# Algorithms and datastructures I
# Lecture 9: RB-trees and hashing

Jan Hubička

Department of Applied Mathematics
Charles University
Prague

March 24 2020

# Set datastructure

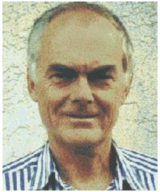
We would like to represent a set (or a dictionary) of some elements from an universe.
We expect that elements of the universum in set can be assigned and compared in $O(1)$.

- INSERT($v$): Insert $v$ to the set.
- DELETE($v$): Delete $v$ from the set.
- FIND($v$): Find $v$ in the set.
- MIN: Return minimum.
- MAX: Return maximum.
- SUCC($v$): Find successor.
- PRED($v$): Find predecessor.

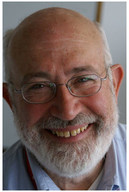
## Basic implementations

| | INSERT | DELETE | FIND | MIN/MAX | SUCC/PRED |
|---|---|---|---|---|---|
| Linked list | $O(n)$ or $O(1)$ | $O(n)$ or $O(1)$ | $O(n)$ | $O(n)$ | $O(n)$ |
| Array | $O(n)$ or $O(1)$ | $O(n)$ or $O(1)$ | $O(n)$ | $O(n)$ | $O(n)$ |
| Sorted array | $O(n)$ | $O(n)$ | $O(\log n)$ | $O(1)$ | $O(\log n)$ or $O(1)$ |
| binary search trees | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ |
| AVL-trees | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ |
| $(r, b)$-trees | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ |

# $(a, b)$-trees (Bayer, McCreight)



Rudolf Bayer



Edward M. McCreight

**Definition (Generalized search tree)**

Generalised search tree is a rooted tree with specified order of sons and two types of vertices:

1. Internal vertices contains non-zero number of keys. If internal vertex has keys $x_1 < \cdots < x_k$ then it has $k+1$ sons $s_0, \ldots, s_k$. Keys separate values in sons, so:
   $T(s_0) < x_1 < T(s_1) < x_2 < \cdots < x_{k-1} < T(s_{k-1}) < x_k < T(s_k)$
2. External vertices contain no keys and are leaf.

**Definition ($(a, b)$-tree)**

$(a, b)$-tree for a given $a \geq 2$, $b \geq 2a - 1$ is a generalised search tree such that:

1. Root has 2 to $b$ sons.
2. Other internal vertices have $a$ to $b$ sons.
3. All external vertices are in the level.

**Lemma**

*Every $(a, b)$-tree with $n$ keys has depth $\Theta(\log n)$.*

# Insert to $(a, b)$-tree

**Insert($v$,$x$)**

Let $u$ be the last internal vertex visited by Find($v$,$x$).

1. If $u$ contains $x$ return.
2. Otherwise add $x$ into $u$ and insert new external vertex
3. If $u$ has more than $b$ sons, split it possibly recursing to father.

It is possible to split preventively if $b \geq 2a$. We will use it today.

# Red-black trees (Bayer 1972; Guibas, Sedgewick 1978; Anderson 1993; Sedgewick 2008)

We can represent $(2, 4)$-trees using binary search trees with colored edges.

Leonidas J. Guibas

Robert Sedgewick

# Red-black trees (Bayer 1972; Guibas, Sedgewick 1978; Anderson 1993; Sedgewick 2008)

We can represent $(2, 4)$-trees using binary search trees with colored edges.

Leonidas J. Guibas

Robert Sedgewick

**Definition (Left leaning red-back tree)**

LLRB-tree is binary search tree with external vertices and edges colored either red or black. It satisfies:

1. There are no two red edges adjacent to each other.
2. If there is only one red edge from a vertex then it is left.
3. Edges to leaves are always black.
4. Every path from root to leaf goes through the same number of black edges.

# Red-black trees (Bayer 1972; Guibas, Sedgewick 1978; Anderson 1993; Sedgewick 2008)

We can represent $(2, 4)$-trees using binary search trees with colored edges.

**Definition (Left leaning red-back tree)**

LLRB-tree is binary search tree with external vertices and edges colored either red or black. It satisfies:

1. There are no two red edges adjacent to each other.
2. If there is only one red edge from a vertex then it is left.
3. Edges to leaves are always black.
4. Every path from root to leaf goes through the same number of black edges.

# Red-black trees (Bayer 1972; Guibas, Sedgewick 1978; Anderson 1993; Sedgewick 2008)

We can represent $(2, 4)$-trees using binary search trees with colored edges.

**Definition (Left leaning red-back tree)**

LLRB-tree is binary search tree with external vertices and edges colored either red or black. It satisfies:

1. There are no two red edges adjacent to each other.
2. If there is only one red edge from a vertex then it is left.
3. Edges to leaves are always black.
4. Every path from root to leaf goes through the same number of black edges.

Optimization: Color of edge may be stored in its destination vertex.

# Depth of LLRB-trees

**Lemma**

*Every LLRB-tree with $n$ keys has depth $\Theta(\log n)$.*

**Proof.**

We know that every LLRB-tree tree corresponds to an $(2, 4)$-tree of height $h = \Theta(\log n)$.
The height $h'$ of LLRB tree is $h \leq h' \leq 2h$. □

# Operations on LLRB-trees

**Observation**

Operations FIND, MIN, MAX, SUCC and PRED run in $\Theta(\log n)$.

# Operations on LLRB-trees

**Observation**

Operations FIND, MIN, MAX, SUCC and PRED run in $\Theta(\log n)$.

Operations INSERT and DELETE can be derived from ones on $(2, 4)$-trees.

# INSERT to an LLRB-tree

Lets see how insertion to $(2, 4)$-tree with preventive splitting translates to RB-tree.

**Insert($v$,$x$)**

1. If $v = \emptyset$: return newly created red vertex with key $x$.
2. If $x = k(v)$: Return $v$.

# INSERT to an LLRB-tree

Lets see how insertion to $(2, 4)$-tree with preventive splitting translates to RB-tree.

**Insert($v$,$x$)**

1. If $v = \emptyset$: return newly created red vertex with key $x$.
2. If $x = k(v)$: Return $v$.
3. If $l(v)$ and $r(v)$ are red: change color of $v$, $l(v)$ and $r(v)$.

# INSERT to an LLRB-tree

Lets see how insertion to $(2, 4)$-tree with preventive splitting translates to RB-tree.

