Hash tables
Two events \( A, B \) are independent if 

\[
\Pr[A \cap B] = \Pr[A] \cdot \Pr[B]
\]

Conditional probability:

\[
\Pr[A|B] = \frac{\Pr[A \cap B]}{\Pr[B]}
\]

The expectation of a (discrete) random variable \( X \) is

\[
E[X] = \sum_k k \cdot \Pr[X = k]
\]

In particular, if the values of \( X \) are 0 or 1, then

\[
E[X] = \Pr[X = 1].
\]
Union bound: For events $A_1, \ldots, A_k$,

$$\Pr \left[ \bigcup_{i=1}^{k} A_i \right] \leq \sum_{i=1}^{k} \Pr[A_i]$$

Linearity of expectation: For random variables $X_1, \ldots, X_k$,

$$E \left[ \sum_{i=1}^{k} X_i \right] = \sum_{i=1}^{k} E[X_i]$$
A dictionary is a data-structure that stores a set $S$ of elements, where each element $x$ has a field $x.key$, and supports the following operations:

- **Search($S, k$)** Return an element $x \in S$ with $x.key = k$
- **Insert($S, x$)** Add $x$ to $S$.
- **Delete($S, x$)** Delete $x$ from $S$.

We will assume that
- The keys are from $U = \{0, \ldots, u - 1\}$.
- The keys are distinct.
**Direct addressing**

- $S$ is stored in an array $T[0..m − 1]$. The entry $T[k]$ contains a pointer to the element with key $k$, if such element exists, and NULL otherwise.
- **Search**$(T, k)$: return $T[k]$.
- **Insert**$(T, x)$: $T[x.key] ← x$.
- **Delete**$(T, x)$: $T[x.key] ←$ NULL.
- What is the problem with this structure?
In order to reduce the space complexity of direct addressing, map the keys to a smaller range \( \{0, \ldots m − 1\} \) using a hash function.

There is a problem of collisions (two or more keys that are mapped to the same value).

There are two ways to handle collisions:

- Chaining
- Open addressing
Let $h : U \rightarrow \{0, \ldots, m - 1\}$ be a hash function ($m < u$).

$S$ is stored in a table $T[0..m - 1]$ of linked lists. The element $x \in S$ is stored in the list $T[h(x.key)]$.

$\text{Search}(T, k)$: Search the list $T[h(k)]$.

$\text{Insert}(T, x)$: Insert $x$ at the head of $T[h(x.key)]$.

$\text{Delete}(T, x)$: Delete $x$ from $T[h(x.key)]$.

$S = \{6, 9, 19, 26, 30\}$

$m = 5$, $h(x) = x \mod 5$
Assumption of simple uniform hashing: any element is equally likely to hash into any of the $m$ slots, independently of where other elements have hashed into.

The above assumption is true when the keys are chosen uniformly and independently at random (with repetitions), and the hash function satisfies $|\{k \in U : h(k) = i\}| = u/m$ for every $i \in \{0, \ldots, m-1\}$.

We want to analyze the performance of hashing under the assumption of simple uniform hashing. This is the balls into bins problem.

Suppose we randomly place $n$ balls into $m$ bins. Let $X$ be the number of balls in bin 1.

The time complexity of a random search in a hash table is $\Theta(1 + X)$. 

Hash tables
Let $\alpha = n/m$.

**Claim**

$E[X] = \alpha$.

**Proof.**

Let $l_j$ be a random variable which is 1 if the $j$-th ball is in bin 1, and 0 otherwise. $X = \sum_{j=1}^{n} l_j$, so

$$E[X] = E[\sum_{j=1}^{n} l_j] = \sum_{j=1}^{n} E[l_j] = \sum_{j=1}^{n} \Pr[l_j = 1] = n \cdot \frac{1}{m}$$
The distribution of $X$

Claim

$\Pr[X = r] \approx \frac{e^{-\alpha} \alpha^r}{r!}$. (i.e., $X$ has approximately Poisson distribution).

Proof.

$\Pr[X = r] = \binom{n}{r} \left(\frac{1}{m}\right)^r \left(1 - \frac{1}{m}\right)^{n-r} = \frac{n(n-1)\cdots(n-r+1)}{r!} \frac{1}{m^r} \left(1 - \frac{1}{m}\right)^{n-r}$

If $m$ and $n$ are large, $n(n-1)\cdots(n-r+1) \approx n^r$ and $(1 - \frac{1}{m})^{n-r} \approx e^{-n/m}$. Thus, $\Pr[X = r] \approx \frac{e^{-n/m} (n/m)^r}{r!}$. 

