November 5, 2020

Simple Data Structures

Data Structure

- Type
- Operations

Operations

- search (H, k)
- Insert (H, x)
- Delete (H, x)

- Minimum (H)
- Maximum (H)
- Previous (H, x)
- Next (H, x)

⋮

Basic Data Structures

→ Stack

→ queue

→ priority queue

→ linked lists

Advanced Data Structures

- → binary search trees (balanced)
- → balanced search trees (for secondary storage)
- → union-find
- → hash tables

⋮

Stack

In a stack the element we can remove is determined, this the last inserted element → LIFO

Operations:

- Push (Insert)
- Pop (Delete)
- Empty ?

Implementation using an array $V[1:n]$
Attribute $Top[V]$ shows the index of the element last inserted, thus the stack consists of elements $V[1], V[2], \ldots, V[Top[V]]$. Initially $Top[V] = 0$, this means that the stack is empty.
We can check whether the stack is empty by operation Empty?

If we want to pop from an empty stack, this is the stack underflow error.

If Top[V] = n then the stack is full; if we want to push to a full stack, this is the stack overflow error.

```
Full? (V)
if Top[V] = n
    then return true
    else return false

Push (V, x)
if Full? (V)
    then
        error "Stack overflow"
    else
    ⋮
```

⋮

$Top[V]++$

$V[Top[V]] := x$

Empty?(V)

if $Top[V] = 0$

then return true

else return false

Pop(V)

if Empty?(V)

then

⋮

⋮ error "Stack underflow"
else
Top[V] --
return V[Top[V]+1]

Time complexity for all operations are $\Theta(1)$

Queue

The element we can remore is determined here, also, but now this in the element first inserted → FIFO