**Insert($v$,$x$)**

1. If $v = \emptyset$: return newly created red vertex with key $x$.
2. If $x = k(v)$: Return $v$.
3. If $l(v)$ and $r(v)$ are red: change color of $v$, $l(v)$ and $r(v)$.
4. If $x < k(v)$: $l(v) \leftarrow$ Insert($l(v)$, $x$).
5. If $x > k(v)$: $r(v) \leftarrow$ Insert($r(v)$, $x$).

# INSERT to an LLRB-tree

Lets see how insertion to $(2,4)$-tree with preventive splitting translates to RB-tree.

**Insert($v$,$x$)**

1. If $v = \emptyset$: return newly created red vertex with key $x$.
2. If $x = k(v)$: Return $v$.
3. If $l(v)$ and $r(v)$ are red: change color of $v$, $l(v)$ and $r(v)$.
4. If $x < k(v)$: $l(v) \leftarrow$ Insert($l(v)$, $x$).
5. If $x > k(v)$: $r(v) \leftarrow$ Insert($r(v)$, $x$).
6. If $l(v)$ is black and $r(v)$ red: rotate edge $(v, r(v))$ and put to $v$ original $r(v)$.

# INSERT to an LLRB-tree

Lets see how insertion to $(2,4)$-tree with preventive splitting translates to RB-tree.

**Insert($v$,$x$)**

1. If $v = \emptyset$: return newly created red vertex with key $x$.
2. If $x = k(v)$: Return $v$.
3. If $l(v)$ and $r(v)$ are red: change color of $v$, $l(v)$ and $r(v)$.
4. If $x < k(v)$: $l(v) \leftarrow$ Insert($l(v)$, $x$).
5. If $x > k(v)$: $r(v) \leftarrow$ Insert($r(v)$, $x$).
6. If $l(v)$ is black and $r(v)$ red: rotate edge $(v, r(v))$ and put to $v$ original $r(v)$.
7. If $l(v)$ and $l(l(v))$ are red: rotate edge $(v, l(v))$ and put to $v$ original $l(v)$.

# INSERT to an LLRB-tree

Lets see how insertion to $(2, 4)$-tree with preventive splitting translates to RB-tree.

**Insert($v$,$x$)**

1. If $v = \emptyset$: return newly created red vertex with key $x$.
2. If $x = k(v)$: Return $v$.
3. If $l(v)$ and $r(v)$ are red: change color of $v$, $l(v)$ and $r(v)$.
4. If $x < k(v)$: $l(v) \leftarrow$ Insert($l(v)$, $x$).
5. If $x > k(v)$: $r(v) \leftarrow$ Insert($r(v)$, $x$).
6. If $l(v)$ is black and $r(v)$ red: rotate edge $(v, r(v))$ and put to $v$ original $r(v)$.
7. If $l(v)$ and $l(l(v))$ are red: rotate edge $(v, l(v))$ and put to $v$ original $l(v)$.
8. Return $v$.

# INSERT to an LLRB-tree

Lets see how insertion to $(2, 4)$-tree with preventive splitting translates to RB-tree.

**Insert($v$,$x$)**

1. If $v = \emptyset$: return newly created red vertex with key $x$.
2. If $x = k(v)$: Return $v$.
3. If $l(v)$ and $r(v)$ are red: change color of $v$, $l(v)$ and $r(v)$.
4. If $x < k(v)$: $l(v) \leftarrow$ Insert($l(v)$, $x$).
5. If $x > k(v)$: $r(v) \leftarrow$ Insert($r(v)$, $x$).
6. If $l(v)$ is black and $r(v)$ red: rotate edge $(v, r(v))$ and put to $v$ original $r(v)$.
7. If $l(v)$ and $l(l(v))$ are red: rotate edge $(v, l(v))$ and put to $v$ original $l(v)$.
8. Return $v$.

Exchanging steps 3 and 7 leads to representation of $(2, 3)$-trees.

# INSERT to an LLRB-tree

Lets see how insertion to $(2, 4)$-tree with preventive splitting translates to RB-tree.

**Insert($v$,$x$)**

1. If $v = \emptyset$: return newly created red vertex with key $x$.
2. If $x = k(v)$: Return $v$.
3. If $l(v)$ and $r(v)$ are red: change color of $v$, $l(v)$ and $r(v)$.
4. If $x < k(v)$: $l(v) \leftarrow$ Insert($l(v)$, $x$).
5. If $x > k(v)$: $r(v) \leftarrow$ Insert($r(v)$, $x$).
6. If $l(v)$ is black and $r(v)$ red: rotate edge $(v, r(v))$ and put to $v$ original $r(v)$.
7. If $l(v)$ and $l(l(v))$ are red: rotate edge $(v, l(v))$ and put to $v$ original $l(v)$.
8. Return $v$.

Exchanging steps 3 and 7 leads to representation of $(2, 3)$-trees.
Fact: DELETE can also be implemented in $\Theta(\log n)$ time.

# INSERT to an LLRB-tree

Lets see how insertion to $(2, 4)$-tree with preventive splitting translates to RB-tree.

**Insert($v$,$x$)**

1. If $v = \emptyset$: return newly created red vertex with key $x$.
2. If $x = k(v)$: Return $v$.
3. If $l(v)$ and $r(v)$ are red: change color of $v$, $l(v)$ and $r(v)$.
4. If $x < k(v)$: $l(v) \leftarrow$ Insert($l(v)$, $x$).
5. If $x > k(v)$: $r(v) \leftarrow$ Insert($r(v)$, $x$).
6. If $l(v)$ is black and $r(v)$ red: rotate edge $(v, r(v))$ and put to $v$ original $r(v)$.
7. If $l(v)$ and $l(l(v))$ are red: rotate edge $(v, l(v))$ and put to $v$ original $l(v)$.
8. Return $v$.

Exchanging steps 3 and 7 leads to representation of $(2, 3)$-trees.
Fact: DELETE can also be implemented in $\Theta(\log n)$ time.

**Theorem**

*Operations INSERT, DELETE, FIND, MIN, MAX, SUCC and PRED on LLRB-tree runs in $\Theta(\log n)$ time.*


And Now For Something
Completely Different

# Tries

Let $\Sigma$ be a fixed alphabet. Let $S \subseteq \Sigma^*$ be a set of words over alphabet $\Sigma$.

