

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 message.
- **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 $\mathcal{O}(\log n)$ 😊

Insert $\mathcal{O}(n)$ 😞

Delete $\mathcal{O}(n)$ 😞

Other idea

Implement dictionary as a linked list

Worst case number of fundamental operations

Search	$\mathcal{O}(n)$	☹️
Insert	$\mathcal{O}(1)$ ¹³	😊
Delete	$\mathcal{O}(n)$	☹️

¹³Provided 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:

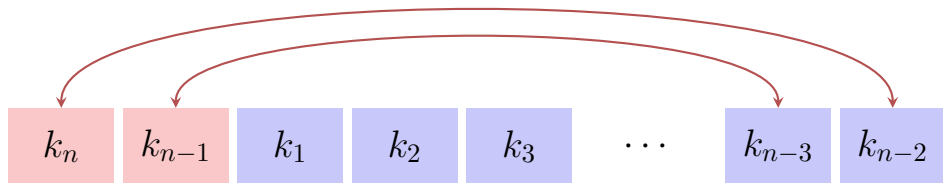


Worst case: Alternating sequence of n accesses to k_{n-1} and k_n .

Runtime: $\Theta(n^2)$

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

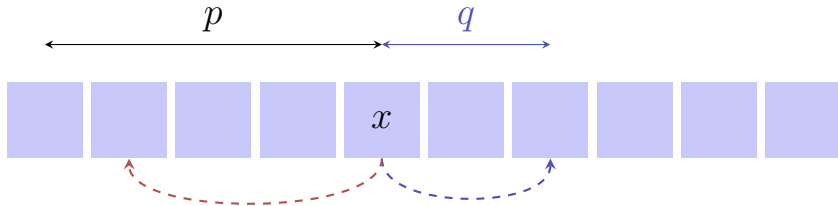
Compare MTF with the best-possible competitor (algorithm) A. How much better can A be?

Assumption: 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 k th 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 .

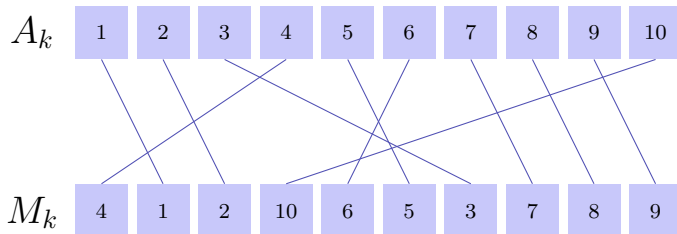
Potential Function

Potential function Φ = Number of inversions of A vs. MTF.

Inversion = Pair x, y such that for the positions of a and y

$$p^{(A)}(x) < p^{(A)}(y) \wedge p^{(M)}(x) > p^{(M)}(y) \text{ or}$$

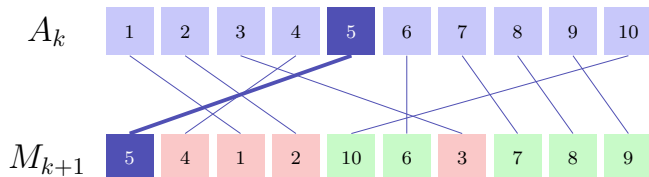
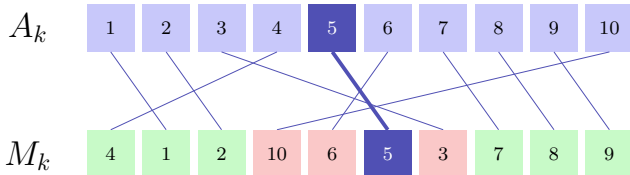
$$p^{(A)}(x) > p^{(A)}(y) \wedge p^{(M)}(x) < p^{(M)}(y)$$



#inversion = #crossings

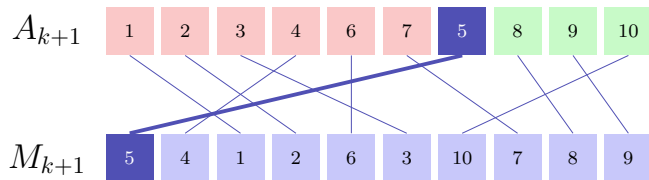
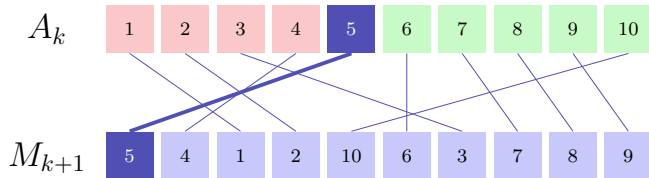
Estimating the Potential Function: MTF

