Algorithms (COMP3600/6466)
Data Structures: Hash Tables

Pascal Bercher

(working in the Intelligent Systems Cluster)

School of Computing
The Australian National University

Tuesday & Wednesday, 26. & 27.9.2023

Introduction

Motivation

- The last two weeks we stored data by using unique keys.
- Our aim was to get "good" runtime for the most common operations like:
  - Search
  - Insert
  - Delete
  - (Maybe others, like Min and Max)
- We had:
  - All are in $O(log(n))$ for balanced trees.
  - For heaps, Min (or Max, depending on whether we use a Min- or a Max-heap) is $O(1)$, but the rest was $O(h)$.
- But ... Could we do even better? Could we achieve constant runtime $O(1)$ for the first three operations? If so, how?

Motivation, cont'd

- To achieve constant runtime for *Search*, *Insert*, and *Delete* we could just use an array!
- Then, we could to the following:
  - Search (data for key) k $\rightarrow$ check whether index k is *true*
  - Insert (data for key) k $\rightarrow$ store *true* (or the data) at index k
  - Delete (data for key) k $\rightarrow$ store *false* (or delete data) at index k
- Are there any problems with this approach?
  - Normally the size of an array equals the number of entries.
    - ▶ Here, it needs to equal the size of the biggest key!(!)
    - → E.g., consider you have 100 entries, but a max. key of 10.000 – that'd be an extremely (memory-)inefficient data structure.
    - ▶ Also what to do if you don't even know the maximal key?
  - What if we want to store multiple (*key*,*data*) entries? (Note that we ignored this issue for the previous chapters!)

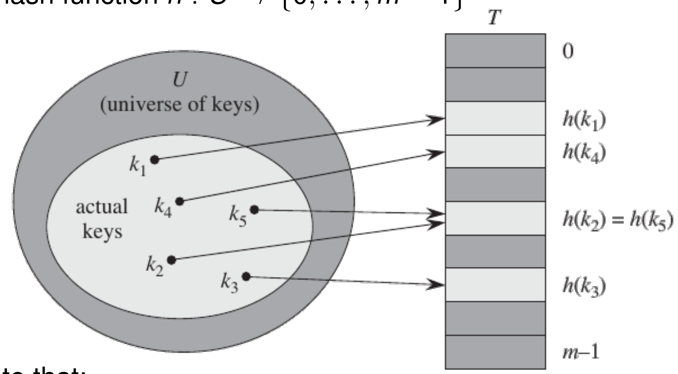
→ ***Hash tables* can be thought of as generalized arrays**:

Instead of fixed keys as array index, we compute a "suitable index", generated from the key using a *hash function*.

Basics

Graphical Illustration and Terminology

- Universe $U = \{k_1, \ldots, k_u\}$ (all possible keys), $|U| = u$.
- Hash table T, an array of size m, with n currently stored keys.
- A hash function $h : U \to \{0, \ldots, m-1\}$



Note that:

- The array size $|T|$ is $m \ll \max_{k \in U} k$.
- Collision may happen! (Dealing with this is a major part!)

Collisions

Recall the previous illustration with k_2 and k_5: h mapped both of them to the same index $i = h(k_2) = h(k_5)$ – that's a collision!

What choices do we have to deal with such a case?

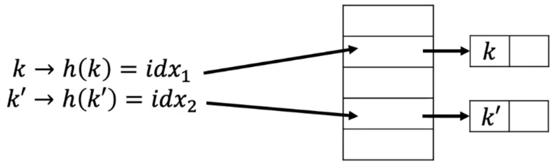
1. Change how data is stored:
   - Hashing with *chaining*: Use a linked list to store all keys that are mapped to index i (at this position).
   - *Open addressing*: Choose the next free position.
2. Minimize likelihood of collisions via:
   - Simple Uniform Hashing: Works well if equal and independent distribution of keys is given.
   - Universal Hashing: Select hash function at random.
   - Perfect Hashing: We *ensure* that there are no collisions.

