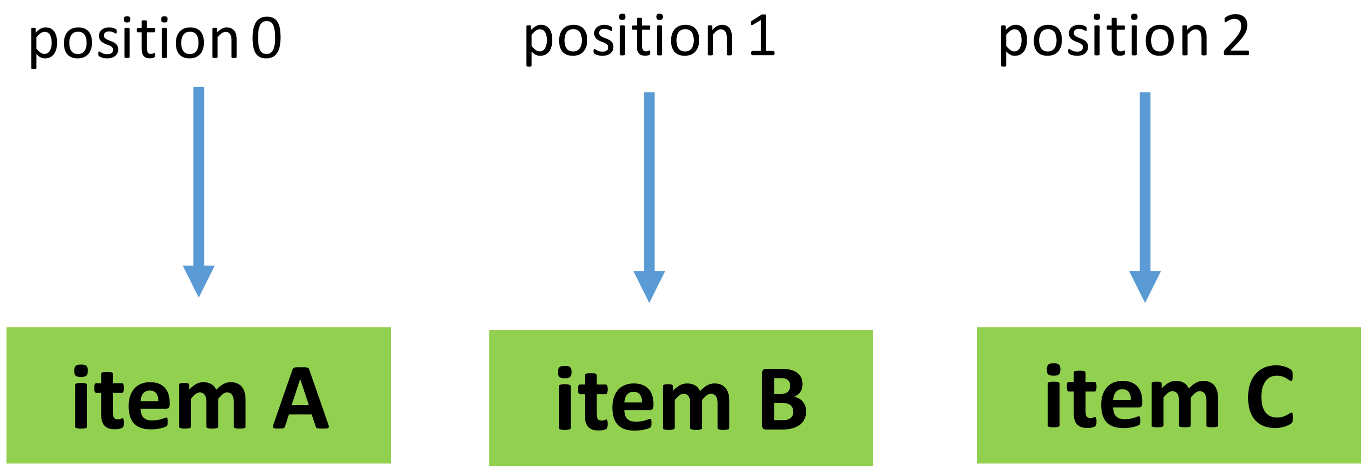
- In the array ADT we had **absolute referencing** using indices and we could **directly access** an item knowing the index.

- However, in a list, we only have **relative referencing** and can only go to the **next or previous element directly** from a specific position.

List

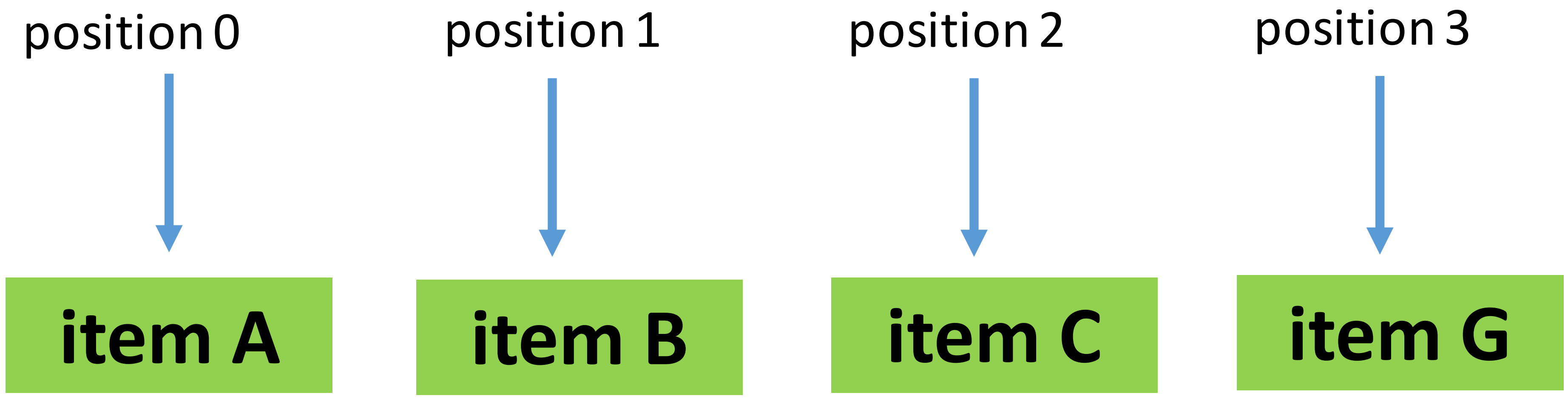
- A list usually supports the following operations:
- $\textsc{Add}(x)$: Add item x at the end of the list.
- $\textsc{Add}(x, i)$: Add item x at position i.
- $\textsc{Contains}(x)$: Returns TRUE if the list contains x, and FALSE otherwise.
- $\textsc{Get}(i)$: Return the item at position i.
- $\textsc{Remove}(i)$: Remove and return the item at position i.
- $\textsc{IsEmpty}()$ and $\textsc{Size}()$ are defined as before.