**Definition (Trie: middle of retrieval, invented by René de la Briandais in 1959; named by Edward Frenklin)**

Trie for some set of words $S$ is a rooted tree where

1. vertices are all prefixes of words $W \in X$, and
2. $W'$ is a son of word $W$ if $W'$ is created from $W$ by extending it by one letter.

# Tries

Let $\Sigma$ be a fixed alphabet. Let $S \subseteq \Sigma^*$ be a set of words over alphabet $\Sigma$.

**Definition (Trie: middle of retrieval, invented by René de la Briandais in 1959; named by Edward Frenklin)**

Trie for some set of words $S$ is a rooted tree where

1. vertices are all prefixes of words $W \in X$, and
2. $W'$ is a son of word $W$ if $W'$ is created from $W$ by extending it by one letter.

**Theorem**

*FIND, INSERT and DELETE for word $X$ can all be implemented in $O(|X|)$.*

# Tries

Let $\Sigma$ be a fixed alphabet. Let $S \subseteq \Sigma^*$ be a set of words over alphabet $\Sigma$.

**Definition (Trie: middle of retrieval, invented by René de la Briandais in 1959; named by Edward Frenklin)**

Trie for some set of words $S$ is a rooted tree where

1. vertices are all prefixes of words $W \in X$, and
2. $W'$ is a son of word $W$ if $W'$ is created from $W$ by extending it by one letter.

**Theorem**

*FIND, INSERT and DELETE for word $X$ can all be implemented in $O(|X|)$.*

To store sets of integers one can see integers as words in some fixed base. Result is known as a radix tree.

# Amortised complexity

Insertion to a (dynamically allocated) growing array.

**Insert($(A, s, n)$,$x$) insert element $x$ to array $A$ of size $s$ containing $n$ elements**

1. if $n = s$:
2. Allocate array $A'$ of size $2s$.
3. For $i = 0, 1, \dots, n-1$: $A'[i] \leftarrow A[i]$.
4. Free $A$.
5. $A \leftarrow A'$, $s \leftarrow 2s$
6. $A[n] \leftarrow x$, $n \leftarrow n + 1$
7. Return $(A, s, n)$.

Worst case complexity of INSERT is $O(n)$.

# Amortised complexity

Insertion to a (dynamically allocated) growing array.

**Insert($(A, s, n)$,$x$) insert element $x$ to array $A$ of size $s$ containing $n$ elements**

1. if $n = s$:
2. Allocate array $A'$ of size $2s$.
3. For $i = 0, 1, \ldots, n-1$: $A'[i] \leftarrow A[i]$.
4. Free $A$.
5. $A \leftarrow A'$, $s \leftarrow 2s$
6. $A[n] \leftarrow x$, $n \leftarrow n + 1$
7. Return $(A, s, n)$.

Worst case complexity of INSERT is $O(n)$.

**Theorem**

*Performing $n$ operations* INSERT *starting from the empty array will run in time $\Theta(n)$.*

**Proof.**

To insert $2^i$ elements one needs $2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^{i-1} = 2^i - 1$ copy operations. □

# Hash functions

Hash function is a function $h$ from universe $\mathcal{U}$ to set $\mathcal{P} = \{0, 1, \ldots, p-1\}$ (of hashes).

**Hash table with separate chaining for set $S \subseteq U$ with hash function $h\colon \mathcal{U} \to \mathcal{P}$.**

Hash table is an array $H$ of linked lists indexed by $\mathcal{P}$. List $H[i]$ contains all elements $e$ of set $S$ such that $h(e) = i$.

# Hash functions

Hash function is a function $h$ from universe $\mathcal{U}$ to set $\mathcal{P} = \{0, 1, \dots, p - 1\}$ (of hashes).

**Hash table with separate chaining for set $S \subseteq U$ with hash function $h\colon \mathcal{U} \to \mathcal{P}$.**

Hash table is an array $H$ of linked lists indexed by $\mathcal{P}$. List $H[i]$ contains all elements $e$ of set $S$ such that $h(e) = i$.

**Assumptions**

1. $h(x)$ can be computed in $O(1)$.
2. $h(x)$ “behaves randomly”.

# Hash functions

Hash function is a function $h$ from universe $\mathcal{U}$ to set $\mathcal{P} = \{0, 1, \ldots, p-1\}$ (of hashes).

**Hash table with separate chaining for set $S \subseteq U$ with hash function $h\colon \mathcal{U} \to \mathcal{P}$.**

Hash table is an array $H$ of linked lists indexed by $\mathcal{P}$. List $H[i]$ contains all elements $e$ of set $S$ such that $h(e) = i$.

**Assumptions**

1. $h(x)$ can be computed in $O(1)$.
2. $h(x)$ “behaves randomly”.

**Observation**

Every entry of the hash table will contain approximately $\frac{|S|}{p}$ elements.

# Hash functions

Hash function is a function $h$ from universe $\mathcal{U}$ to set $\mathcal{P} = \{0, 1, \ldots, p-1\}$ (of hashes).

**Hash table with separate chaining for set $S \subseteq U$ with hash function $h\colon \mathcal{U} \to \mathcal{P}$.**

Hash table is an array $H$ of linked lists indexed by $\mathcal{P}$. List $H[i]$ contains all elements $e$ of set $S$ such that $h(e) = i$.

**Assumptions**

1. $h(x)$ can be computed in $O(1)$.
2. $h(x)$ “behaves randomly”.

**Observation**

Every entry of the hash table will contain approximately $\frac{|S|}{p}$ elements.

**Corollary**

Operations FIND, INSERT and DELETE will run in $O(|S|)$ however expected (average) runtime is only $O(\frac{|S|}{p})$.

# Hash functions

Hash function is a function $h$ from universe $\mathcal{U}$ to set $\mathcal{P} = \{0, 1, \ldots, p-1\}$ (of hashes).

**Hash table with separate chaining for set $S \subseteq U$ with hash function $h \colon \mathcal{U} \to \mathcal{P}$.**

Hash table is an array $H$ of linked lists indexed by $\mathcal{P}$. List $H[i]$ contains all elements $e$ of set $S$ such that $h(e) = i$.

**Assumptions**

1. $h(x)$ can be computed in $O(1)$.
2. $h(x)$ “behaves randomly”.

**Observation**

Every entry of the hash table will contain approximately $\frac{|S|}{p}$ elements.

