# Hash Tables

ESO207

Indian Institute of Technology, Kanpur

# Introduction

- Consider a dynamic set that supports dictionary operations, INSERT, SEARCH and DELETE.
- Previously, the LIST (and even STACK and QUEUE) data structures can support the above operations, but not efficiently.
- Hash table is an effective data structure for implementing dictionaries.
- Searching for an element in a hash table can take as long as in a linked list – $\Theta(n)$ in the worst case, but
- In practice, hashing performs extremely well. Under reasonable assumptions, expected time to search for an element in a hash table is $O(1)$.

# Introduction

- Consider a dynamic set that supports dictionary operations, INSERT, SEARCH and DELETE.
- Previously, the LIST (and even STACK and QUEUE) data structures can support the above operations, but not efficiently.
- Hash table is an effective data structure for implementing dictionaries.
- Searching for an element in a hash table can take as long as in a linked list – $\Theta(n)$ in the worst case, but
- In practice, hashing performs extremely well. Under reasonable assumptions, expected time to search for an element in a hash table is $O(1)$.

# Introduction

- Consider a dynamic set that supports dictionary operations, INSERT, SEARCH and DELETE.
- Previously, the LIST (and even STACK and QUEUE) data structures can support the above operations, but not efficiently.
- Hash table is an effective data structure for implementing dictionaries.
- Searching for an element in a hash table can take as long as in a linked list – $\Theta(n)$ in the worst case, but
- In practice, hashing performs extremely well. Under reasonable assumptions, expected time to search for an element in a hash table is $O(1)$.

# Introduction

- Consider a dynamic set that supports dictionary operations, INSERT, SEARCH and DELETE.
- Previously, the LIST (and even STACK and QUEUE) data structures can support the above operations, but not efficiently.
- Hash table is an effective data structure for implementing dictionaries.
- Searching for an element in a hash table can take as long as in a linked list – $\Theta(n)$ in the worst case, but
- In practice, hashing performs extremely well. Under reasonable assumptions, expected time to search for an element in a hash table is $O(1)$.

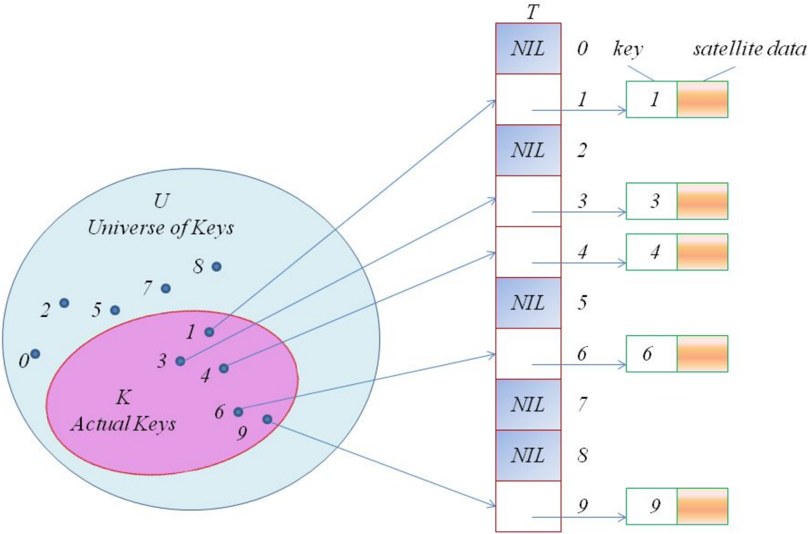
# Introduction

- Consider a dynamic set that supports dictionary operations, INSERT, SEARCH and DELETE.
- Previously, the LIST (and even STACK and QUEUE) data structures can support the above operations, but not efficiently.
- Hash table is an effective data structure for implementing dictionaries.
- Searching for an element in a hash table can take as long as in a linked list – $\Theta(n)$ in the worst case, but
- In practice, hashing performs extremely well. Under reasonable assumptions, expected time to search for an element in a hash table is $O(1)$.

# Introduction (contd.)

- A hash table generalizes an array's ability to directly address by an index into it in $O(1)$ time.
- Notation: Let $U$ be the universe of the possible values of the keys.
- The dynamic set $S$ at any time is a subset of $U$.
- For simplicity (although not general), assume that $U = \{0, 1, \ldots, m - 1\}$ is a set of numbered keys.
- No two elements of the set have the same key.

