Data Structures and Algorithms

Conf. dr. ing. Guillaume Ducoffe

guillaume.ducoffe@fmi.unibuc.ro

Randomized Data Structures

- Randomized Algorithms
  - (Pseudo)Random Generator
- An application to Lists: **Skip Lists**
- An application to vectors: **Hashing**
  - generalized frequency vector

Non-deterministic Algorithms

An algorithm is:

- deterministic if, for any fixed input, the output is always the same.
- non-deterministic otherwise.

A non-deterministic algorithm can always be (re)written as a deterministic algorithm with one additional input (generated non-deterministically), sometimes called a **certificate**.

$\Longrightarrow$ "checker"

Non-deterministic Complexity: Size of the certificate + Runtime of the (det.) checker

Deterministic Complexity: $2^{\text{Size of the Certificate}}$ + Runtime of the checker

$\Longrightarrow$ Brute-force enumeration of all possible certificates

Randomized Algorithms

Somehow between deterministic and non-deterministic.

- Deterministic checker + a random certificate

  (according to some distribution over the search space)

The random certificate is computed by using a **random generator**

- In its simplest form, a random generator output a single bit, that is set to 1 (resp., 0) with probability 1/2.
- Repeated uses of a random gen. allows one to:
  - output a random number within some fixed range
  - output a random even/odd/prime/etc. number within some fixed range.
  - output a single bit that is set to 1 with some probability $p \neq 1/2$
  - ...

Pseudorandom generators

(Rough intuition) An algorithm cannot generate a "true" random number because its output can be deduced – at least partially – from both its input and the list of its operations.

$\Longrightarrow$ need for an external source of randomness (*e.g.*, a physical phenomenon such as the temperature at the surface of a processor). Sometimes called a seed.

But measurement is inherently biased, and so the external source of randomness must be "corrected".

$\Longrightarrow$ use of chaotic sequences, whose output may drastically change depending on the input.

Ex: **linear congruences** $X_{n+1} = (a \cdot X_n + b) \mod m$.

Implementation in C/C++

The <cstdlib> library proposes a default random generator.

```
int rand(void)
```

$\longrightarrow$ Pseudo-uniform generation of an integer between 0 and some pre-defined constant `RAND_MAX`

Can be turned into a generator within any fixed interval $[a; b]$:

```
int s = (rand() % (b-a+1)) + a
```

Generation of a random bit with probability $1/p$, $p \in \mathbb{N}$:

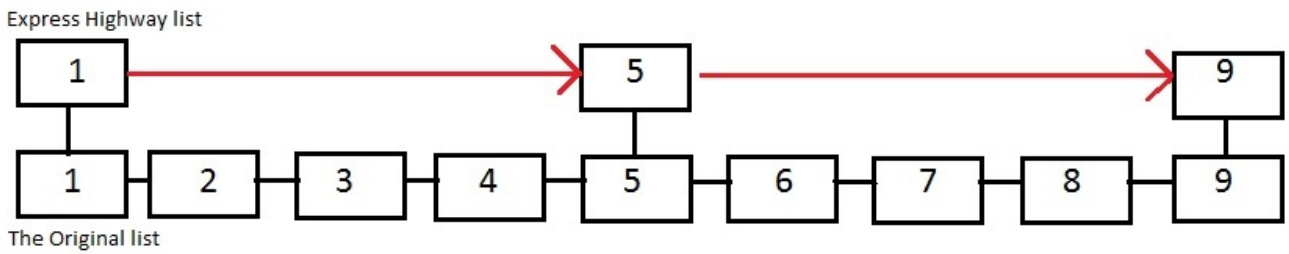
```
int s = rand() % p; if(s == 0) { ...}
```

Do not forget the seed! The function `void srand(int)` must be called before using the generator. Ex: `srand(time(NULL));`

Skip Lists

- Invented by W. Pugh in order to speed up list operations from $\mathcal{O}(n)$ to $\mathcal{O}(\log n)$ time **in expectation** (over all possible random choices).

