11. Fundamental Data Structures

Abstract data types stack, queue, implementation variants for linked lists, amortized analysis [Ottman/Widmayer, Kap. 1.5.1-1.5.2, Cormen et al, Kap. 10.1.-10.2,17.1-17.3]

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

- $\mathtt{push}(x, S)$: 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.

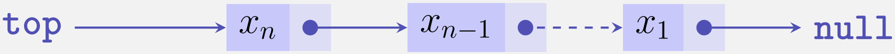
Abstract Data Types

We recall

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

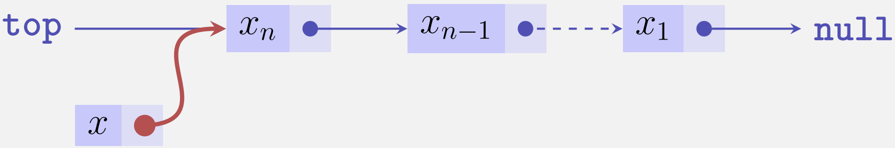
- **push**(x, S): Puts element x on the stack S.
- **pop**(S): Removes and returns top most element of S or **null**
- **top**(S): Returns top most element of S or **null**.
- **isEmpty**(S): Returns **true** if stack is empty, **false** otherwise.
- **emptyStack**$()$: Returns an empty stack.

Implementation Push



push(x, S):

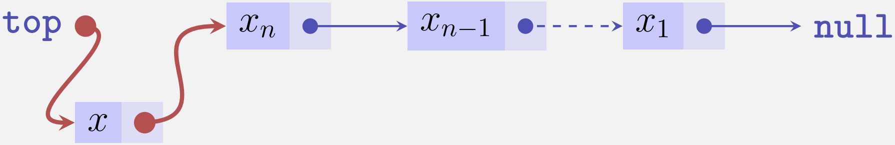
Implementation Push



`push`(x, S):

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

Implementation Push



push(x, S):

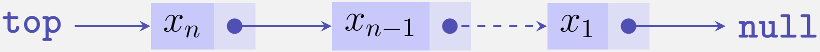
1. Create new list element with x and pointer to the value of **top**.
2. Assign the node with x to **top**.

Implementation Pop

$\texttt{top} \longrightarrow x_n \; \bullet \longrightarrow x_{n-1} \; \bullet \dashrightarrow x_1 \; \bullet \longrightarrow \texttt{null}$

$\texttt{pop}(S)$:

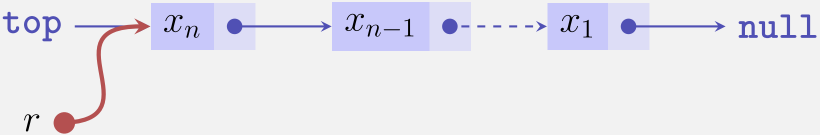
Implementation Pop



`pop`(S):

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

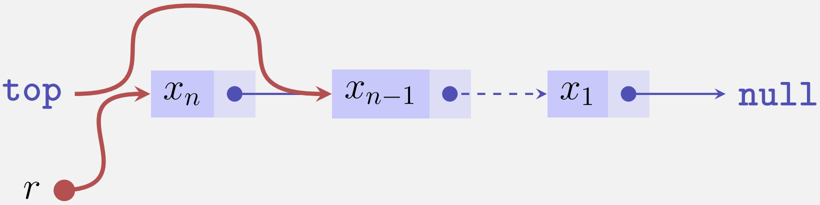
Implementation Pop



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

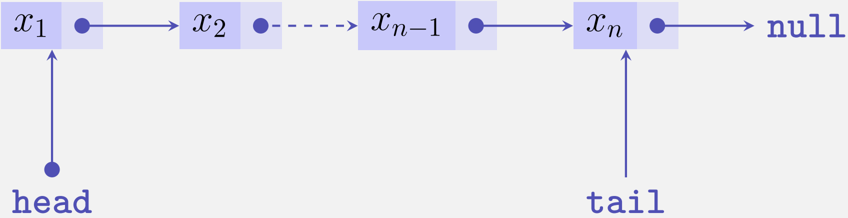
- `enqueue`(x, Q): adds x to the tail (=end) of the queue.
- `dequeue`(Q): removes x from the head of the queue and returns x (`null` otherwise)
- `head`(Q): returns the object from the head of the queue (`null` otherwise)
- `isEmpty`(Q): return `true` if the queue is empty, otherwise `false`

Queue (fifo)

A queue is an ADT with the following operations

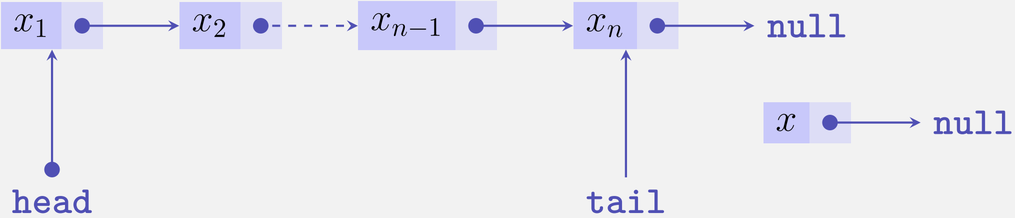
- `enqueue`(x, Q): adds x to the tail (=end) of the queue.
- `dequeue`(Q): removes x from the head of the queue and returns x (`null` otherwise)
- `head`(Q): returns the object from the head of the queue (`null` otherwise)
- `isEmpty`(Q): return `true` if the queue is empty, otherwise `false`
- `emptyQueue`$()$: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

Implementation Queue



`enqueue`(x, S):