(We won't cover these options in the same order given here.)

Common Hash Functions

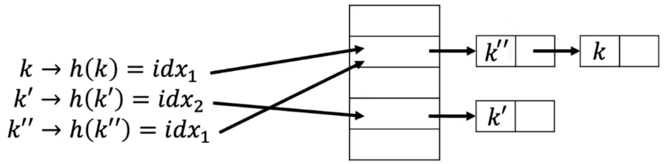
Hashing with Chaining: Insertion, Deletion, Search

- Each slot in the hash table contains a linked list.
- For insertion, if a key is hashed into a non-empty slot, place the new pair (key plus satellite data) at the front of the respective list:



Hashing with Chaining: Insertion, Deletion, Search

- Each slot in the hash table contains a linked list.
- For insertion, if a key is hashed into a non-empty slot, place the new pair (key plus satellite data) at the front of the respective list:



- For search and deletion, iterate through the respective list. In case of deletion, cut out the respective element.
- Runtimes are linear, but not $O(n)$, where n is the number of keys in the table (here: 3), but $O(n_i)$, where n_i is the number of keys mapped to index i (here: 2 for $i = 1$). The actual runtime then depends on the hash function's distribution of hash values.

Hashing with Chaining: Simple Uniform Hashing

- A *uniform hash function* is a function where any given key is equally likely to map onto any of the m slots, independently of where any other key has been mapped to. Thus, it is a function h such that:

$$P(h(k) = v) = \frac{1}{m} \text{ for all } k \in U \text{ and } v \in \{0, \ldots, m-1\}$$

  (We also say that h "hashes" to a certain position.)
  Note how uniformity depends on the Universe!
- Given a uniform hash function and assuming the input keys are uniformly distributed and independent, we get the following collision probability:

$$P(h(k_1) = h(k_2)) = \frac{1}{m}, k_1 \neq k_2$$

  (Is this right? Shouldn't it be $\frac{1}{m^2}$?)

Hashing with Chaining: Time Complexity

Assume we use Uniform Hashing (with uniformly distributed and independent keys) with Chaining.
Do we have constant time access?

- Time complexity of searching a key (regardless of success) on average $\Theta(1 + \alpha)$, where $\alpha = \frac{n}{m}$
  - Recall: n are number of keys stored so far, m the size of the array.
  - The expression α is usually called the *load factor*. If we keep it constant, average complexity will be constant too!
  - We might have to resize the hash table to achieve this. (Not covered in this lecture.)
- How to prove that?
  - Case 1: When the key is not found (easy!)
  - Case 2: When the key is found (more challenging)

Hashing with Chaining: Time Complexity Proof (Key not Found)

Steps for searching: **case: Key *not* found!**

- Compute $h(k)$, which runs in $O(1)$.
- Average number of checks to know that the key isn't contained:
  - Isn't that just *one*? The list is empty! (Case "key not found"!)
  - The list may not be empty since *other* keys that map to $h(k)$ might be stored in that list!
  - Thus we'll have to run to the end of said list.
  - Thus we get the average linked list length as runtime! This is exactly our load factor $\alpha = \frac{n}{m}$ due to all our assumptions! (Hash function is uniform, and keys are uniformly distributed and independent.)
- Thus, average is $\Theta(1+\alpha)$, worst case is $O(n)$, since m is constant.