- Hierarchy of lists: each element of the i^{th} list is included in the next $(i+1)^{th}$ list with some probability p (parameter of the data structure). The other elements are skipped in order to speed up Searching.



NB: The first element is always included.

Analysis

- Each element of the original list (level 0) is contained into $k+1$ lists of the hierarchy with probability:

$$p^k \cdot (1-p)$$

Therefore, in expectation, an element is contained into

$$1 + (1-p)\sum_{k \geq 0} k \cdot p^k = \frac{1}{1-p} = \mathcal{O}(1)$$

- The i^{th} list contains in expectation $p^i \cdot n$ elements.

Therefore, the expected number of lists is at most $\log_{1/p}(n)$.

Implementation

```
struct node {
    int value;
    node *next_elt; //next element into the list
    node *prev_lvl; //pointer to this element in the previous level
    int pos; //position in the original list – level 0
};

//pointer to the head of the list at the highest level
typedef node *SkipList;
```

Initialization from a list

Adding one level...

```
//Assumption: list with > 1 elements
SkipList L = ...

//Copy the former list
SkipList S = L;
L = new node;
L->value = S->value; L->prev_lvl = S; L->pos = 0;
srand(time(0));
S = S->next_elt; node *n = L;
while(S != nullptr){
   if(rand() % q == 0) { //q = 1/p
      node *m = new node;
      m->value = S -> value; m->prev_lvl = S; m->pos = S->pos;
      n->next_elt = m; n = m;
   }
   S = S->next_elt;
}
n->next_elt = nullptr;
```

Search by index

```
int get(SkipList& L, int i) {
   node *n = L; //head of the list
   while(n->next_elt != nullptr && n->next_elt->pos <= i)
      n = n->next;
   if(n->prev_lvl != nullptr)
      return get(n->prev_lvl,i); //continue in the previous level
   else return n->value;
}
```

Expected complexity: $\mathcal{O}(\log n)$ (see next slide)

Search by index: Analysis

Consider any element at level 1. We skip the k next elements with probability $(1-p)^k$.

$\Longrightarrow$ The expected distance between two consecutive selected elements equals:

$$1 + \sum_{k \geq 0} k(1-p)^k p = \frac{1}{p}$$

$\Longrightarrow$ The expected distance (in the original list) between two consecutive vertices at level j grows **exponentially** with j

$\Longrightarrow$ Access to element i requires $\mathcal{O}(\log n)$ (number of lists) + $\mathcal{O}(\log i)$ (number of traversed elements) in expectation. $\underline{= \mathcal{O}(\log n).}$

Search by value

The original motivation for skip lists was to replace **binary search** for **sorted** lists.

```
// Assumption: sorted list

int find(SkipList& L, int e){
    node *n = L; //head of the list
    while(n->next_elt != nullptr && n->next_elt->value <= e)
        n = n->next;
    if(n->prev_lvl != nullptr)
        return find(n->prev_lvl,e); //continue in the previous level
    else {
        if(n->value == e) return n->pos;
        else return -1;
    }
}
```

Expected Complexity: $\mathcal{O}(\log n)$.

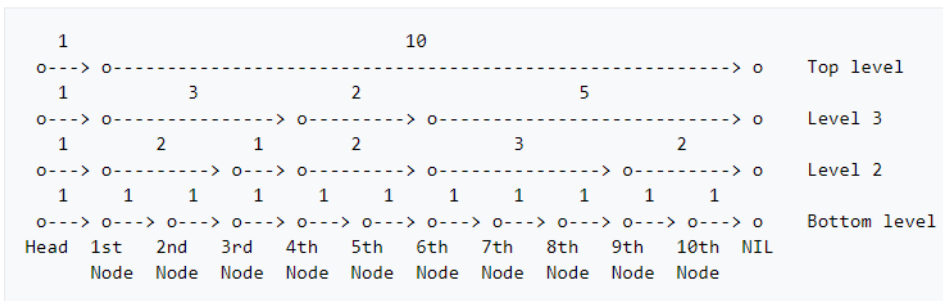
Insertion/Removal

- We constructed SkipLists from existing linked lists.