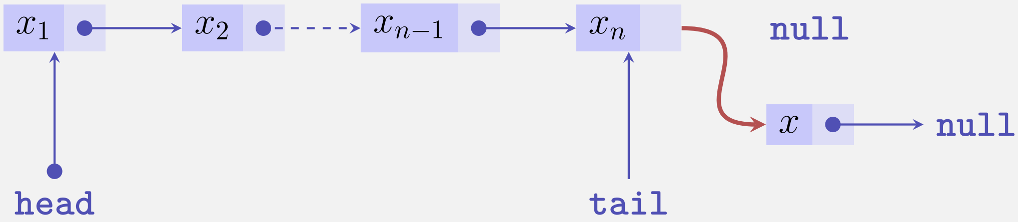
Implementation Queue



`enqueue`(x, S):

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

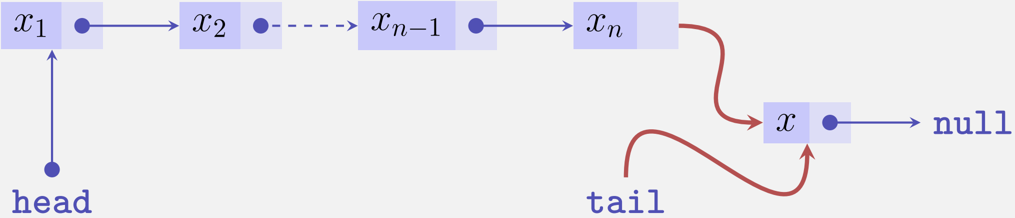
Implementation Queue



`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.

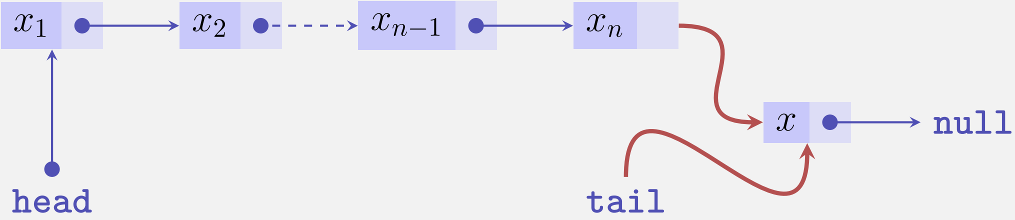
Implementation Queue



`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.

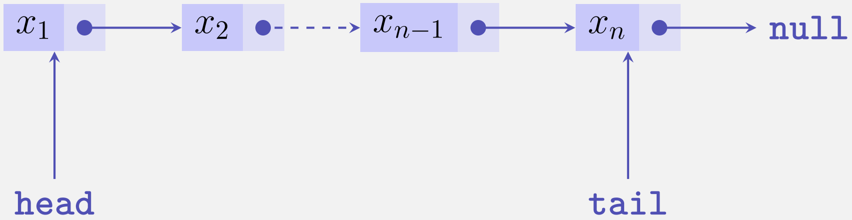
Implementation Queue



enqueue(x, S):

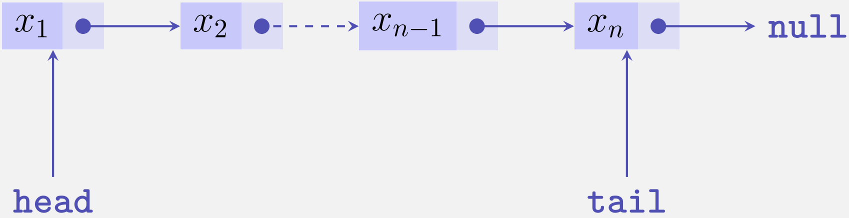
1. Create a new list element with x and pointer to **null**.
2. If **tail** $\neq$ **null**, then set **tail.next** to the node with x.
3. Set **tail** to the node with x.
4. If **head** $=$ **null**, then set **head** to **tail**.

Invariants



With this implementation it holds that

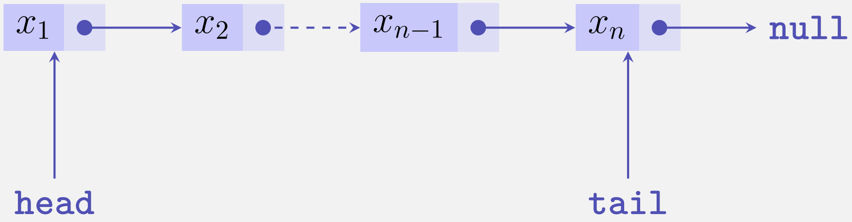
Invariants



With this implementation it holds that

- either `head` $=$ `tail` $=$ `null`,

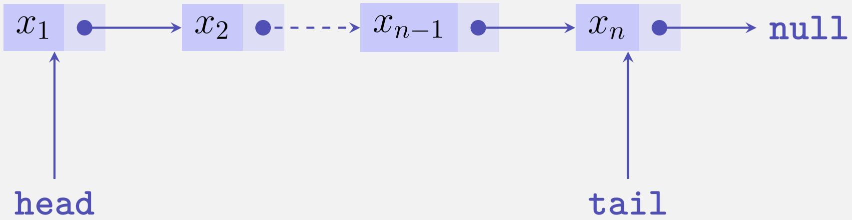
Invariants



With this implementation it holds that

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

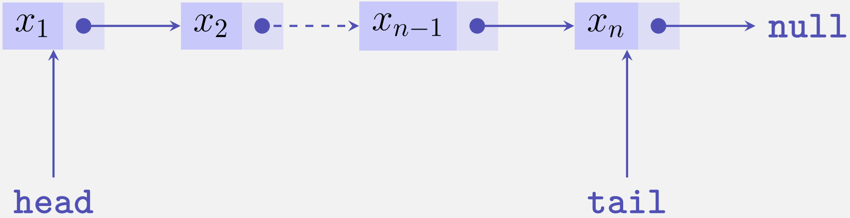
Invariants



With this implementation it holds that

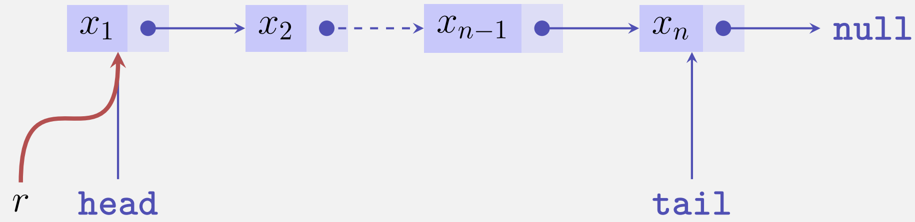
- either $\texttt{head} = \texttt{tail} = \texttt{null}$,
- or $\texttt{head} = \texttt{tail} \neq \texttt{null}$ and $\texttt{head.next} = \texttt{null}$
- or $\texttt{head} \neq \texttt{null}$ and $\texttt{tail} \neq \texttt{null}$ and $\texttt{head} \neq \texttt{tail}$ and $\texttt{head.next} \neq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