# Direct Addressing



# Direct Address Tables

- Direct addressing works well when the universe is small.
- An array called the **direct-address table** is used and denoted by $T[0, \dots, m-1]$.
- Each position or **slot** of the array corresponds to a key in the universe $U$.
- If the dynamic set is $S$, then, for each $k \in S$, $T[k]$ (that is, slot $k$) points to the element of the set with key $k$.
- For each $k \in U - S$, $T[k]$ is NIL.

# Operations: Direct Address Tables

DIRECT-ADDRESS-SEARCH$(T, k)$
1. **return** $T[k]$

DIRECT-ADDRESS-INSERT$(T, x)$
1. $T[x.key] = x$

DIRECT-ADDRESS-DELETE$(T, x)$
1. $T[x.key] =$ NIL

# Direct Address Tables

- For some applications, direct address table can hold the elements directly.
- That is, rather than storing the element's key and satellite data in an object external to the table, with a pointer from the slot, we can store the object in the slot itself.
- For this, empty slots will need to be indicated by a special key value.

# Hash Tables

- Down-side of direct addressing:
  1. If the universe $U$ is large, storing a table $T$ of size $|U|$ may be impractical or impossible.
  2. Further, the set of keys $K$ *actually stored* in the table $T$ may be very small. This would lead to wastage of space.
  3. Hash tables typically work with space $O(|K|)$, while still being able to search in expected $O(1)$ time. [For Direct-Address, this was worst-case $O(1)$ time].

# Hash Tables

- Keep a table $T$ consisting of $m$ slots numbered $T[0, \ldots, m-1]$.
- A **hash function** $h$ maps each key value of the universe $U$ to some slot in $T$, that is,

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

- We say that an element with key $k$ hashes to slot $h(k)$.
- In general, $m \ll |U|$ (and that is where the benefit in the storage requirement of hash tables arises).
- Hence, two keys from $U$ may hash to the same value, that is,

$$\textbf{Collision} : h(k_1) = h(k_2), \quad k_1, k_2 \in U, k_1 \neq k_2 \ .$$

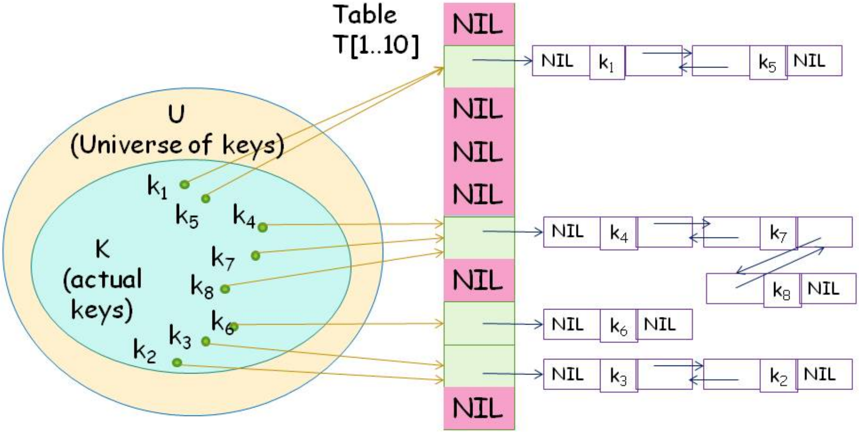
In such a case we say that the keys $k_1$ and $k_2$ collide under the hash function $h$.

# Collisions

- Collisions cannot be avoided, since $|U| > m$.
- One of the goals of the design of hash functions is to make $h$ appear “random”.
- The very term “to hash” evokes images of random mixing and chopping.
- We will explain the notion of randomness of hash functions later — note that a hash function $h$ is deterministic, that is, given an input key $k$, the output $h(k)$ is the same value.

# Collision resolution: by chaining

- Simplest form of resolving collisions is to keep *a linked list of the elements that hash to each bucket*.
- Each slot $j$ contains a pointer to the head of the list of all the stored elements that hash to the slot $j$.
- If there are no such elements, slot $j$ contains NIL.




Collision Resolution by chaining. Each hash table slot T[j] contains a doubly linked list of all the keys whose hash value is j. Here $k_1$ and $k_4$ collide, that is, $h(k_1) = h(k_4)$, and $h(k_5) = h(k_7) = h(k_2)$.

# Code for Insert, Search and Delete

Chained-Hash-Insert$(T, x)$
1. Insert $x$ at the head of the list $T[h(x.key)]$

Chained-Hash-Search$(T, k)$
1. Search for an element with key $k$ in list $T[h(k)]$

Chained-Hash-Delete$(T, x)$
1. Delete $x$ from the list $T[h(x.key)]$

- Worst case time for insertion: $O(1)$. Assumes element being inserted is not already present.
- Otherwise, we can check for this assumption by searching for an element whose key is $x.key$ before inserting.