Hash tables
The maximum number of balls in a bin

Let $Y$ be the maximum number of balls in some bin.

**Claim**

For $m = n$, $\Pr[Y \geq R] \leq \frac{1}{n^{1.5}}$, where $R = e \ln n / \ln \ln n$.

**Proof.**

Let $X_i$ be the number of balls in bin $i$.

$$
\Pr[Y \geq r] \leq \sum_{i=1}^{n} \Pr[X_i \geq r] = n \cdot \Pr[X \geq r]
$$

$$
\Pr[X \geq r] \leq \binom{n}{r} \left(\frac{1}{n}\right)^r \leq \frac{n^r}{r!} \cdot \frac{1}{n^r} = \frac{1}{r!} \leq \left(\frac{e}{r}\right)^r
$$

where the last inequality is true since $e^r = \sum_{i=0}^{\infty} \frac{r^i}{i!} \geq \frac{r^r}{r!}$.
Proof (continued).

\[ \Pr[Y \geq R] \leq n \cdot \left( \frac{e}{R} \right)^R \]

\[
= n \left( \frac{\ln \ln n}{\ln n} \right)^{e \ln n / \ln \ln n}
\]

\[
= e^{\ln n} \cdot e^{(\ln \ln \ln n - \ln \ln n) \cdot e \ln n / \ln \ln n}
\]

\[
= e^{- (e-1) \ln n + \ln n \cdot \frac{e \ln \ln n}{\ln \ln n}}
\]

\[
\leq e^{-1.5 \ln n} = \frac{1}{n^{1.5}}.
\]
We wish to maintain \( n = O(m) \) in order to have \( \Theta(1) \) search time.

This can be achieved by rehashing. Suppose we want \( n \leq m \). When the table has \( m \) elements and a new element is inserted, create a new table of size \( 2m \) and copy all elements into the new table.

The cost of rehashing is \( \Theta(n) \).
Universal hash functions

Definition
A collection \( \mathcal{H} \) of hash functions is a **universal** if for every pair of distinct keys \( x, y \in U \), \( \Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{2}{m} \).

Example
Let \( p \) be a prime number larger than \( u \).
\[
f_{a,b}(x) = ((ax + b) \mod p) \mod m
\]
\( \mathcal{H}_{p,m} = \{ f_{a,b} | a \in \{1, 2, \ldots, p - 1\}, b \in \{0, 1, \ldots, p - 1\} \} \)
Universal hash functions

**Theorem**

Suppose that \( \mathcal{H} \) is a universal collection of hash functions. If a hash table for \( S \) is built using a randomly chosen \( h \in \mathcal{H} \), then for every \( k \in U \), the expected time of \( \text{Search}(S, k) \) is \( \Theta(1 + n/m) \).

**Proof.**

Let \( X = \text{length of } T[h(k)] \).

\[
X = \sum_{y \in S} l_y \text{ where } l_y = 1 \text{ if } h(y.\text{key}) = h(k) \text{ and } l_y = 0 \text{ otherwise.}
\]

\[
E[X] = E \left[ \sum_{y \in S} l_y \right] = \sum_{y \in S} E[l_y] = \sum_{y \in S} \Pr_{h \in \mathcal{H}} [h(y.\text{key}) = h(k)] \\
\leq 1 + n \cdot \frac{2}{m}.
\]
Under the assumption of simple uniform hashing, the expected time of a search is $\Theta(1 + \alpha)$ time.

If $\alpha = \Theta(1)$, and under the assumption of simple uniform hashing, the worst case time of a search is $\Theta(\log n / \log \log n)$, with probability at least $1 - 1/n^{\Theta(1)}$.

If the hash function is chosen from a universal collection at random, the expected time of a search is $\Theta(1 + \alpha)$.

The worst case time of insert is $\Theta(1)$ if there is no rehashing.
Interpreting keys as natural numbers

- How can we convert floats or ASCII strings to natural numbers?
- An ASCII string can be interpreted as a number in base 128.

**Example**

For the string CLRS, the ASCII values of the characters are C = 67, L = 76, R = 82, S = 83. So CLRS is \((67 \cdot 128^3) + (76 \cdot 128^2) + (82 \cdot 128^1) + (83 \cdot 128^0) = 141,764,947.\)
Horner’s rule

- Horner’s rule:
  \[a_dx^d + a_{d-1}x^{d-1} + \cdots + a_1x + a_0 = (\cdots ((a_dx + a_{d-1})x + a_{d-2})x + \cdots )x + a_0.\]

- Code:
  
  ```
  y = a_d 
  for i = d - 1 to 0 
  y = a_i + xy 
  ```

- If \(d\) is large the value of \(y\) is too big.