**Corollary**

Operations Find, Insert and Delete will run in $O(|S|)$ however expected (average) runtime is only $O(\frac{|S|}{p})$.

**Corollary**

Putting $p \sim |S|$ we get Find, Insert and Delete is running on average approximately in $O(1)$.

# Hash functions

**Example (Integers: $h\colon \mathbb{N} \to \{0, 1, \ldots, p-1\}$)**

$$h(x) = ax \mod p$$

where $a, p$ are prime numbers.

# Hash functions

## Example (Integers: $h \colon \mathbb{N} \to \{0, 1, \ldots, p-1\}$)

$$h(x) = ax \mod p$$

where $a, p$ are prime numbers.

## Example (Strings: $h \colon \mathbb{N}^* \to \{0, 1, \ldots, p-1\}$)

$$h(x) = \left( \sum_{i=1}^{|x|} x_i a^{|x|-i} \mod p \right).$$

# Hash functions

**Example (Integers: $h\colon \mathbb{N} \to \{0, 1, \dots, p-1\}$)**

$$h(x) = ax \mod p$$

where $a, p$ are prime numbers.

**Example (Strings: $h\colon \mathbb{N}^* \to \{0, 1, \dots, p-1\}$)**

$$h(x) = \left( \sum_{i=1}^{|x|} x_i a^{|x|-i} \mod p \right).$$

Can be effectively computed as (Horner's method):

$$\begin{aligned}
h_1 &= x_1 \\
h_2 &= (h_1 a + x_2) \mod p \\
h_3 &= (h_2 a + x_3) \mod p \\
\dots & \quad \dots \\
h_{|x|} &= (h_{|x|-1} a + x_{|x|}) \mod p
\end{aligned}$$

# Open addressing

An alternative way of solving collisions is to use hash function $h(x, i)$ such that for every $x \in \mathcal{U}$ sequence $h(x, 0), h(x, 1), \ldots, h(x, p-1)$ is a permutation of $(0, 1, \ldots, p-1)$.

**Insert(x)**

1. For $i = 0, \ldots, p-1$:
2. $\quad j \leftarrow h(x, i)$
3. $\quad$ If $H[j] = \emptyset$: put $H[j] \leftarrow x$ and return.
4. Report that table is full.

**Find(x)**

1. For $i = 0, \ldots, p-1$:
2. $\quad j \leftarrow h(x, i)$
3. $\quad$ If $H[j] = x$: return $j$.
4. Return $\emptyset$.

# Open addressing

An alternative way of solving collisions is to use hash function $h(x, i)$ such that for every $x \in \mathcal{U}$ sequence $h(x, 0), h(x, 1), \ldots, h(x, p-1)$ is a permutation of $(0, 1, \ldots, p-1)$.

**Insert(x)**

1. For $i = 0, \ldots, p-1$:
2. $\quad j \leftarrow h(x, i)$
3. $\quad$ If $H[j] = \emptyset$: put $H[j] \leftarrow x$ and return.
4. Report that table is full.

**Find(x)**

1. For $i = 0, \ldots, p-1$:
2. $\quad j \leftarrow h(x, i)$
3. $\quad$ If $H[j] = x$: return $j$.
4. Return $\emptyset$.

We can not remove values from the table, just mark them as removed.

# Open addressing

An alternative way of solving collisions is to use hash function $h(x, i)$ such that for every $x \in \mathcal{U}$ sequence $h(x, 0), h(x, 1), \dots, h(x, p-1)$ is a permutation of $(0, 1, \dots, p-1)$.

**Insert(x)**

1. For $i = 0, \dots, p-1$:
2. $j \leftarrow h(x, i)$
3. If $H[j] = \emptyset$: put $H[j] \leftarrow x$ and return.
4. Report that table is full.

**Find(x)**

1. For $i = 0, \dots, p-1$:
2. $j \leftarrow h(x, i)$
3. If $H[j] = x$: return $j$.
4. Return $\emptyset$.

**Theorem**

*Assuming that the hash function is giving random permutations, the average number of visited entries during unsuccessful find is* $\frac{1}{(1-\alpha)}$ *for* $\alpha = \frac{n}{m}$.

# Open addressing

An alternative way of solving collisions is to use hash function $h(x, i)$ such that for every $x \in \mathcal{U}$ sequence $h(x, 0), h(x, 1), \dots, h(x, p-1)$ is a permutation of $(0, 1, \dots, p-1)$.

**Insert(x)**

1. For $i = 0, \dots, p-1$:
2. $\quad j \leftarrow h(x, i)$
3. $\quad$ If $H[j] = \emptyset$: put $H[j] \leftarrow x$ and return.
4. Report that table is full.

**Find(x)**

1. For $i = 0, \dots, p-1$:
2. $\quad j \leftarrow h(x, i)$
3. $\quad$ If $H[j] = x$: return $j$.
4. Return $\emptyset$.

**Theorem**

*Assuming that the hash function is giving random permutations, the average number of visited entries during unsuccessful find is* $\frac{1}{(1-\alpha)}$ *for* $\alpha = \frac{n}{m}$.

**Proof.**

Let $p_i$ be probability that we will search at least $i$ entries.

# Open addressing

An alternative way of solving collisions is to use hash function $h(x, i)$ such that for every $x \in \mathcal{U}$ sequence $h(x, 0), h(x, 1), \ldots, h(x, p-1)$ is a permutation of $(0, 1, \ldots, p-1)$.

**Insert(x)**

1. For $i = 0, \ldots, p-1$:
2. $\quad j \leftarrow h(x, i)$
3. $\quad$ If $H[j] = \emptyset$: put $H[j] \leftarrow x$ and return.
4. Report that table is full.

**Find(x)**

1. For $i = 0, \ldots, p-1$:
2. $\quad j \leftarrow h(x, i)$
3. $\quad$ If $H[j] = x$: return $j$.
4. Return $\emptyset$.

**Theorem**

*Assuming that the hash function is giving random permutations, the average number of visited entries during unsuccessful find is* $\frac{1}{(1-\alpha)}$ *for* $\alpha = \frac{n}{m}$.

**Proof.**

Let $p_i$ be probability that we will search at least $i$ entries. $p_1 = 1$,

# Open addressing

An alternative way of solving collisions is to use hash function $h(x, i)$ such that for every $x \in \mathcal{U}$ sequence $h(x, 0), h(x, 1), \ldots, h(x, p-1)$ is a permutation of $(0, 1, \ldots, p-1)$.

