8. Fundamental Data Structures

Abstract data types stack, queue, implementation variants for linked lists, [Ottman/Widmayer, Kap. 1.5.1-1.5.2, Cormen et al, Kap. 10.1.-10.2]

Abstract Data Types

We recall

A *stack* is an abstract data type (ADR) with operations

Abstract Data Types

We recall

A *stack* is an abstract data type (ADR) with operations

- $\texttt{push}(x, S)$: Puts element x on the stack S.

Abstract Data Types

We recall

A *stack* is an abstract data type (ADR) with operations

- **push**(x, S): Puts element x on the stack S.
- **pop**(S): Removes and returns top most element of S or **null**

Abstract Data Types

We recall

A *stack* is an abstract data type (ADR) with operations

- **push**(x, S): Puts element x on the stack S.
- **pop**(S): Removes and returns top most element of S or **null**
- **top**(S): Returns top most element of S or **null**.

Abstract Data Types

We recall

A *stack* is an abstract data type (ADR) with operations

- `push`(x, S): Puts element x on the stack S.
- `pop`(S): Removes and returns top most element of S or `null`
- `top`(S): Returns top most element of S or `null`.
- `isEmpty`(S): Returns `true` if stack is empty, `false` otherwise.

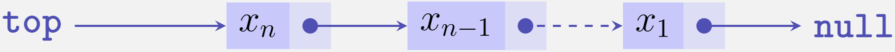
Abstract Data Types

We recall

A *stack* is an abstract data type (ADR) with operations

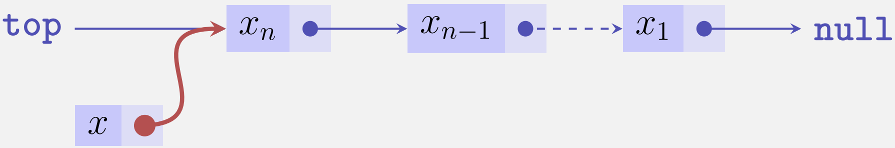
- **push**(x, S): Puts element x on the stack S.
- **pop**(S): Removes and returns top most element of S or **null**
- **top**(S): Returns top most element of S or **null**.
- **isEmpty**(S): Returns **true** if stack is empty, **false** otherwise.
- **emptyStack**$()$: Returns an empty stack.

Implementation Push



push(x, S):

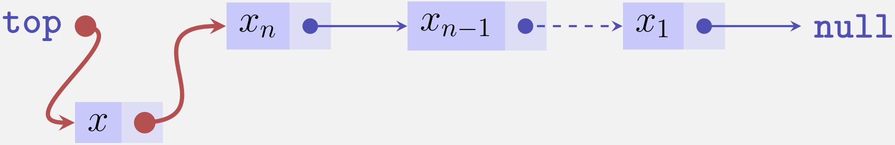
Implementation Push



`push`(x, S):

1. Create new list element with x and pointer to the value of `top`.

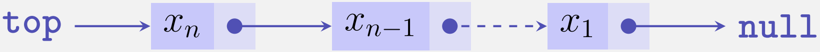
Implementation Push



push(x, S):

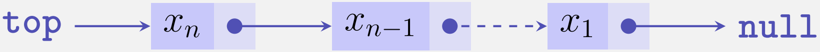
1. Create new list element with x and pointer to the value of **top**.
2. Assign the node with x to **top**.

Implementation Pop



$\texttt{pop}(S)$:

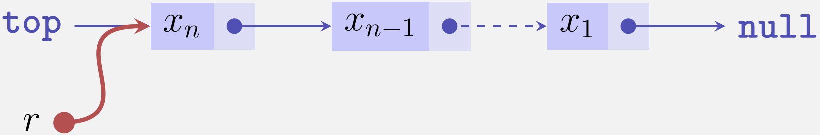
Implementation Pop



`pop`(S):