List - example



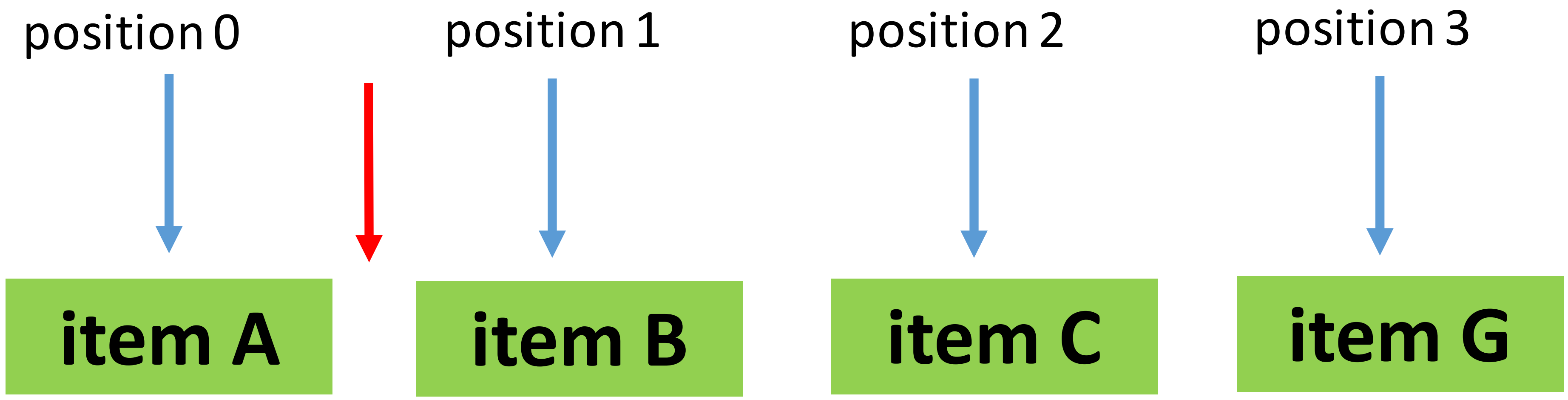
List - example

Add(item G): item G is inserted at the end of the list



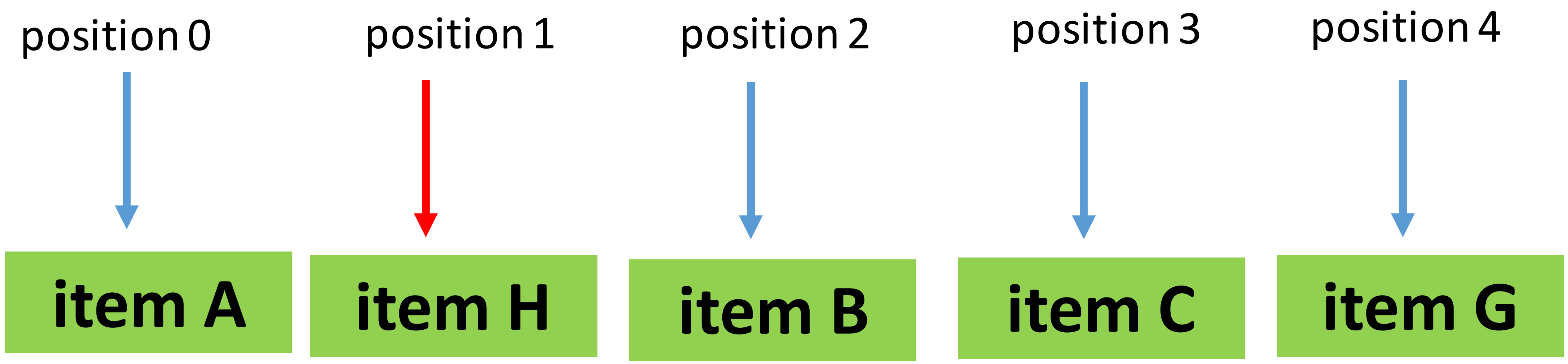
List - example

Add(item H, 1): item H is inserted at position 1



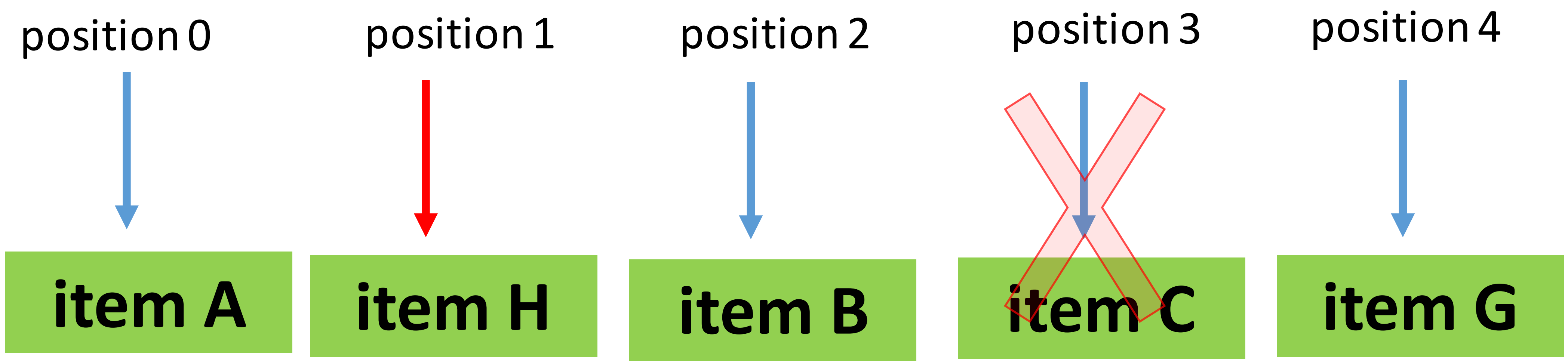
List - example

Add(item H, 1): item H is inserted at position 1



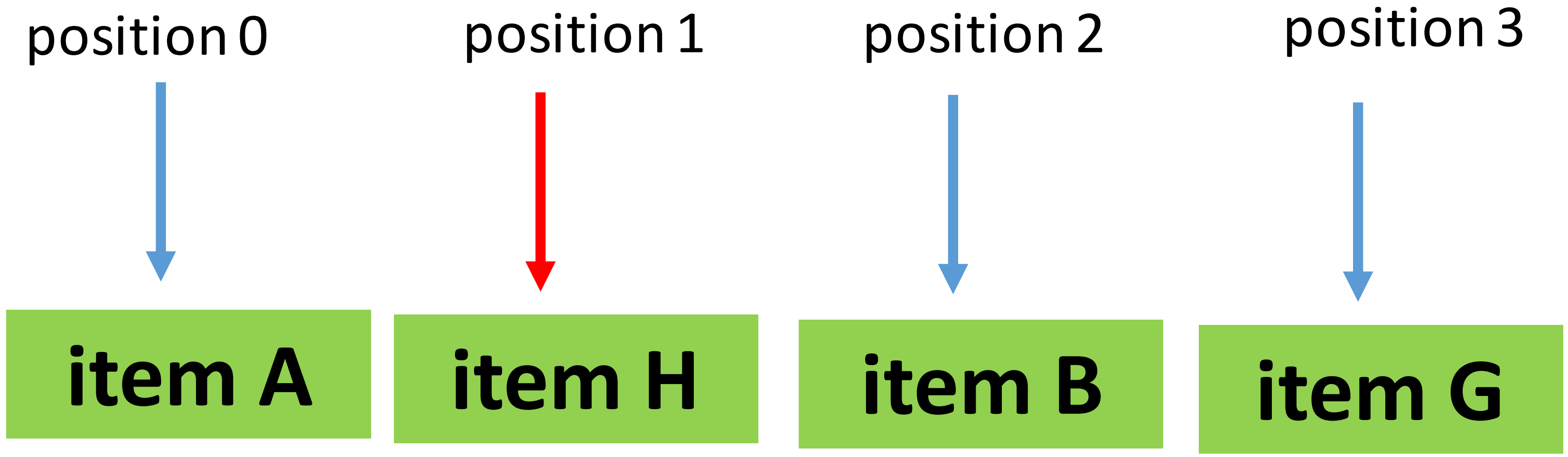
List - example

Remove(3): item at position 3 is removed



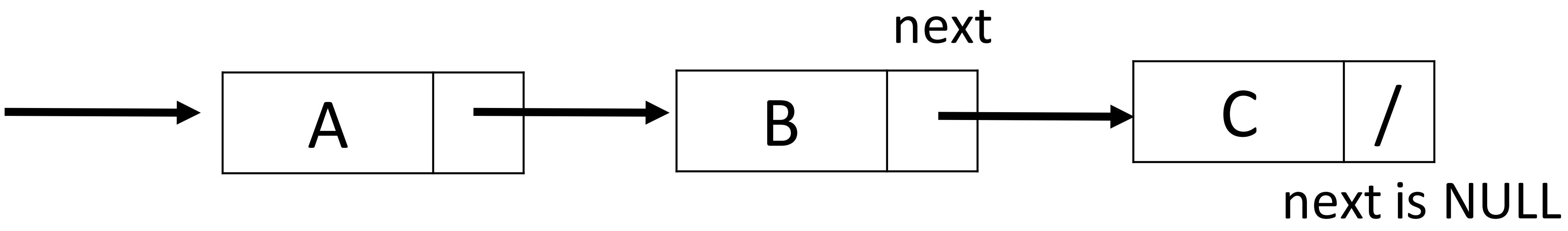
List - example

Remove(3): item at position 3 is removed



Implementation – linked lists

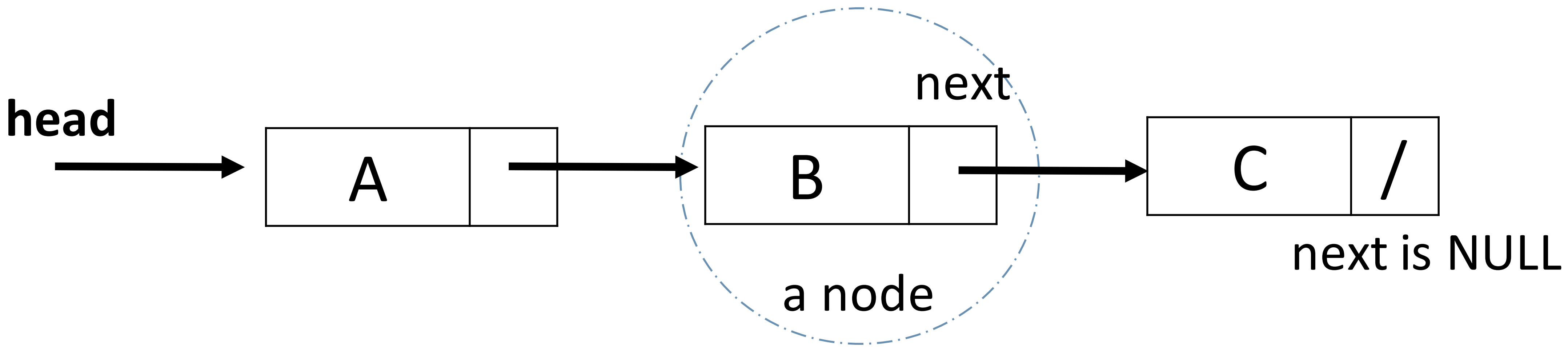
- A ***singly linked list*** is a data structure that stores each element as a separate object, and includes pointers in each object that point to the **next element.**



Implementation – linked lists

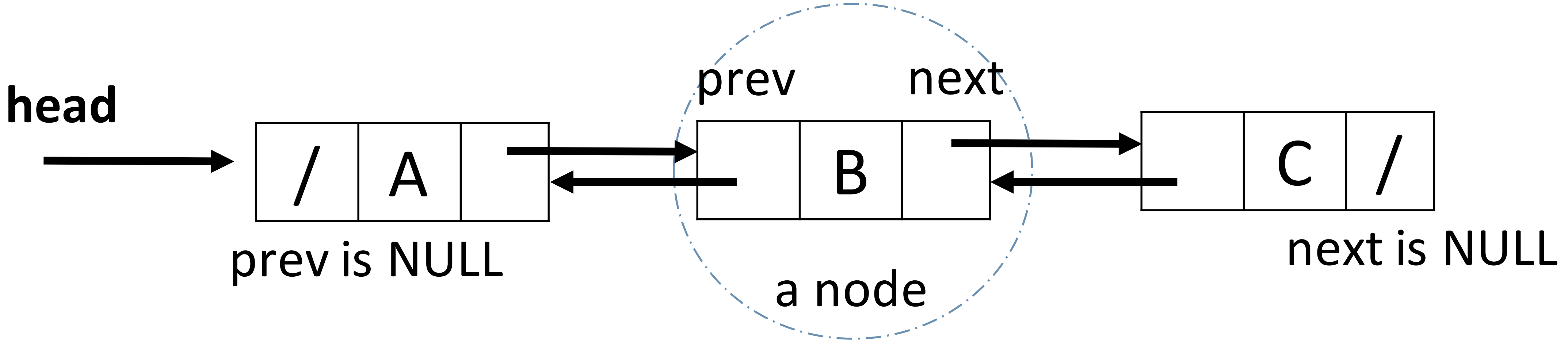
- We call each of these objects a **node** of the linked list.

- Also a **head pointer** points to the first element in the list



Implementation – linked lists

- A ***doubly linked list*** is a very similar data structure where each node stores an additional pointer to the **previous element,** as well.



We will use a doubly linked list to implement the list ADT.

linked lists - $\textsc{Size}()$, $\textsc{IsEmpty}()$

- To implement these two methods we can again use a variable $size$ (just like in the queue implementation). We can increment it on add operations and decrement it on remove operations.

- Next we will explain the more troublesome operations $\textsc{Add}(x)$, $\textsc{Add}(x, i)$, and $\textsc{Remove}(i)$

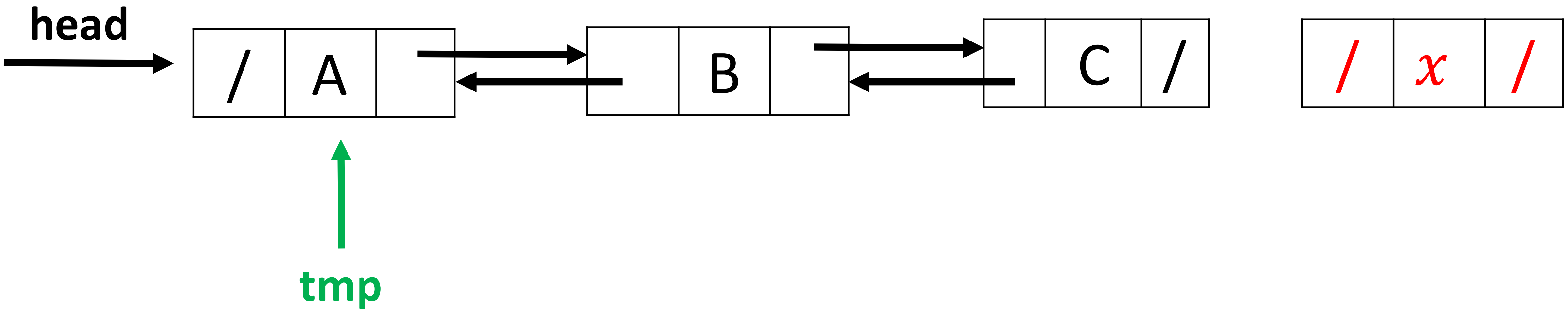
Node class

- To do the implementation we can assume that each node is an object of the following class:

```
Class Node{
        Node next; //points to the next element
        Node prev; //points to the previous element
        Item item; //holds the actual item
}
```

linked lists - $\textsc{Add}(x)$

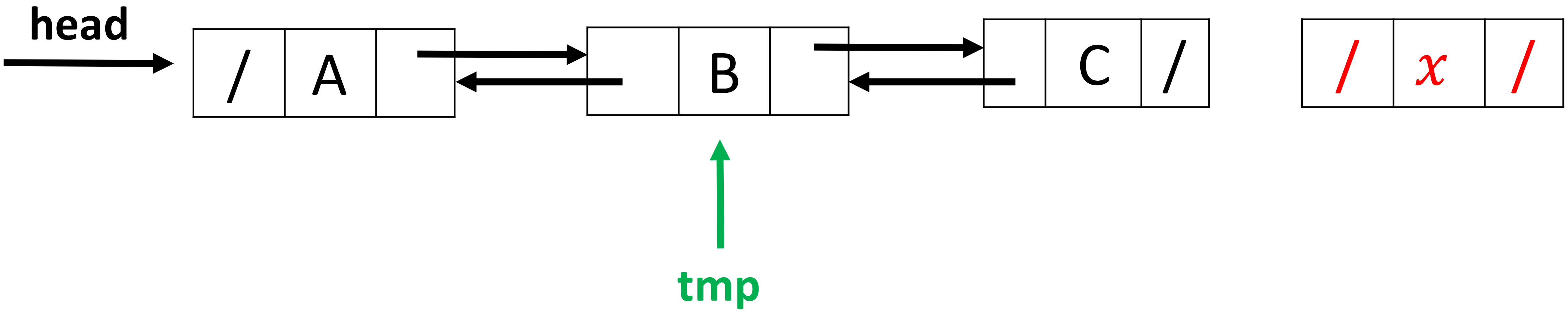
- In order to add to the end of the list, first we use an auxiliary node pointer ***tmp*** to traverse the list and find the last element. Then, we update the pointers.



1) tmp = head

linked lists - $\textsc{Add}(x)$

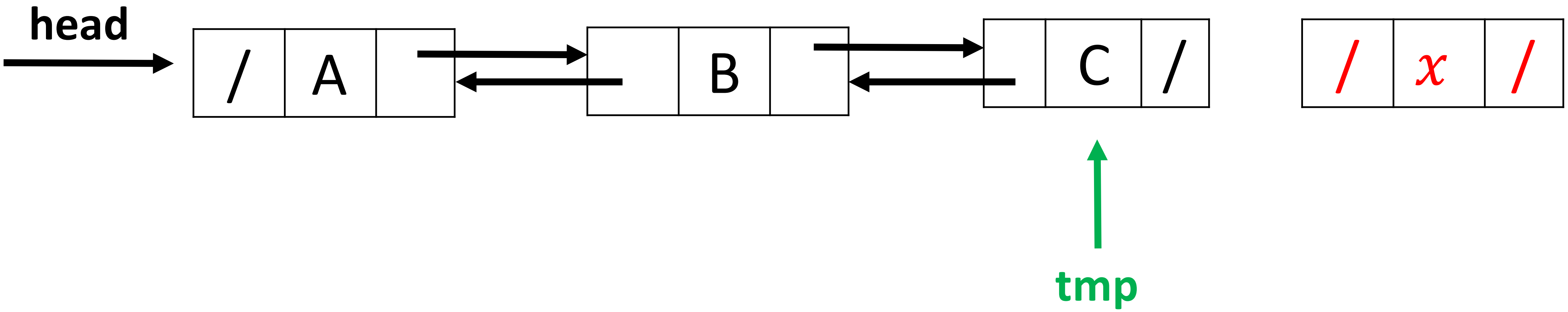
- In order to add to the end of the list, first we use an auxiliary node pointer ***tmp*** to traverse the list and find the last element. Then, we update the pointers.