**Insert(x)**

1. For $i = 0, \ldots, p-1$:
2. $j \leftarrow h(x, i)$
3. If $H[j] = \emptyset$: put $H[j] \leftarrow x$ and return.
4. Report that table is full.

**Find(x)**

1. For $i = 0, \ldots, p-1$:
2. $j \leftarrow h(x, i)$
3. If $H[j] = x$: return $j$.
4. Return $\emptyset$.

**Theorem**

*Assuming that the hash function is giving random permutations, the average number of visited entries during unsuccessful find is* $\frac{1}{(1-\alpha)}$ *for* $\alpha = \frac{n}{m}$.

**Proof.**

Let $p_i$ be probability that we will search at least $i$ entries. $p_1 = 1, p_2 = \frac{n}{m} = \alpha,$

# Open addressing

An alternative way of solving collisions is to use hash function $h(x, i)$ such that for every $x \in \mathcal{U}$ sequence $h(x, 0), h(x, 1), \ldots, h(x, p-1)$ is a permutation of $(0, 1, \ldots, p-1)$.

**Insert(x)**

1. For $i = 0, \ldots, p-1$:
2. $\quad j \leftarrow h(x, i)$
3. $\quad$ If $H[j] = \emptyset$: put $H[j] \leftarrow x$ and return.
4. Report that table is full.

**Find(x)**

1. For $i = 0, \ldots, p-1$:
2. $\quad j \leftarrow h(x, i)$
3. $\quad$ If $H[j] = x$: return $j$.
4. Return $\emptyset$.

**Theorem**

*Assuming that the hash function is giving random permutations, the average number of visited entries during unsuccessful find is* $\frac{1}{(1-\alpha)}$ *for* $\alpha = \frac{n}{m}$.

**Proof.**

Let $p_i$ be probability that we will search at least $i$ entries. $p_1 = 1, p_2 = \frac{n}{m} = \alpha, p_3 = \alpha\frac{n-1}{m-1} \leq \alpha^2, \ldots$.

# Open addressing

An alternative way of solving collisions is to use hash function $h(x, i)$ such that for every $x \in \mathcal{U}$ sequence $h(x, 0), h(x, 1), \ldots, h(x, p-1)$ is a permutation of $(0, 1, \ldots, p-1)$.

**Insert(x)**

1. For $i = 0, \ldots, p-1$:
2. $\quad j \leftarrow h(x, i)$
3. $\quad$ If $H[j] = \emptyset$: put $H[j] \leftarrow x$ and return.
4. Report that table is full.

**Find(x)**

1. For $i = 0, \ldots, p-1$:
2. $\quad j \leftarrow h(x, i)$
3. $\quad$ If $H[j] = x$: return $j$.
4. Return $\emptyset$.

**Theorem**

*Assuming that the hash function is giving random permutations, the average number of visited entries during unsuccessful find is* $\frac{1}{(1-\alpha)}$ *for* $\alpha = \frac{n}{m}$.

**Proof.**

Let $p_i$ be probability that we will search at least $i$ entries. $p_1 = 1, p_2 = \frac{n}{m} = \alpha, p_3 = \alpha \frac{n-1}{m-1} \leq \alpha^2, \ldots$.

$$S = \sum_{i \geq 1} i(p_i - p_{i+1})$$

# Open addressing

An alternative way of solving collisions is to use hash function $h(x, i)$ such that for every $x \in \mathcal{U}$ sequence $h(x, 0), h(x, 1), \ldots, h(x, p-1)$ is a permutation of $(0, 1, \ldots, p-1)$.

**Insert(x)**

1. For $i = 0, \ldots, p-1$:
2. $\quad j \leftarrow h(x, i)$
3. $\quad$ If $H[j] = \emptyset$: put $H[j] \leftarrow x$ and return.
4. Report that table is full.

**Find(x)**

1. For $i = 0, \ldots, p-1$:
2. $\quad j \leftarrow h(x, i)$
3. $\quad$ If $H[j] = x$: return $j$.
4. Return $\emptyset$.

**Theorem**

*Assuming that the hash function is giving random permutations, the average number of visited entries during unsuccessful find is* $\frac{1}{(1-\alpha)}$ *for* $\alpha = \frac{n}{m}$.

**Proof.**

Let $p_i$ be probability that we will search at least $i$ entries. $p_1 = 1, p_2 = \frac{n}{m} = \alpha, p_3 = \alpha\frac{n-1}{m-1} \leq \alpha^2, \ldots$.

$$S = \sum_{i \geq 1} i(p_i - p_{i+1}) = \sum_{i \geq 1} (i - (i-1))p_i$$

# Open addressing

An alternative way of solving collisions is to use hash function $h(x, i)$ such that for every $x \in \mathcal{U}$ sequence $h(x, 0), h(x, 1), \ldots, h(x, p-1)$ is a permutation of $(0, 1, \ldots, p-1)$.

**Insert(x)**

1. For $i = 0, \ldots, p-1$:
2. $\quad j \leftarrow h(x, i)$
3. $\quad$ If $H[j] = \emptyset$: put $H[j] \leftarrow x$ and return.
4. Report that table is full.

**Find(x)**

1. For $i = 0, \ldots, p-1$:
2. $\quad j \leftarrow h(x, i)$
3. $\quad$ If $H[j] = x$: return $j$.
4. Return $\emptyset$.

**Theorem**

*Assuming that the hash function is giving random permutations, the average number of visited entries during unsuccessful find is* $\frac{1}{(1-\alpha)}$ *for* $\alpha = \frac{n}{m}$.

**Proof.**

Let $p_i$ be probability that we will search at least $i$ entries. $p_1 = 1, p_2 = \frac{n}{m} = \alpha, p_3 = \alpha\frac{n-1}{m-1} \leq \alpha^2, \ldots$.

$$S = \sum_{i \geq 1} i(p_i - p_{i+1}) = \sum_{i \geq 1} (i - (i-1))p_i = \sum_{i \geq 1} p_i \leq \sum_{i \geq 1} \alpha^{i-1}$$

# Open addressing

An alternative way of solving collisions is to use hash function $h(x, i)$ such that for every $x \in \mathcal{U}$ sequence $h(x, 0), h(x, 1), \ldots, h(x, p-1)$ is a permutation of $(0, 1, \ldots, p-1)$.