1. If `top`=`null`, then return `null`

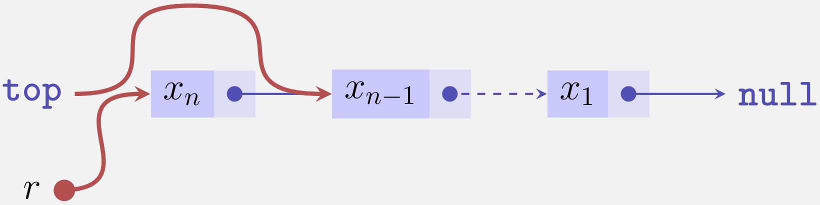
Implementation Pop



pop(S):

1. If top=null, then return null
2. otherwise memorize pointer p of top in r.

Implementation Pop



pop(S):

1. If top=null, then return null
2. otherwise memorize pointer p of top in r.
3. Set top to $p.next$ and return r

Analysis

Each of the operations `push`, `pop`, `top` and `isEmpty` on a stack can be executed in $\mathcal{O}(1)$ steps.

Queue (fifo)

A queue is an ADT with the following operations

Queue (fifo)

A queue is an ADT with the following operations

- `enqueue`(x, Q): adds x to the tail (=end) of the queue.

Queue (fifo)

A queue is an ADT with the following operations

- `enqueue`(x, Q): adds x to the tail (=end) of the queue.
- `dequeue`(Q): removes x from the head of the queue and returns x (`null` otherwise)

Queue (fifo)

A queue is an ADT with the following operations

- `enqueue`(x, Q): adds x to the tail (=end) of the queue.
- `dequeue`(Q): removes x from the head of the queue and returns x (`null` otherwise)
- `head`(Q): returns the object from the head of the queue (`null` otherwise)

Queue (fifo)

A queue is an ADT with the following operations

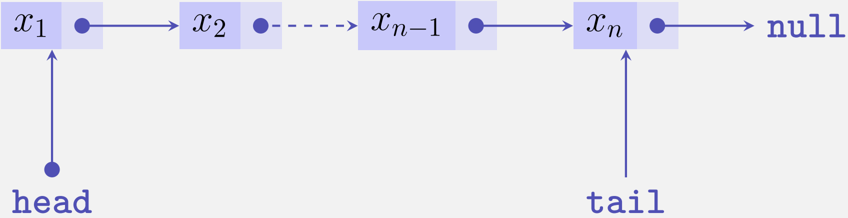
- `enqueue`(x, Q): adds x to the tail (=end) of the queue.
- `dequeue`(Q): removes x from the head of the queue and returns x (`null` otherwise)
- `head`(Q): returns the object from the head of the queue (`null` otherwise)
- `isEmpty`(Q): return `true` if the queue is empty, otherwise `false`

Queue (fifo)

A queue is an ADT with the following operations

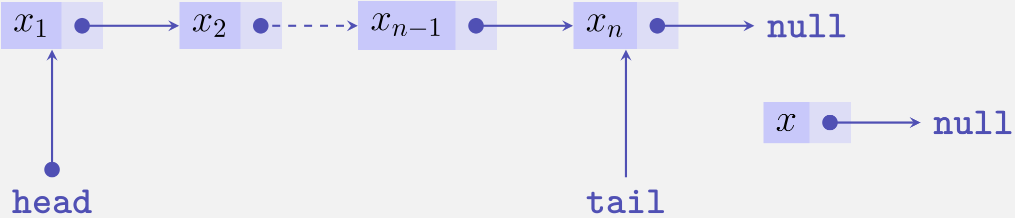
- `enqueue`(x, Q): adds x to the tail (=end) of the queue.
- `dequeue`(Q): removes x from the head of the queue and returns x (`null` otherwise)
- `head`(Q): returns the object from the head of the queue (`null` otherwise)
- `isEmpty`(Q): return `true` if the queue is empty, otherwise `false`
- `emptyQueue`$()$: returns empty queue.