2) tmp = tmp.next

linked lists - ADD(x)

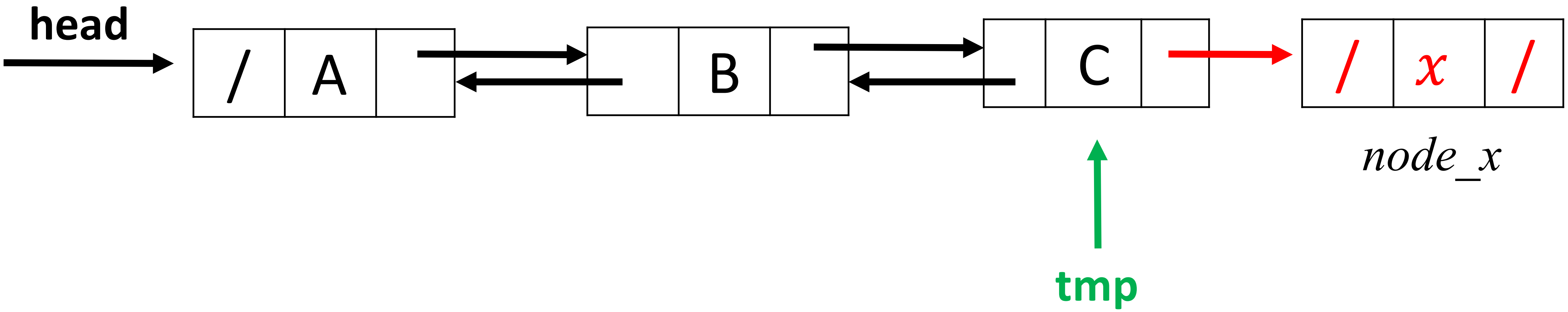
- In order to add to the end of the list, first we use an auxiliary node pointer ***tmp*** to traverse the list and find the last element. Then, we update the pointers.



2) tmp = tmp.next

linked lists - $\textsc{Add}(x)$

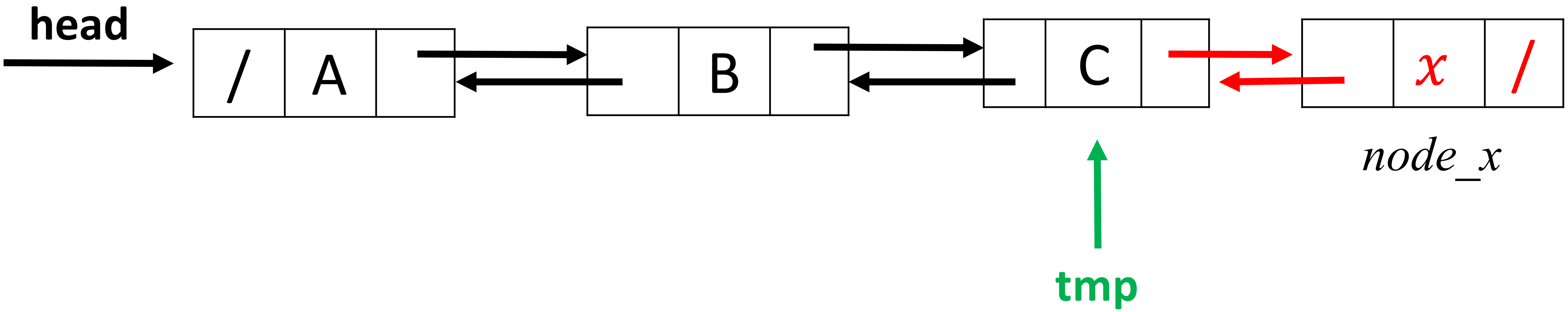
- In order to add to the end of the list, first we traverse the list to get to to the last element, and then update the pointers as follows.



3) tmp.next = node_x

linked lists - ADD(x)

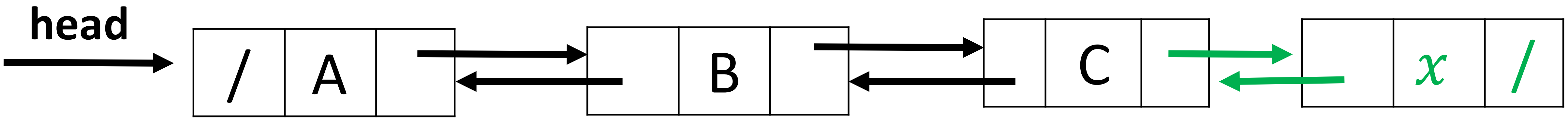
- In order to add to the end of the list, first we traverse the list to get to to the last element, and then update the pointers as follows.



4) node_x.prev = tmp

linked lists - ADD(x)

- In order to add to the end of the list, first we traverse the list to get to to the last element, and then update the pointers as follows.



linked lists - **Add**(x)

```
if head == NULL
    head = node_x //a new node with x as its item
    return
tmp = head
while tmp.next != NULL
    tmp = tmp.next
tmp.next = node_x
node_x.prev = tmp
size = size + 1
```

linked lists - **ADD**(x)

```
1   if head == NULL
2       head = node_x //a new node with x as its item
3       return
4   tmp = head
5   while tmp.next != NULL
6       tmp = tmp.next
7   tmp.next = node_x
8   node_x.prev = tmp
9   size = size + 1
```

We can use the same implementation in a singly linked list by just removing line 8

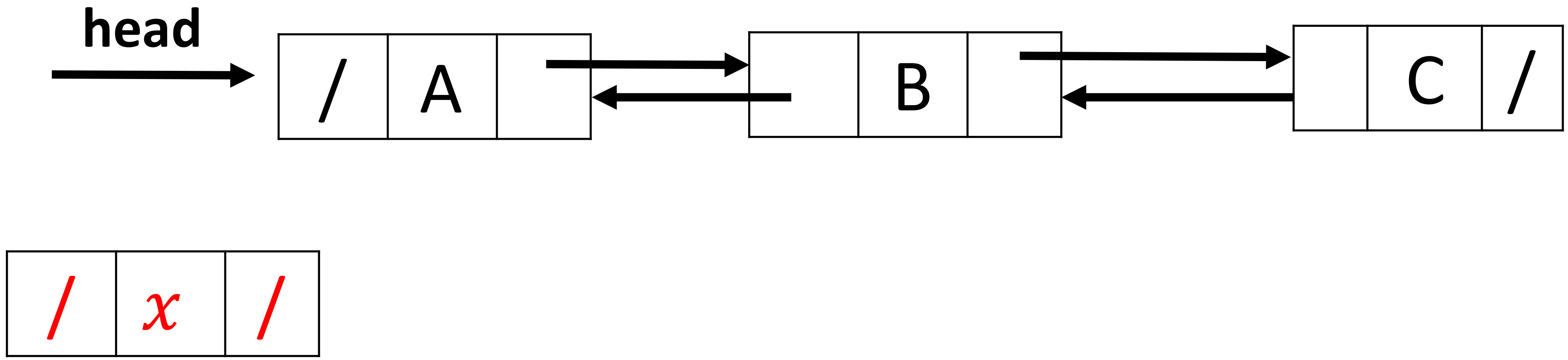
linked lists - CONTAINS(x)

```
tmp = head
while tmp != NULL
    if tmp.item == x
        return TRUE
    tmp = tmp.next
return FALSE
```

The same for singly and double linked list.

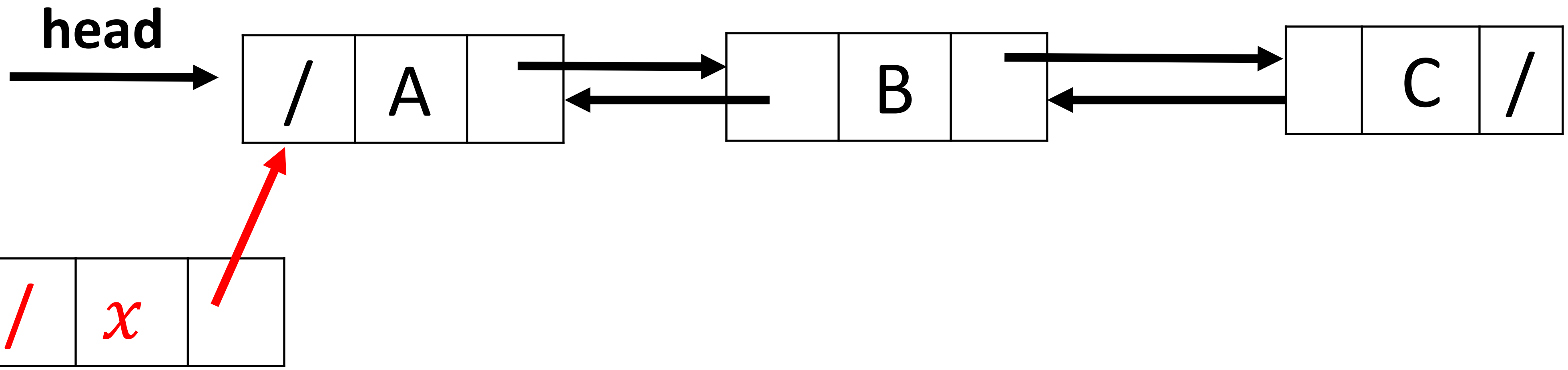
linked lists - $\textsc{Add}(x, i)$

- For simplicity, we check the two cases of $i = 0$ and $i = size()$ separately.



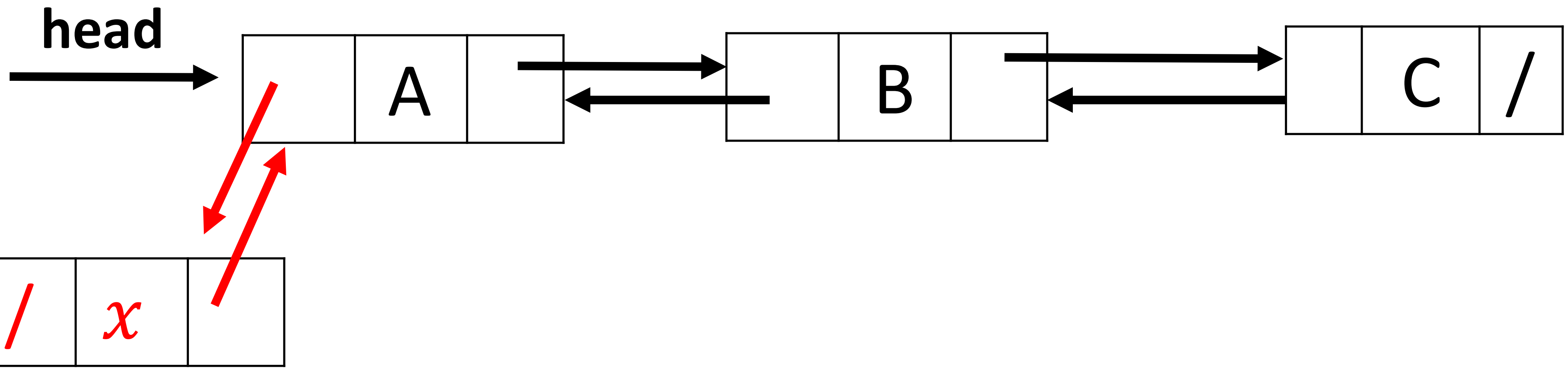
linked lists - $\textbf{Add}(\boldsymbol{x}, \boldsymbol{i})$

- When $i = 0$.