**Insert(x)**

1. For $i = 0, \ldots, p-1$:
2. $\quad j \leftarrow h(x, i)$
3. $\quad$ If $H[j] = \emptyset$: put $H[j] \leftarrow x$ and return.
4. Report that table is full.

**Find(x)**

1. For $i = 0, \ldots, p-1$:
2. $\quad j \leftarrow h(x, i)$
3. $\quad$ If $H[j] = x$: return $j$.
4. Return $\emptyset$.

**Theorem**

*Assuming that the hash function is giving random permutations, the average number of visited entries during unsuccessful find is* $\frac{1}{(1-\alpha)}$ *for* $\alpha = \frac{n}{m}$.

**Proof.**

Let $p_i$ be probability that we will search at least $i$ entries. $p_1 = 1, p_2 = \frac{n}{m} = \alpha, p_3 = \alpha\frac{n-1}{m-1} \leq \alpha^2, \ldots$.

$$S = \sum_{i \geq 1} i(p_i - p_{i+1}) = \sum_{i \geq 1} (i - (i-1))p_i = \sum_{i \geq 1} p_i \leq \sum_{i \geq 1} \alpha^{i-1} = \sum_{i \geq 0} \alpha_i$$

# Open addressing

An alternative way of solving collisions is to use hash function $h(x, i)$ such that for every $x \in \mathcal{U}$ sequence $h(x, 0), h(x, 1), \dots, h(x, p-1)$ is a permutation of $(0, 1, \dots, p-1)$.

**Insert(x)**

1. For $i = 0, \dots, p-1$:
2. $\quad j \leftarrow h(x, i)$
3. $\quad$ If $H[j] = \emptyset$: put $H[j] \leftarrow x$ and return.
4. Report that table is full.

**Find(x)**

1. For $i = 0, \dots, p-1$:
2. $\quad j \leftarrow h(x, i)$
3. $\quad$ If $H[j] = x$: return $j$.
4. Return $\emptyset$.

**Theorem**

*Assuming that the hash function is giving random permutations, the average number of visited entries during unsuccessful find is* $\frac{1}{(1-\alpha)}$ *for* $\alpha = \frac{n}{m}$.

Linear addressing: $h(x, i) = f(x) + i \mod p$.

# Open addressing

An alternative way of solving collisions is to use hash function $h(x, i)$ such that for every $x \in \mathcal{U}$ sequence $h(x, 0), h(x, 1), \ldots, h(x, p-1)$ is a permutation of $(0, 1, \ldots, p-1)$.

**Insert(x)**

1. For $i = 0, \ldots, p-1$:
2. $\quad j \leftarrow h(x, i)$
3. $\quad$ If $H[j] = \emptyset$: put $H[j] \leftarrow x$ and return.
4. Report that table is full.

**Find(x)**

1. For $i = 0, \ldots, p-1$:
2. $\quad j \leftarrow h(x, i)$
3. $\quad$ If $H[j] = x$: return $j$.
4. Return $\emptyset$.

**Theorem**

*Assuming that the hash function is giving random permutations, the average number of visited entries during unsuccessful find is* $\frac{1}{(1-\alpha)}$ *for* $\alpha = \frac{n}{m}$.

Linear addressing: $h(x, i) = f(x) + i \mod p$.
Not a random permutation: expected number of visited entries increases to: $\frac{1}{(1-\alpha)^2}$.

# Open addressing

An alternative way of solving collisions is to use hash function $h(x, i)$ such that for every $x \in \mathcal{U}$ sequence $h(x, 0), h(x, 1), \ldots, h(x, p-1)$ is a permutation of $(0, 1, \ldots, p-1)$.

**Insert(x)**

1. For $i = 0, \ldots, p-1$:
2. $\quad j \leftarrow h(x, i)$
3. $\quad$ If $H[j] = \emptyset$: put $H[j] \leftarrow x$ and return.
4. Report that table is full.

**Find(x)**

1. For $i = 0, \ldots, p-1$:
2. $\quad j \leftarrow h(x, i)$
3. $\quad$ If $H[j] = x$: return $j$.
4. Return $\emptyset$.

**Theorem**

*Assuming that the hash function is giving random permutations, the average number of visited entries during unsuccessful find is $\frac{1}{(1-\alpha)}$ for $\alpha = \frac{n}{m}$.*

Linear addressing: $h(x, i) = f(x) + i \mod p$.
Not a random permutation: expected number of visited entries increases to: $\frac{1}{(1-\alpha)^2}$.
Double hashing: $h(x, i) = (f(x) + i(g(x) + 1)) \mod m$ for $f$ and $g$ being two different hash functions.

# Universal hashing

**Definition ($c$-universal system of hash functions)**

System $\mathcal{S}$ of hash functions from universe $\mathcal{U}$ to $\{0, 1, \ldots, p-1\}$ is $c$-universal for given $c \geq 1$ if for every $x, y \in \mathcal{U}$, $x \neq y$

$$\Pr_{h \in \mathcal{S}}[h(x) = h(y)] \leq \frac{c}{p}.$$

# Universal hashing

**Definition ($c$-universal system of hash functions)**

System $\mathcal{S}$ of hash functions from universe $\mathcal{U}$ to $\{0, 1, \ldots, p-1\}$ is $c$-universal for given $c \geq 1$ if for every $x, y \in \mathcal{U}$, $x \neq y$

$$\Pr_{h \in \mathcal{S}}[h(x) = h(y)] \leq \frac{c}{p}.$$

**Lemma**

*Let $\mathcal{S}$ be $c$-universal system of hash functions $\mathcal{U} \to \{0, 1, \ldots, p\}$. Let $x_1, x_2, \ldots, x_n, y$ be pairwise different elements of $\mathcal{U}$. Then*

$$\mathbb{E}_{h \in \mathcal{S}}[\#i \colon h(x_i) = h(y)] \leq \frac{cn}{p}.$$

# Universal hashing

**Definition ($c$-universal system of hash functions)**

System $\mathcal{S}$ of hash functions from universe $\mathcal{U}$ to $\{0, 1, \ldots, p-1\}$ is $c$-universal for given $c \geq 1$ if for every $x, y \in \mathcal{U}$, $x \neq y$

$$\Pr_{h \in \mathcal{S}}[h(x) = h(y)] \leq \frac{c}{p}.$$

**Lemma**

*Let $\mathcal{S}$ be $c$-universal system of hash functions $\mathcal{U} \to \{0, 1, \ldots, p\}$. Let $x_1, x_2, \ldots, x_n, y$ be pairwise different elements of $\mathcal{U}$. Then*

$$\mathbb{E}_{h \in \mathcal{S}}[\#i\colon h(x_i) = h(y)] \leq \frac{cn}{p}.$$

The lemma shows expected runtime of INSERT, FIND and DELETE with separate chaining.