Implementation Queue



dequeue(S):

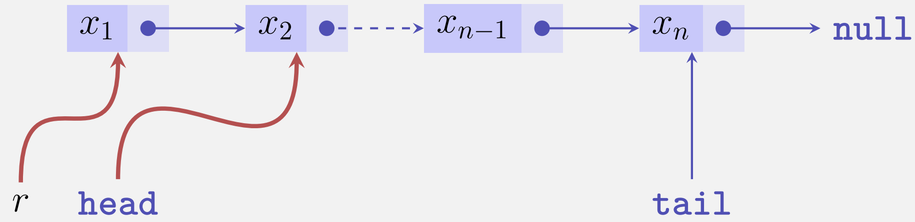
Implementation Queue



dequeue(S):

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

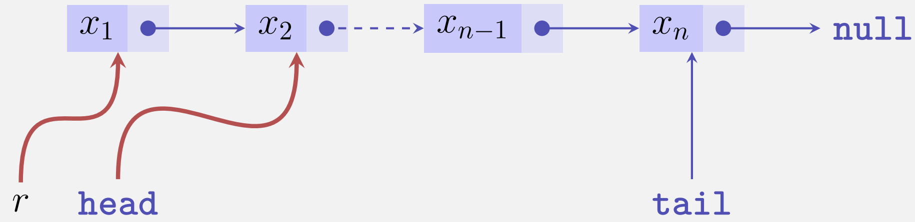
Implementation Queue



dequeue(S):

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

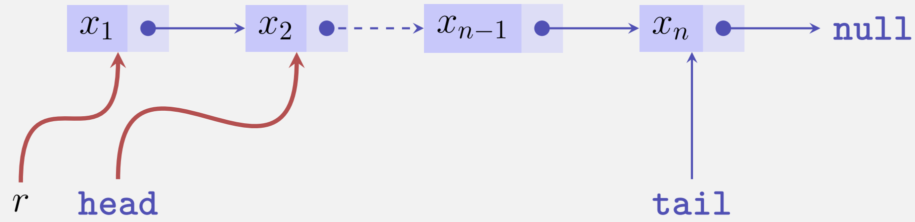
Implementation Queue



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):

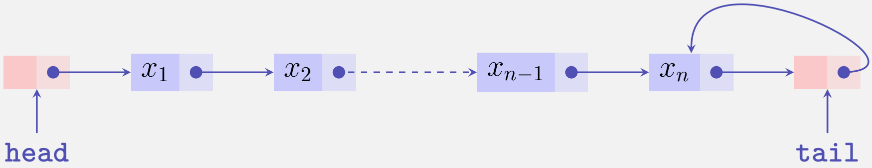
1. Store pointer to head in r. If $r = \texttt{null}$, then return r .
2. Set the pointer of head to head.next.
3. Is now $\texttt{head} = \texttt{null}$ then set tail to null.
4. Return the value of r.

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

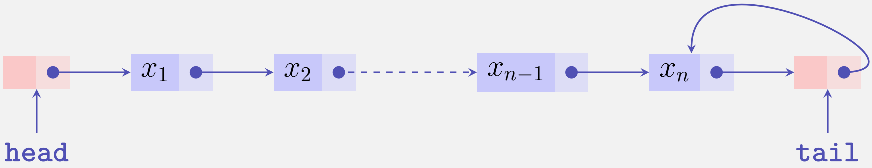
List with dummy elements (sentinels).



Advantage: less special cases

Implementation Variants of Linked Lists

List with dummy elements (sentinels).

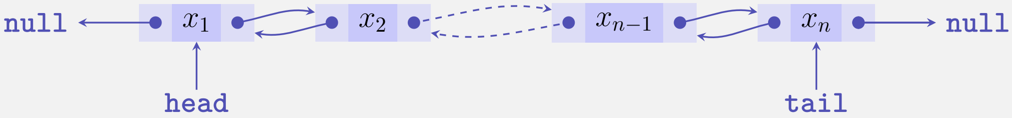


Advantage: less special cases

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

priority queue

Priority Queue

Operations

- `insert(x,p,Q)`: Enter object x with priority p.
- `extractMax(Q)`: Remove and return object x with highest priority.

Implementation Priority Queue

With a Max Heap

Thus

- `insert` in $\mathcal{O}(?)$ and
- `extractMax` in $\mathcal{O}(?)$.

Implementation Priority Queue

With a Max Heap

Thus

- `insert` in $\mathcal{O}(\log n)$ and
- `extractMax` in $\mathcal{O}(?)$.

Implementation Priority Queue

With a Max Heap

Thus

- `insert` in $\mathcal{O}(\log n)$ and
- `extractMax` in $\mathcal{O}(\log n)$.

Multistack

Multistack adds to the stack operations below

`multipop(s,S)`: remove the $\min(\text{size}(S), k)$ most recently inserted objects and return them.

Implementation as with the stack. Runtime of `multipop` is $\mathcal{O}(k)$.

Academic Question

If we execute on a stack with n elements a number of n times `multipop(k,S)` then this costs $\mathcal{O}(n^2)$?

Academic Question

If we execute on a stack with n elements a number of n times `multipop(k,S)` then this costs $\mathcal{O}(n^2)$?