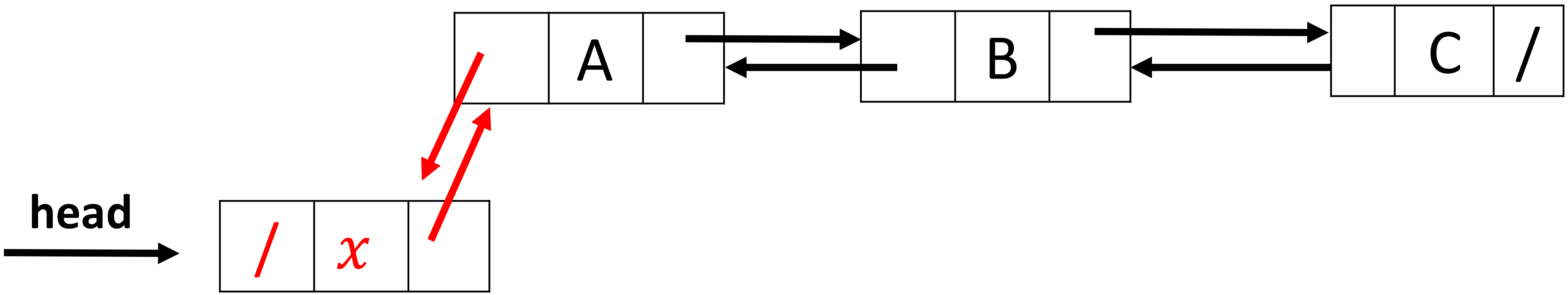
linked lists - $\textbf{A}\text{DD}(\boldsymbol{x}, \boldsymbol{i})$

- For simplicity, we check the two cases of $i = 0$ and $i = size()$ separately.



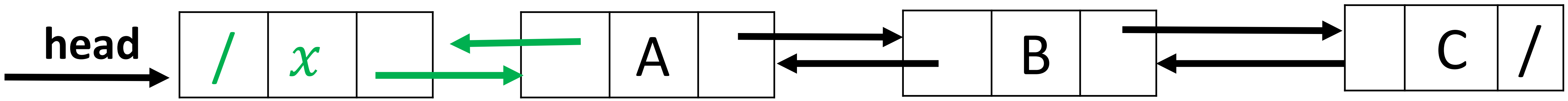
linked lists - $\textbf{Add}(\boldsymbol{x}, \boldsymbol{i})$

- For simplicity, we check the two cases of $i = 0$ and $i = size()$ separately.



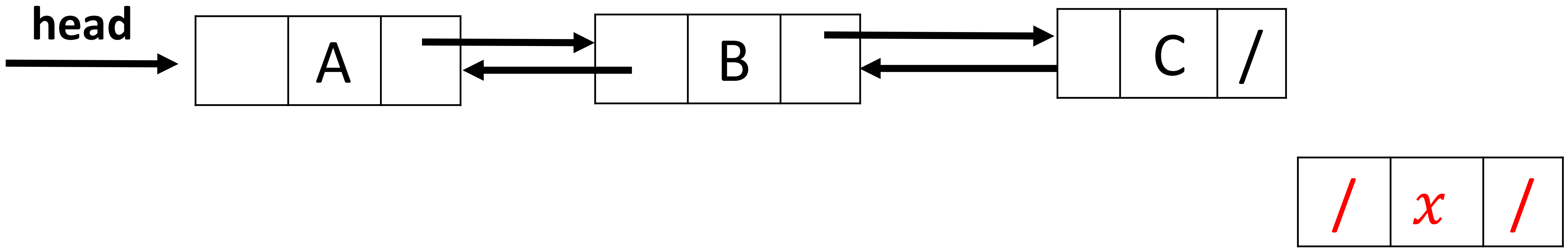
linked lists - $\textbf{A}\textsc{dd}(\boldsymbol{x}, \boldsymbol{i})$

- For simplicity, we check the two cases of $i = 0$ and $i = size()$ separately.



linked lists - $\textbf{A}\textsc{dd}(\boldsymbol{x}, \boldsymbol{i})$

- When $i = size()$ we are basically adding to the end of the list so we can just call Add(x) instead.

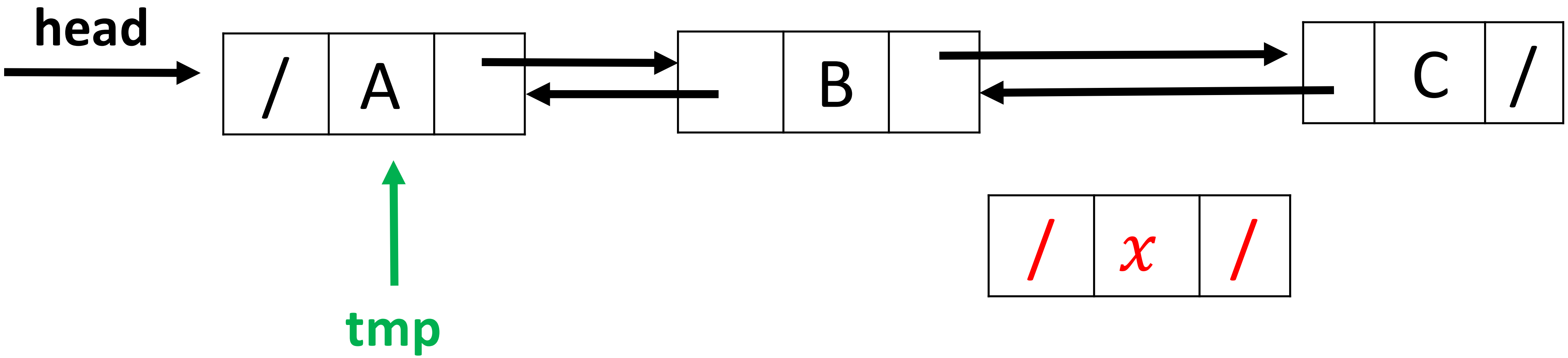


linked lists - ADD(x, i)

```
if i > SIZE()
    return "invalid position error"
if i == SIZE()
    ADD(x); //just add to the end
    return;
if i == 0
    node_x.next = head
    head.prev = node_x
    head = node_x
    return
```

linked lists - $\textbf{A}\textsc{dd}(\boldsymbol{x}, \boldsymbol{i})$

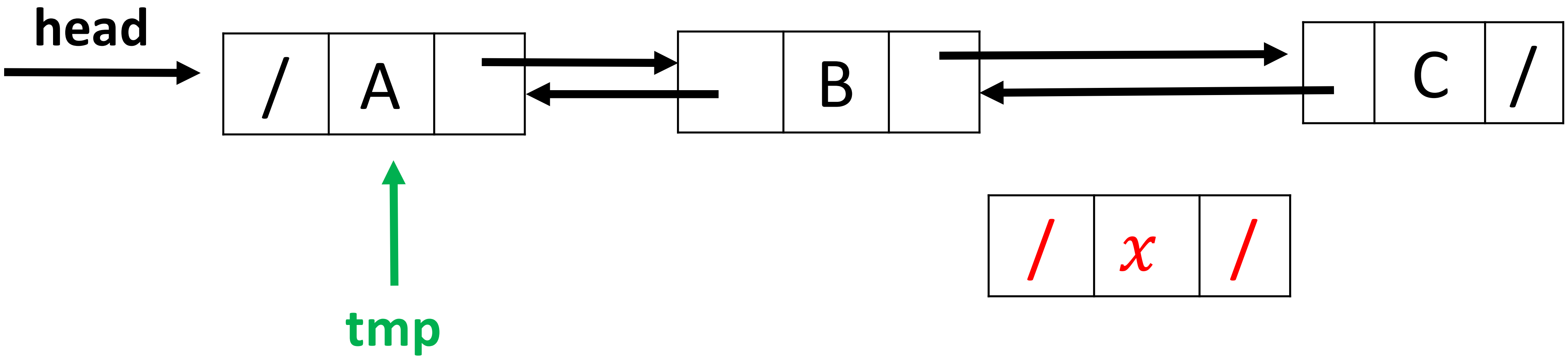
- First we find position i, and update the pointer for the new node, and the nodes at the previous and the next positions as follows.