Hashing with Chaining: Time Complexity Proof (Key Found)

Steps for searching: **case: Key found!**

- Compute $h(k)$, which runs in $O(1)$.
- Average number of checks until k is found:
  - Intuition: Expected search runtime in average-long list.
  - Formally, required: Average number of elements that lie before k.
  - Let the random variable $X_{i,j} = \begin{cases} 1 & \text{if } h(k_i) = h(k_j) \\ 0 & \text{otherwise} \end{cases}$, where k_j denotes the jth inserted key.
  - We know that $P_{i \neq j}(X_{i,j} = 1) = E[X_{i,j}] = \frac{1}{m}$

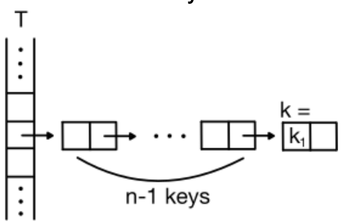
Hashing with Chaining: Time Complexity Proof (Key Found, cont'd)

Will show that the average search runtime is:

$$E\left[\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}X_{i,j})\right]$$

The number of examined elements until success is one more than the number of elements appearing in the list in $T[h(k)]$ *before* k, which is the inner equation. (k could be any k_i!)

Assume the key k we search for is k_1, inserted first!



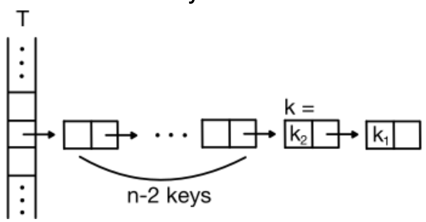
Hashing with Chaining: Time Complexity Proof (Key Found, cont'd)

Will show that the average search runtime is:

$$E\left[\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}X_{i,j})\right]$$

The number of examined elements until success is one more than the number of elements appearing in the list in $T[h(k)]$ *before* k, which is the inner equation. (k could be any k_i!)

Assume the key k we search for is k_2, inserted second:



Now we "just" need to show that the above equation is in $\Theta(1+\alpha)$.

Hashing with Chaining: Time Complexity Proof (Key Found), cont'd

$$E\left[\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}X_{i,j})\right]=\dots$$

(next equation follows from linearity of expectation)

$$=\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}E[X_{i,j}])=\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}\frac{1}{m})$$

$$=\frac{1}{n}\sum_{i=1}^{n}1+\frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{1}{m}=1+\frac{1}{n\cdot m}\sum_{i=1}^{n}\sum_{j=i+1}^{n}1$$

$$=1+\frac{1}{n\cdot m}\sum_{i=1}^{n}(\underbrace{(1+\dots+1)}_{n-1}+\underbrace{(1+\dots+1)}_{n-2}+\dots+\underbrace{1}_{1})$$

$$=1+\frac{1}{n\cdot m}\sum_{i=1}^{n}(n-i)=1+\frac{1}{n\cdot m}(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i)=1+\frac{1}{n\cdot m}(n^2-\frac{n}{2}(n+1))$$

$$=1+\frac{n}{m}-\frac{n+1}{2m}=1+\frac{n-1}{2m}=1+\frac{n}{2m}-\frac{1}{2m}$$

$$=1+\frac{1}{2}\alpha-\frac{1}{2m}\in\Theta(1+\alpha)\quad(\ni 1+\frac{1}{2}\alpha(1-\frac{1}{n}))$$

Hashing with Chaining: Commonly Used Hash Functions

- Simplest:
  - $h(k)=\lfloor k\cdot m\rfloor$ when the key is a real number independently and uniformly distributed in $[0,1)$
- Division method:
  - $h(k)=k \mod m$ when the key is an integer.
  - Choosing m to be prime might lead to fewer collisions if keys are not independently and uniformly distributed. If they are (which we assume) it doesn't matter.
- Multiplication method:
  - $h(k)=\lfloor((k\cdot A) \mod 1)\cdot m\rfloor$, where A is a constant (Real) in the range $0<A<1$.
  - What? Why does $\mod 1$ make sense?
    - Recall that A is real! So that's what remains.
    - $(k\cdot A) \mod 1 = k\cdot A-\lfloor k\cdot A\rfloor$ is the fractional component of $k\cdot A$.
  - E.g., let key $k=100$, constant $A=0.042$, and table size $m=16$.
    $h(k)=\lfloor(100\times 0.042 \mod 1)\times 16\rfloor=\lfloor(4.2 \mod 1)\times 16\rfloor$
    $=\lfloor 0.2\times 16\rfloor=\lfloor 3.2\rfloor=3$
  - Reduces dependencies on number of slots in the hash table.