Certainly correct because each `multipop` may take $\mathcal{O}(n)$ steps.

Academic Question

If we execute on a stack with n elements a number of n times `multipop(k,S)` then this costs $\mathcal{O}(n^2)$?

Certainly correct because each `multipop` may take $\mathcal{O}(n)$ steps.

How to make a better estimation?

Idea (accounting)

Introduction of a cost model:

- Each call of `push` costs 1 CHF and additional 1 CHF will be put to account.
- Each call to `pop` costs 1 CHF and will be paid from the account.

Account will never have a negative balance. Thus: maximal costs = number of `push` operations times two.

More Formal

Let t_i denote the real costs of the operation i. Potential function $\Phi_i \geq 0$ for the "account balance" after i operations. $\Phi_i \geq \Phi_0 \; \forall i$.

Amortized costs of the ith operation:

$$a_i := t_i + \Phi_i - \Phi_{i-1}.$$

It holds

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (t_i + \Phi_i - \Phi_{i-1}) = \left(\sum_{i=1}^{n} t_i \right) + \Phi_n - \Phi_0 \geq \sum_{i=1}^{n} t_i.$$

Goal: find potential function that evens out expensive operations.

Example stack

Potential function Φ_i = number element on the stack.

- **push**(x, S): real costs $t_i = 1$. $\Phi_i - \Phi_{i-1} = 1$. Amortized costs $a_i = 2$.
- **pop**(S): real costs $t_i = 1$. $\Phi_i - \Phi_{i-1} = -1$. Amortized costs $a_i = 0$.
- **multipop**(k, S): real costs $t_i = k$. $\Phi_i - \Phi_{i-1} = -k$. amortized costs $a_i = 0$.

All operations have *constant amortized cost*! Therefore, on average Multipop requires a constant amount of time. [14]

[14] Note that we are not talking about the probabilistic mean but the (worst-case) average of the costs.

Example Binary Counter

Binary counter with k bits. In the worst case for each count operation maximally k bitflips. Thus $\mathcal{O}(n \cdot k)$ bitflips for counting from 1 to n. Better estimation?

Real costs t_i = number bit flips from 0 to 1 plus number of bit-flips from 1 to 0.

$$...0\underbrace{1111111}_{l \text{ Einsen}}+1 = ...1\underbrace{0000000}_{l \text{ Zeroes}}.$$

$$\Rightarrow t_i = l + 1$$

Example Binary Counter

$$\ldots 0\underbrace{1111111}_{l \text{ Einsen}}+1 = \ldots 1\underbrace{0000000}_{l \text{ Nullen}}$$

potential function Φ_i: number of 1-bits of x_i.

$$\Rightarrow \Phi_i - \Phi_{i-1} = 1 - l,$$

$$\Rightarrow a_i = t_i + \Phi_i - \Phi_{i-1} = l + 1 + (1 - l) = 2.$$

Amortized constant cost for each count operation. ☺

12. Dictionaries

Dictionary, Self-ordering List, Implementation of Dictionaries with Array / List /Skip lists. [Ottman/Widmayer, Kap. 3.3,1.7, Cormen et al, Kap. Problem 17-5]

Dictionary

ADT to manage keys from a set $\mathcal{K}$ with operations

- `insert`(k, D): Insert $k \in \mathcal{K}$ to the dictionary D. Already exists $\Rightarrow$ error messsage.
- `delete`(k, D): Delete k from the dictionary D. Not existing $\Rightarrow$ error message.
- `search`(k, D): Returns `true` if $k \in D$, otherwise `false`

Idea

Implement dictionary as sorted array

Worst case number of fundamental operations

Search
Insert
Delete

Idea

Implement dictionary as sorted array

Worst case number of fundamental operations

Search	$\mathcal{O}(\log n)$ ☺
Insert	
Delete	

Idea

Implement dictionary as sorted array

Worst case number of fundamental operations

Search	$\mathcal{O}(\log n)$ 🙂
Insert	$\mathcal{O}(n)$ 🙁
Delete	

Idea

Implement dictionary as sorted array

Worst case number of fundamental operations

Search	$\mathcal{O}(\log n)$
Insert	$\mathcal{O}(n)$
Delete	$\mathcal{O}(n)$

Other idea

Implement dictionary as a linked list

Worst case number of fundamental operations

Search
Insert
Delete

[15] Provided that we do not have to check existence.

Other idea

Implement dictionary as a linked list

Worst case number of fundamental operations

Search	$\mathcal{O}(n)$ ☹
Insert	
Delete	

[15] Provided that we do not have to check existence.

Other idea

Implement dictionary as a linked list

Worst case number of fundamental operations

Search	$\mathcal{O}(n)$ ☹
Insert	$\mathcal{O}(1)$[15] ☺
Delete	

[15] Provided that we do not have to check existence.

Other idea

Implement dictionary as a linked list

Worst case number of fundamental operations

Search	$\mathcal{O}(n)$	☹
Insert	$\mathcal{O}(1)$[15]	☺
Delete	$\mathcal{O}(n)$	☹

[15] Provided that we do not have to check existence.

Self Ordered Lists

Problematic with the adoption of a linked list: linear search time

Idea: Try to order the list elements such that accesses over time are possible in a faster way

For example

- Transpose: For each access to a key, the key is moved one position closer to the front.
- Move-to-Front (MTF): For each access to a key, the key is moved to the front of the list.

Transpose

Transpose:

k_1	k_2	k_3	k_4	k_5	$\cdots$	k_{n-1}	k_n

Worst case: Alternating sequence of n accesses to k_{n-1} and k_n.

Transpose

Transpose:

$$k_1 \quad k_2 \quad k_3 \quad k_4 \quad k_5 \quad \cdots \quad k_n \quad k_{n-1}$$

Worst case: Alternating sequence of n accesses to k_{n-1} and k_n.