1) tmp = head

linked lists - $\textbf{ADD}(\boldsymbol{x}, \boldsymbol{i})$

- Example: $\textsc{Add}(x, 2)$

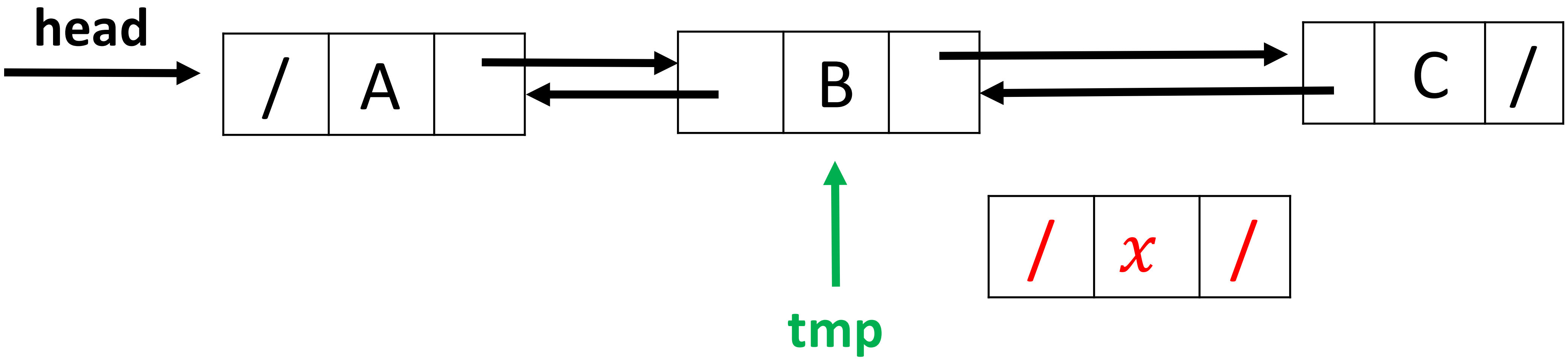


1) tmp = head

x's new position should be 2

linked lists - $\textsc{Add}(\boldsymbol{x}, \boldsymbol{i})$

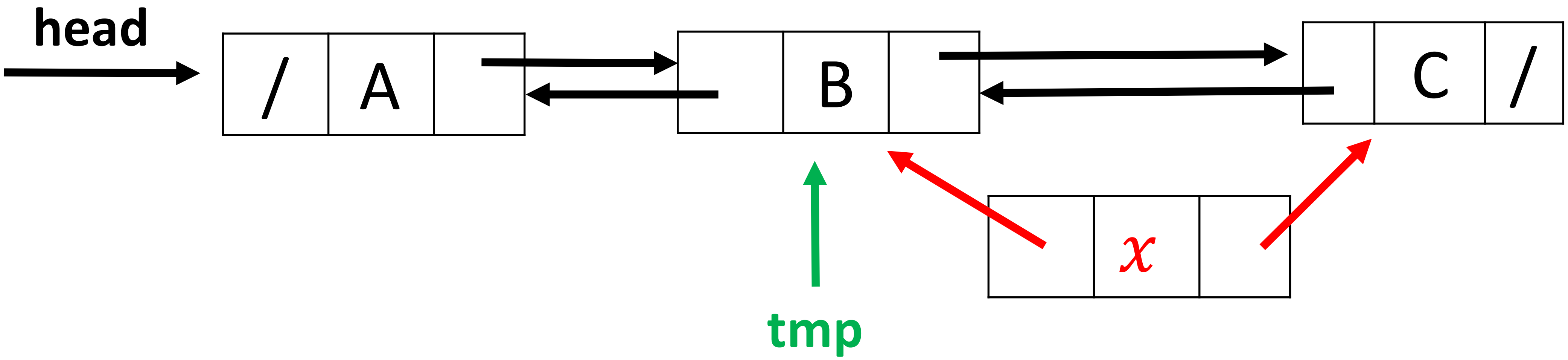
- Example: $\textsc{Add}(x, 2)$



2) tmp = tmp.next

linked lists - $\textbf{Add}(\boldsymbol{x}, \boldsymbol{i})$

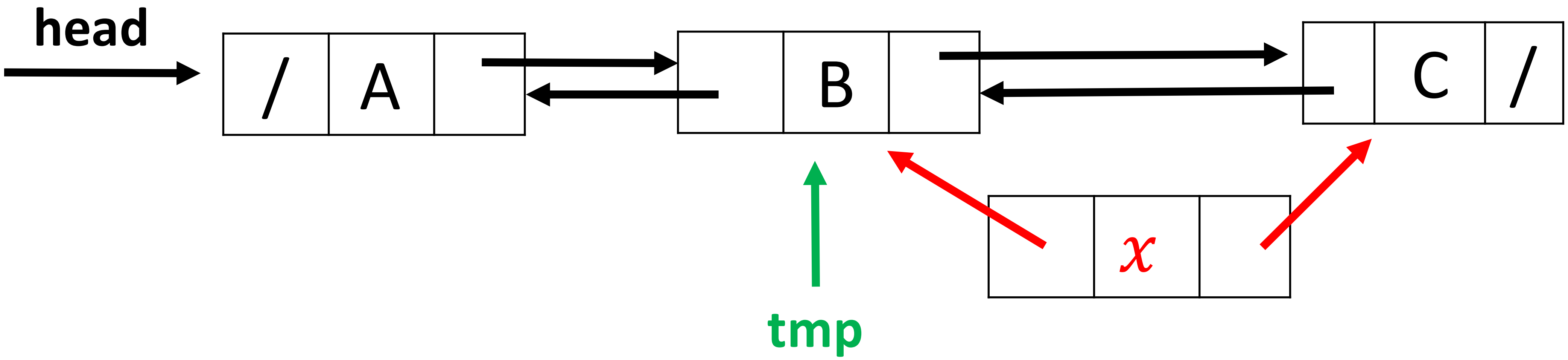
- Example: $\textsc{Add}(x, 2)$



3) node_x.next= tmp.next

linked lists - $\textbf{A}\textsc{dd}(\boldsymbol{x}, \boldsymbol{i})$

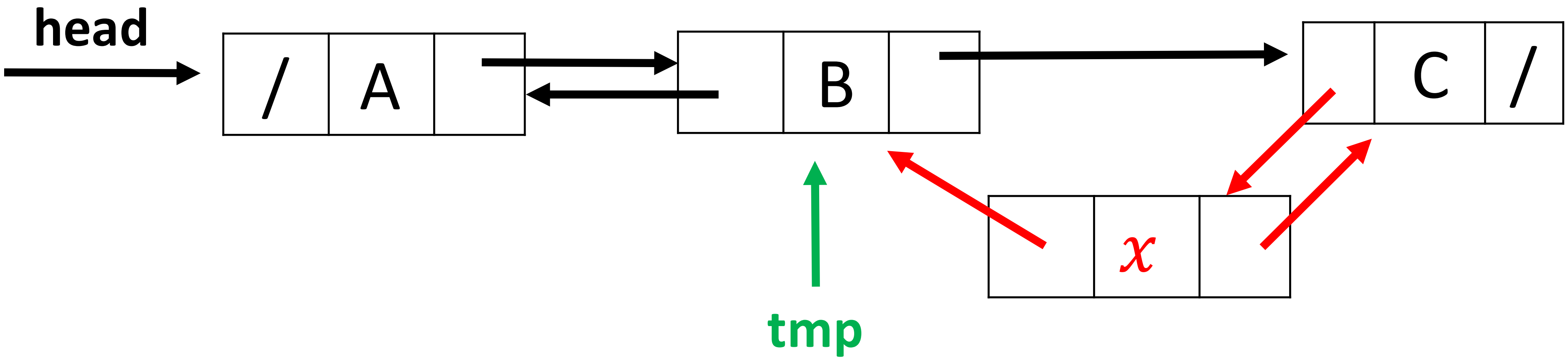
- Example: $\textsc{Add}(x, 2)$



4) node_x.prev = tmp

linked lists - $\textbf{Add}(\boldsymbol{x}, \boldsymbol{i})$

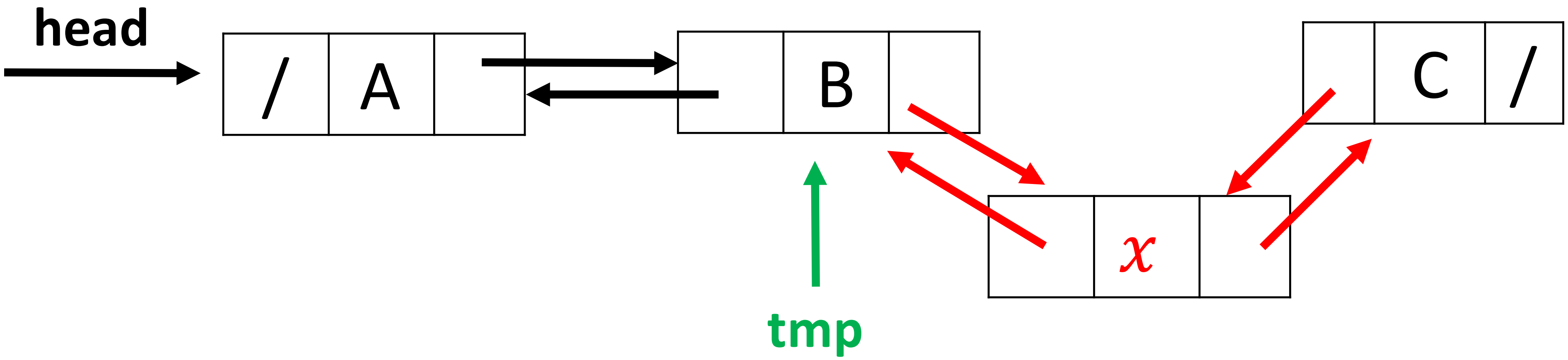
- Example: $\textsc{Add}(x, 2)$



5) node_x.next.prev= node_x

linked lists - $\textbf{A}\small\textbf{DD}(\boldsymbol{x}, \boldsymbol{i})$

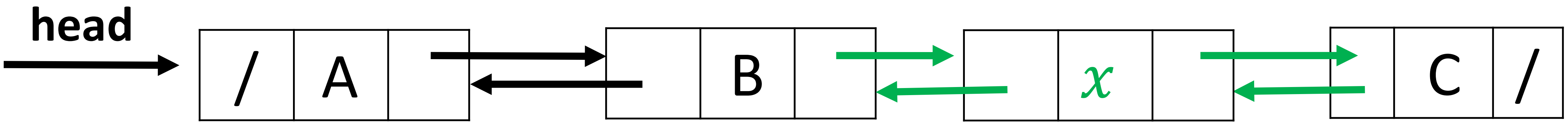
- Example: $\text{A}\small\text{DD}(x, 2)$



6) node_x.prev.next= node_x

linked lists - $\textbf{A}\textbf{DD}(\boldsymbol{x}, \boldsymbol{i})$

- Example: $\textsc{Add}(x, 2)$

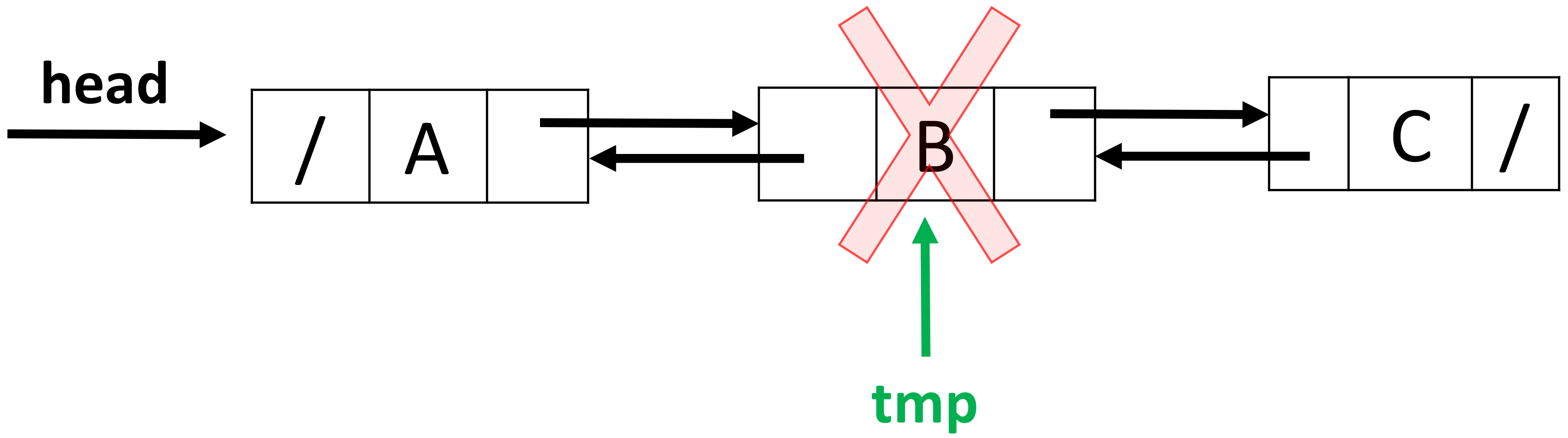


linked lists - ADD(x, i) – when i is not 0 or size()

```
tmp = head
counter = 0
while tmp.next != NULL and counter < i − 1
    tmp = tmp.next
    counter = counter + 1
Update pointers as in steps 3-6 of the slides
size = size + 1
```

