Datenstrukturen und Algorithmen

Exercise 4

FS 2018

Program of today

1. Feedback of last exercise
2. Repetition theory
   - Amortized Analysis
   - Skip Lists
3. Programming Task

Sorting

Bubblesort	min	max
Comparisons	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Sequence	any	any
Swaps	0	$\mathcal{O}(n^2)$
Sequence	$1, 2, \ldots, n$	$n, n-1, \ldots, 1$

Sorting

InsertionSort	min	max
Comparisons	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
Sequence	$1, 2, \ldots, n$	$n, n-1, \ldots, 1$
Swaps	0	$\mathcal{O}(n^2)$
Sequence	$1, 2, \ldots, n$	$n, n-1, \ldots, 1$

SelectionSort	min	max
Comparisons	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Sequence	any	any
Swaps	0	$\mathcal{O}(n)$
Sequence	$1, 2, \ldots, n$	$n, n-1, \ldots, 1$

Sorting

QuickSort	min	max
Comparisons	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$
Sequence	complex	$1, 2, \ldots, n$
Swaps	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$
Sequence	$1, 2, \ldots, n$	complex

complex: Sequence must be made such that the pivot halves the sorting range. For example ($n = 7$): $4, 5, 7, 6, 2, 1, 3$

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

Three Methods

- Aggregate Analysis
- Account Method
- Potential Method

Example: simple dictionary

Supports operations insert and find. Idea:

- Collection of arrays A_i with Length 2^i
- Every array is either entirely empty or entirely full and stores items in a sorted order
- Between the arrays there is no further relationship

data $\{1, 8, 10, 18, 20, 24, 36, 48, 50, 75, 99\}$, $n = 11$

$$\begin{aligned} A_0&: \quad [50] \\ A_1&: \quad [8, 99] \\ A_2&: \quad \emptyset \\ A_3&: \quad [1, 10, 18, 20, 24, 36, 48, 75] \end{aligned}$$

Example: simple dictionary

data $\{1, 8, 10, 18, 20, 24, 36, 48, 50, 75, 99\}$, $n = 11$

$$
\begin{aligned}
A_0 &: \quad [50] \\
A_1 &: \quad [8, 99] \\
A_2 &: \quad \emptyset \\
A_3 &: \quad [1, 10, 18, 20, 24, 36, 48, 75]
\end{aligned}
$$

Algorithm `Find`:

Example: simple dictionary

data $\{1, 8, 10, 18, 20, 24, 36, 48, 50, 75, 99\}$, $n = 11$

$$\begin{array}{ll} A_0: & [50] \\ A_1: & [8, 99] \\ A_2: & \emptyset \\ A_3: & [1, 10, 18, 20, 24, 36, 48, 75] \end{array}$$

Algorithm `Find`: Run through all arrays and make a binary search each
Worst-case Runtime :

Example: simple dictionary

data $\{1, 8, 10, 18, 20, 24, 36, 48, 50, 75, 99\}$, $n = 11$

$$\begin{aligned} A_0 &: \quad [50] \\ A_1 &: \quad [8, 99] \\ A_2 &: \quad \emptyset \\ A_3 &: \quad [1, 10, 18, 20, 24, 36, 48, 75] \end{aligned}$$

Algorithm **`Find`**: Run through all arrays and make a binary search each
Worst-case Runtime : $\Theta(\log^2 n)$,

$$\log 1 + \log 2 + \log 4 + \cdots + \log 2^k = \sum_{i=0}^{k} \log_2 2^i = \frac{k \cdot (k+1)}{2} \in \Theta(\log^2 n).$$

$(k = \lfloor \log_2 n \rfloor)$

Example: simple dictionary

Algorithm `Insert(x)`:

Example: simple dictionary

Algorithm `Insert(x)`:

- New array $A'_0 \leftarrow [x]$, $i \leftarrow 0$

Example: simple dictionary

Algorithm **`Insert(x)`**:

- New array $A'_0 \leftarrow [x]$, $i \leftarrow 0$
- while $A_i \neq \emptyset$, set A'_{i+1} =Merge(A_i, A'_i), $A_i \leftarrow \emptyset$, $i \leftarrow i+1$

Example: simple dictionary

Algorithm `Insert(x)`:

- New array $A'_0 \leftarrow [x]$, $i \leftarrow 0$
- while $A_i \neq \emptyset$, set A'_{i+1} =Merge(A_i, A'_i), $A_i \leftarrow \emptyset$, $i \leftarrow i+1$
- Set $A_i \leftarrow A'_i$

Example: simple dictionary

Algorithm `Insert(x)`:

- New array $A'_0 \leftarrow [x]$, $i \leftarrow 0$
- while $A_i \neq \emptyset$, set A'_{i+1} =Merge(A_i, A'_i), $A_i \leftarrow \emptyset$, $i \leftarrow i+1$
- Set $A_i \leftarrow A'_i$

Example: simple dictionary

Algorithm **`Insert(x)`**:

- New array $A'_0 \leftarrow [x]$, $i \leftarrow 0$
- while $A_i \neq \emptyset$, set A'_{i+1} =Merge(A_i, A'_i), $A_i \leftarrow \emptyset$, $i \leftarrow i+1$
- Set $A_i \leftarrow A'_i$

Insert(11)

$A'_0 = [11]$, $A'_1 = [11, 50]$, $A'_2 = [8, 11, 50, 99]$

$$\begin{array}{ll} A_0: & [50] \\ A_1: & [8, 99] \\ A_2: & \emptyset \\ A_3: & [1, 10, 18, 20, 24, 36, 48, 75] \end{array} \xrightarrow{\text{Insert}(11)} \begin{array}{ll} A_0: & \emptyset \\ A_1: & \emptyset \\ A_2: & [8, 11, 50, 99] \\ A_3: & [1, 10, 18, 20, 24, 36, 48, 75] \end{array}$$

Costs Insert

Notation in the following $n = 2^k$, $k = \log_2 n$

Assumption: creating new array A'_i with length 2^i (and, for $i > 0$ subsequent merge of A'_{i-1} and A_{i-1}) has costs $\Theta(2^i)$

In the worst case inserting an element into the data structure provides $\log_2 n$ such operations. $\Rightarrow$ Worst-case Costs Insert:

$$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1 \in \Theta(n).$$

Aggregate Analysis

Level	Costs	Example Array
0	1	$[*]$
1	2	$[*, *]$
2	4	$[*, *, *, *]$
3	8	$\emptyset$
4	16	$[*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Observation: when you start with an empty container, an insertion sequence merges reaches level 0 each time, level 1 (with costs 2) every second time, level 2 (with costs 4) every fourth time, level 3 (with costs 8) every eighth time etc.

Aggregate Analysis

Level	Costs	Example Array
0	1	$[*]$
1	2	$[*,*]$
2	4	$[*,*,*,*]$
3	8	$\emptyset$
4	16	$[*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*]$

Observation: when you start with an empty container, an insertion sequence merges reaches level 0 each time, level 1 (with costs 2) every second time, level 2 (with costs 4) every fourth time, level 3 (with costs 8) every eighth time etc.

Total costs: $1 \cdot \frac{n}{1} + 2 \cdot \frac{n}{2} + 4 \cdot \frac{n}{4} + \cdots + 2^k \cdot \frac{n}{2^k} = (k+1)n \in \Theta(n \log n)$.
Amortized cost per operation: $\Theta((n \log n)/n) = \Theta(\log n)$.

Account Method

Every element i $(1 \leq i \leq n)$ pays $a_i = \log_2 n$ coins when it is inserted into the data structure. The element pays the allocation of the first array and every subsequent merge-step that can occur until the element has reached array A_{k+1} $(k = \lfloor \log_2 \rfloor n)$. The account provides enough credit to pay for all Merge operations of the n elements.