Universal Hashing

Basics: Issues with Hash Functions

- All previous (commonly used) hash functions are deterministic. This is on the one hand important, but on the other there might be situations where we might perform at the worst case.
  - E.g., if the data is provided by an adversary, then he/she is in control of the distribution and may exploit this to make us perform at the worst-case!
  - Even without an adversary keys might not be uniformly distributed (depending on the application) and thus cause (close to?) worst-case performance.
  - Remember: Worst-case is a single list, i.e., $O(n)$.

Basics: Idea Behind Universal Hashing

- In *Universal Hashing*, we choose a random hash function h from a collection of hash functions, denoted by H.
- A collection H of hash functions is called universal when for each pair of distinct keys k, k', the number of hash functions for which $h(k) = h(k')$ is at most $\frac{|H|}{m}$, $|\{h \in H \mid h(k) = h(k')\}| \leq \frac{|H|}{m}$.
- Equivalently, this means that for each specific $h \in H$, the probability of a hash collision between any distinct keys k and k' is at most $\frac{1}{m}$ if $h(k)$ and $h(k')$ were randomly and independently chosen from all possible hash values $\{0, \ldots, m-1\}$.
- We later see how such a "family" H can be constructed by parametrizing hash functions.

Basics: Average Number of Collisions

Suppose we have a hash table T that uses chaining and universal hashing that's already filled with n distinct keys. Terminology:

- Let n_i denote the number of elements stored in $T[i]$.
- $E[n_i]$ is the expected number of elements stored in $T[i]$.

Given a key k, if $h(k) = i$ for a hash function h selected uniformly at random from the collection H, then it holds:

- if k is not already in T: $E[n_i] \leq \frac{n}{m} = \alpha$
- if k is already in T: $E[n_i] \leq \frac{n-1}{m} + 1 < \alpha + 1$

We now prove that these bounds hold without assumptions on the distribution on keys, but solely depend on the hash function.

Basics: Average Number of Collisions, Proof

- Let the random variable $X_{k,l}$ describe that keys k and l are colliding:

  $$X_{k,l} = \begin{cases} 1 & \text{if } h(k) = h(l) \\ 0 & \text{otherwise} \end{cases}$$

  $P(X_{k,l} = 1) = E[X_{k,l}] \leq \frac{1}{m}$
- WLOG, suppose we are looking for key k.
- Define random variable Y_k as the number of keys other than k, which are in T and hash to the same index as k. First, let $keys(T)$ be the set of keys already in T. Then we get:
- $$E[Y_k] = E\left[\sum_{\substack{l \neq k \\ l \in keys(T)}} X_{k,l}\right] = \sum_{\substack{l \neq k \\ l \in keys(T)}} E[X_{k,l}] \leq \sum_{\substack{l \neq k \\ l \in keys(T)}} \frac{1}{m}$$

Basics: Average Number of Collisions, Proof cont'd

Recall from last slide: $E[Y_k] \leq \sum_{\substack{l \neq k \\ l \in keys(T)}} \frac{1}{m}$

Thus:

- If k is *not* already in T:

  $E[n_i] = E[Y_k] \leq \frac{n}{m} = \alpha$
- If k *is* already in T:

  $E[n_i] = E[Y_k] + 1 \leq \frac{n-1}{m} + 1 < \alpha + 1.$

Basics: Examples for Universal Hash Functions

- Example from our CLRS textbook (p. 267, PDF page 288):
  - $h_{a,b,p}(k) = ((a \cdot k + b) \bmod p) \bmod m$
  - a, b, and p are constants. p prime, $p > m$.
  - $H = \{h_{a,b,p} \mid a, b \in \{0, \ldots, p-1\}, a \neq 0\}$
- Dot Product Hash Family
  - Covered in detail next!