# Search, Delete

CHAINED-HASH-INSERT$(T, x)$

1. Insert $x$ at the head of the list $T[h(x.key)]$

CHAINED-HASH-SEARCH$(T, k)$

1. Search for an element with key $k$ in list $T[h(k)]$

CHAINED-HASH-DELETE$(T, x)$

1. Delete $x$ from the list $T[h(x.key)]$

- Worst case time for *Search*: length of the linked list.
- Worst case time for *Delete*: $O(1)$. Note that deletion takes (pointer to the element) $x$ to be deleted.
- Since we have doubly-linked lists, deletion is fast.

# Comment

Hash tables with open chaining is a very popular data structure in the industry.

# Analysis of Chaining: introduction

- **Load factor**: Given a hash table $T$ with $m$ slots that stores $n$ elements, the ***load factor*** $\alpha$ for $T$ is $n/m$, that is, the average number of elements stored in a chain.
- A well-designed hash table in practice attempts to keep $\alpha$ close to 1.
- *Worst-case performance.* In the worst case, the input may consist of $n$ keys all of whom hash to the same slot. This would give a chain length of $n$ at this slot, and chain lengths of 0 at all other slots. The worst-case time for searching would be $O(n)$.
- Hash tables are not used for their worst-case performance.

# Analysis: introduction

- Hash tables are immensely popular because of their *average-case* performance.
- The average case performance of hashing depends on how well the hash function $h$ distributes the set of keys to be stored among the $m$ slots $\{0, 1, \ldots, m-1\}$, on the average.
- Idealized assumption about hash functions for analysis:
- ***Simple uniform hashing.*** A key value is equally likely to hash into any of the $m$ slots, independently of where any other element hashes to.

# Analysis

- Corresponding to slot $j \in \{0, 1, \ldots, m-1\}$, denote the length of the chain at slot $j$ to be $n_j$. Then,

$$n = n_0 + n_1 + \ldots + n_{m-1}$$

- The average chain length is the sum of all chain lengths divided by the number of slots, so this is

$$\mathbb{E}\left[n_j\right] = \frac{\sum_{k=0}^{m-1} n_k}{m} = \frac{n}{m} = \alpha$$

  which is the load factor of the hash table.

- For the analysis, we will make an important assumption, namely, that

  ***Hash value $h(k)$ is computed in time $O(1)$.***

# Analysis

- Expected time required to search for a key in a hash table.
- Step 1: compute $h(k)$ in $O(1)$ time. Then search through the list at slot number $h(k)$.
- Length of this list is $n_{h(k)}$.
- Hence, an *unsuccessful search* will go over all the keys in this list requiring time $O(n_{h(k)})$.

# Expected time for searching

- By uniform hashing assumption, $h(k) = j$ with equal probability $1/m$.
- the average time for search is

$$\mathbb{E}\left[n_{h(k)}\right] = \frac{1}{m}\sum_{j=0}^{m-1} n_j = \alpha$$

  as calculated before.
- The ***average case time complexity of search*** is $O(1+\alpha)$, where, the $O(1)$ time is required for computing the hash value $h(k)$.

## Expectation Analysis: contd.

- One would expect a *successful search* to take slightly less on average, since, the entire chain need not be browsed, rather, the search may halt once the key is found.
- A slightly detailed analysis shows that the average time for a successful search is $1 + \alpha/2 - \alpha/(2m) = \Theta(1 + \alpha)$.
- All this analysis shows that if $\alpha = n/m$ is $O(1)$, then, the search operation requires on the average $O(1)$ time.
- Using doubly linked lists for storing the chains, deletion takes $O(1)$ time.
- Insertion (without searching for duplicates) takes $O(1)$ time, hence, all the hash table operations can be performed in $O(1)$ time, on average, if $\alpha = O(1)$.

# Good Hash Functions

- A *good hash* function should come close to realizing the idealized assumption made by simple uniform hashing.
- Many hash functions assume that the universe of keys is the set of natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$.
- Under this assumption, either the keys themselves are natural numbers, or they are transformed into natural numbers (e.g., strings of characters may be viewed as long numbers over a larger alphabet).

# Good and bad hash functions

- The simplest hash function is of the form

$$h(k) = k \mod m$$

- This hash function is very efficient, although it does not have good uniformity properties.
- It is often not recommended to use $m = 2^r$ for some value of $r$, since, $k \mod 2^r$ gives the low order $r$ bits of $k$.
- Unless we know that the low order $r$ bits of the input keys are equally likely, we are better off designing a hash function whose value depends on all the bits of the key.

