Datenstrukturen und Algorithmen

Exercise 4

FS 2018

Program of today

1. Feedback of last exercise
2. Repetition theory
   - Self Ordering
   - Skiplisten
3. Programming Task

Heapsort structure

- Which functions to implement for heapsort?

Heapsort structure

```
void sink(...);
void heapify(...);
void heapsort(...);
```

- heapify can be done inline
- Signature of the functions (for std::vector)?

Heapsort structure

```
void sink(vector<int>& A, size_t index, size_t size
void heapify(vector<int>& A);
void heapsort(vector<int>& A);
```

- Generic (e.g., for MyVector)?

Heapsort structure

```
template <typename X>
void sink(X& A, size_t index, size_t size);

template <typename X>
void heapify(X& A);

template <typename X>
void heapsort(X& A);
```

2. Repetition theory

Amortized Analysis

Let t_i denote the real costs of the operation i.

Potential function $\Phi_i \geq 0$ for the "account balance" after i operations.

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.

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

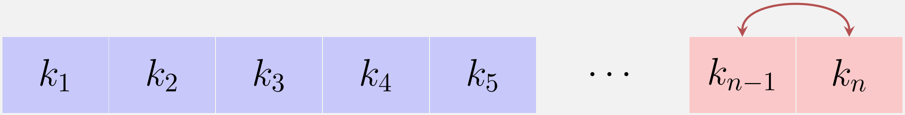
Transpose:

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

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

Transpose

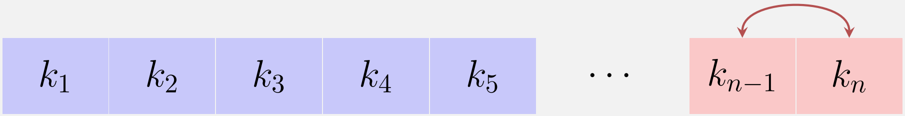
Transpose:



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

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

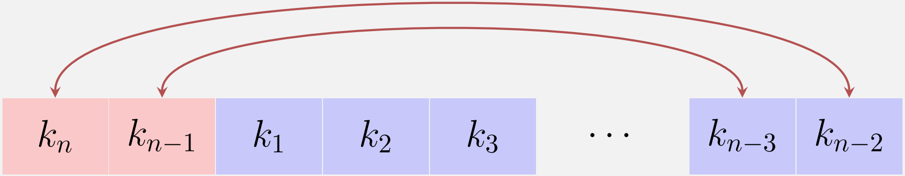
Move-to-Front:

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

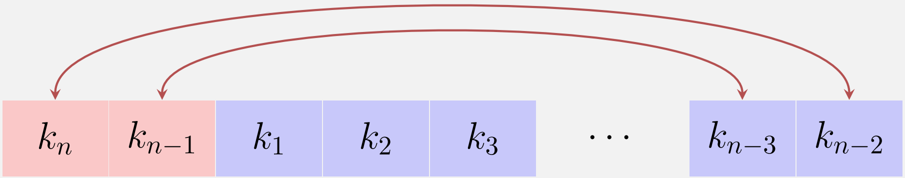
Move-to-Front:



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

Move-to-Front

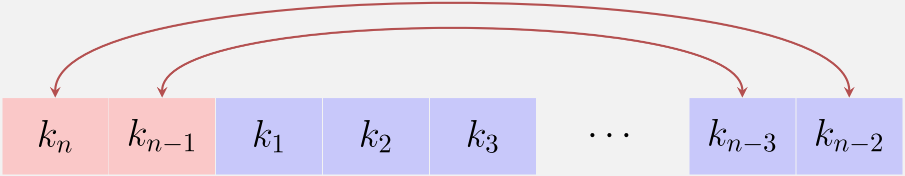
Move-to-Front:



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

Move-to-Front

Move-to-Front:

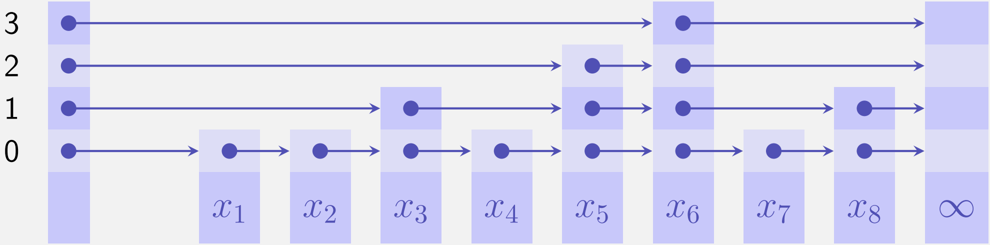


Alternating sequence of n accesses to k_{n-1} and k_n. Runtime: $\Theta(n)$

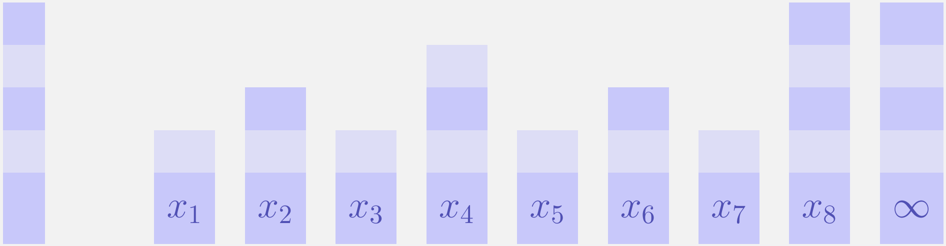
Also here we can provide a sequence of accesses with quadratic runtime, e.g. access to the last element. But there is no obvious strategy to counteract much better than MTF..

Randomized Skip List

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

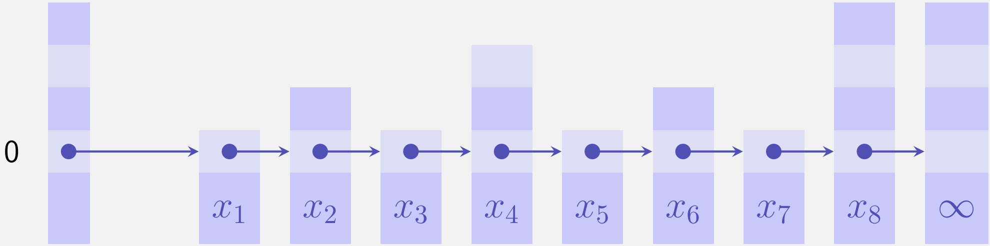


Randomized Skip List: finding element



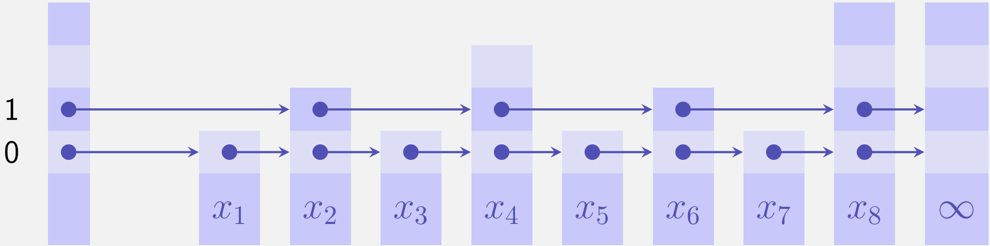
$$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9.$$

Randomized Skip List: finding element



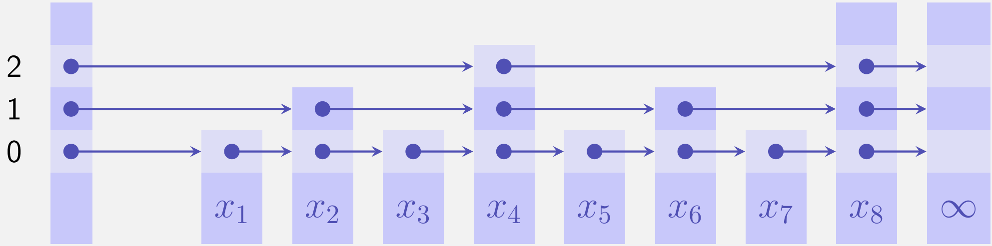
$$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9.$$

Randomized Skip List: finding element



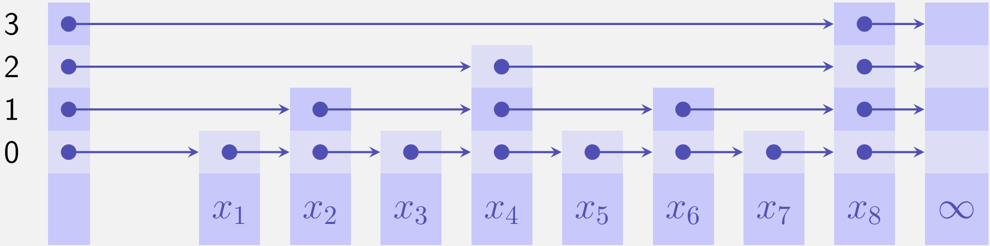
$$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9.$$