- Solution: evaluate the polynomial modulo \(p\):
  
  ```
  y = a_d 
  for i = d - 1 to 0 
  y = (a_i + xy) \mod p 
  ```
Suppose we are given strings $P$ and $T$, and we want to find all occurrences of $P$ in $T$.

The Rabin-Karp algorithm is as follows:

- Compute $h(P)$ for every substring $T'$ of $T$ of length $|P|$.
- If $h(T') = h(P)$ check whether $T' = P$.

The values $h(T')$ for all $T'$ can be computed in $\Theta(|T|)$ time using rolling hash.

Example

Let $T = \text{BGUCS}$ and $P = \text{GUC}$. Let $T_1 = \text{BGU}$, $T_2 = \text{GUC}$.

$$h(T_1) = (66 \cdot 128^2 + 71 \cdot 128 + 85) \mod p$$

$$h(T_2) = (71 \cdot 128^2 + 85 \cdot 128 + 67) \mod p$$

$$= (h(T_1) - 66 \cdot 128^2) \cdot 128 + 67 \mod p$$
Applications

- **Data deduplication**: Suppose that we have many files, and some files have duplicates. In order to save storage space, we want to store only one instance of each distinct file.

- **Distributed storage**: Suppose we have many files, and we want to store them on several servers.
Let $m$ denote the number of servers.

The simple solution is to use a hash function $h : U \rightarrow \{1, \ldots, m\}$, and assign file $x$ to server $h(x)$.

The problem with this solution is that if we add a server, we need to do rehashing which will move most files between servers.
Suppose that the server have identifiers \( s_1, \ldots, s_m \).

Let \( h : U \to [0, 1] \) be a hash function.

For each server \( i \) associate a point \( h(s_i) \) on the unit circle.

For each file \( f \), assign \( f \) to the server whose point is the first point encountered when traversing the unit cycle anti-clockwise starting from \( h(f) \).
Suppose that the server have identifiers $s_1, \ldots, s_m$.

Let $h : U \rightarrow [0, 1]$ be a hash function.

For each server $i$ associate a point $h(s_i)$ on the unit circle.

For each file $f$, assign $f$ to the server whose point is the first point encountered when traversing the unit cycle anti-clockwise starting from $h(f)$. 
Distributed storage: Consistent hashing

- When a new server $m + 1$ is added, let $i$ be the server whose point is the first server point after $h(s_{m+1})$.
- We only need to reassign some of the files that were assigned to server $i$.
- The expected number of files reassignments is $n/(m + 1)$.
Linear probing

- In the following, we assume that the elements in the hash table are keys with no satellite information.
- To insert an element \( k \), try to insert to \( T[h(k)] \).
  If \( T[h(k)] \) is not empty, try
  \( T[h(k) + 1 \mod m] \), then try
  \( T[h(k) + 2 \mod m] \) etc.
- Code:
  ```
  for i = 0, \ldots, m - 1
    j = h(k) + i \mod m
  if T[j] = NULL OR ...
    T[j] = k
  return error “hash table overflow”
  ```
Linear probing

- In the following, we assume that the elements in the hash table are keys with no satellite information.

- To insert an element \( k \), try to insert to \( T[h(k)] \).

  If \( T[h(k)] \) is not empty, try \( T[h(k) + 1 \text{ mod } m] \), then try \( T[h(k) + 2 \text{ mod } m] \) etc.

- Code:

  ```
  for i = 0, \ldots, m - 1
  j = h(k) + i \text{ mod } m
  if T[j] = NULL OR ...
    T[j] = k
  return error "hash table overflow"
  ```

Hash tables

0 1 2 3 4 14 5 6 7 8 9

insert(T,14)
Linear probing

- In the following, we assume that the elements in the hash table are keys with no satellite information.
- To insert an element $k$, try to insert to $T[h(k)]$.
  - If $T[h(k)]$ is not empty, try $T[h(k) + 1 \mod m]$, then try $T[h(k) + 2 \mod m]$ etc.
- Code:
  ```
  for i = 0, \ldots, m - 1
  j = h(k) + i \mod m
  if T[j] = NULL OR ...
  j = h(k) + i \mod m
  T[j] = k
  return error “hash table overflow”
  ```
In the following, we assume that the elements in the hash table are keys with no satellite information.

To insert an element $k$, try to insert to $T[h(k)]$.

If $T[h(k)]$ is not empty, try $T[h(k) + 1 \mod m]$, then try $T[h(k) + 2 \mod m]$ etc.