- 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 i .
- $X_k^{(A)}$: number movements to the back (otherwise 0).
- access costs for i : $C_k^{(A)} = i$
- A increases the number of inversions by $X_k^{(A)}$.



Estimation

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

Amortized costs of MTF in step k :

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

Estimation

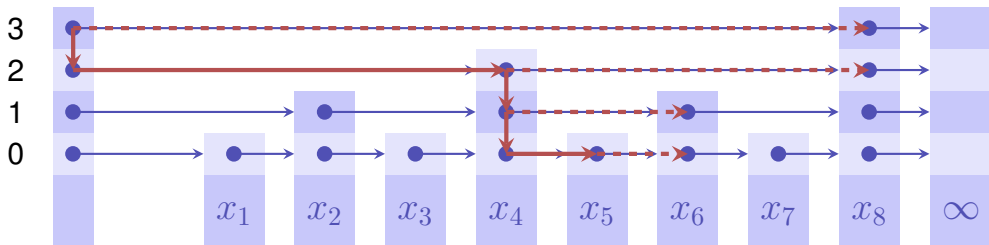
Summing up costs

$$\begin{aligned}\sum_k G_k^{(M)} &= \sum_k C_k^{(M)} \leq \sum_k a_k^{(M)} \leq \sum_k 2 \cdot C_k^{(A)} - 1 + X_k^{(A)} \\ &\leq \sum_k 2 \cdot C_k^{(A)} + X_k^{(A)} \leq 2 \cdot \sum_k C_k^{(A)} + X_k^{(A)} \\ &= 2 \cdot \sum_k G_k^{(A)}\end{aligned}$$

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

Cool idea: skip lists

Perfect skip list



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

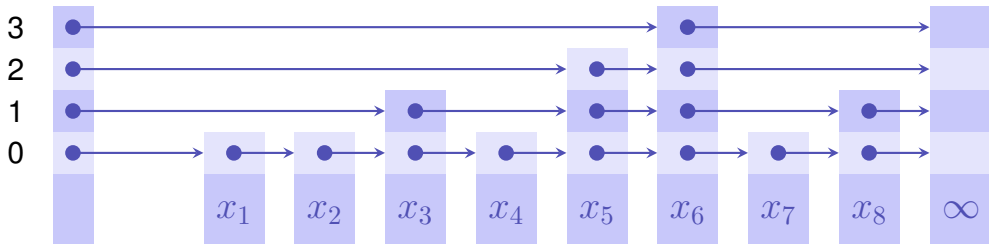
Example: search for a key x with $x_5 < x < x_6$.

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 $\mathbb{P}(H = i) = \frac{1}{2^{i+1}}$.



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 course observes the length of a path from a searched node back to the starting point in the highest level.

13. C++ advanced (III): Functors and Lambda

13.1 Appendix to previous C++ chapters

Appendix about Move-Semantics

```
// nonsense implementation of a "vector" for demonstration purposes
class vec{
public:
    vec () {
        std::cout << "default constructor\n";}
    vec (const vec&) {
        std::cout << "copy constructor\n";}
    vec& operator = (const vec&) {
        std::cout << "copy assignment\n"; return *this;}
    ~vec() {}
};
```

How many Copy Operations?

```
vec operator + (const vec& a, const vec& b){  
    vec tmp = a;  
    // add b to tmp  
    return tmp;  
}  
  
int main (){  
    vec f;  
    f = f + f + f + f;  
}
```

Output

default constructor

copy constructor

copy constructor

copy constructor

copy assignment

4 copies of the vector

Appendix about Move-Semantics

```
// nonsense implementation of a "vector" for demonstration purposes
class vec{
public:
    vec () { std::cout << "default constructor\n";}
    vec (const vec&) { std::cout << "copy constructor\n";}
    vec& operator = (const vec&) {
        std::cout << "copy assignment\n"; return *this;}
    ~vec() {}
    // new: move constructor and assignment
    vec (vec&&) {
        std::cout << "move constructor\n";}
    vec& operator = (vec&&) {
        std::cout << "move assignment\n"; return *this;}
};
```

