Implementing Dictionaries

Many applications require a dynamic set that supports dictionary-type operations such as Insert, Delete, and Search. E.g., a symbol table in a compiler, cache memory.

With an ordinary array, we store an element k at position k of the array. Given a key k, we find the element by looking in the kth position of the array.

This is known as **direct addressing**. Direct addressing is applicable when we can afford to allocate an array with one position for every possible key.

Direct-address tables

Each element has a key drawn from a universe U={0,1,…,m-1} and no two keys are equal.

Direct-addressing works well when the universe U of keys is reasonably small.

Approach 1: Direct-Addressing

Direct-Address Table: Assume U = {0, 1, 2, ..., m}.

- **Insert(x)**: T[key[x]] := x
- **Search(k)**: return T[k]
- **Delete(x)**: T[key[x]] := NIL

Running Times: (assume n elements in table)

- **Insert(x)**: O(1) time
- **Search(k)**: O(1) time
- **Delete(x)**: O(1) time

Great running time! (can't do any better)

Space Usage:

- O(m) space always!
- good if n = (m)
- bad if n << m

Approach 2: Hashing

Hashing

- Hash table (a 1D array) H[0..m-1], where m << |U| and m is prime
- hash function h is a deterministic mapping of keys to indices in H
  h : U \rightarrow (0, 1, ..., m-1)

Problem: there will be some collisions; that is, h will map some keys to the same position in H (h(k₁) = h(k₂) for k₁ ≠ k₂).

Different methods of resolving collisions:

1. chaining: use a general hash function and put all elements that hash to the same location in a linked list at that location (as is done in bucket sort).
2. open addressing: use a general hash function as in chaining, and then increment the original position until an empty slot (or the element you are looking for) is found. One index position for each element in table.

Hash Tables

- U is the "key space", the set of all possible keys
- K ⊆ U is the set of keys being used

A hash table is an ordinary array, but it typically uses a size proportional to the number of keys to be stored, not to the key space.

Use a hash table when K is small relative to U.

IDEA: Given a key, k, don't just use k as the index into the array. Instead, compute a function of k, and use that value to index into the array. We call this function a hash function.

Goals:

- fast implementation of all operations -- O(1) time
- space efficient data structure -- O(n) space if n elements in dictionary

Hash Functions

- The mapping of keys to indexes of a hash table is called a hash function

Purpose of a hash function is to translate an extremely large key space into a reasonably small range of integers, i.e., to map each key k in our dictionary to a position in the hash table.

- An essential requirement of the hash function is to map equal keys to equal indexes
- A "good" hash function minimizes the probability of collisions
Choosing Hash Functions

Ideally, a hash function satisfies the Simple Uniform Hashing Assumption.

Simple Uniform Hashing (SUH): Any key is equally likely to hash to any location (index) in hash table.

Choosing Hash Functions

Realistically, it’s usually not possible to satisfy SUH requirement because we don’t know in advance the probability distribution the keys are drawn from.

We often use heuristics based on the domain of the keys to create a hash function that performs well.

Hash functions assume that keys are natural numbers. When they’re not, have to find a way to interpret them as natural numbers.

Hash-Code Maps

1) Component sum: for numeric types with more than 32 bits, we can add the 32-bit components, i.e., sum the high-order bits with the low-order bits. Integer result is the hash code.

Hash-Code Maps

2) Polynomial accumulation: for strings of a natural language, combine the character values (ASCII or Unicode) \(a_0, a_1, ... a_n\) by viewing them as the coefficients of a polynomial:

\[ a_0 + a_1 x + a_2 x^2 + ... + a_n x^{n-1} \]

Common hash codes (cont.):
Example: Interpret a character string as an integer expressed in some radix notation. Suppose the string is CLRS:
- ASCII values: C = 67, L = 76, R = 82, S = 83
- There are 128 basic ASCII values.
- So interpret CLRS as \((67 \times 128^3) + (76 \times 128^2) + (82 \times 128) + (83 \times 128) = 145,794,947\)

Compression Maps

Normally, the range of possible hash codes generated for a set of keys will exceed the range of the array. Thus, we need a way to map this integer into the range \([0, m-1]\).

Division method: Take the integer produced as a hash code and mod it by the table size.

\[ h(k) = k \mod m \]

- The table size \(m\) is usually chosen as a prime number to help “spread out” the distribution of hashed values.
- The prime number should not be close to an exact power of 2.

Compression Maps

Multiplication method:
1. Choose constant \(A\) in range \(0 < A < 1\)
2. Multiply key \(k\) by \(A\)
3. Extract the fractional part of \(kA\)
4. Multiply the fractional part by \(m\)
5. Truncate (take the floor of the result)

Advantages of this method are that the value of \(m\) is not critical (need not be a prime number).