- However, SkipLists can also support insertion/deletion of new elements, at arbitrary indices, in expected $\mathcal{O}(\log n)$ time.

- This requires changing the implementation because adding/removing an element might require actualizing the field `pos` for $\mathcal{O}(n)$ elements!

- Solution: replace `pos` by another integer field `width`, that represents the number of skipped elements between a node and the next node in the list.



Frequency Vectors

A simple (*non-randomized*) data structure for Searching.

Example: consider a vector `v[] = {1, 1, 10, 5, 7, 2, 11, 19, 3}`

- Let $m = \max\{\texttt{v[i]} \mid 0 \leq i < n\}$. – Here, $m = 19$.

  $\rightarrow$ More generally, let M be the upper-limit of the integer range considered ($\approx$ size of the search space).

- Construct an array `num` of length $m + 1$ so that $\widehat{\texttt{num[i]}}$ represents the number of occurrences of element i in the vector.

  – Here, `num[] = {0,2,1,1,0,1,0,1,0,0,1,1,0,0,0,0,0,0,0,1}`

Pre-processing time: $\mathcal{O}(n + m)$ – **exponential** in $\log m$

Query time: $\mathcal{O}(1)$

Hashing

Definition

A Hash function $f : \mathbb{N} \to \mathbb{N}$ maps the data to a *smaller domain*.

- Data domain $\neq$ Vector size!

Ex: how many strings of length n over the alphabet $\{A, T, C, G\}$?

Hashing

Definition

A Hash function $f : \mathbb{N} \to \mathbb{N}$ maps the data to a *smaller domain*.

- Data domain $\neq$ Vector size!

Ex: how many strings of length n over the alphabet $\{A, T, C, G\}$?
$\Longrightarrow 4^n$ (Exponential)

Hashing

Definition

A Hash function $f : \mathbb{N} \to \mathbb{N}$ maps the data to a *smaller domain*.

- Data domain $\neq$ Vector size!

Ex: how many strings of length n over the alphabet $\{A, T, C, G\}$?
$\Longrightarrow 4^n$ (Exponential)

- Typical scenario: we shall map the n integer values stored in a vector to $\{1, 2, \ldots, c \cdot n\}$ for some small constant c.

Hashing

Definition

A Hash function $f : \mathbb{N} \to \mathbb{N}$ maps the data to a *smaller domain*.

- Data domain $\neq$ Vector size!

Ex: how many strings of length n over the alphabet $\{A, T, C, G\}$?
$\Longrightarrow 4^n$ (Exponential)

- Typical scenario: we shall map the n integer values stored in a vector to $\{1, 2, \ldots, c \cdot n\}$ for some small constant c.

Questions:

- How to choose the hash function?
- What if there exist $x \neq y$ s.t. $f(x) = f(y)$ (**collision**) ?

Selection of a random hash function

- A simple case: n-size vector whose each element is in $\mathcal{O}(n)$.

$\Longrightarrow$ frequency vector! (Deterministic choice)

- General case: we consider a family $\mathcal{H}$ (possibly infinite) of hash functions and we select one $h \in \mathcal{H}$ uniformly at random.

- The hash function h is randomly selected but it is itself a **Deterministic** function: For any *fixed* value x, the value $h(x)$ cannot change between two different calls to the hash function.

– otherwise, nothing could work...

Universal Hashing

Motivation: minimizing collisions

$\longrightarrow$ One possible way consists to minimizing the size of pre-images $h^{-1}(i) = \{x \mid h(x) = i\}$.

Observation: if we map the elements of an n-size vector to $\{0, 1, \ldots, m-1\}$ then some pre-image has size $\Omega(n/m)$.

Definition (Universality)

$\forall x, y \ Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq 1/m$

- Implies that all pre-images have $\mathcal{O}(n/m)$ size in expectation!
- Can be relaxed to $\forall x, y \ Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq c/m$ for some small constant c.

Simple hash functions

• **The identity function** is always universal, and even perfect (no collisions).

Remark: space-efficient only if the data domain is small.

• **The trivial hash functions**. Fix a subset of m bits in the binary representation.