Dot Product Hash Family: Example for Generating Universal Hash Functions

- Suppose m is a prime number.
- Express key k (which has base 10) with another base m.
  - Recall: With our "normal" numbers that we use every day, we have base $m = 10$, so a number k, e.g., $k = 42$ (i.e., $\vec{k} = \langle 4, 2\rangle$, since $42 = 4 \cdot 10^1 + 2 \cdot 10^0$), consists of digits in $\{0, \ldots, 9\}$.
  - Thus, express k as sequence/vector $\vec{k} = \langle k_{r-1}, \ldots, k_0\rangle$, where $k_i \in \{0, \ldots, m-1\}$ for all $0 \leq i \leq r-1$, and r being the arity that we need to express k with base m.
- Note that $r = \lfloor log_m(\max_{k \in U} k)\rfloor + 1$ (since $\max_{k \in U} k \leq m^r - 1$). E.g., if $U = \{0, \ldots, 42\}$ and $m = 2$, the highest key is $\vec{42} = \langle 1, 0, 1, 0, 1, 0\rangle$, so $r = \lfloor log_2(42)\rfloor = \lfloor 5.39\ldots\rfloor + 1 = 6$ (Note that $m = 2$ is unrealistic since that's our array size!)
- Dot product hash family:
  - Choose a random number a and compute $\vec{a} = \langle a_{r-1}, \ldots, a_0\rangle$ (again expressed with base m, and with the same length r as k)
  - Define $h_a(k) = (\vec{a} \cdot \vec{k}) \bmod m = \sum_{i=0}^{r-1} a_i \cdot k_i \bmod m$
  - The dot product hash family is $H = \{h_a \mid a \in \{0, \ldots, m^r - 1\}\}$.

Dot Product Hash Family: Example 1

Suppose the possible keys are integers $U = \{0, \ldots, 20\}$.

- Assume $m = 5$ (the array size).
- So we get $r = \lfloor log_5(20)\rfloor + 1 = \lfloor 1.891\ldots\rfloor + 1 = 2$. So two decimals suffice: $\vec{20} = 4 \cdot 5^1 + 0 \cdot 5^0$
- Assume the key to insert is $k = 19$. Since $19 = 3 \cdot 5^1 + 4 \cdot 5^0$ we get $\vec{19} = \langle 3, 4\rangle$
- Assume our randomly picked $a \in \{0, \ldots, 5^2 - 1\}$ is 15. In base 5 that's $\vec{a} = \langle 3, 0\rangle$.
- So we get $h_a(k) = h_{15}(19) = (\vec{15} \cdot \vec{19}) \bmod 5$ $= (3 \cdot 3 + 0 \cdot 4) \bmod 5 = 9 \bmod 5 = 4$

Dot Product Hash Family: Example 2

Let's say we hash IPs. (IPv4)

- IPs like 127.0.0.1 (the local host) consist of four numbers, each in $\{0, \ldots, 255\}$. Let's interpret IPs as those "vectors".
- Recall that our k_i in $\vec{k} = \langle \ldots k_i \ldots\rangle$ were within $\{0, \ldots, m-1\}$. So if we see an IP as a number with 4 "digits" each between 0 and 255, then $m - 1 = 255$, so $m = 256$. We need m prime, so we may pick $m = 257$ (which is the smallest prime ≥ 256).
- So the local host IP would be $\vec{lh} = \langle 127, 0, 0, 1\rangle$ (representing the number $lh = 2,130,706,433$ in base 10, but that doesn't matter!)
- Now we pick a random key a, $\vec{a} = \langle a_3, a_2, a_1, a_0\rangle$, such that $a_i \in \{0, \ldots, m-1\}$ for all $0 \leq i \leq 3$.
- Then proceed as before: compute $h_a(lh)$ as $(\vec{a} \cdot \vec{lh}) \bmod 257$
- Note that here the given key lh is to base $m = 256$, but the random key a is to base $m = 257$, which we normally don't do.