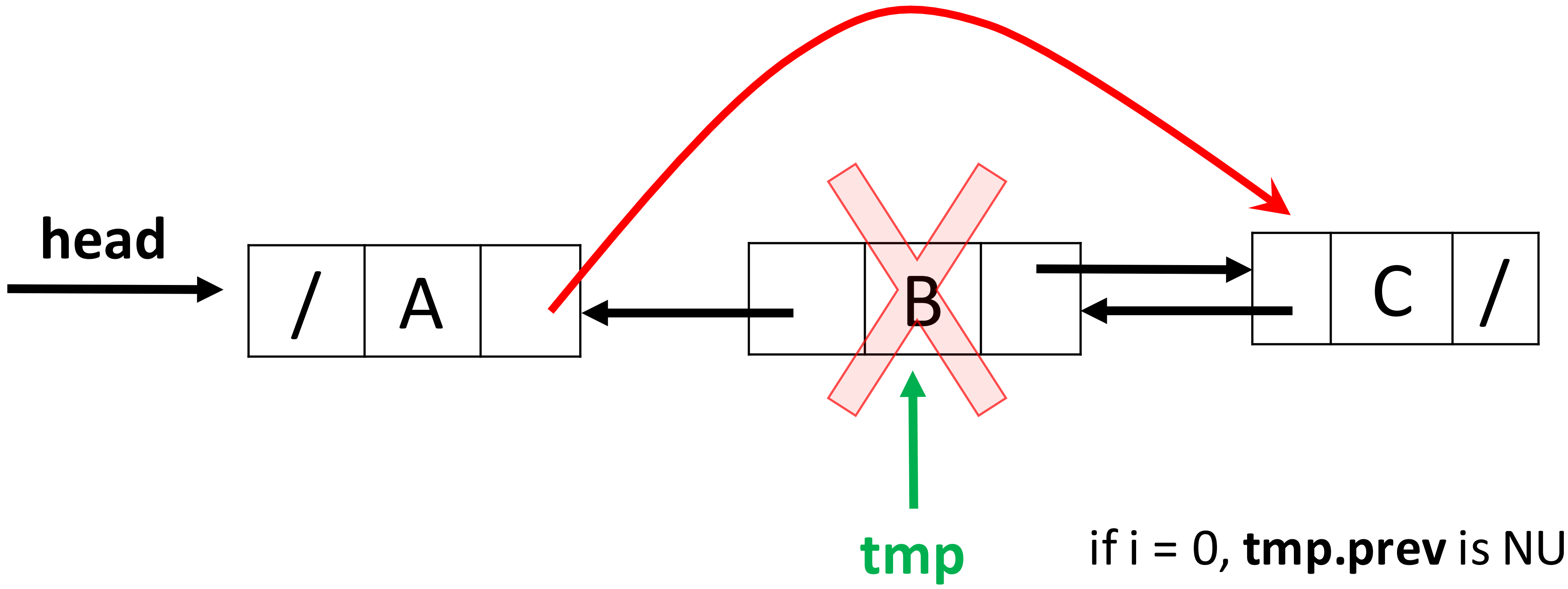
linked lists - $\textsc{Remove}(i)$

- Example: $\textsc{Remove}(1)$



linked lists - $\textsc{Remove}(i)$

- Example: $\textsc{Remove}(1)$

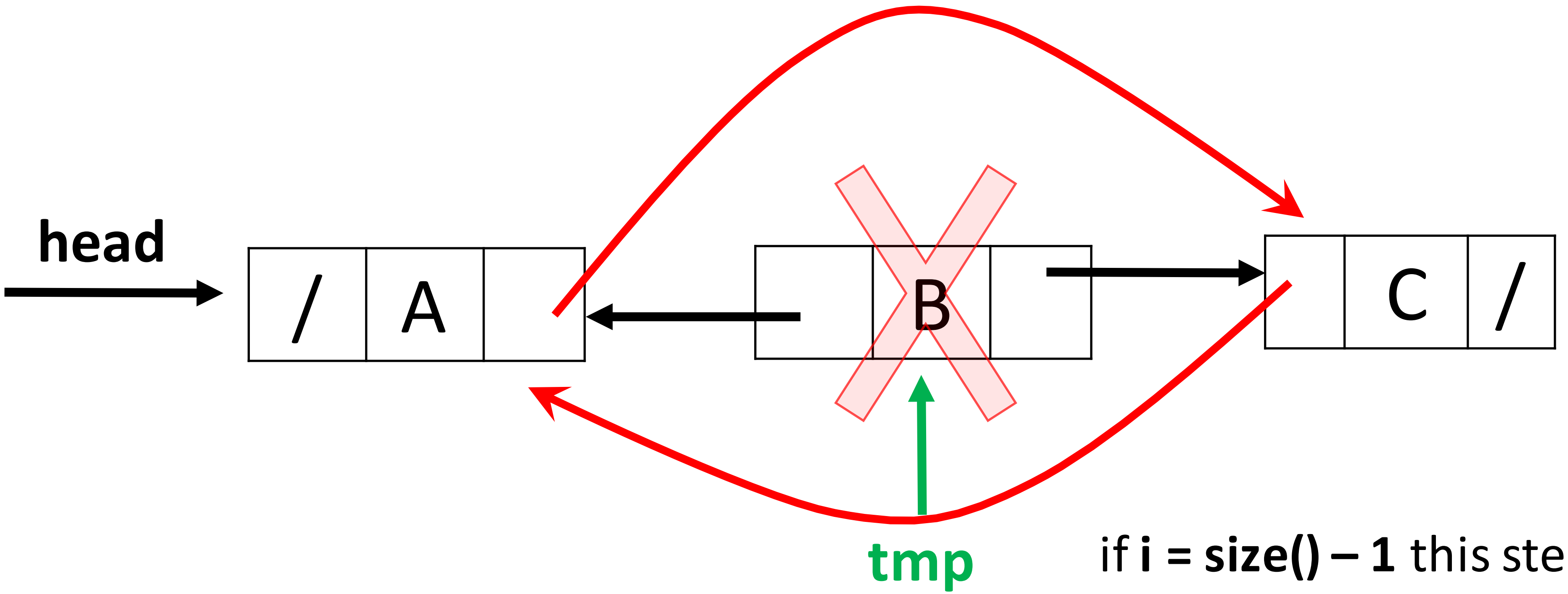


if i = 0, **tmp.prev** is NULL so this step is not necessary. Instead we should update **head = tmp.next**.

1) tmp.prev.next= tmp.next

linked lists - $\textsc{Remove}(i)$

- Example: $\textsc{Remove}(1)$

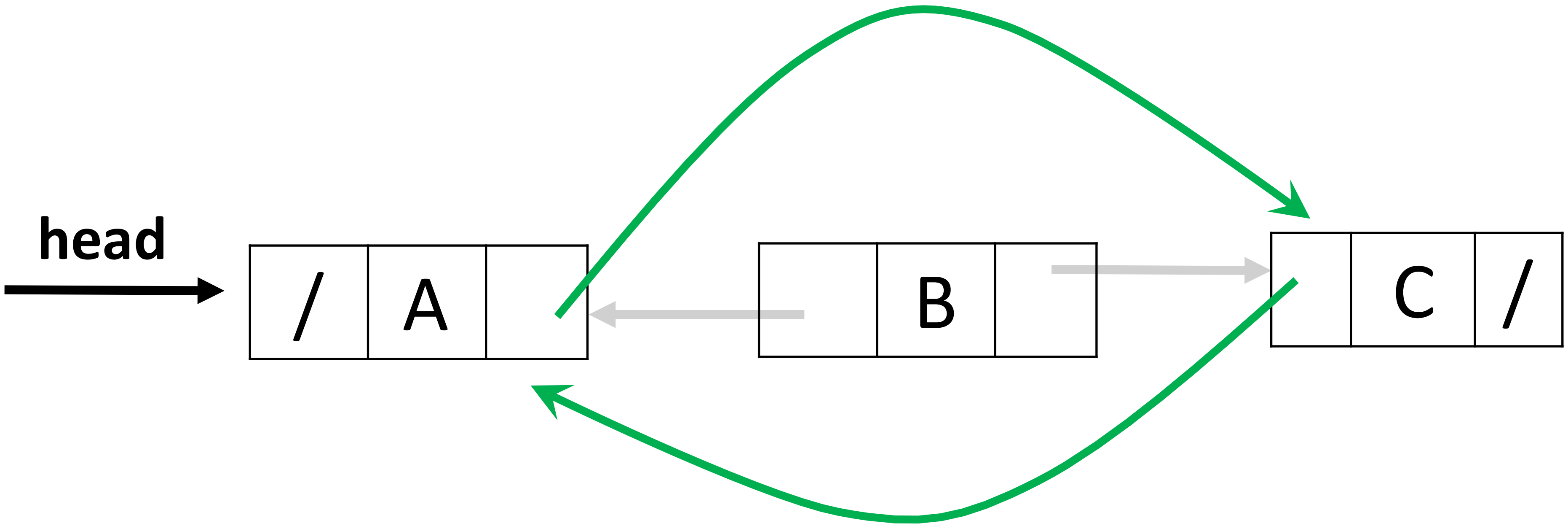


if **i = size() – 1** this step is not necessary. In fact, tmp.next would be NULL in that case.

2) tmp.next.prev= tmp.prev

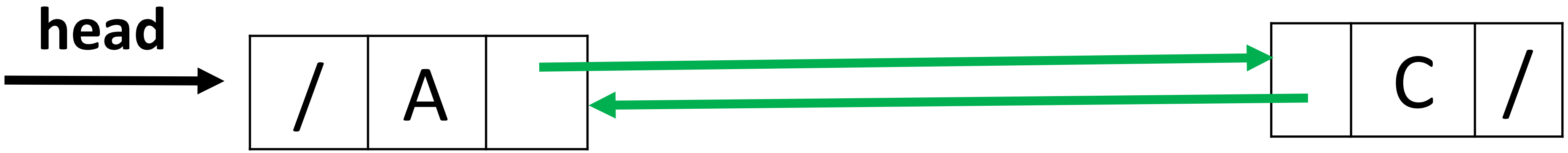
linked lists - **REMOVE(*i*)**

- Example: REMOVE(1)



linked lists - $\textsc{Remove}(i)$

- Example: $\textsc{Remove}(1)$



linked lists - **Remove(i)**

```
if i ≥ Size()
    return "invalid position error"
tmp = head
counter = 0
while tmp.next != NULL and counter < i
    tmp = tmp.next
    counter = counter + 1
Update pointers as in steps 1, 2 in the slides
size = size – 1
return tmp
```

linked lists - REMOVE(i)

```
if i ≥ SIZE()
    return "error"
tmp = head
counter = 0
prev = NULL
while tmp.next != NULL and counter < i
    prev = tmp
    tmp = tmp.next
    counter = counter + 1
.....
```

To do Remove(i) on a singly linked list we have to keep an extra pointer that points to the previous element of *tmp and use that instead of tmp.prev when we want to update.*

linked lists - $\textbf{G}\textsc{et}(\boldsymbol{i})$

- $\textsc{Get}(i)$ can be implemented similar to $\textsc{Remove}(i)$ but instead of removing the tmp node we just return $tmp.item$

Analysis

- If the size of the list is currently n
- $\textsc{Add}(x)$ — $O(n)$ time
- $\textsc{Add}(x, i)$ — $O(i)$ time
- $\textsc{Contains}(x)$ — $O(n)$ time
- $\textsc{Remove}(i)$ — $O(i)$ time
- $\textsc{Get}(i)$ — $O(i)$ time
- $\textsc{IsEmpty}()$ and $\textsc{Size}()$ — $O(1)$ time

Dealing with boundary conditions

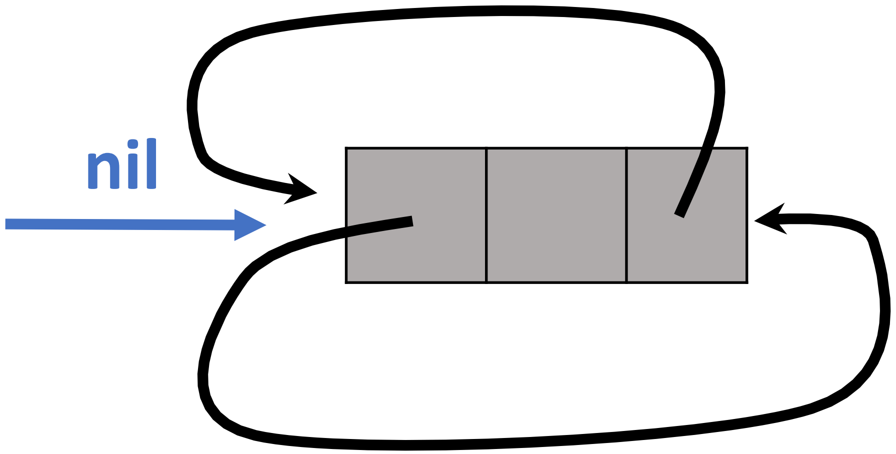
- We can add a **sentinel node** called *nil*, and whenever a *prev* or *next* pointer wants to point to NULL we point to this node.

- This will result in a **circular doubly linked list**. In fact, we assume that the *prev* pointer of the *nil* node is that last node.

- Head pointer is not needed anymore.