Transpose

Transpose:

$k_1 \quad k_2 \quad k_3 \quad k_4 \quad k_5 \quad \cdots \quad k_{n-1} \quad k_n$

Worst case: Alternating sequence of n accesses to k_{n-1} and k_n.

Transpose

Transpose:

Worst case: Alternating sequence of n accesses to k_{n-1} and k_n.
Runtime: $\Theta(n^2)$

Move-to-Front

Move-to-Front:

k_1	k_2	k_3	k_4	k_5	$\cdots$	k_{n-1}	k_n

Alternating sequence of n accesses to k_{n-1} and k_n.

Move-to-Front

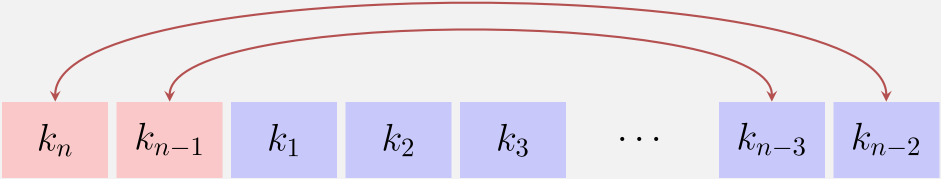
Move-to-Front:

k_{n-1} k_1 k_2 k_3 k_4 $\cdots$ k_{n-2} k_n

Alternating sequence of n accesses to k_{n-1} and k_n.

Move-to-Front

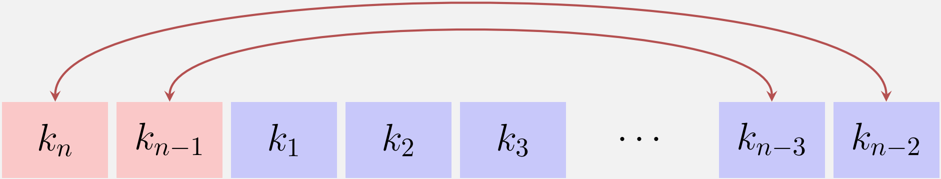
Move-to-Front:



Alternating sequence of n accesses to k_{n-1} and k_n.

Move-to-Front

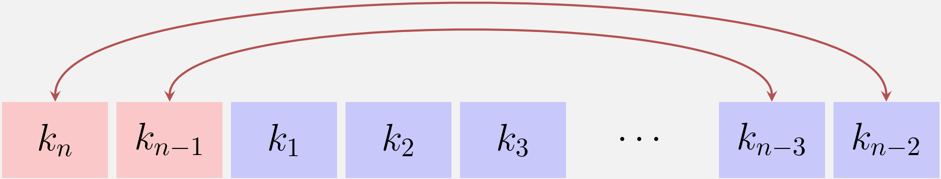
Move-to-Front:



Alternating sequence of n accesses to k_{n-1} and k_n. Runtime: $\Theta(n)$

Move-to-Front

Move-to-Front:



Alternating sequence of n accesses to k_{n-1} and k_n. Runtime: $\Theta(n)$

Also here we can provide a sequence of accesses with quadratic runtime, e.g. access to the last element. But there is no obvious strategy to counteract much better than MTF..

Analysis

Compare MTF with the best-possible competitor (algorithm) A. How much better can A be?

Assumptions:

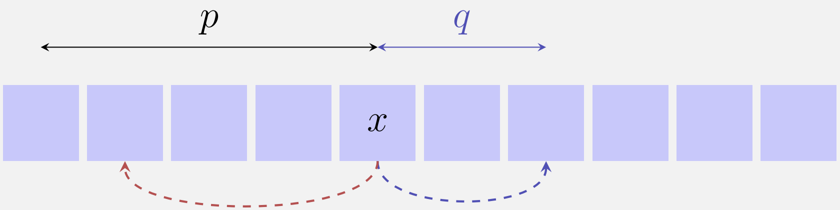
- MTF and A may only move the accessed element.
- MTF and A start with the same list.

Let M_k and A_k designate the lists after the kth step. $M_0 = A_0$.

Analysis

Costs:

- Access to x: position p of x in the list.
- No further costs, if x is moved before p
- Further costs q for each element that x is moved back starting from p.



Amortized Analysis

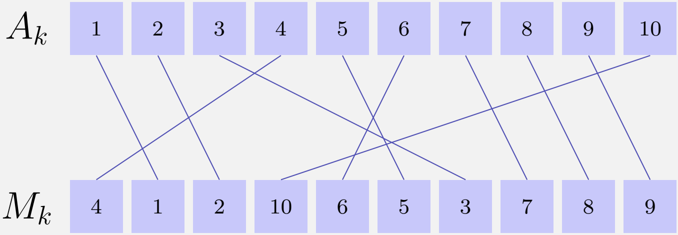
Let an arbitrary sequence of search requests be given and let $G_k^{(M)}$ and $G_k^{(A)}$ the costs in step k for Move-to-Front and A, respectively. Want estimation of $\sum_k G_k^{(M)}$ compared with $\sum_k G_k^{(A)}$.

$\Rightarrow$ Amortized analysis with potential function Φ.

Potential Function

Potential function $\Phi =$ Number of inversions of A vs. MTF.