How many Copy Operations?

```
vec operator + (const vec& a, const vec& b){  
    vec tmp = a;  
    // add b to tmp  
    return tmp;  
}  
  
int main (){  
    vec f;  
    f = f + f + f + f;  
}
```

Output

default constructor

copy constructor

copy constructor

copy constructor

move assignment

3 copies of the vector

How many Copy Operations?

```
vec operator + (vec a, const vec& b){  
    // add b to a  
    return a;  
}
```

```
int main (){  
    vec f;  
    f = f + f + f + f;  
}
```

Output

default constructor

copy constructor

move constructor

move constructor

move constructor

move assignment

1 copy of the vector

Explanation: move semantics are applied when an x-value (expired value) is assigned. R-value return values of a function are examples of x-values.

How many Copy Operations?

```
void swap(vec& a, vec& b){  
    vec tmp = a;  
    a=b;  
    b=tmp;  
}
```

```
int main (){  
    vec f;  
    vec g;  
    swap(f,g);  
}
```

Output

default constructor
default constructor
copy constructor
copy assignment
copy assignment

3 copies of the vector

Forcing x-values

```
void swap(vec& a, vec& b){  
    vec tmp = std::move(a);  
    a=std::move(b);  
    b=std::move(tmp);  
}  
  
int main (){  
    vec f;  
    vec g;  
    swap(f,g);  
}
```

Output

default constructor
default constructor
move constructor
move assignment
move assignment

0 copies of the vector

Explanation: With `std::move` an l-value expression can be transformed into an x-value. Then move-semantics are applied. <http://en.cppreference.com/w/cpp/utility/move>

13.2 Functors and Lambda-Expressions

Functors: Motivation

A simple output filter

```
template <typename T, typename function>
void filter(const T& collection, function f){
    for (const auto& x: collection)
        if (f(x)) std::cout << x << " ";
    std::cout << "\n";
}
```

Functors: Motivation

```
template <typename T, typename function>  
void filter(const T& collection, function f);
```

```
template <typename T>  
bool even(T x){  
    return x % 2 == 0;  
}
```

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};  
filter(a,even<int>); // output: 2,4,6,16
```

Functor: object with overloaded operator ()

```
class LargerThan{  
    int value; // state  
public:  
    LargerThan(int x):value{x}{};  
  
    bool operator() (int par){  
        return par > value;  
    }  
};
```

Functor is a callable object. Can be understood as a stateful function.

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};  
int value=8;  
filter(a,LargerThan(value)); // 9,11,16,19
```

Functor: object with overloaded operator ()

```
template <typename T>
class LargerThan{
    T value;
public:
    LargerThan(T x):value{x}{};

    bool operator() (T par){
        return par > value;
    }
};
```

also works with a template, of course

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int value=8;
filter(a,LargerThan<int>(value)); // 9,11,16,19
```

The same with a Lambda-Expression

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};  
int value=8;
```

```
filter(a, [value](int x) {return x>value;} );
```

Sum of Elements – Old School

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};  
int sum = 0;  
for (auto x: a)  
    sum += x;  
std::cout << sum << "\n"; // 83
```

Sum of Elements – with Functor

```
template <typename T>
struct Sum{
    T & value = 0;
    Sum (T& v): value{v} {}

    void operator() (T par){
        value += par;
    }
};

std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};
int s=0;
Sum<int> sum(s);
sum = std::for_each(a.begin(), a.end(), sum);
std::cout << s << "\n"; // 83
```

Sum of Elements – with Λ

```
std::vector<int> a {1,2,3,4,5,6,7,9,11,16,19};  
int s=0;
```

```
std::for_each(a.begin(), a.end(), [&s] (int x) {s += x;} );
```

```
std::cout << s << "\n";
```

Sorting, different

```
// pre: i >= 0
// post: returns sum of digits of i
int q(int i){
    int res =0;
    for(;i>0;i/=10)
        res += i % 10;
    return res;
}
```