Randomized Skip List: finding element



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

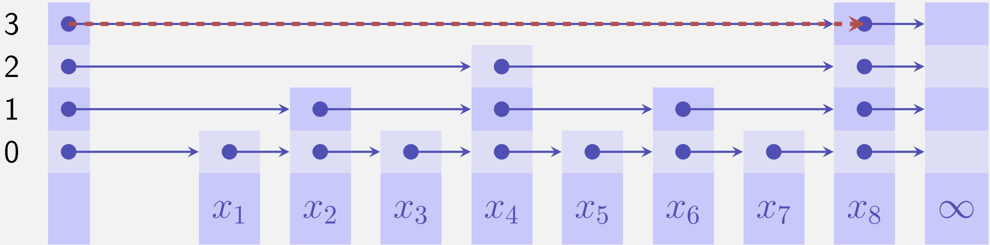
Randomized Skip List: finding element



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

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

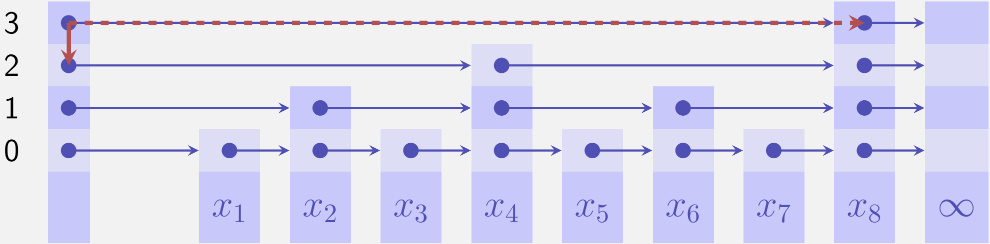
Randomized Skip List: finding element



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

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

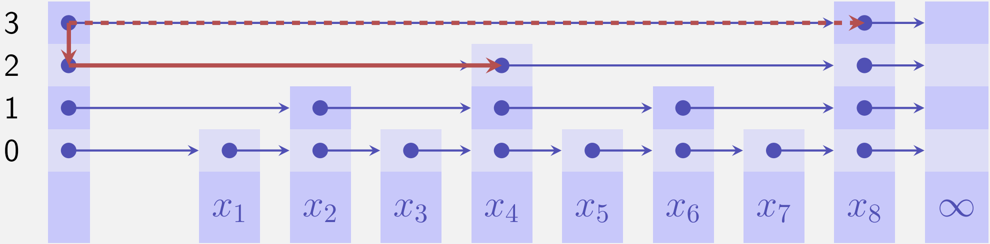
Randomized Skip List: finding element



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

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

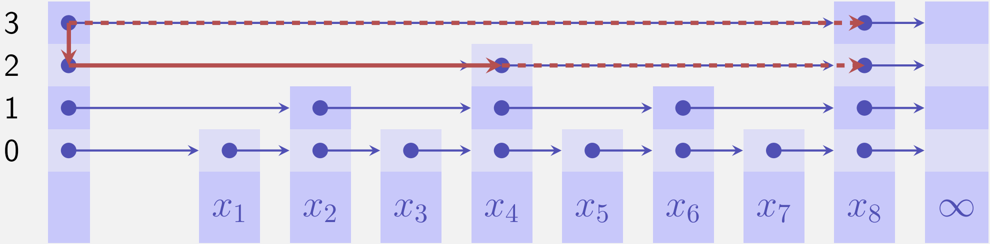
Randomized Skip List: finding element



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

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

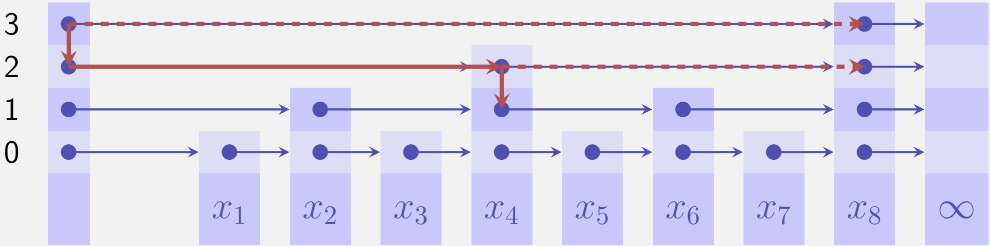
Randomized Skip List: finding element



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

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

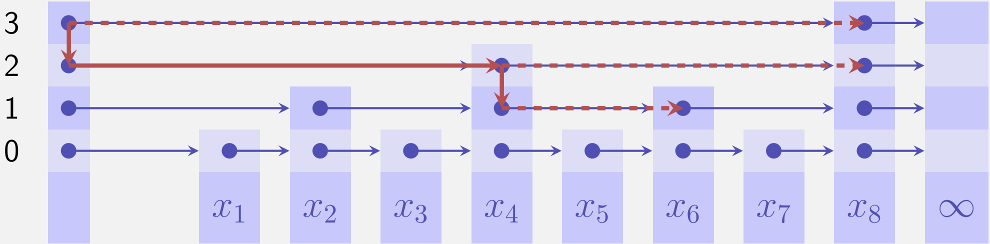
Randomized Skip List: finding element



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

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

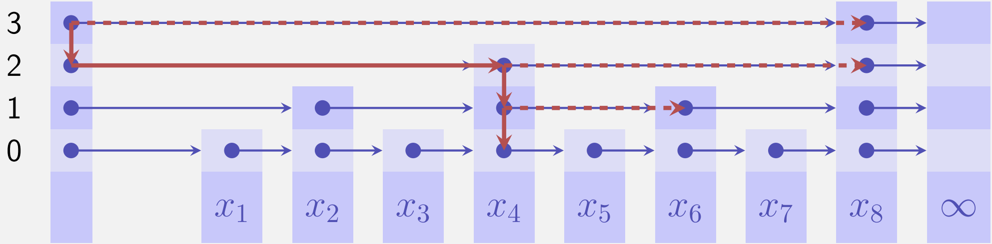
Randomized Skip List: finding element



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

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

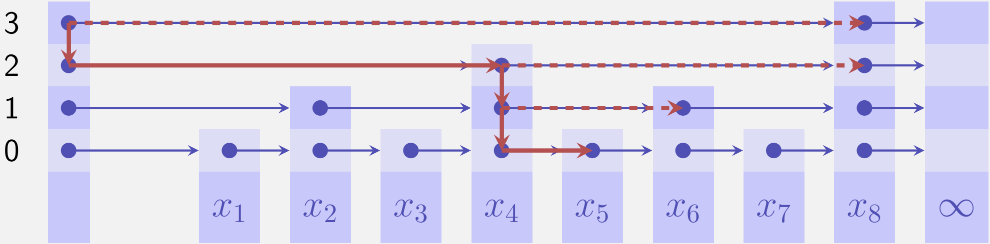
Randomized Skip List: finding element



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

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

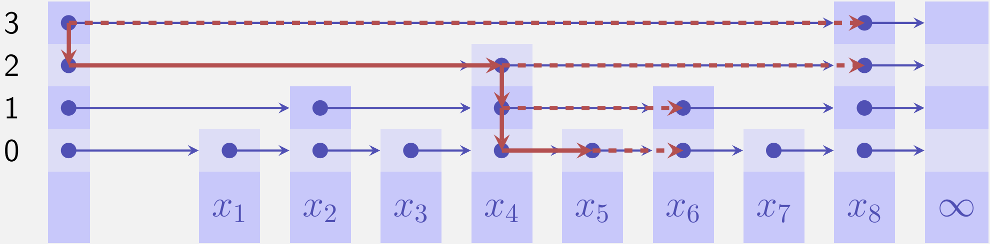
Randomized Skip List: finding element



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

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

Randomized Skip List: finding element



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

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

Skip Lists Interface

```