Implementation Queue



enqueue(x, S):

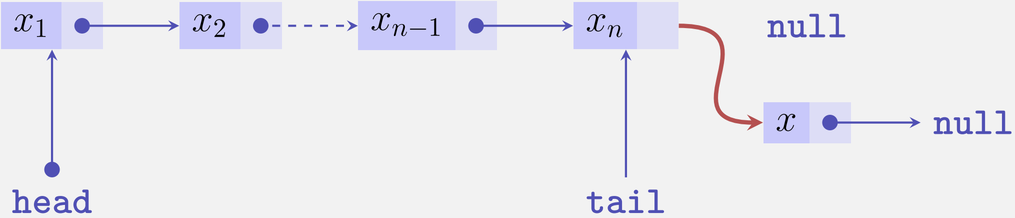
Implementation Queue



`enqueue`(x, S):

1. Create a new list element with x and pointer to `null`.

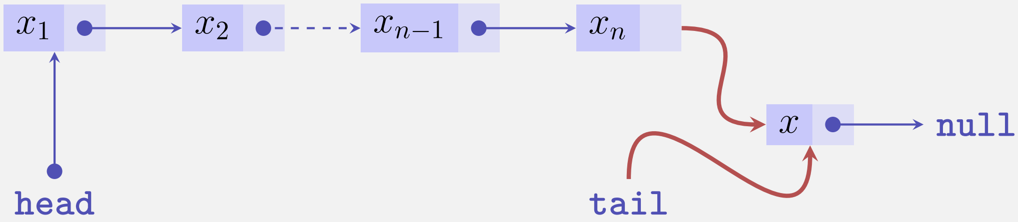
Implementation Queue



$\texttt{enqueue}(x, S)$:

1. Create a new list element with x and pointer to `null`.
2. If `tail` $\neq$ `null`, then set `tail.next` to the node with x.

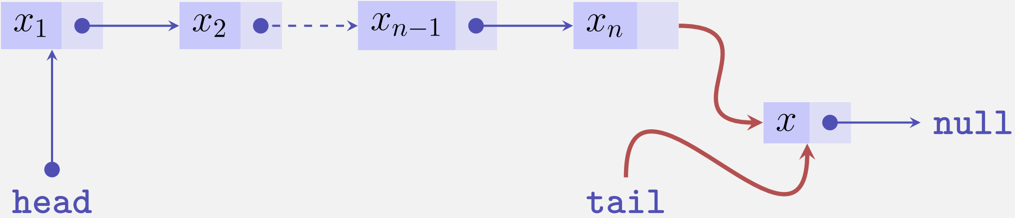
Implementation Queue



`enqueue`(x, S):

1. Create a new list element with x and pointer to `null`.
2. If `tail` $\neq$ `null`, then set `tail.next` to the node with x.
3. Set `tail` to the node with x.

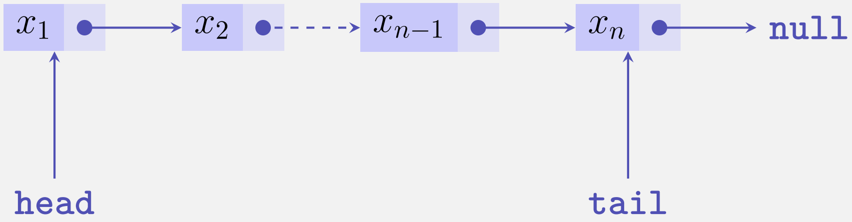
Implementation Queue



enqueue(x, S):

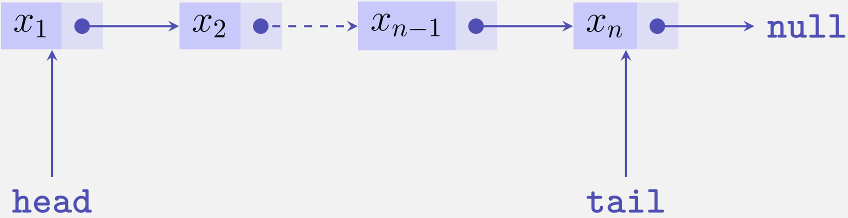
1. Create a new list element with x and pointer to **null**.
2. If **tail** $\neq$ **null**, then set **tail.next** to the node with x.
3. Set **tail** to the node with x.
4. If **head** $=$ **null**, then set **head** to **tail**.