$\Rightarrow$ Amortized costs for insertion $\mathcal{O}(\log n)$

Potential Method

We know from the account method that each element on the way to higher levels requires $\log n$ coins, i.e. that an element on level i still needs to posess $k-i$ coins. We use the potential

$$\Phi_i = \sum_{0\le i\le k:A_i\neq\emptyset}(k-i)\cdot 2^i$$

Potential Method

For the change of the potential $\Phi_i - \Phi_{i-1}$ we only have to consider the lower l levels that are occupied at time point $i-1$ (in analogy to the binary counter). Let l be the smallest index such that array A_l is empty. After merging array $A_0 \dots A_{l-1}$ arrays $A_i, 0 \leq i < l$ are now empty and array A_l is now full. Therefore:

$$\Phi_i - \Phi_{i-1} = (k-l) \cdot 2^l - \sum_{i=0}^{l-1} (k-i) \cdot 2^i$$

Real costs:

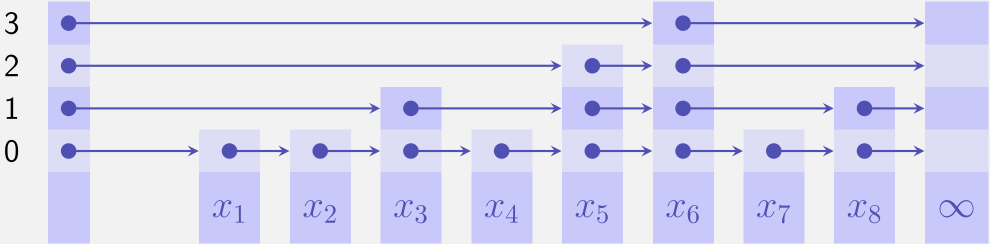
$$t_i = \sum_{i=0}^{l} 2^i = 2^{l+1} - 1$$

Potential Method

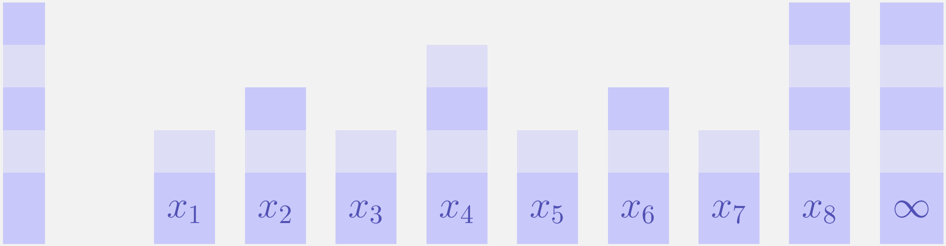
$$\begin{aligned}
\Phi_i - \Phi_{i-1} &= (k-l)\cdot 2^l - \sum_{i=0}^{l-1}(k-i)\cdot 2^i \\
&= (k-l)\cdot 2^l - k\cdot(2^l - 1) + \sum_{i=0}^{l-1} i\cdot 2^i \\
&= (k-l)\cdot 2^l - k\cdot(2^l - 1) + l\cdot 2^l - 2^{l+1} + 2 \\
&= k - 2^{l+1} + 2 \\
\Phi_i - \Phi_{i-1} + t_i &= k - 2^{l+1} + 2 + 2^{l+1} - 1 = k + 1 \in \Theta(\log n)
\end{aligned}$$