Dot Product Hash Family: Universality

Is the dot product hash family universal?

- Yes!
- Proof:
  - It wouldn't be presented in that section, otherwise! :)
  - (The actual proof is skipped, but you should know which properties should hold.)

Open Addressing

Procedure

- All items are stored directly in the hash table (i.e., no linked list, one item per slot). So the main idea is to look for another free index in case of collision.
- Here, the hash function maps "(key, number of trials)" to an index in the hash table:

$$h : U \times \{0, \dots, m-1\} \to \{0, \dots, m-1\}$$

- The sequence $\langle h(k,0), h(k,1), \dots, h(k,m-1)\rangle$ is called a *probe sequence*.
- Each probe sequence must be a permutation of $\langle 0, \dots, m-1\rangle$.
- Why the permutation requirement? So that each slot gets eventually filled! (Since the permutation is the sequence of array positions that are tried, in the order of trying.)

Generic Example

- Say that we want to store key k. Since we don't know whether $h(k)$ is still free or not, we just assume so and thus try the first trial: We compute $h(k,0)$, which, say, is i. If it's free, store it there, i.e., at $T[i]$.
- Say we attempt to store a second key k'. Again we have to try the first trial again, so we compute $h(k',0)$. Assume that's a collision with i, so we get $h(k',0) = i$ as well.
- We will thus have to try again and compute $h(k',1)$, which will get another index. We repeat until we get a free index.

Insertion and Search

```
HASH-INSERT(T, k)
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j] == NIL
5          T[j] = k
6          return j
7      else i = i + 1
8  until i == m
9  error "hash table overflow"
```

```
HASH-SEARCH(T, k)
1  i = 0
2  repeat
3      j = h(k, i)
4      if T[j] == k
5          return j
6      i = i + 1
7  until T[j] == NIL or i == m
8  return NIL
```

An issues here?

- One issue here is that search might take $O(m)$. If we assume $m > n$ (which we must in this context), it's in $O(n)$.
- Deletion doesn't (easily) work! (Next slides show why.)

Deletion

- Imagine key k has been stored in $T[h(k,3)]$.
- Suppose we delete $T[h(k,2)]$ (i.e., some other key lies there, which happens to also have the hash value $h(k,2)$). Thus, assume that we just assigned $T[h(k,2)] = NIL$ for this purpose.
- Now we can no longer find k in the hash table according to the previous search routine, since that stops once $T[h(k,2)] = NIL$ is found.
- Solution:
  - Fix delete procedure: Set $T[h(k,2)]$ as *deleted* instead of NIL:
    - *Solved:* Search will continue on 'deleted' entries and still stop on 'empty' (NIL) ones.
    - *New issue:* Insert can so far only delete when entry is NIL.
  - Fix insert procedure: Modify line 4 (insertion) such that it inserts data into cells that are NIL or *deleted*.

Probing Strategies: The Problem

- Recall that Insert and Search increase the i which encodes the trial number thus jumping from some $h(k,i)$ to $h(k,i+1)$.
- But the actual values $\langle h(k,0), h(k,1), \dots, h(k,m-1)\rangle$ of the probe are still "open" (i.e., not set)! We only demanded that it's some permutation of $\langle 0, \dots, m-1\rangle$, but which?
- That's called *probing strategy*.