# Example hash function: key modulo prime

- Choosing $m$ to be a prime $p$ that is not too close to a power of 2 is often a good choice.
- For example, suppose there are $n = 2000$ keys to be hashed. Let $p = 701$, which would give a load factor of $2000/701 \approx 3$ and $h(k) = k \mod 701$.

# Multiplicative method for creating hash functions

- The ***multiplication method*** for creating hash functions works in two steps.
  1. Choose a number $A$ in the range $0 < A < 1$ and multiply the key $k$ by $A$ and take the fractional part of $kA$, that is, take $kA - \lfloor kA \rfloor$.
  2. This is also referred to as
  $$kA \mod 1 = kA - \lfloor kA \rfloor$$
  3. Now multiply this by the number of slots $m$ and return the floor, that is,
  $$h(k) = \lfloor m(kA \mod 1) \rfloor$$

# Multiplication method

- Advantage: the value of $m$ is not critical.
- An efficient way of implementing the multiplication method:
  1. Choose $m = 2^p$, a power of 2.
  2. Suppose the word size of the machine is $w$ bits and a key value fits into a word.
  3. Choose $A$ to be of the form $s/2^w$ so that $A \cdot 2^w = s$.
  4. Since $k$ and $s$ are both $w$-bit integers, $ks$ is a $2w$-bit integer that we write as

     $$ks = r_1 2^w + r_0$$

     where, $r_1$ is the high order $w$-bit of $ks$ and $r_0$ is the low order $w$-bit.

# Multiplication method

1. Then,
$$kA = \frac{ks}{2^w} = r_1 + \frac{r_0}{2^w}$$
2. Hence the fractional part of $kA$ is $r_0/2^w$ (since, $0 < r_0 \leq 2^w - 1$).
3. So, $kA \mod 1 = r_0/2^w$.
4. Hence,
$$h(k) = \lfloor m(kA \mod 1) \rfloor = \left\lfloor 2^p \frac{r_0}{2^w} \right\rfloor$$
5. Since, $r_0$ is a $w$-bit number, multiplying $r_0/2^w$ by $2^p$ gives the top $p$ bits of $r_0$.

- Why? write $r_0$ as a $w$-bit binary number:

$$r_0 = b_1 + b_2 \cdot 2 + b_2 \cdot 2^2 \ldots + b_w 2^{w-1}$$

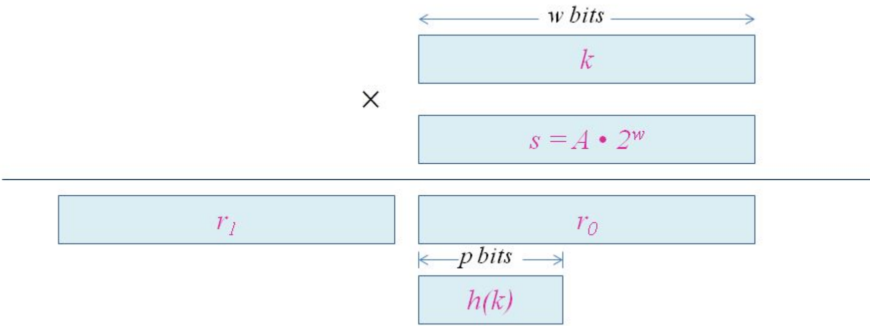
- Then,

$$\begin{aligned}
\frac{r_0 \cdot 2^p}{2^w} &= \frac{1}{2^w}\left(b_1 \cdot 2^p + b_2 \cdot 2^{p+1} + \ldots + b_w \cdot 2^{w+p-1}\right) \\
&= \frac{1}{2^{w-p}}\left(b_1 + b_2 \cdot 2 + \ldots + b_{w-p} \cdot 2^{w-p-1}\right) \\
&\qquad + b_{w-p+1} + b_{w-p+2} \cdot 2 + \ldots + b_w \cdot 2^{p-1}
\end{aligned}$$

- Thus,

$$\left\lfloor \frac{r_0 \cdot 2^p}{2^w} \right\rfloor = b_{w-p+1} + b_{w-p+2} \cdot 2 + \ldots + b_w \cdot 2^{p-1}$$

which is the number corresponding to the top-$p$ bits of $r_0$.


w bits
k
×
s = A • 2^w
r_1
r_0
p bits
h(k)

# Multiplicative method

- Although this method works with any value of the constant $A$, experiments show that it works better with some values than others.
- Knuth suggests that

$$A \approx (\sqrt{5} - 1)/2 = 0.6180339887\ldots$$

  is likely to work reasonably well.