Randomized Skip List

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

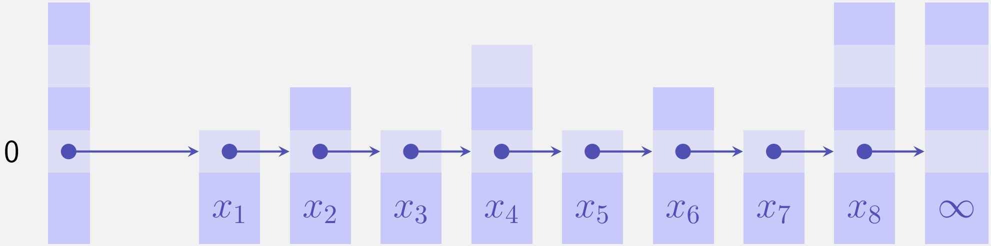


Randomized Skip List: finding element



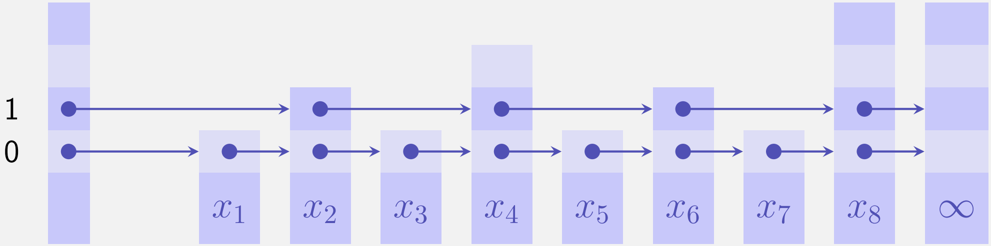
$$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9.$$

Randomized Skip List: finding element



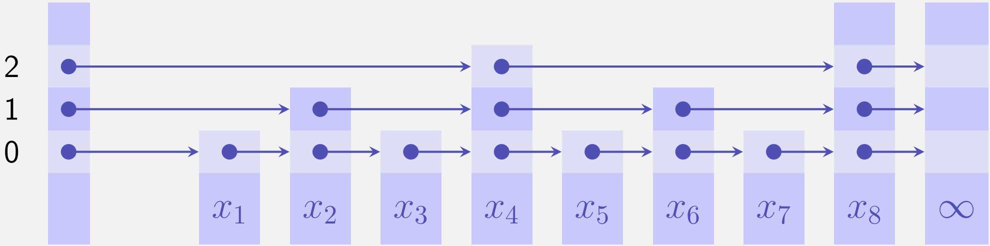
$$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9.$$

Randomized Skip List: finding element



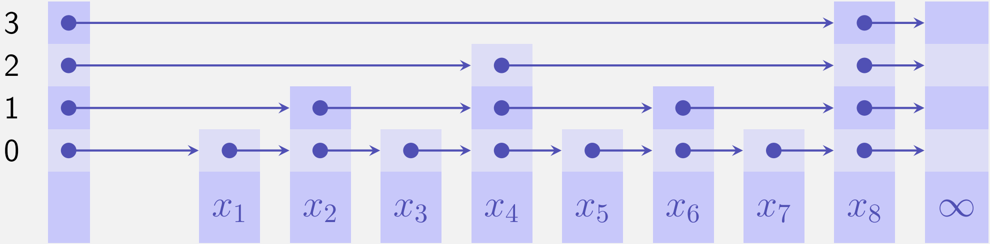
$$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_9.$$