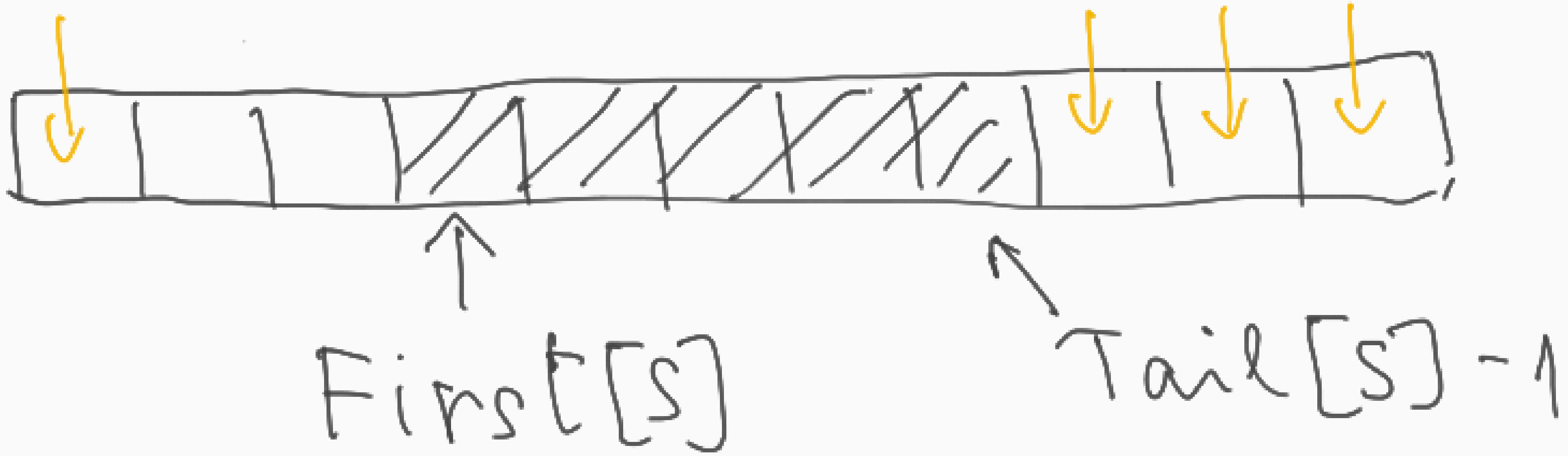
Operations

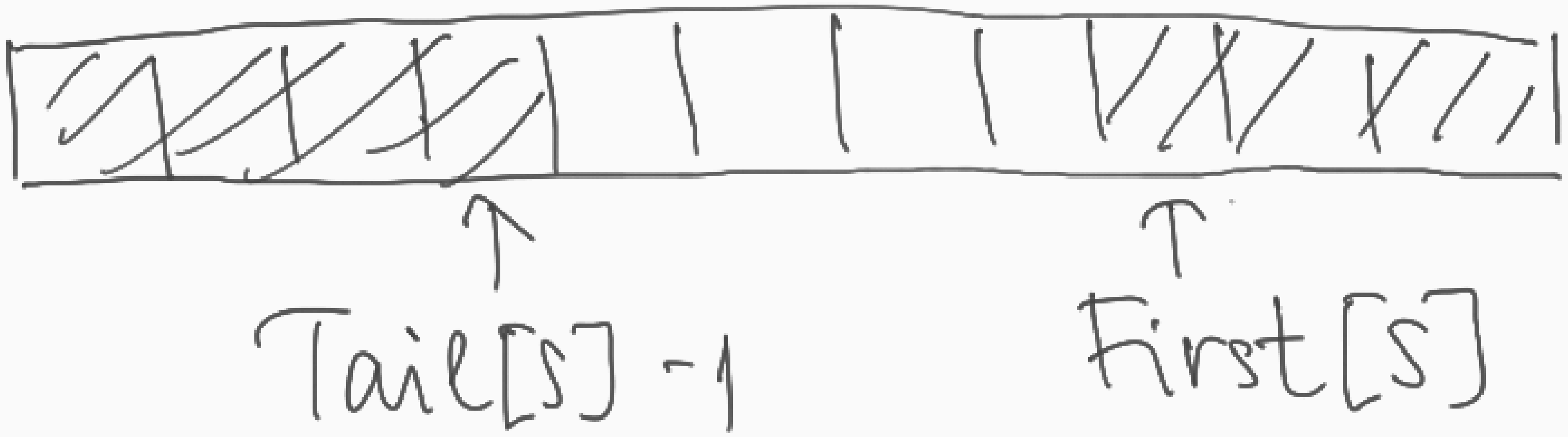
- Enqueue (Insert)
- Dequeue (Delete)

A queue can be implemented by an array $S[1:n]$

Attribute First[S] shows the index of the element first inserted, attribute Tail[S] shows the index of the element we will insert next

It means that the queue looks like this





Initially (empty queue) First[S] = Tail[S] = 1

The third attribute length[S] will show the number of elements in the queue.
Initially Length[S] = 0.

```
Enqueue(S, x)
if length[S] = n
   then
     error "queue overflow"
   else
     S[tail[S]] = x
```

```
    ⋮
    if Taie[S] = n
        then Tail[S] = 1
        else Tail [S]++
    Length[S]++

Dequene
if Length[S] = 0
    then
        error "queue underflow"
    else
    ⋮
```

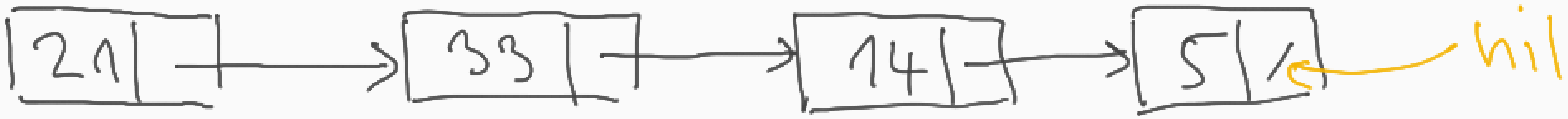
⋮

```
x = S[First[S]]
if First[S] = n
    then First[S] = 1
    else First[S]++
Length[S]--
return x
```

The operations takes $\Theta(1)$ time.

Linked lists

In this data structures the objects follow each other in a linear order This linear order is given by the indices in an array, but in a linked list this is given by pointers



Each object contains a pointer pointing to the next object of list.