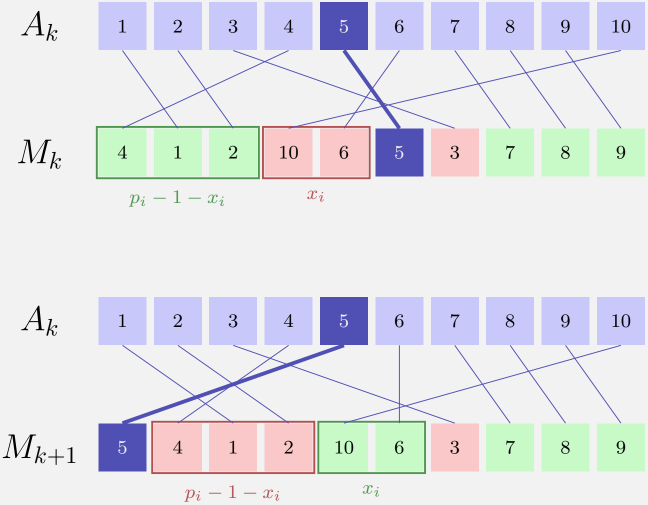
Inversion = Pair x, y such that for the positions of a and y
$\left(p^{(A)}(x) < p^{(A)}(y)\right) \neq \left(p^{(M)}(x) < p^{(M)}(y)\right)$



#inversion = #crossings

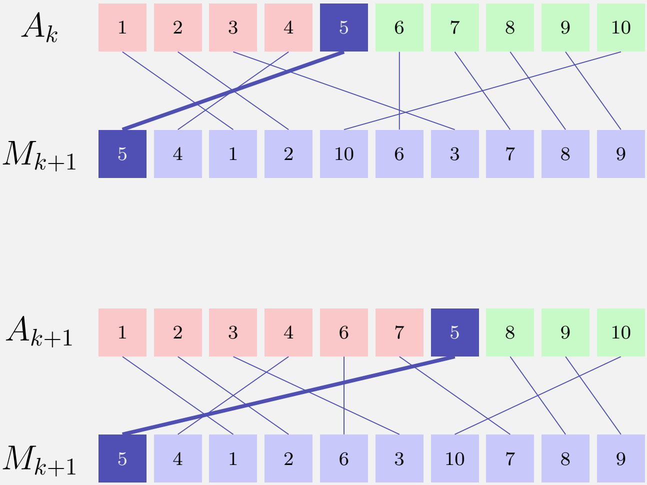
Estimating the Potential Function: MTF

- Element i at position $p_i := p^{(M)}(i)$.
- access costs $C_k^{(M)} = p_i$.
- x_i: Number elements that are in M before p_i and in A after i .
- MTF removes x_i inversions.
- $p_i - x_i - 1$: Number elements that in M are before p_i and in A are before i.
- MTF generates $p_i - 1 - x_i$ inversions.



Estimating the Potential Function: A

- Wlog element i at position $p^{(A)}(i)$.
- $X_k^{(A)}$: number movements to the back (otherwise 0).
- access costs for i: $C_k^{(A)} = p^{(A)}(i) \geq p^{(M)}(i) - x_i$.
- A increases the number of inversions maximally by $X_k^{(A)}$.



Estimation

$$\Phi_{k+1} - \Phi_k \le -x_i + (p_i - 1 - x_i) + X_k^{(A)}$$

Amortized costs of MTF in step k:

$$\begin{aligned} a_k^{(M)} &= C_k^{(M)} + \Phi_{k+1} - \Phi_k \\ &\le p_i - x_i + (p_i - 1 - x_i) + X_k^{(A)} \\ &= (p_i - x_i) + (p_i - x_i) - 1 + X_k^{(A)} \\ &\le C_k^{(A)} + C_k^{(A)} - 1 + X_k^{(A)} \le 2 \cdot C_k^{(A)} + X_k^{(A)}. \end{aligned}$$

Estimation

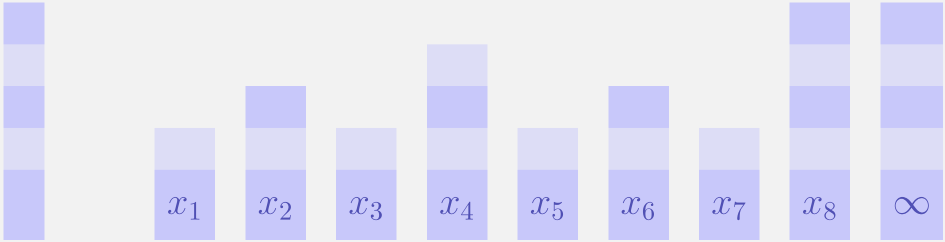
Summing up costs

$$\begin{aligned}\sum_k G_k^{(M)} = \sum_k C_k^{(M)} &\leq \sum_k a_k^{(M)} \leq \sum_k 2 \cdot C_k^{(A)} + X_k^{(A)} \\ &\leq 2 \cdot \sum_k C_k^{(A)} + X_k^{(A)} \\ &= 2 \cdot \sum_k G_k^{(A)}\end{aligned}$$

In the worst case MTF requires at most twice as many operations as the optimal strategy.