# Universal hashing

**Definition ($c$-universal system of hash functions)**

System $\mathcal{S}$ of hash functions from universe $\mathcal{U}$ to $\{0, 1, \ldots, p-1\}$ is $c$-universal for given $c \geq 1$ if for every $x, y \in \mathcal{U}$, $x \neq y$

$$\Pr_{h \in \mathcal{S}}[h(x) = h(y)] \leq \frac{c}{p}.$$

**Lemma**

*Let $\mathcal{S}$ be $c$-universal system of hash functions $\mathcal{U} \to \{0, 1, \ldots, p\}$. Let $x_1, x_2, \ldots, x_n, y$ be pairwise different elements of $\mathcal{U}$. Then*

$$\mathbb{E}_{h \in \mathcal{S}}[\#i \colon h(x_i) = h(y)] \leq \frac{cn}{p}.$$

he lemma shows expected runtime of INSERT, FIND and DELETE with separate chaining.

**Proof.**

We define indicators $I_1, I_2, \ldots, I_n$:

$$I_i = \begin{cases} 0 & \text{if } h(x_i) \neq h(y) \\ 1 & \text{if } h(x_i) = h(y). \end{cases}$$

# Universal hashing

**Definition ($c$-universal system of hash functions)**

System $\mathcal{S}$ of hash functions from universe $\mathcal{U}$ to $\{0, 1, \ldots, p-1\}$ is $c$-universal for given $c \geq 1$ if for every $x, y \in \mathcal{U}$, $x \neq y$

$$\Pr_{h \in \mathcal{S}}[h(x) = h(y)] \leq \frac{c}{p}.$$

**Lemma**

*Let $\mathcal{S}$ be $c$-universal system of hash functions $\mathcal{U} \to \{0, 1, \ldots, p\}$. Let $x_1, x_2, \ldots, x_n, y$ be pairwise different elements of $\mathcal{U}$. Then*

$$\mathbb{E}_{h \in \mathcal{S}}[\# i \colon h(x_i) = h(y)] \leq \frac{cn}{p}.$$

he lemma shows expected runtime of INSERT, FIND and DELETE with separate chaining.

**Proof.**

We define indicators $I_1, I_2, \ldots, I_n$:

$$I_i = \begin{cases} 0 & \text{if } h(x_i) \neq h(y) \\ 1 & \text{if } h(x_i) = h(y). \end{cases}$$

$\mathbb{E}[I_i] = Pr[I_i = 1] \leq \frac{c}{p}$. (by universality)

# Universal hashing

## Definition ($c$-universal system of hash functions)

System $\mathcal{S}$ of hash functions from universe $\mathcal{U}$ to $\{0, 1, \ldots, p-1\}$ is $c$-universal for given $c \geq 1$ if for every $x, y \in \mathcal{U}$, $x \neq y$

$$\Pr_{h \in \mathcal{S}}[h(x) = h(y)] \leq \frac{c}{p}.$$

## Lemma

*Let $\mathcal{S}$ be $c$-universal system of hash functions $\mathcal{U} \to \{0, 1, \ldots, p\}$. Let $x_1, x_2, \ldots, x_n, y$ be pairwise different elements of $\mathcal{U}$. Then*

$$\mathbb{E}_{h \in \mathcal{S}}[\#i\colon h(x_i) = h(y)] \leq \frac{cn}{p}.$$

he lemma shows expected runtime of INSERT, FIND and DELETE with separate chaining.

## Proof.

We define indicators $I_1, I_2, \ldots, I_n$:

$$I_i = \begin{cases} 0 & \text{if } h(x_i) \neq h(y) \\ 1 & \text{if } h(x_i) = h(y). \end{cases}$$

$\mathbb{E}[I_i] = Pr[I_i = 1] \leq \frac{c}{p}$. (by universality)

$\mathbb{E}_{h \in \mathcal{S}}[\#i\colon h(x_i) = h(y)] = \sum_{1 \leq i \leq n} \mathbb{E}[I_i]$.

# Universal hashing

**Definition ($c$-universal system of hash functions)**

System $\mathcal{S}$ of hash functions from universe $\mathcal{U}$ to $\{0, 1, \ldots, p-1\}$ is $c$-universal for given $c \geq 1$ if for every $x, y \in \mathcal{U}$, $x \neq y$

$$\Pr_{h \in \mathcal{S}}[h(x) = h(y)] \leq \frac{c}{p}.$$

**Lemma**

*Let $\mathcal{S}$ be $c$-universal system of hash functions $\mathcal{U} \to \{0, 1, \ldots, p\}$. Let $x_1, x_2, \ldots, x_n, y$ be pairwise different elements of $\mathcal{U}$. Then*

$$\mathbb{E}_{h \in \mathcal{S}}[\#i\colon h(x_i) = h(y)] \leq \frac{cn}{p}.$$

he lemma shows expected runtime of INSERT, FIND and DELETE with separate chaining.

**Proof.**

We define indicators $I_1, I_2, \ldots, I_n$:

$$I_i = \begin{cases} 0 & \text{if } h(x_i) \neq h(y) \\ 1 & \text{if } h(x_i) = h(y). \end{cases}$$

$\mathbb{E}[I_i] = Pr[I_i = 1] \leq \frac{c}{p}$. (by universality)

$\mathbb{E}_{h \in \mathcal{S}}[\#i\colon h(x_i) = h(y)] = \sum_{1 \leq i \leq n} \mathbb{E}[I_i]$.

$\mathbb{E}_{h \in \mathcal{S}}[\#i\colon h(x_i) = h(y)] = \sum_{1 \leq i \leq n} Pr[I_i = 1] \leq \frac{cn}{p}$.