Randomized Skip List: finding element



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

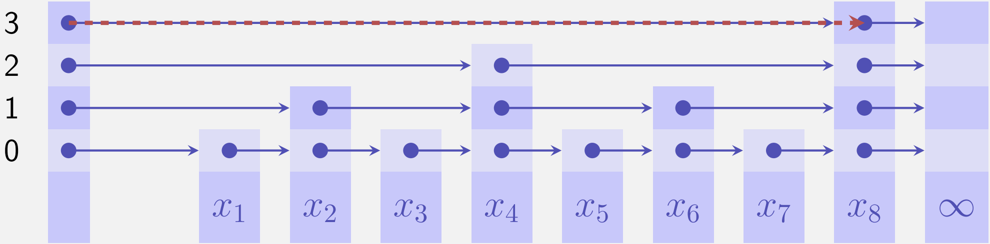
Randomized Skip List: finding element



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

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

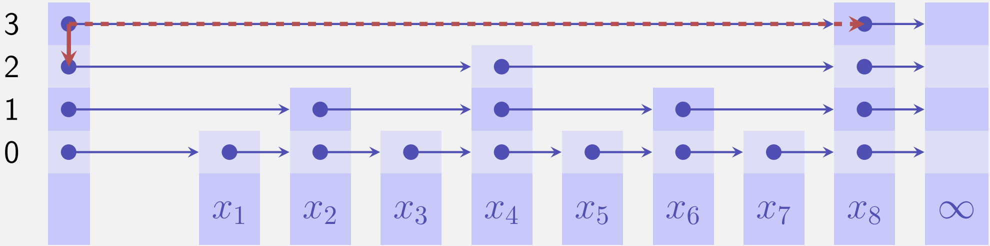
Randomized Skip List: finding element



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

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

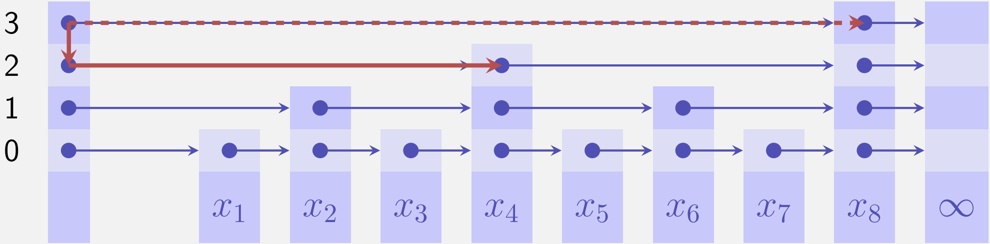
Randomized Skip List: finding element



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

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

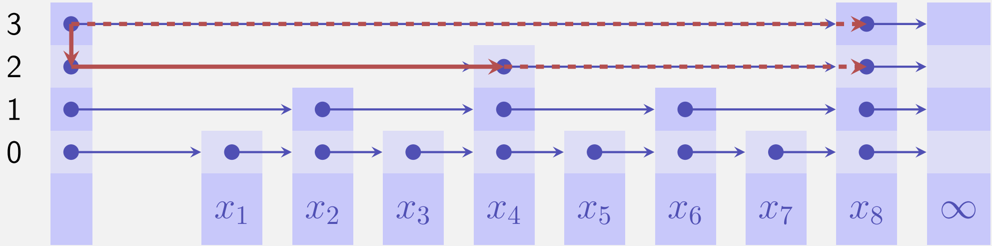
Randomized Skip List: finding element



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

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

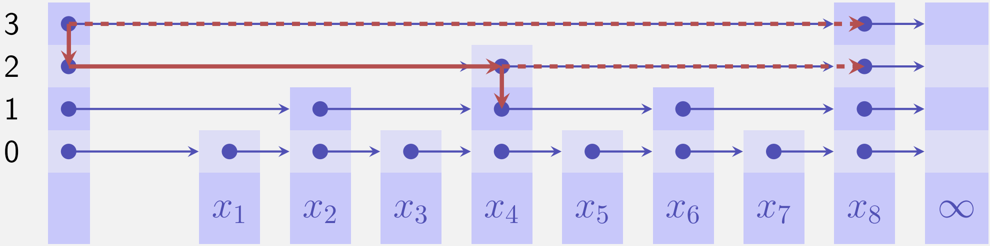
Randomized Skip List: finding element



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

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

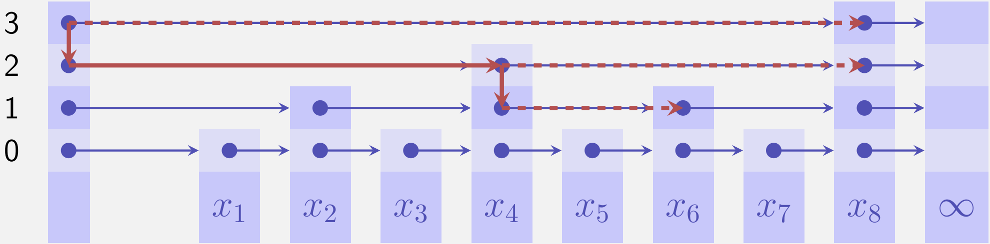
Randomized Skip List: finding element



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

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

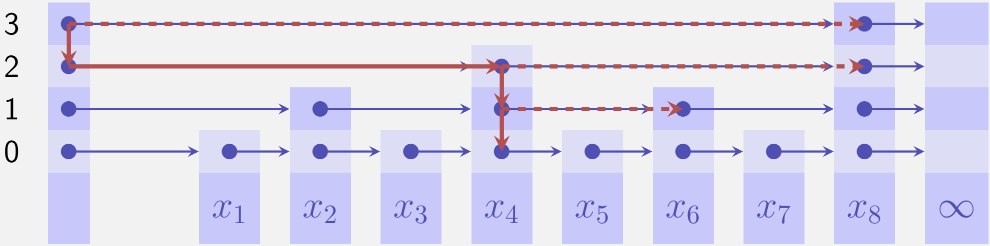