The first object of the list (a pointer to the first object) is a list attribute

We can identify the list by this first object pointer first[L]

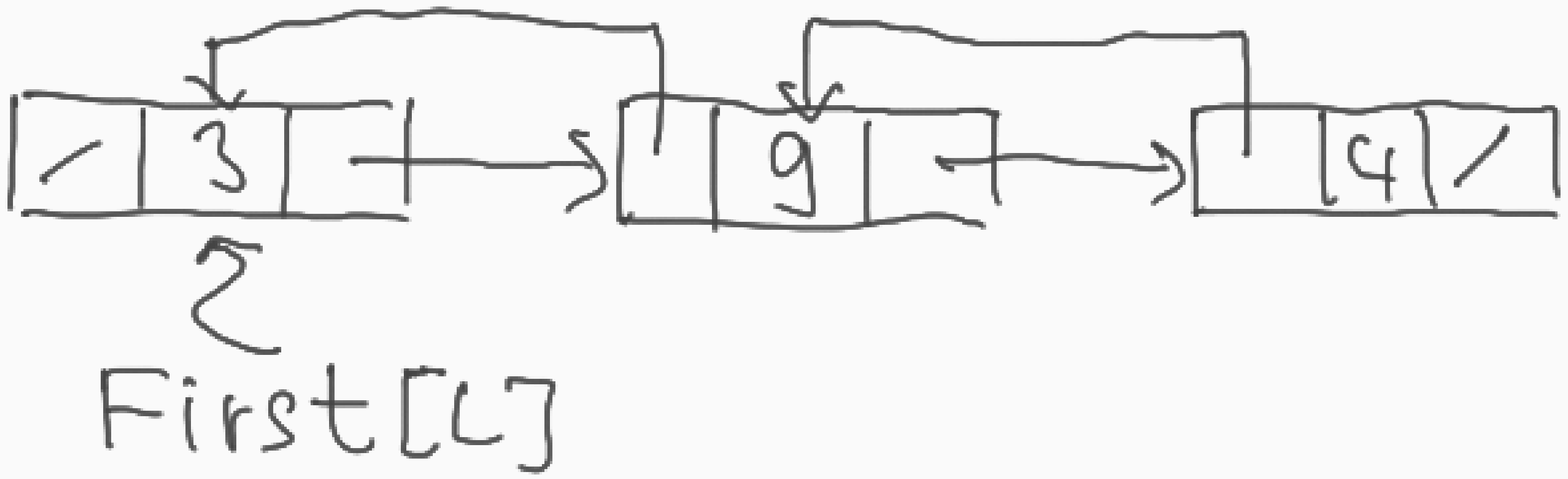
This list is empty if First[L] = nil

nil is a special pointer pointing nothing

There are many kinds of linked lists

→ single linked lists

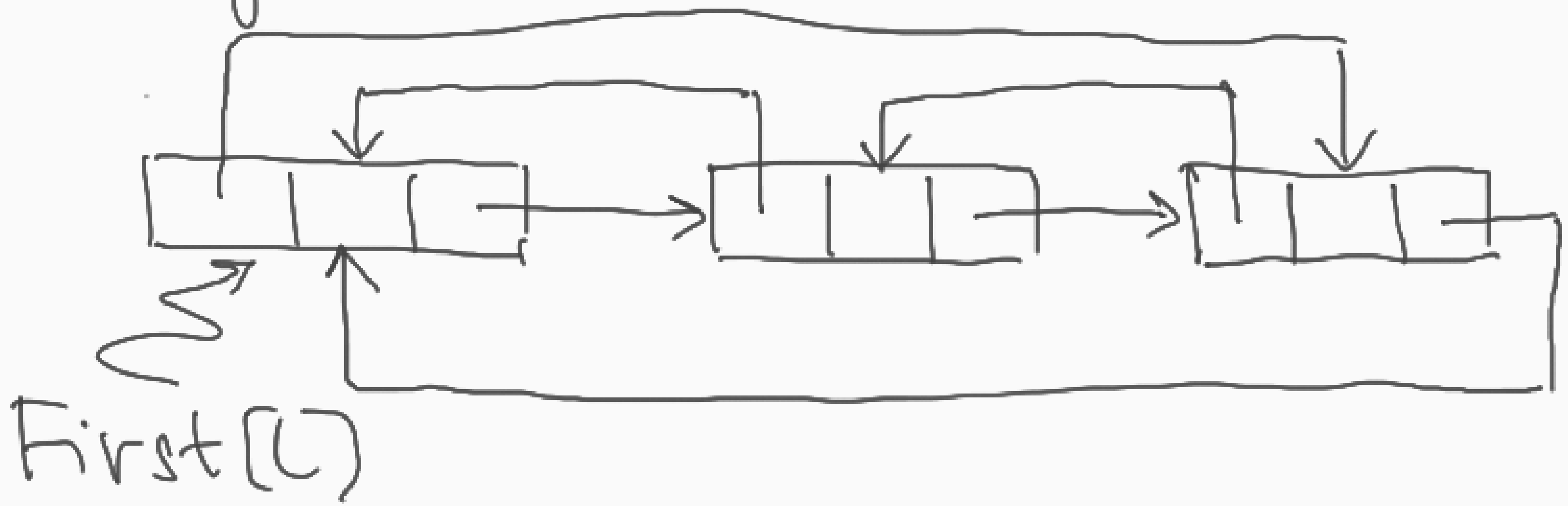
→ double linked lists



→ unordered linked lists

→ ordered linked lists

→ cyclic lists



Operations on single linked lists

Search (L, k)