template<typename T> class SkipList {
public:
  SkipList();
  ~SkipList();

  void insert(const T& value);
  void erase(const T& value);

  // iterator implementation ...
};
```

Partially implemented:

- A class `Node` saves an element `value` of type `T` and a `std::vector` called `forward` with pointers to successive nodes.
- First Node (without value): `head`.
- `forward[0]` points to the following element in the list.
- We use this in an already implemented iterator.

Implementing `insert` and `erase`

`insert(const T& value)`

- create new node
- choose random number of levels
- for each level, find the first smaller node
- set pointers from previous nodes and new node

Implementing `insert` and `erase`

`insert(const T& value)`

- create new node
- choose random number of levels
- for each level, find the first smaller node
- set pointers from previous nodes and new node

`erase(const T& value)`

- find first smaller node
- check if next node has the according value
- set pointers accordingly
- delete node if necessary

Implementing `insert` and `erase`

`insert(const T& value)`

- create new node
- choose random number of levels
- for each level, find the first smaller node
- set pointers from previous nodes and new node

`erase(const T& value)`

- find first smaller node
- check if next node has the according value
- set pointers accordingly
- delete node if necessary

Warning: The same value can appear multiple times.

Recap dynamic allocated memory

Important: Every `new` needs its `delete` and only one!

Recap dynamic allocated memory

Important: Every `new` needs its `delete` and only one!

Therefore “Rule of three”:

- constructor
- copy constructor
- destructor

Recap dynamic allocated memory

Important: Every `new` needs its `delete` and only one!

Therefore "Rule of three":

- constructor
- copy constructor
- destructor

being lazy " Rule of two":

- never copy (unsure)
- make copy constructor private (save)

Questions?

Questions?

Let's get to work.