Randomized Skip List: finding element



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

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

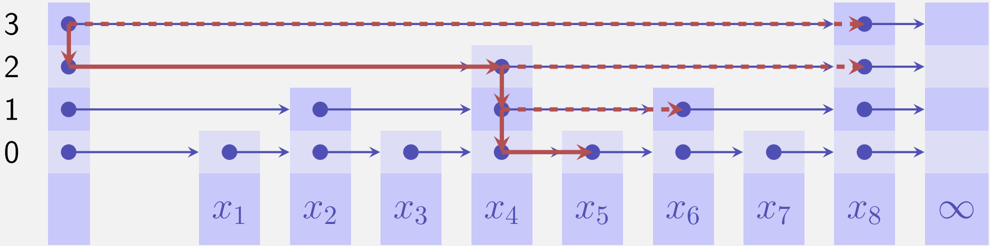
Randomized Skip List: finding element



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

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

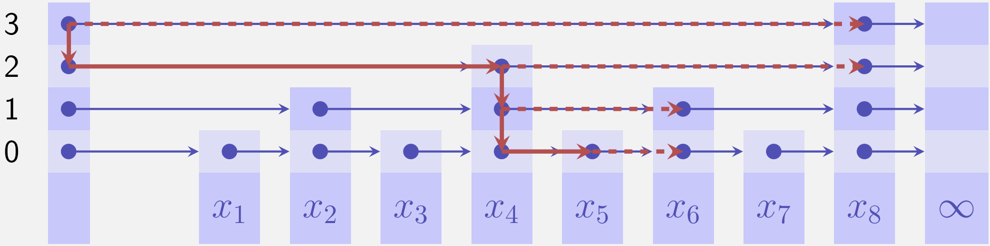
Randomized Skip List: finding element



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

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.

Types as Template Parameters

```
template <typename ElementType>
class vector{
        size_t size;
        T* elem;
public:
        ...
        vector(size_t s):
        size{s},
        elem{new ElementType[s]}{}
        ...
        ElementType& operator[](size_t pos){
                return elem[pos];
        }
        ...
}
```

Function Templates

```
template <typename T> // square number
T sq(T x){
        return x*x;
}
template <typename Container, typename F>
void apply(Container& c, F f){ // x <- f(x) forall x in c
        for(auto& x: c)
        x = f(x);
}
int main(){
        std::vector<int> v={1,2,3};
        apply(v,sq<int>);
        output(v); // 1 4 9
}
```

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?