Inf 2B: Hash Tables
Lecture 4 of ADS thread

Kyriakos Kalorkoti

School of Informatics
University of Edinburgh
Dictionaries

A *Dictionary* stores key–element pairs, called *items*. Several elements might have the same key. Provides three methods:

- **findElement**(\(k\))**: If the dictionary contains an item with key \(k\), then return its element; otherwise return the special element **NO_SUCH_KEY**.

- **insertItem**(\(k, e\))**: Insert an item with key \(k\) and element \(e\).

- **removeItem**(\(k\))**: If the dictionary contains an item with key \(k\), then delete it and return its element; otherwise return **NO_SUCH_KEY**.
List Dictionaries

- Items are stored in a \textit{singly linked list} (in any order).
- Algorithms for all methods are straightforward.
- Running Time:

  \begin{align*}
  \text{insertItem} & : \Theta(1) \\
  \text{findElement} & : \Theta(n) \\
  \text{removeItem} & : \Theta(n)
  \end{align*}

\textit{(n always denotes the number of items stored in the dictionary)}
Direct Addressing

Suppose:

- Keys are integers in the range $0, \ldots, N - 1$.
- All elements have distinct keys.

A data structure realising *Dictionary* (sometimes called a *direct address table)*:

- Elements are stored in array $B$ of length $N$.
- The element with key $k$ is stored in $B[k]$.
- Running Time: $\Theta(1)$ for all methods.
Bucket Arrays

Suppose:

- Keys are integers in the range $0, \ldots, N - 1$.
- Several elements might have the same key, so collisions may occur.

What do we do about these collisions?

Store them all together in a List pointed to by $B[k]$ (sometimes called chaining).
Bucket Arrays

Bucket array implementation of Dictionary:

- Bucket array $B$ of length $N$ holding Lists
- Element with key $k$ is stored in the List $B[k]$.
- Methods of Dictionary are implemented using insertFirst(), first(), and remove($p$) of List

Running Time: $\Theta(1)$ for all methods (with linked list implementation of List - $p$ is always the first pointer, so we can easily keep track of it).

- Works because findElement($k$) and removeItem($k$) only need 1 item with key $k$.

A good solution if $N$ is not much larger than the number of keys (a small constant multiple).
Dictionary implementation for arbitrary keys (not necessarily all distinct).

Two components:

- **Hash function** \( h \) mapping keys to integers in the range \( 0, \ldots, N - 1 \) (for some suitable \( N \in \mathbb{N} \)).
- **Bucket array** \( B \) of length \( N \) to hold the items.

Item (key–element pair) with key \( k \) is stored in the bucket \( B[h(k)] \).
Issues for Hash Tables

- Need to consider collision handling. (Here we might have $h(k_1) = h(k_2)$ even for $k_1 \neq k_2$, so List implementation is more complicated.)
- Analyse the running time.
- Find good hash functions.
- Choose appropriate $N$. 
Implementation

**Problem:** Elements with distinct keys might go into the same bucket.

**Solution:** Let buckets be *list dictionaries* storing the *items* (key-element pairs).

**The methods:**

```
Algorithm findElement(k)
1. Compute h(k)
2. return B[h(k)].findElement(k)
```
Implementation

**Algorithm** InsertItem\((k, e)\)
1. Compute \(h(k)\)
2. \(B[h(k)].insertItem(k, e)\)

**Algorithm** removeItem\((k)\)
1. Compute \(h(k)\)
2. **return** \(B[h(k)].removeItem(k)\)
Implementation

Running time?
Depends on the list methods
- $B[h(k)].\text{findElement}(k)$,
- $B[h(k)].\text{insertItem}(k, e)$, and
- $B[h(k)].\text{removeItem}(k)$.

Assume we Insert at front (or end):
- $\Theta(1)$ time for $B[h(k)].\text{insertItem}(k, e)$. 
Analysis

- Let $T_h$ be the running time required for computing $h$ (more precisely: $T_h(n_{\text{key}})$, where $n_{\text{key}}$ is the size of the key).
- Let $m$ be the maximum size of a bucket. Then the running time of the hash table methods is:

  \[
  \begin{align*}
  \text{insertItem} & : T_h + \Theta(1) \\
  \text{findElement} & : T_h + \Theta(m) \\
  \text{removeItem} & : T_h + \Theta(m)
  \end{align*}
  \]

  Worst case:

  \[
  m = n.
  \]

$m$ depends on hash function \textit{and} on input distribution of keys.
Hash functions

Hash function $h$ maps keys to $\{0, \ldots, N - 1\}$.