Invariants



With this implementation it holds that

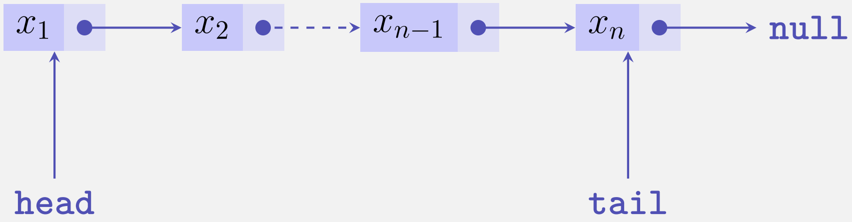
Invariants



With this implementation it holds that

- either head = tail = null,

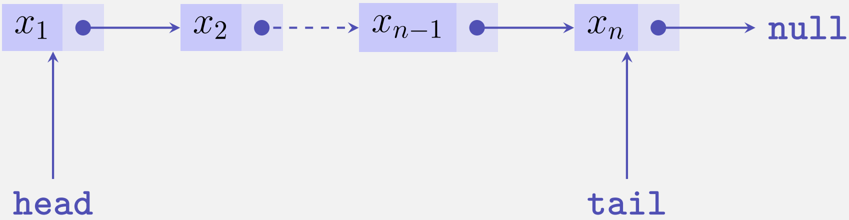
Invariants



With this implementation it holds that

- either `head` $=$ `tail` $=$ `null`,
- or `head` $=$ `tail` $\neq$ `null` and `head.next` $=$ `null`

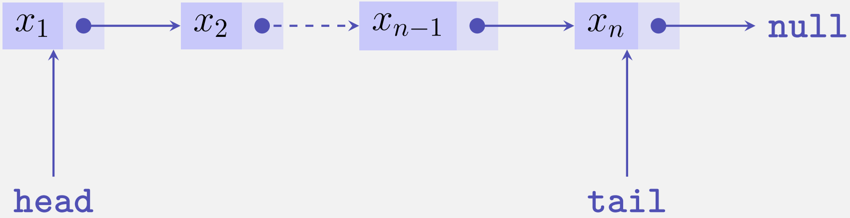
Invariants



With this implementation it holds that

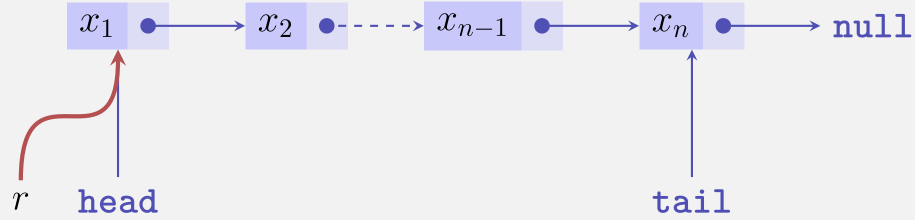
- either $\texttt{head} = \texttt{tail} = \texttt{null}$,
- or $\texttt{head} = \texttt{tail} \neq \texttt{null}$ and $\texttt{head.next} = \texttt{null}$
- or $\texttt{head} \neq \texttt{null}$ and $\texttt{tail} \neq \texttt{null}$ and $\texttt{head} \neq \texttt{tail}$ and $\texttt{head.next} \neq \texttt{null}$.

Implementation Queue



dequeue(S):

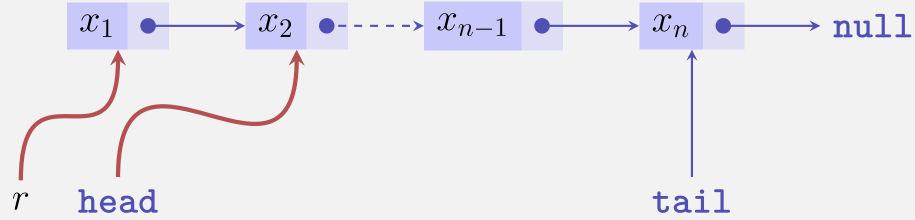
Implementation Queue



$\texttt{dequeue}(S)$:

1. Store pointer to head in r. If $r = \texttt{null}$, then return r .

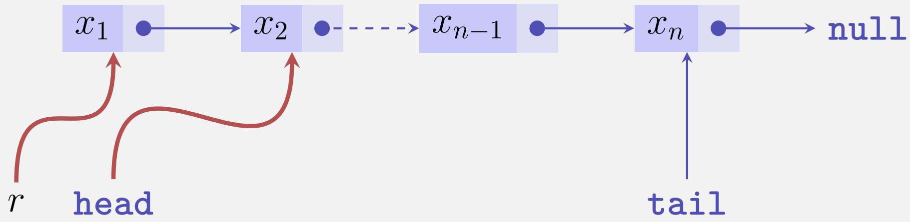
Implementation Queue



dequeue(S):

1. Store pointer to head in r. If $r =$ null, then return r .
2. Set the pointer of head to head.next.

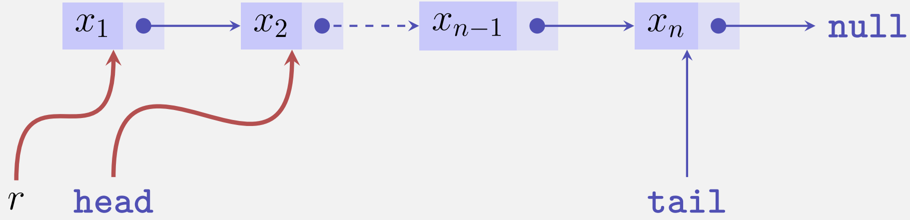
Implementation Queue



dequeue(S):

1. Store pointer to head in r. If $r = $ null, then return r .
2. Set the pointer of head to head.next.
3. Is now head $=$ null then set tail to null.

Implementation Queue



dequeue(S):

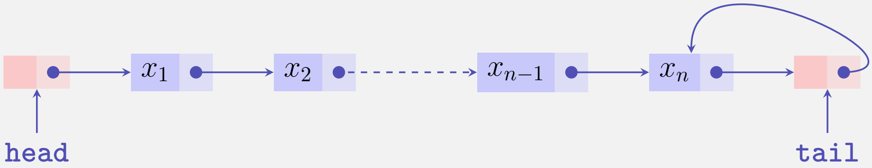
1. Store pointer to head in r. If $r = \texttt{null}$, then return r .
2. Set the pointer of head to head.next.
3. Is now $\texttt{head} = \texttt{null}$ then set tail to null.
4. Return the value of r.

Analysis

Each of the operations `enqueue`, `dequeue`, `head` and `isEmpty` on the queue can be executed in $\mathcal{O}(1)$ steps.

Implementation Variants of Linked Lists

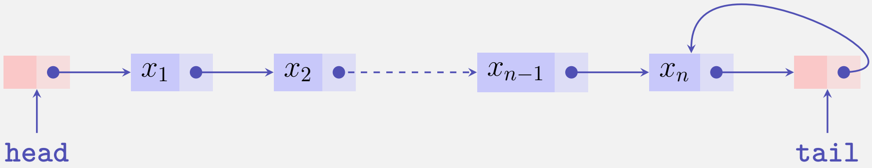
List with dummy elements (sentinels).



Advantage: less special cases

Implementation Variants of Linked Lists

List with dummy elements (sentinels).

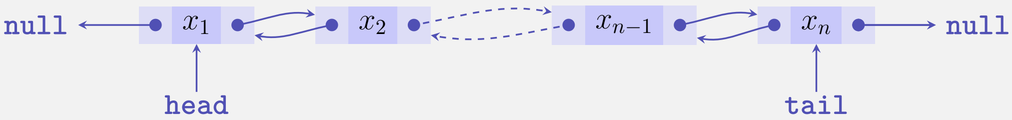


Advantage: less special cases

Variant: like this with pointer of an element stored singly indirect. (Example: pointer to x_3 points to x_2.)

Implementation Variants of Linked Lists

Doubly linked list



Overview

	enqueue	delete	search	concat
(A)	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
(B)	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
(C)	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
(D)	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$

(A) = singly linked
(B) = Singly linked with dummy element at the beginning and the end
(C) = Singly linked with indirect element addressing
(D) = doubly linked