Formally, $h : 2^n \mapsto 2^m$. In particular, $\mathcal{H}$ has cardinality $\binom{n}{m}$.

Ex: $n = 10$ bits, select the $m = 3$ bits $1, 4, 9$

$$0011011001 \rightarrow 000$$

If x, y are random n-bit numbers then $\forall h,\ Pr[h(x) = h(y)] = \frac{1}{2^m}$.
However these are not universal hash functions

(*e.g.*, case of x, y differing in a single bit. . .)

Carter and Wegman's Hashing functions

Context: mapping the n values in a vector to $\{0, 1, \ldots, m-1\}$.

- Fix a prime number $p \geq n$ – *e.g.*, $p \in [n; 2n]$
- Select random numbers $a, b \in \mathbb{Z}_p$, $a \neq 0$.

$$h_{a,b}(x) = ((a \cdot x + b) \pmod{p}) \pmod{m}$$

$\rightarrow$ Double use of Division Hashing

$\rightarrow$ "Linear" function: preserves many interesting properties.

A hash function **preserves** a property $\mathcal{P}$ if $\mathcal{P}(x, y) \Longrightarrow \mathcal{P}(f(x), f(y))$.

Example: $x \leq y \Longrightarrow f(x) \leq f(y)$.

$\rightarrow$ There exist some "algebraic" variants (*i.e.*, using polynomials)

Analysis

Theorem

The hashing functions $h_{a,b}$ (for fixed p, n, m) are universal.

We have $h(x) = h(y)$ if and only if $ax + b = ay + b + km \pmod{p}$ for some $k \neq 0$.

In particular: $a = km(x - y)^{-1} \pmod{p}$

There are only $\mathcal{O}(p/m)$ possible values for k, and so, also for a.

Hash Tables

Store a collection of pairs `(key,value)`.

Example: `key` = element from a vector, `value` = position in the vector.

- Three operations:
  - **int** `search(int);` Returns the value associated to some key (if it is present in the table)
  - **void** `insert(int,int);` Adds a new pair `(key,value)` (if the key is not already present in the table)
  - **void** `delete(int);` Deletes a pair `(key,value)` – given its key.

Elements are stored in an m-size array using a universal hashing function.

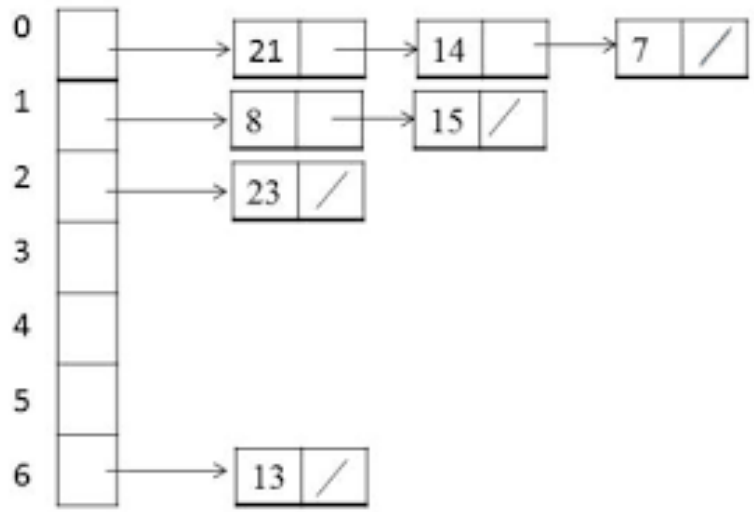
$\rightarrow$ Needs for **rehashing** in case of resize

Load $\alpha = n/m$ ($= \mathcal{O}(1)$ provided $m = \Omega(n)$)

Resolving collisions

Separate chaining

- For every hash value i, we store $f^{-1}(i) = \{x \mid f(x) = i\}$ in a linked list.



- Insertion of an element in $\mathcal{O}(1)$ time – Worst-Case.

- Search/Removal in $\mathcal{O}(\max_i |f^{-1}(i)|)$. This is $\mathcal{O}(\alpha)$ on average.