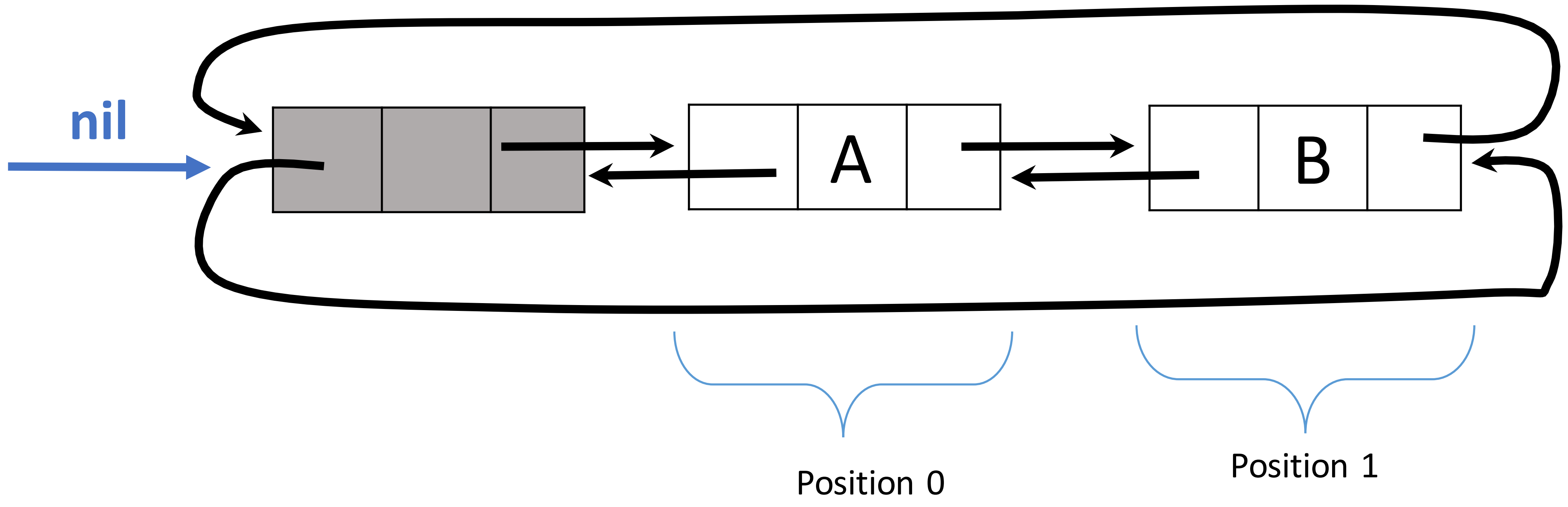
Dealing with boundary conditions

- We can add a **sentinel node** called nil, and whenever a $prev$ or $next$ pointer wants to point to NULL it points to this node.



Dealing with boundary conditions

- We can add a **sentinel node** called *nil*, and whenever a *prev* or *next* pointer wants to point to NULL it points to this node.



Modified ADD(x, i) //assuming i is valid

```
tmp = nil
counter = -1
while counter < i − 1
    tmp = tmp.next
    counter = counter + 1
node_x.next = tmp.next
node_x.prev = tmp
node_x.next.prev = node_x
node_x.prev.next = node_x
size = size + 1
```

Modified ADD(x, i)

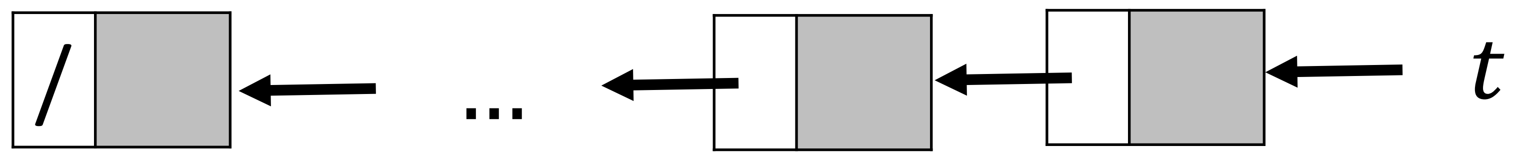
```
tmp = nil
counter = -1
while counter < i – 1
    tmp = tmp.next
    counter = counter + 1
node_x.next = tmp.next
node_x.prev = tmp
node_x.next.prev = node_x
node_x.prev.next = node_x
size = size + 1
```

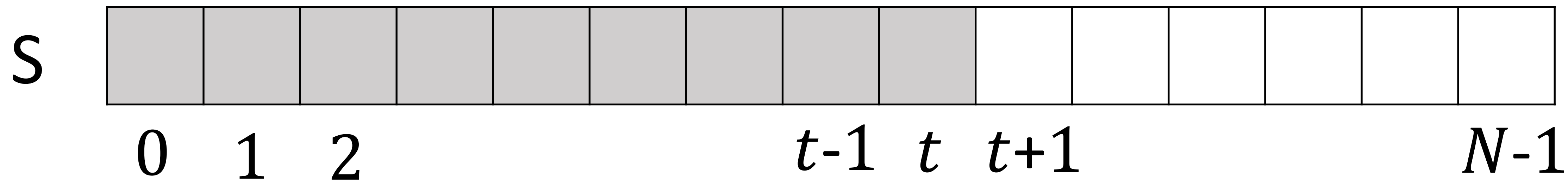
There will be no NULL pointer exception!

Stack and queue using a linked list

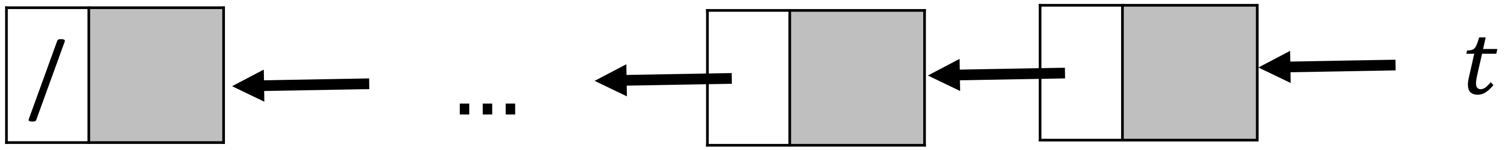
Stack with a singly linked list

t acts as the head pointer in linked list



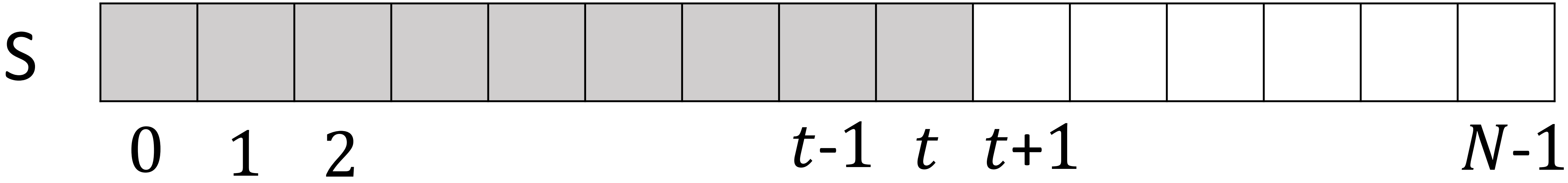


Stack with a singly linked list



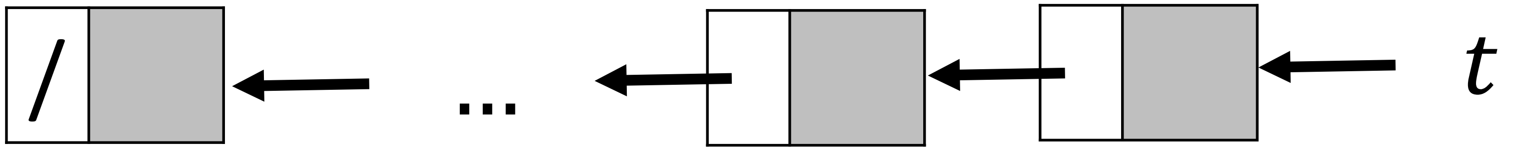
PUSH(x) is equivalent to ADD(x, 0)

POP() is equivalent to REMOVE(0)



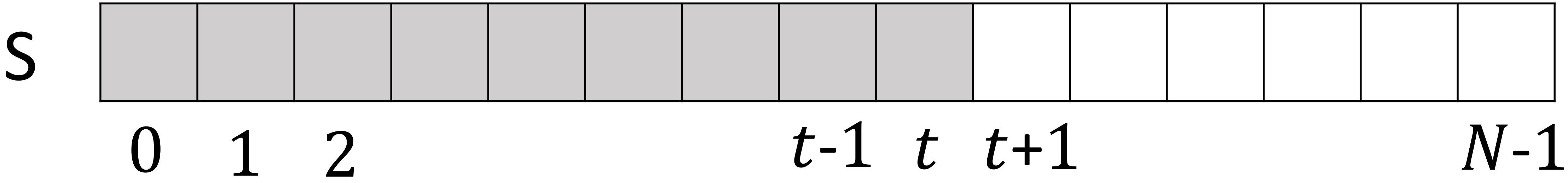
Stack with a singly linked list

Question: Why don't we implement the linked list in a way that the head pointer corresponds to the **bottom** of the stack?



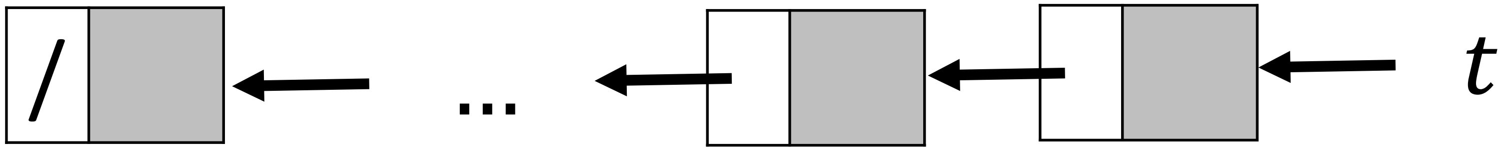
$\text{Push}(x)$ is equivalent to $\text{Add}(x, 0)$

$\text{Pop}()$ is equivalent to $\text{Remove}(0)$

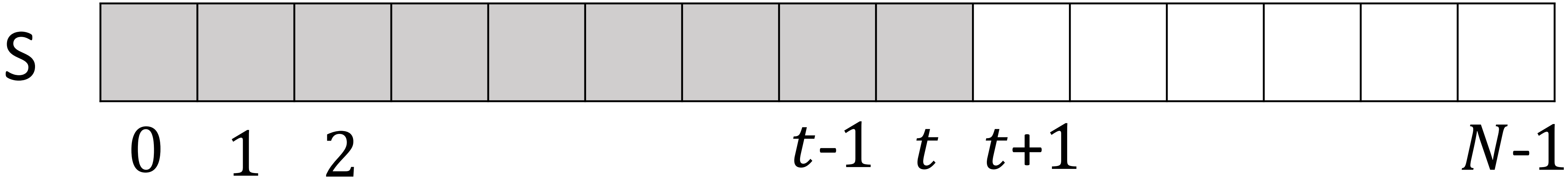


Stack with a singly linked list

Answer: Then, each operation takes $O(n)$ time instead of $O(1)$ if n is the current size of the stack



$\textsc{Push}(x)$ is equivalent to $\textsc{Add}(x, 0)$
$\textsc{Pop}()$ is equivalent to $\textsc{Remove}(0)$



Queue with a linked list

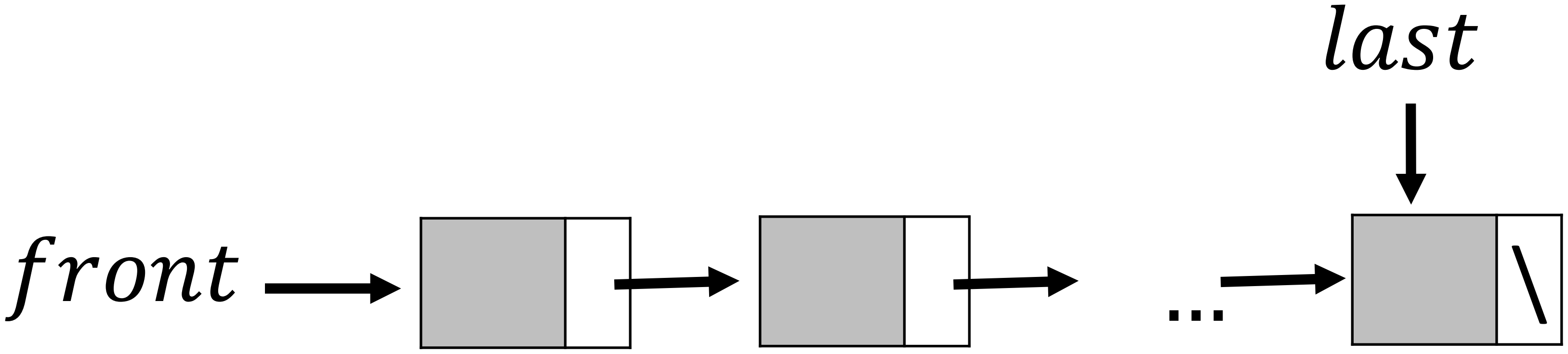
- In a queue **we are modifying both ends** of the queue, so using a simple singly/doubly linked list would be inefficient.