□

# 1-universal system

**System of functions $\mathcal{S} \colon \mathbb{Z}_p^d \to \{0, 1, \ldots, p-1\}$**

Let $p$ be a prime number, $\mathcal{P} = \mathbb{Z}_p$ (ring modulo $p$), $\mathcal{U} = \mathbb{Z}_p^d$ (vectors of length $d$ in $\mathbb{Z}_p$).

$$\mathcal{S} = \{h_{\vec{a}} \colon \vec{a} \in \mathbb{Z}_p^d, \vec{a} \neq 0\} \text{ where } h_{\vec{a}}(x) = \vec{a}\vec{x} = \sum_{i=1}^{d} a_i x_i \mod p. \text{ ($a_i x_i$ is the scalar product).}$$

# 1-universal system

**System of functions $\mathcal{S}\colon \mathbb{Z}_p^d \to \{0, 1, \ldots, p-1\}$**

Let $p$ be a prime number, $\mathcal{P} = \mathbb{Z}_p$ (ring modulo $p$), $\mathcal{U} = \mathbb{Z}_p^d$ (vectors of length $d$ in $\mathbb{Z}_p$).

$$\mathcal{S} = \{h_{\vec{a}}\colon \vec{a} \in \mathbb{Z}_p^d, \vec{a} \neq 0\} \text{ where } h_{\vec{a}}(x) = \vec{a}\vec{x} = \sum_{i=1}^{d} a_i x_i \mod p.$$

($a_i x_i$ is the scalar product).

**Theorem**

*$\mathcal{S}$ is 1-universal.*

# 1-universal system

## System of functions $\mathcal{S}\colon \mathbb{Z}_p^d \to \{0, 1, \ldots, p-1\}$

Let $p$ be a prime number, $\mathcal{P} = \mathbb{Z}_p$ (ring modulo $p$), $\mathcal{U} = \mathbb{Z}_p^d$ (vectors of length $d$ in $\mathbb{Z}_p$).

$$\mathcal{S} = \{h_{\vec{a}}\colon \vec{a} \in \mathbb{Z}_p^d, \vec{a} \neq 0\} \text{ where } h_{\vec{a}}(x) = \vec{a}\vec{x} = \sum_{i=1}^{d} a_i x_i \mod p. \ (a_i x_i \text{ is the scalar product}).$$

## Theorem

*$\mathcal{S}$ is 1-universal.*

## Proof.

Set $\vec{x} \neq \vec{y} \in \mathbb{Z}_p^d$. WLOG $x_d \neq y_d$. What is $\Pr_{\vec{a} \in \mathbb{Z}_p^d}[\vec{a}\vec{x} = \vec{a}\vec{y} \mod p]$?

# 1-universal system

## System of functions $\mathcal{S}\colon \mathbb{Z}_p^d \to \{0, 1, \dots, p-1\}$

Let $p$ be a prime number, $\mathcal{P} = \mathbb{Z}_p$ (ring modulo $p$), $\mathcal{U} = \mathbb{Z}_p^d$ (vectors of length $d$ in $\mathbb{Z}_p$).

$$\mathcal{S} = \{h_{\vec{a}}\colon \vec{a} \in \mathbb{Z}_p^d, \vec{a} \neq 0\} \text{ where } h_{\vec{a}}(x) = \vec{a}\vec{x} = \sum_{i=1}^{d} a_i x_i \mod p. \ (a_i x_i \text{ is the scalar product}).$$

## Theorem

*$\mathcal{S}$ is 1-universal.*

## Proof.

Set $\vec{x} \neq \vec{y} \in \mathbb{Z}_p^d$. WLOG $x_d \neq y_d$. What is $\Pr_{\vec{a} \in \mathbb{Z}_p^d}[\vec{a}\vec{x} = \vec{a}\vec{y} \mod p]$?

Put $\vec{z} = \vec{x} - \vec{y}$. ($\vec{a}\vec{x} \equiv \vec{a}\vec{y}$ means $\vec{a}\vec{x} = \vec{a}\vec{y} \mod p$)

$$\Pr_{\vec{a} \in \mathbb{Z}_p^d}[\vec{a}\vec{x} \equiv \vec{a}\vec{y}] = \Pr_{\vec{a} \in \mathbb{Z}_p^d}\left[\sum_{i=1}^{d} a_i z_i \equiv 0\right] = \Pr_{\vec{a} \in \mathbb{Z}_p^d}\left[\sum_{i=1}^{d-1} a_i z_i + a_d z_d \equiv 0\right].$$

# 1-universal system

## System of functions $\mathcal{S} \colon \mathbb{Z}_p^d \to \{0, 1, \dots, p-1\}$

Let $p$ be a prime number, $\mathcal{P} = \mathbb{Z}_p$ (ring modulo $p$), $\mathcal{U} = \mathbb{Z}_p^d$ (vectors of length $d$ in $\mathbb{Z}_p$).

$$\mathcal{S} = \{h_{\vec{a}} \colon \vec{a} \in \mathbb{Z}_p^d, \vec{a} \neq 0\} \text{ where } h_{\vec{a}}(x) = \vec{a}\vec{x} = \sum_{i=1}^{d} a_i x_i \mod p. \text{ } (a_i x_i \text{ is the scalar product}).$$

## Theorem

*$\mathcal{S}$ is 1-universal.*

## Proof.

Set $\vec{x} \neq \vec{y} \in \mathbb{Z}_p^d$. WLOG $x_d \neq y_d$. What is $\Pr_{\vec{a} \in \mathbb{Z}_p^d}[\vec{a}\vec{x} = \vec{a}\vec{y} \mod p]$?

Put $\vec{z} = \vec{x} - \vec{y}$. ($\vec{a}\vec{x} \equiv \vec{a}\vec{y}$ means $\vec{a}\vec{x} = \vec{a}\vec{y} \mod p$)

$$\Pr_{\vec{a} \in \mathbb{Z}_p^d}\left[\vec{a}\vec{x} \equiv \vec{a}\vec{y}\right] = \Pr_{\vec{a} \in \mathbb{Z}_p^d}\left[\sum_{i=1}^{d} a_i z_i \equiv 0\right] = \Pr_{\vec{a} \in \mathbb{Z}_p^d}\left[\sum_{i=1}^{d-1} a_i z_i + a_d z_d \equiv 0\right].$$

$\sum_{i=1}^{d-1} a_i z_i + a_d z_d \equiv 0$ happens only if $\sum_{i=1}^{d-1} a_i z_i \equiv -a_d z_d$. This has probability $\frac{1}{p}$. $\square$