Probing Strategies: Strategy Overview

- *Linear Probing:* $h(k,i) = (h'(k) + i) \mod m$, where h' is a usual hash function.
  - Meaning? Just use the next cell!
  - Issue: Clustering, i.e., consecutive group of occupied slots becomes longer.
- *Double Hashing:* $h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod m$, where h_1 and h_2 are usual hash functions. It must also hold that:
  - h_2 can never be 0 (for good reasons!)
  - $h_2(k)$ must be relatively prime to m (greatest common divisor is 1)

Perfect Hashing

General Idea and Properties

- If the data is static (i.e., no insert and delete), i.e., all data is given in advance, then we can ensure worst-case $O(1)$ search time.
- Idea: 2-level hashing
  - Each slot points to another hash table (instead of to a linked list).
  - Use universal hashing in both levels.
- Properties:
  - $O(1)$ search time in the worst case.
  - $O(n)$ expected space.

How to Build a Perfect Hash Function?

1. Set up outer hash table:
   - Set m to n. (Though usually m is prime.)
   - Pick a universal hash function $h_1 \in H$.
2. Set up inner has tables:
   - Set m_i to n_i^2.
   - If $\sum_{i=0}^{m-1} m_i > c \cdot n$, for a selected constant c, redo 1.
     This guarantees $O(n)$ size of the entire table!
   - Choose a universal hash function $h_2^i \in H$ to be the hash function for this inner hash table at position i and insert its elements.
   - As long as there are two k, k' with $h_2^i(k) = h_2^i(k')$ for any $k \neq k'$, pick a different h_2^i and rehash those elements in that inner table according to the new function.

Why Inner Table Size n_i^2? (Runtime)

Theorem:

Let m be n^2. The probability that there are *any* collisions (using a hash function randomly chosen from a set of universal hash functions) is strictly less than 50%.

Proof:

- There exist $\binom{n}{2}$ pairs of keys (worst case number of collisions).
- Each collision has probability $\frac{1}{m}$.
- Let X be a random variable counting the number of collisions.
- $E[X] = \binom{n}{2} \cdot \frac{1}{m} = \frac{n \cdot (n-1)}{2} \cdot \frac{1}{n^2} = \frac{1}{2} \cdot \frac{n-1}{n} < \frac{1}{2} \cdot \frac{n}{n} = \frac{1}{2}$.

So it's more likely (than not) that we get no collisions!
(Note that this result generalizes to m_i and n_i.)

Why Inner Table Size n_i^2? (Space)

Theorem:

Let m be n and m_i be n_i^2 for each $0 \leq i < m$.

Then it holds: $E\left[\sum_{i=0}^{m-1} m_i\right] < 2n$

Proof:

Some auxiliary equality: $a^2 = a + 2 \cdot \binom{a}{2}$ (we use it for $m_i = n_i^2$)

$$E\left[\sum_{i=0}^{m-1} m_i\right] = E\left[\sum_{i=0}^{m-1} n_i + 2 \cdot \binom{n_i}{2}\right] = E\left[\sum_{i=0}^{m-1} n_i\right] + 2 \cdot E\left[\sum_{i=0}^{m-1} \binom{n_i}{2}\right]$$

$$= n + 2 \cdot E\left[\sum_{i=0}^{m-1} \binom{n_i}{2}\right] = n + 2 \cdot \left(\binom{n}{2} \cdot \frac{1}{m}\right) = n + 2 \cdot \left(\frac{n \cdot (n-1)}{2} \cdot \frac{1}{n}\right)$$

$$= n + (n-1) = 2n - 1 < 2n$$ (Outer table size still missing.)

So the expected perfect hash table size is almost exactly $2n + n = 3n$.

Book shows: probability that the size is $\geq 4n$ is $< 50\%$.

Summary

Summary

This week we covered **hash functions**.

Hash functions considered:

- Simple Uniform Hashing $\rightarrow$ assumes equal hash distribution
- Universal Hashing $\rightarrow$ select hash function randomly
- Perfect Hashing $\rightarrow$ no collisions at all

Data storage types:

- Hashing with Chaining $\rightarrow$ use linked list
- Open Addressing $\rightarrow$ take another free position

The main motivation is to obtain constant time access to data.