$x = \text{First}[L]$

while $x \neq nil$ and $x \to key \neq k$ do

$\quad x := x \to next$

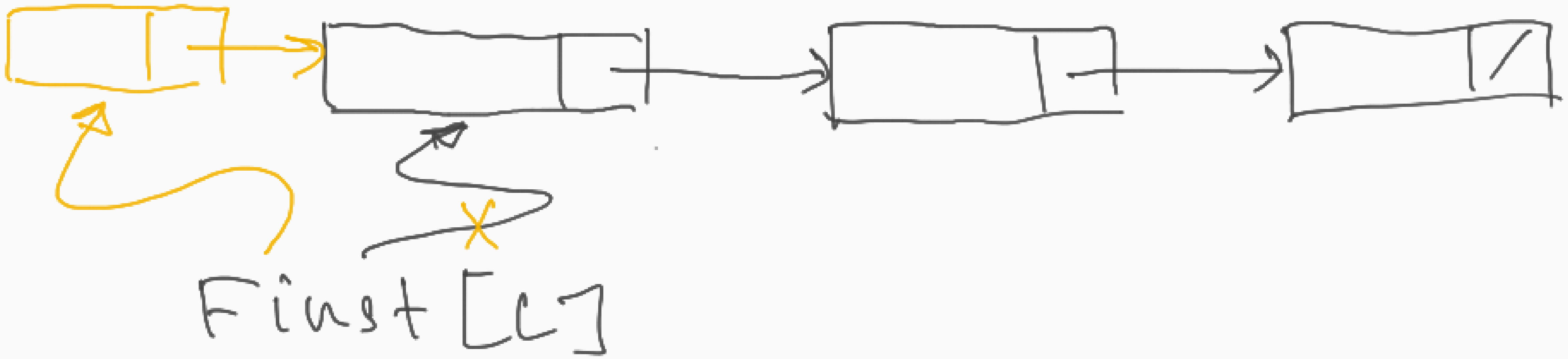
return x

Notation $x \to next$, $x \to key$

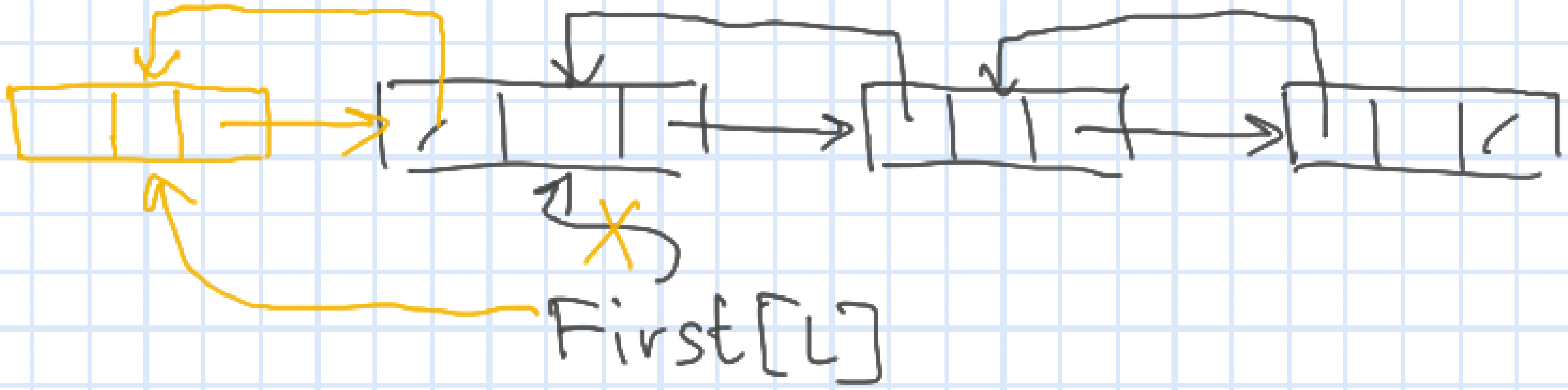
they are the next pointer and key tag of x, resp.

Cost : $O(n)$ in an n element list (in the worst case)

Insertion (the new element will be the first in the list for the sake of simplicity)