- We could use a **circular doubly linked list** to access both the first and the last element in $O(1)$.
- **Note:** $nil.next$ is the first node, and $nil.prev$ is the last node.

- The **memory efficient** solution is to use a **singly linked list with an added last pointer.**
- If you care more about **simplicity** use the **circular linked list.**

Queue with a singly linked list

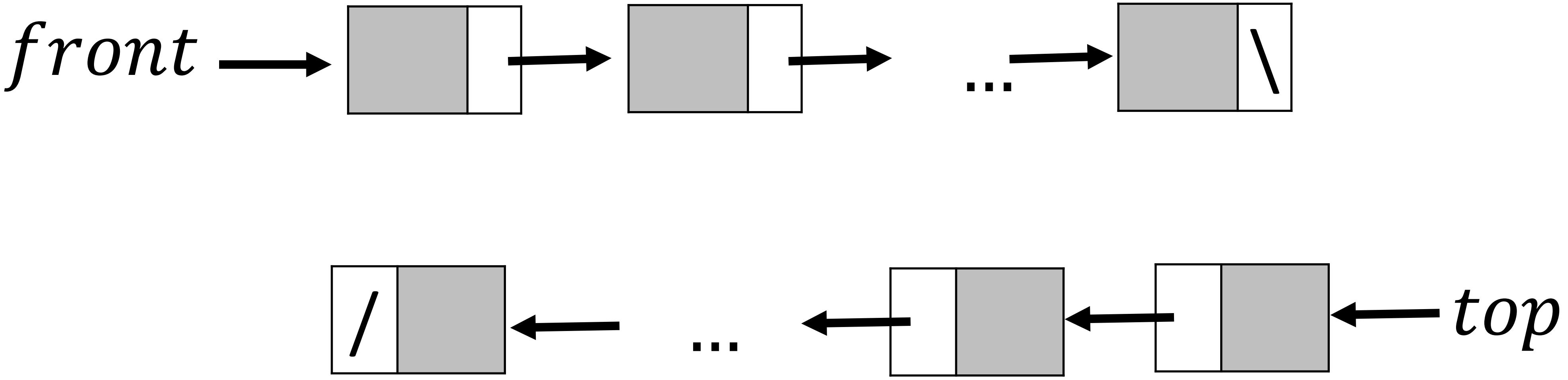
- In order to do Enqueue and Dequeue in $O(1)$ time, we need an extra pointer to the last element of the queue.



- $\textbf{E}\small{\textbf{NQUEUE}}(\boldsymbol{x})$ is equivalent to $\textsc{Add}(x)$, which using the ***last*** pointer can be implemented **in $\boldsymbol{O(1)}$ time.**
- $\textbf{D}\small{\textbf{EQUEUE}}(\ )$ is equivalent to $\textsc{Remove}(0)$.

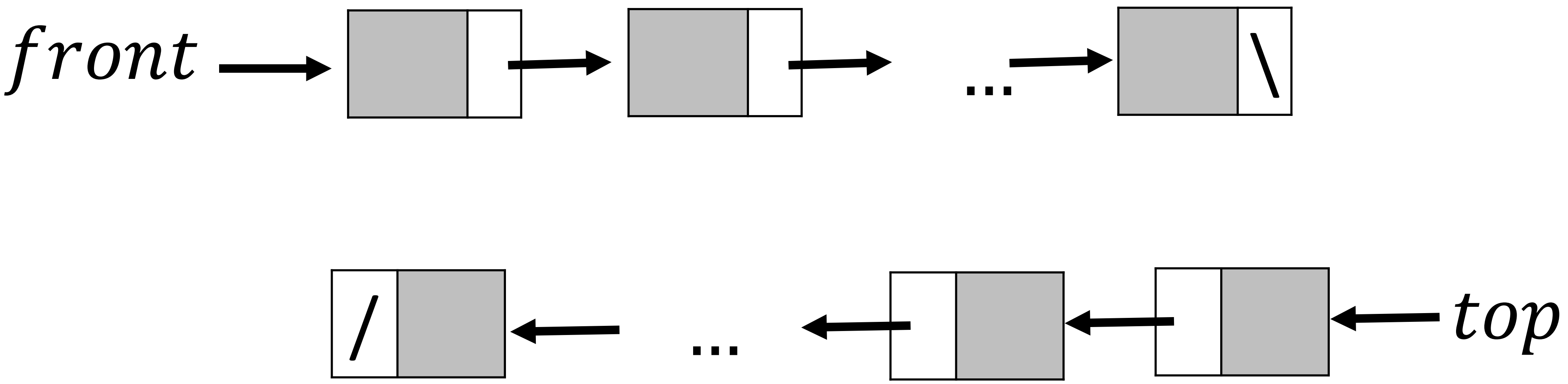
Queue front and stack top

- **Question:** Which linked list operation can be used to return the front of the queue and the top of the stack?



Queue front and stack top

- **Question:** Which linked list operation can be used to return the front of the queue and the top of the stack?
- **Answer:** Get(0)



Contiguous vs linked structures

- There are two types of data structures based on how the memory is allocated for them:

1. **Contiguous structures** are composed of neighboring blocks of memory. Ex. arrays, matrices.
2. **Linked structures** that are composed of non-neighboring blocks of memory **bound together** with pointers. Ex. linked lists, trees (if represented by pointers)

- The items that an array stores are **physically** beside each other in the memory.

Arrays vs linked lists

Arrays advantages:

- The stored objects are **physically continuous in the memory** which allows for fast access time even if we are just iterating through the elements. (known as memory locality)
- Allows for **constant time access** given the index.
- Consist **purely of data** and no extra memory is wasted on pointers.

Linked lists advantages:

- **Flexible** for insert and delete operations. Inserting in the middle of an array may require shifting other elements.
- Overflow error **never happens** in linked lists.
- When working with large objects, it’s more **efficient to work with pointers** than copying the actual object.

Resizable arrays

- Resizable arrays are very interesting data structures that provide the benefits of arrays and linked lists at the same time.

- Technically, resizable arrays (also called dynamic arrays) are arrays.

- However, they have **efficient strategies** for **resizing** if more memory is needed, or if memory is being wasted.

Resizable arrays

- A dynamic array can be used as the **underlying data structure** in the **array-based implementation** of the stack and the queue.

- Stack, ArrayList, Vector in Java use resizable arrays.

- We need to do **amortized analysis** to assess the efficiency of this data structure which looks at the average complexity **over a number of operations** instead of a single operation.

The basic idea

- Say we have a **dynamic array $\boldsymbol{R}$** where R's initial capacity is $N = 1$.

- N shows the **current capacity** of the dynamic array.

- Assume for now that we are implementing a stack using a dynamic array. The extension to queue is not very hard.

The basic idea

- We resize the array R as follows:

1. After a push, if $\text{size} = \boldsymbol{N}$:

Allocate a new array of size $\mathbf{2}\boldsymbol{N}$, copy R to the new array, and replace R with the new array

2. After a pop, if $\text{size} < \boldsymbol{N}/\mathbf{4}$:

Allocate a new array of size $\boldsymbol{N}/\mathbf{2}$, copy R to it and replace R with the new array

The basic idea

- We resize the array R as follows:

1. After a push, if $\text{size} = \boldsymbol{N}$: push takes amortized $O(1)$ time

Allocate a new array of size $\boldsymbol{2N}$, copy R to the new array, and replace R with the new array

2. After a pop, if $\text{size} < \boldsymbol{N/4}$: pop takes amortized $O(1)$ time

Allocate a new array of size $\boldsymbol{N/2}$, copy R to it and replace R with the new array

The basic idea

N = 4

A	B		

The basic idea

N = 4

A	B	C	

The basic idea

N = 4

t = N, so we double the capacity

A	B	C	D

The basic idea

N = 8

t = N, so we double the capacity

A	B	C	D				

The basic idea

$N = 8$

A	B	C	D	E			

The basic idea

N = 8

A	B	C	D	E	F		

The basic idea

N = 8

A	B	C	D	E	F	G	

The basic idea

N = 8

t = N, so we double the size

A	B	C	D	E	F	G	H

The basic idea

N = 16

A	B	C	D	E	F	G	H								

The basic idea

N = 16

Say we have pushed 8 more items

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P

The basic idea

N = 16

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	

The basic idea

N = 16

A	B	C	D	E	F	G	H	I	J	K	L	M	N		

The basic idea

N = 16

A	B	C	D	E	F	G	H	I	J	K	L	M			

The basic idea

N = 16

A	B	C	D	E	F	G	H	I	J	K	L				

The basic idea

N = 16

A	B	C	D	E	F	G	H	I	J	K					

The basic idea

N = 16

A	B	C	D	E	F	G	H	I	J						

The basic idea

N = 16

A	B	C	D	E	F	G	H	I							

The basic idea

N = 16

A	B	C	D	E	F	G	H								

The basic idea

N = 16

A	B	C	D	E	F	G									

The basic idea

A	B	C	D	E	F										

The basic idea

$N = 16$

A	B	C	D	E											

The basic idea

$N = 16$

$t = N/4$, so we halve the size

A	B	C	D												

The basic idea

$N = 8$

A	B	C	D				

Amortized analysis

- There are different techniques for doing amortized analysis but here we use the **aggregate method**.

- In this method, we compute the total running time over $\boldsymbol{n}$ **push** operations, and then **divide by** $\boldsymbol{n}$ to get the amortized time complexity for a single push operation.

$$\textbf{Amortized time per operation} = \frac{\boldsymbol{total\ time\ of\ n\ operatoins}}{\boldsymbol{n}}$$

Amortized analysis

- It is true that a single operation may take a lot of time, but **in reality** we are doing **a lot of operations**.

- So, it makes sense to look at the **amortized cost** in the long run instead of the maximum time per operation.

- Amortized analysis is a **very practical** way to analyze the complexity of various operations in many data structures.

Theorem

- **Theorem:** The amortized time for a push operation is O(1).

- **Proof idea:** Assume that t_i is the running time on the ith push operation. The total running time is then $\sum_{i=1}^{n} t_i$ for n push operations.

- Assume that we call a push operation **heavy** if it causes the size to double, and **light** if it doesn't.

Proof

- We assume that the **initial capacity is 1**.
- A light push takes **constant time c**
- A heavy push is when size becomes a power of 2, i.e. 2^k (for some $k \geq 0$), and takes $c2^{k+1}$
- $\sum_{i=1}^{n} t_i = \sum \mathbf{light\ pushes} + \sum \boldsymbol{heavy\ pushes}$
- Amortized push time = $\frac{\sum_{i=1}^{n} t_i}{n}$

Proof

$$\sum_{i=1}^{n} t_i = \sum_{i=1, i \neq 2^k}^{n} t_i + \sum_{k=0}^{\log n} t_{2^k}$$

Proof

$$\sum_{i=1}^{n} t_i = \sum_{i=1, i \neq 2^k}^{n} t_i + \sum_{k=0}^{\log n} t_{2^k} \leq cn + 4cn = 5cn$$

So, the amortized push time is $\frac{5cn}{n} = 5c = O(1)$

Note that since in operation t_{2^k} , $2^k \leq\ n$, we know that k is at most $\log n$.

Final notes

- We can use the same idea to prove that **each pop** operation also takes **amortized $O(1)$ time**.

- We can also use a dynamic array to do array-based implementation for a queue.

- **Question:** Even though we are taking the average over the cost of operations, why are we not calling this average-case analysis?

Final notes

- We can use the same idea to prove that **each pop** operation also takes **amortized $\boldsymbol{O(1)}$ time**.

- We can also use a dynamic array to do array-based implementation for a queue.

- **Answer:** We are not getting the average over different inputs. Instead, it's an average over a sequence of operations. Usually, amortized analysis is used to analyze the efficiency of an operation in a data structure.