Code:

```c
for i = 0, \ldots, m - 1
    j = h(k) + i \mod m
    if T[j] = NULL OR ...
        T[j] = k
        return error "hash table overflow"
```

Hash tables
Linear probing

In the following, we assume that the elements in the hash table are keys with no satellite information.

To insert an element $k$, try to insert to $T[h(k)]$. If $T[h(k)]$ is not empty, try $T[h(k) + 1 \mod m]$, then try $T[h(k) + 2 \mod m]$ etc.

Code:

```c
for i = 0, \ldots, m - 1
    j = h(k) + i \mod m
    if T[j] = NULL OR ...
        T[j] = k
    return error “hash table overflow”
```
In the following, we assume that the elements in the hash table are keys with no satellite information.

To insert an element $k$, try to insert to $T[h(k)]$. If $T[h(k)]$ is not empty, try $T[h(k) + 1 \mod m]$, then try $T[h(k) + 2 \mod m]$ etc.

Code:

```
for i = 0, \ldots, m - 1
    j = h(k) + i \mod m
    if T[j] = NULL OR ...
        T[j] = k
        return
error "hash table overflow"
```
Linear probing

- In the following, we assume that the elements in the hash table are keys with no satellite information.
- To insert an element $k$, try to insert to $T[h(k)]$. If $T[h(k)]$ is not empty, try $T[h(k) + 1 \mod m]$, then try $T[h(k) + 2 \mod m]$ etc.
- Code:

```c
for i = 0, \ldots, m - 1
    j = h(k) + i \mod m
    if T[j] = NULL OR ...
        T[j] = k
        return
error “hash table overflow”
```

insert(T,49)
Insert

How to perform Search?

search(T,55)
Search($T, k$) is performed as follows:

\[
\text{for } i = 0, \ldots , m - 1 \\
\quad j = h(k) + i \mod m \\
\quad \text{if } T[j] = k \\
\quad \quad \text{return } j \\
\quad \text{if } T[j] = \text{NULL} \\
\quad \quad \text{return } \text{NULL} \\
\text{return } \text{NULL}
\]
Delete method 1: To delete element $k$, store in $T[h(k)]$ a special value DELETED.
Delete method 2: Erase \( k \) from the table (replace it by NULL) and also erase all the elements in the block of \( k \). Then, reinsert the latter elements to the table.
Delete method 2: Erase $k$ from the table (replace it by NULL) and also erase all the elements in the block of $k$. Then, reinsert the latter elements to the table.
Delete method 2: Erase $k$ from the table (replace it by NULL) and also erase all the elements in the block of $k$. Then, reinsert the latter elements to the table.
Open addressing is a generalization of linear probing.

Let
\[ h : U \times \{0, \ldots, m - 1\} \rightarrow \{0, \ldots, m - 1\} \]
be a hash function such that
\( \{h(k, 0), h(k, 1), \ldots, h(k, m - 1)\} \) is a permutation of \( \{0, \ldots, m - 1\} \) for every \( k \in U \).

The slots examined during search/insert are \( h(k, 0) \), then \( h(k, 1) \), \( h(k, 2) \) etc.

In the example on the right,
\[ h(k, i) = (h_1(k) + ih_2(k)) \mod 13 \]
where
\begin{align*}
    h_1(k) &= k \mod 13 \\
    h_2(k) &= 1 + (k \mod 11)
\end{align*}
Insertion is the same as in linear probing:

```
for i = 0, ..., m - 1
    j = h(k, i)
    if T[j] = NULL OR T[j] = DELETED
        T[j] = k
    return error “hash table overflow”
```

Deletion is done using delete method 1 defined above (using special value DELETED).
Double hashing

- In the double hashing method,
  \[ h(k, i) = (h_1(k) + ih_2(k)) \mod m \]
  for some hash functions \( h_1 \) and \( h_2 \).
- The value \( h_2(k) \) must be relatively prime to \( m \) for the entire hash table to be searched. This can be ensured by either
  - Taking \( m \) to be a power of 2, and the image of \( h_2 \) contains only odd numbers.
  - Taking \( m \) to be a prime number, and the image of \( h_2 \) contains integers from \( \{1, \ldots, m - 1\} \).
- For example,
  \[ h_1(k) = k \mod m \]
  \[ h_2(k) = 1 \mod m' \]
  where \( m \) is prime and \( m' < m \).
Assume uniform hashing: the probe sequence of each key is equally likely to be any of the $m!$ permutations of $\{0, \ldots, m - 1\}$.

Assuming uniform hashing and no deletions, the expected number of probes in a search is

- At most $\frac{1}{1 - \alpha}$ for unsuccessful search.
- At most $\frac{1}{\alpha} \ln \frac{1}{1 - \alpha}$ for successful search.