Criteria for a good hash function:

(H1) $h$ evenly distributes the keys over the range of buckets (hope input keys are well distributed originally).

(H2) $h$ is easy to compute.
Hash functions

- Simpler if we start with keys that are already integers.
- Trickier if the original key is not Integer type (e.g., string).

**One approach:** Split hash function into:

- hash code and
- compression map.
Hash Codes

- Keys (of any type) are just sequences of bits in memory.
- Basic idea: Convert bit representation of key to a binary integer, giving the hash code of the key.
- But computer integers have bounded length (say 32 bits).
  - consider bit representation of key as sequence of 32-bit integers $a_0, \ldots, a_{\ell-1}$
- Summation method: Hash code is
  \[ a_0 + \cdots + a_{\ell-1} \mod N \]

- Polynomial method: Hash code is
  \[ a_0 + a_1 \cdot x + a_2 \cdot x^2 + \cdots + a_{\ell-1} \cdot x^{\ell-1} \mod N \]
  (for some integer $x$).

Sometimes $N = 2^{32}$. 
Evaluating Polynomials

*Horner’s Rule:*

\[
a_0 + a_1 \cdot x + a_2 \cdot x^2 + \cdots + a_{\ell-1} \cdot x^{\ell-1} = a_0 + a_1 \cdot x + a_2 \cdot x \cdot x + \cdots + a_{\ell-1} \cdot x \cdot x \cdots x = a_0 \cdot x(a_1 + x(a_2 + \cdots + x(a_{\ell-2} + x \cdot + a_{\ell-1}) \cdots ))
\]

[\(\Theta(\ell^2)\) operations]

[\(\Theta(\ell)\) operations]

Has been *proved* to be best possible.

**Note:** Sensible to reduce mod \(N\) after each operation.

**Warning:** Deciding what is a “good hash function” is something of a “black art”.

Polynomials look good because it is harder to see regularities (many keys mapping to the same hash value).

**Warning:** we haven’t proved anything! For some situations there are bad regularities, usually due to a bad choice of \(N\).
Hash functions for character strings

Characters are 7-bit numbers (0, \ldots, 127).

- $x = 128, N = 96$. Bad for small words. (because $gcd(96, 128) = 32$. NOT coprime)

- $x = 128, N = 97$, good.

- $x = 127, N = 96$, good.
Compression Map

Integer $k$ is mapped to

$$|ak + b| \mod N,$$

where $a, b$ are randomly chosen integers.

Whole point of hashing is to “Compress” (evenly).

Works particularly well if $a, N$ are coprime (experimental observation only).
Quick quiz question

Consider the hash function

\[ h(k) = 3k \mod 9. \]

Suppose we use \( h \) to hash exactly one item for every key \( k = 0, \ldots, 9M - 1 \) (for some big \( M \)) into a bucket array with 9 buckets \( B[0], B[1], \ldots, B[8] \). How many items end up in bucket \( B[5] \)?

1. 0.
2. \( M \).
3. \( 2M \).
4. \( 4M \).

Answer is 0.
Load Factors and Re-hashing

- Number of items: \( n \)
  Length of bucket array: \( N \)

\[
\text{Load factor:} \quad \frac{n}{N}
\]

- High load factor (definitely) causes many collisions (large buckets).
  Low load factor - waste of memory space.
  *Good compromise*: Load factor around \( 3/4 \).

- Choose \( N \) to be a prime number around \((4/3)n\).

- If load factor gets too high or too low, **re-hash** (amortised analysis similar to *dynamic arrays*).
JVC and HashMap

- No duplicate keys.
- Will hash many different types of key.
- **User can specify** - initial capacity (def. N=16), load factor (def. 3/4).
- *Dynamic* Hash table - “re-hash” takes place frequently behind scenes.
- Different hash functions for different key domains. For **String**, uses polynomial hash code with \( a = 31 \).
- Hashtable is more-or-less identical.
Reading and Resources

- If you have [GT]: The “Maps and Dictionaries” chapter.
- If you have [CLRS]: The “Hash tables” chapter.
- Two nice exercises on Lecture Note 4 (handed out).