Cool idea: skip lists

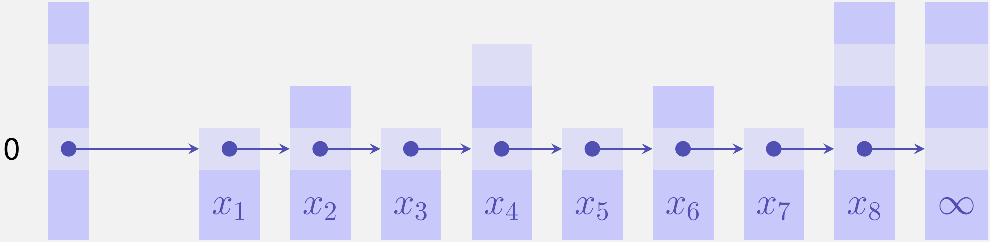
skip list



$$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9.$$

Cool idea: skip lists

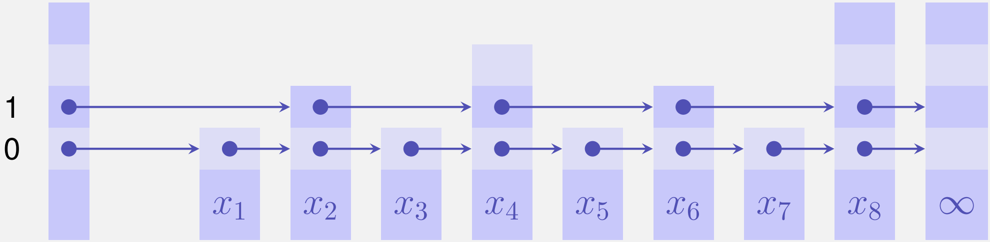
skip list



$$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9.$$

Cool idea: skip lists

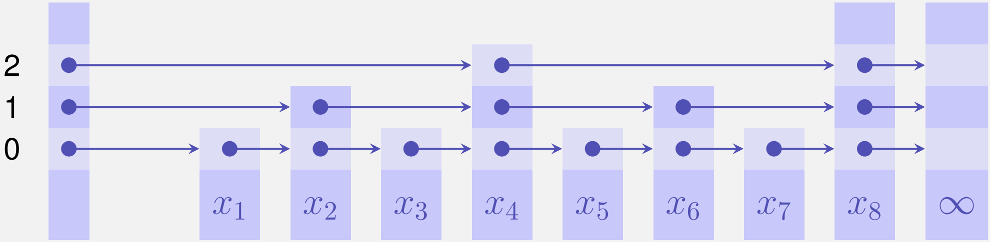
skip list



$$x_1 \le x_2 \le x_3 \le \cdots \le x_9.$$

Cool idea: skip lists

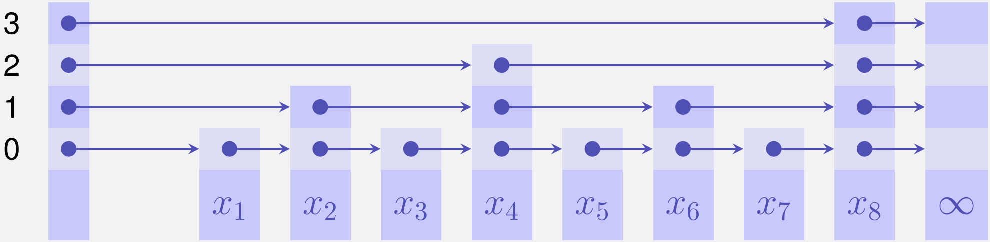
skip list



$$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9.$$

Cool idea: skip lists

Perfect skip list

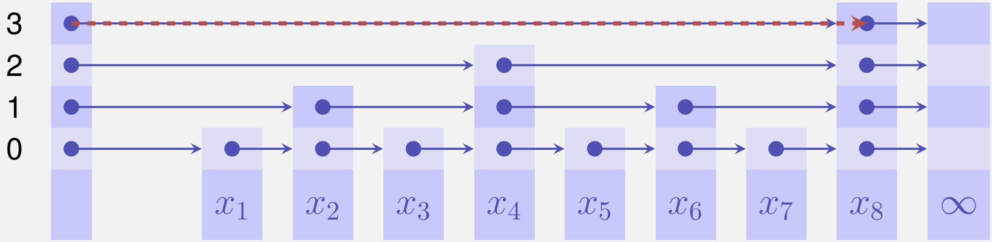


$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

Example: search for a key x with $x_5 < x < x_6$.

Cool idea: skip lists

Perfect skip list

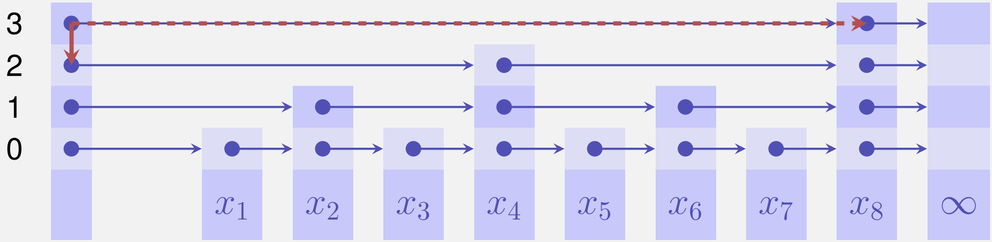


$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

Example: search for a key x with $x_5 < x < x_6$.

Cool idea: skip lists

Perfect skip list

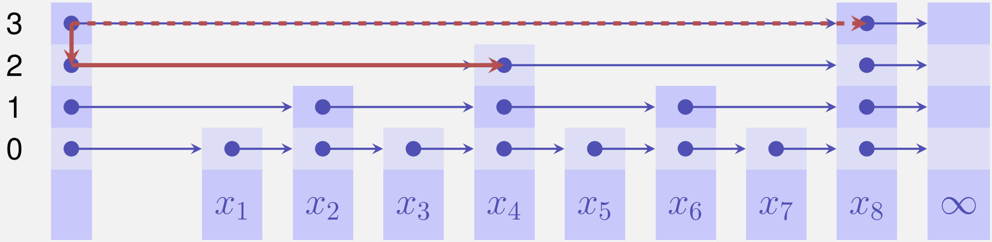


$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

Example: search for a key x with $x_5 < x < x_6$.

Cool idea: skip lists

Perfect skip list

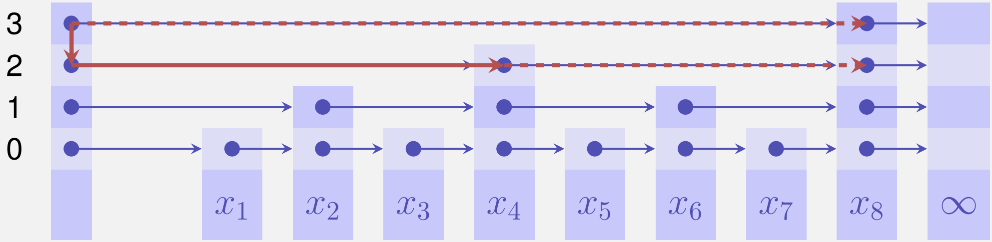


$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

Example: search for a key x with $x_5 < x < x_6$.