```
std::vector<int> v {10,12,9,7,28,22,14};
std::sort (v.begin(), v.end(),
    [] (int i, int j) { return q(i) < q(j);}
);
```

Now $v = 10, 12, 22, 14, 7, 9, 28$ (sorted by sum of digits)

Lambda-Expressions in Detail

`[value] (int x) ->bool {return x>value;}`

capture parameters return
type statement

Closure

```
[value] (int x) ->bool {return x>value;}
```

- Lambda expressions evaluate to a temporary object – a closure
- The closure retains the execution context of the function, the captured objects.
- Lambda expressions can be implemented as functors.

Simple Lambda Expression

```
[] () -> void {std::cout << "Hello World";}
```

call:

```
[] () -> void {std::cout << "Hello World";}();
```

Minimal Lambda Expression

`[] {}`

- Return type can be inferred if ≤ 1 return statement.

```
[] () {std::cout << "Hello World";}
```

- If no parameters and no return type, then `()` can be omitted.

```
[] {std::cout << "Hello World";}
```

- `[...]` can never be omitted.

Examples

```
[] (int x, int y) {std::cout << x * y;} (4,5);
```

Output: 20

Examples

```
int k = 8;  
[](int& v) {v += v;} (k);  
std::cout << k;
```

Output: 16

Examples

```
int k = 8;  
[] (int v) {v += v;} (k);  
std::cout << k;
```

Output: 8

Capture – Lambdas

For Lambda-expressions the capture list determines the context accessible

Syntax:

- `[x]`: Access a copy of x (read-only)
- `[&x]`: Capture x by reference
- `[&x, y]`: Capture x by reference and y by value
- `[&]`: Default capture all objects by reference in the scope of the lambda expression
- `[=]`: Default capture all objects by value in the context of the Lambda-Expression

Capture – Lambdas

```
int elements=0;
int sum=0;
std::for_each(v.begin(), v.end(),
    [&] (int k) {sum += k; elements++;} // capture all by reference
)
```

Capture – Lambdas

```
template <typename T>
void sequence(vector<int> & v, T done){
    int i=0;
    while (!done()) v.push_back(i++);
}
```

```
vector<int> s;
sequence(s, [&] {return s.size() >= 5;} )
```

now v = 0 1 2 3 4

The capture list refers to the context of the lambda expression.

Capture – Lambdas

When is the value captured?

```
int v = 42;  
auto func = [=] {std::cout << v << "\n"};  
v = 7;  
func();
```

Output: 42

Values are assigned when the lambda-expression is created.

Capture – Lambdas

(Why) does this work?

```
class Limited{
    int limit = 10;
public:
    // count entries smaller than limit
    int count(const std::vector<int>& a){
        int c = 0;
        std::for_each(a.begin(), a.end(),
            [=,&c] (int x) {if (x < limit) c++;}
        );
        return c;
    }
};
```

The `this` pointer is implicitly copied by value

Capture – Lambdas

```
struct mutant{  
    int i = 0;  
    void do(){ [=] {i=42;}();}  
};
```

```
mutant m;  
m.do();  
std::cout << m.i;
```

Output: 42

The `this` *pointer* is implicitly copied by value

Lambda Expressions are Functors

```
[x, &y] () {y = x;}
```

can be implemented as

```
unnamed {x,y};
```

with

```
class unnamed {  
    int x; int& y;  
    unnamed (int x_, int& y_) : x (x_), y (y_) {}  
    void operator () () {y = x;}};  
};
```

Lambda Expressions are Functors

```
[=] () {return x + y;}
```

can be implemented as

```
unnamed {x,y};
```

with

```
class unnamed {  
    int x; int y;  
    unnamed (int x_, int y_) : x (x_), y (y_) {}  
    int operator () () {return x + y;}  
};
```

Polymorphic Function Wrapper `std::function`

```
#include <functional>

int k= 8;
std::function<int(int)> f;
f = [k](int i){ return i+k; };
std::cout << f(8); // 16
```

Kann verwendet werden, um Lambda-Expressions zu speichern.

Other Examples

```
std::function<int(int,int)>;
std::function<void(double)> ...
```

<http://en.cppreference.com/w/cpp/utility/functional/function>