Let's see this for double linked lists



Insert (L, x)

x → next := First[L]

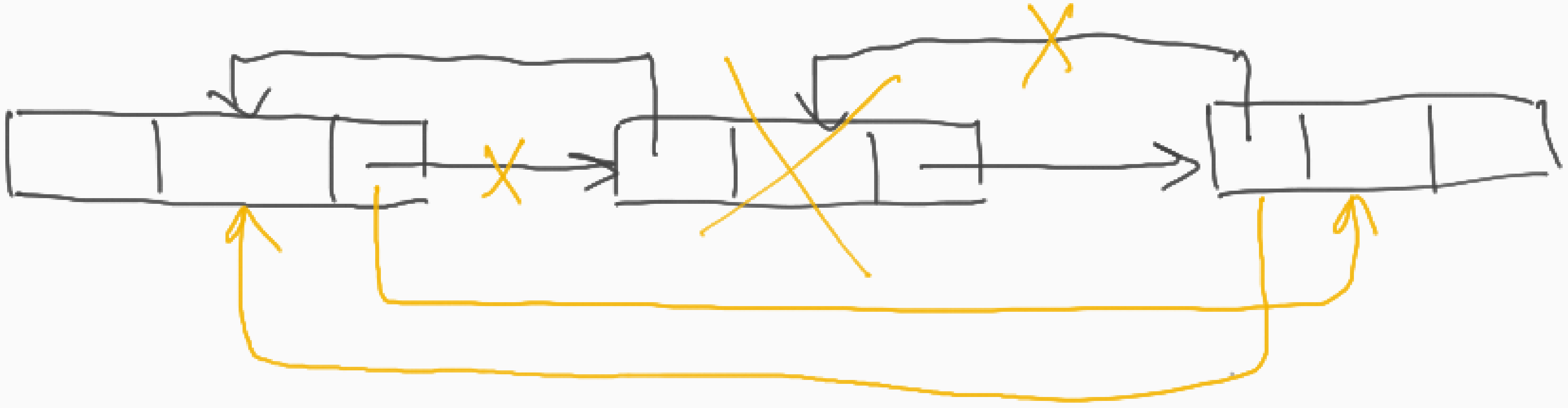
if First[L] ≠ nil then

    First[L] → prev = x

First[L] := x

x → prev := nil

Delete (L, x)   we know x



Delete (L, x)

if $x \to prev \neq nil$

then $x \to prev \to next := x \to next$

else $First[L] := x \to next$

if $x \to next \neq nil$ then

$x \to next \to prev := x \to prev$