Cool idea: skip lists

Perfect skip list

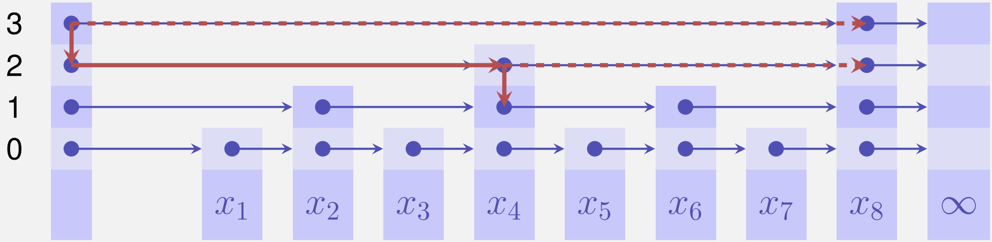


$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

Example: search for a key x with $x_5 < x < x_6$.

Cool idea: skip lists

Perfect skip list

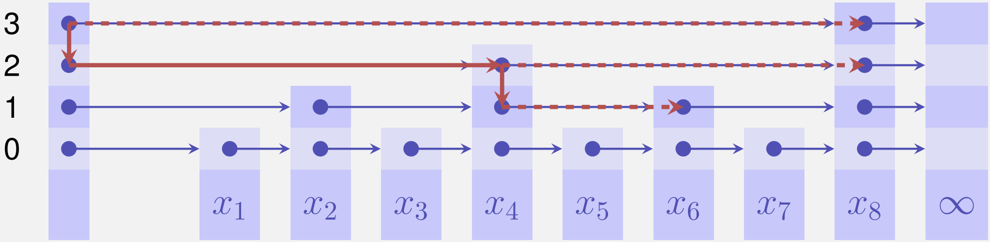


$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

Example: search for a key x with $x_5 < x < x_6$.

Cool idea: skip lists

Perfect skip list

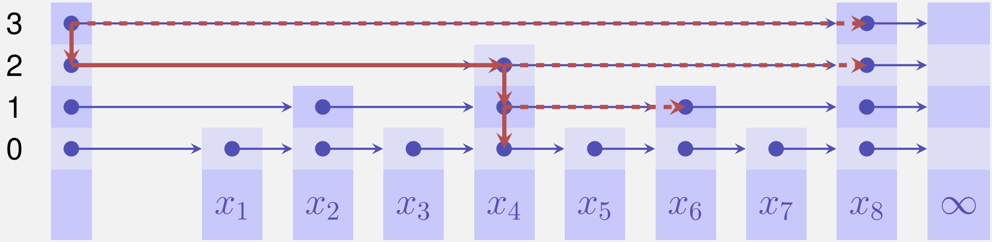


$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

Example: search for a key x with $x_5 < x < x_6$.

Cool idea: skip lists

Perfect skip list

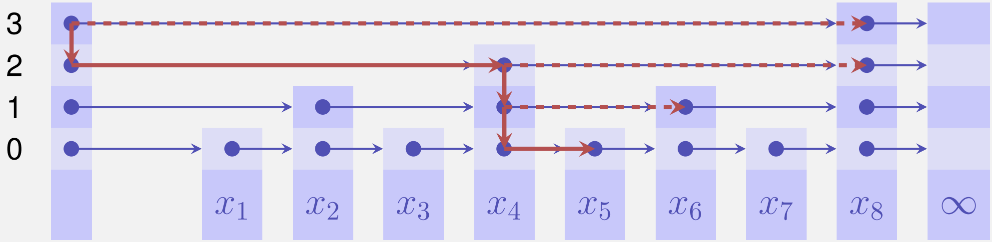


$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

Example: search for a key x with $x_5 < x < x_6$.

Cool idea: skip lists

Perfect skip list

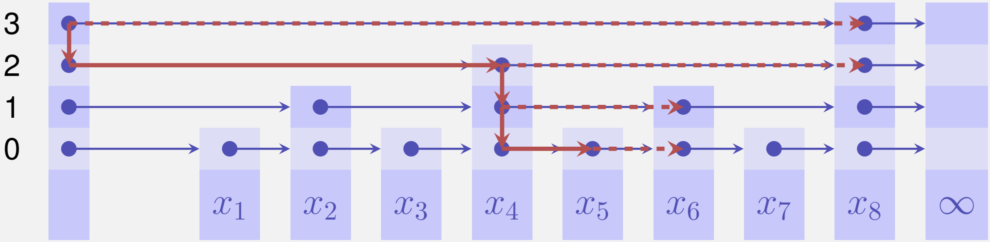


$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

Example: search for a key x with $x_5 < x < x_6$.

Cool idea: skip lists

Perfect skip list



$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9$.

Example: search for a key x with $x_5 < x < x_6$.

Analysis perfect skip list (worst cases)

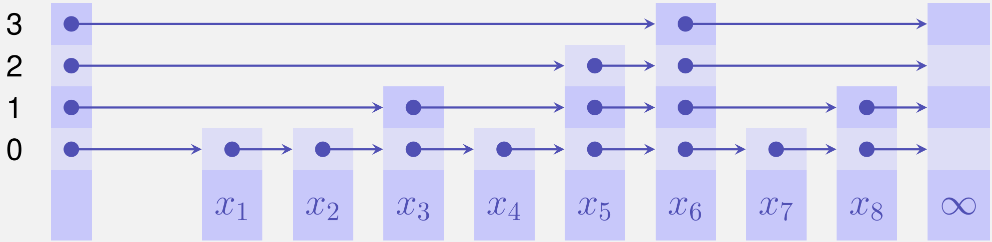
Search in $\mathcal{O}(\log n)$. Insert in $\mathcal{O}(n)$.

Randomized Skip List

Idea: insert a key with random height H with $\mathbb{P}(H = i) = \frac{1}{2^{i+1}}$.

Randomized Skip List

Idea: insert a key with random height H with $\mathbb{P}(H = i) = \frac{1}{2^{i+1}}$.



Analysis Randomized Skip List

Theorem

The expected number of fundamental operations for Search, Insert and Delete of an element in a randomized skip list is $\mathcal{O}(\log n)$.

The lengthy proof that will not be presented in this courseobserves the length of a path from a searched node back to the starting point in the highest level.