Resolving collisions

Linear probing

- When inserting an element x: if the cell $f(x)$ is already occupied, then go to the cell $f(x)+1$. Repeat until you find an empty cell.

Interpretation: if f is uniform, then so are $f+1, f+2, \ldots$.

If $f(x) = i$ and the existing keys $x_1, x_1, \ldots, x_n$ are in positions $i_1, i_2, \ldots, i_n$ then we can insert x in average time:

$$\leq \frac{1}{(1-\alpha)^2}$$

More about Linear probing

- Searching for an element x?

- Deleting an element x?

More about Linear probing

- Searching for an element x?

$\longrightarrow$ Start from the cell $f(x)$ and continue until either you find x or an empty cell.

Complexity?

- Deleting an element x?

More about Linear probing

- Searching for an element x?

$\longrightarrow$ Start from the cell $f(x)$ and continue until either you find x or an empty cell.

Complexity? $\mathcal{O}((1-\alpha)^{-2})$-time on average.

- Deleting an element x?

More about Linear probing

- Searching for an element x?

$\longrightarrow$ Start from the cell $f(x)$ and continue until either you find x or an empty cell.

Complexity? $\mathcal{O}((1-\alpha)^{-2})$-time on average.

- Deleting an element x?

$\longrightarrow$ Find element x in cell i and delete it.

While there is an element y on the next cell $i+1$ such that $f(y) \neq i+1$: write element y in cell i; $i \leftarrow i+1$.

Complexity?

More about Linear probing

- Searching for an element x?

$\longrightarrow$ Start from the cell $f(x)$ and continue until either you find x or an empty cell.

Complexity? $\mathcal{O}((1-\alpha)^{-2})$-time on average.

- Deleting an element x?

$\longrightarrow$ Find element x in cell i and delete it.

While there is an element y on the next cell $i+1$ such that $f(y) \neq i+1$: write element y in cell i; $i \leftarrow i+1$.

Complexity? $\mathcal{O}((1-\alpha)^{-2})$-time on average.

Resolving collisions

Quadratic probing

Same as linear probing (**open addressing**), but when inserting an element x: try the cells

$$f(x), f(x) + u(1), f(x) + u(2), \ldots, f(x) + u(k), \ldots$$

for some quadratic function $u : i \to a \cdot i^2 + b \cdot i$.

Ex: if $a = 1, b = 0$ then go to cells $f(x), f(x) + 1, f(x) + 4, f(x) + 9, \ldots$.

- Comparison with linear probing: avoid the clustering effect (all elements consecutive in the hash table).

Resolving collisions

Double hashing

A sort of generalization of the other open addressing methods.

Choose a **second hash function** $h \neq f$ such that:

- $\forall i,\ h(i) \neq 0$ (we do not come back on the first cell)
- h cycles through the whole indices (if a cell is empty, we will find one).
- h is efficient to compute
- h is independent from f
  (to ensure uniformity + other properties of $f + h$)

$\longrightarrow$ Try the cells $f(x), f(x) + h(1), f(x) + h(2), \ldots, f(x) + h(i), \ldots$.

Average complexity: $\sum_{j \geq 1} \alpha^j (1 - \alpha) = \frac{1}{1-\alpha}$

Resolving collisions

Cuckoo Hashing

We maintain two different tables T_1, T_2, each using a different hash function h_1, h_2.

- An element x can only be found in either $T_1[h_1(x)]$ or $T_2[h_2(x)]$.

$\Longrightarrow$ Search/Deletion in $\mathcal{O}(1)$ Worst-Case time.

- Insertion of x:
  - If $T_1[h_1(x)]$ is empty, then we set $T_1[h_1(x)] := x$.
  - Otherwise, let $y = T_1[h_1(x)]$. Set $T_1[h_1(x)] := x$.
  - If $T_2[h_2(y)]$ is empty, then we set $T_2[h_2(y)] := y$.
  - Otherwise, let $z = T_2[h_2(y)]$. Set $T_2[h_2(y)] := y$.
  - If $T_1[h_1(z)]$ is empty, then we set $T_1[h_1(z)] := z$.
  - ...

Questions