Disadvantage is that it takes longer to compute.
Collision Resolution by Chaining

**Running Times:** (assume \( n \) elements in list)
- **Insert(x):** \( O(1) \) time to insert node at head of list
- **Search(k):** Proportional to length of list \( T[h(k)] \)
- **Delete(x):** Proportional to length of list \( T[h(k)] \)

**Hash function**

\[ h(x) = n \mod m \]

- Every key maps into \( H \).

**Efficiency of search with chaining**

Depends on the lengths of the linked lists, which depend on the table size, the number of dictionary elements, and the quality of the hash function.

If hash function distributes \( n \) keys among \( m \) cells of the hash table about evenly, each list will have about \( n/m \) keys.

Ratio \( \alpha = n/m \) is called the load factor.

The average number of pointers inspected in unsuccessful searches is \( 1 + \alpha \) and in successful searches is \( 1 + \alpha/2 \).

**Average-Case Analysis for Search(x) with Collision Resolution by Chaining**

Assuming simple uniform hashing:
- Let \( H = \{0...m\} \)
- \( n \) be the number of elements currently in \( H \)
- \( \alpha = n/m \) (the load factor of \( H \))

Average search time:
- \( t(1) \) to compute \( h(x) \) + average time to examine list \( H[h(x)] \)
- For \( x \)
  - \( t(1) + t(\text{average length of list}) \)
  - \( t(1) + t(n/m) \)
  - \( t(1 + \alpha) \)

**Collision Resolution by Open Addressing**

Alternative to chaining for handling collisions.

**Main Idea:**
- All elements are stored in the hash table array itself. No linked lists.
- Each array position contains either a key or NIL.
- Idea: Compute a hash function as in the chaining scheme, but if collision occurs, successively index into (i.e., probe) \( H \) until we find \( x \), or an empty slot for \( x \).

The sequence in which slots are probed depends on both the key of the element and the probe increment.
Collision Resolution by Open Addressing

In this method, the hash function includes the probe number (i.e., how many attempts have been made to find a slot for this key) as an argument.

- The probe sequence for key \( k = h(k,0), h(k,1), ..., h(k,m-1) \)

In the worst case, every slot in table will be examined, so stop either when the item with key \( k \) is found or when a slot with NIL is found.

What might happen if we delete elements from \( H \)? Need to handle deletions by giving slot different status. Not straight-forward as in chaining.

Open Addressing with Linear Probing

Linear Probing: Simplest rehashing function (e.g., + 1 each probe).

The \( i \)th probe \( h(k,i) \) is

\[
h(k,i) = (h(k) + ci + yi^2) \mod m
\]

- \( h(k) \) is ordinary hashing function, tells where to start the search.
- \( c_1, c_2 \) are constants
- \( yi^2 \) is a polynomial polynomial \( i \) (the probe number).

How many distinct probe sequences are there? \( m \)

- each starting point gives a probe sequence
- there are \( m \) starting points
- therefore, not uniform (need \( m! \) probe sequences)

Collision Resolution by Open Addressing

Ideally, we have:

Note: This is different from simple uniform hashing

- simple uniform hashing: each key hashes to each of \( m \) slots with probability \( 1/m \).
- uniform hashing: each key hashes to each of \( m! \) probe sequences with probability \( 1/m \).

Problem: this is hard to implement, so we approximate it with techniques that guarantee the probe sequence is a permutation of \( (0,1,\ldots,m-1) \).

Open Addressing with Quadratic Probing

Quadratic Probing: the \( i \)th probe \( h(k,i) \) is

\[
h(k,i) = (h(k) + ci + yi) \mod m
\]

- \( c_1, c_2 \) are constants
- \( h(k) \) is ordinary hashing function, tells where to start the search.
- later probes are offset by an amount quadratic in \( i \) (the probe number).

How many distinct probe sequences are there? \( m \)

- each starting point gives a probe sequence
- there are \( m \) starting points
- therefore, not uniform (need \( m! \) probe sequences)

Collision Resolution by Open Addressing

Linear Probing Example

- Insert keys: 18 41 22 44 59 32 31 73 (in that order)

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
 18 & 41 & 59 & 32 & 22 & 31 & 44 & 73 \\
\end{array}
\]

How many collisions occur in this case?

- If a collision occurs, when \( j = h(k) \), try \( j + 1 \mod m \), then \( j + 2 \mod m \), and so on. When an empty position is found the item is inserted.
- Linear probing is easy to implement, but leads to clustering (long run of occupied slots). Clusters can lead to poor performance, both for inserting and finding keys.

Quadratic Probing

- Insert keys: 18 41 22 44 59 32 31 73 (in that order)

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
 18 & 41 & 59 & 32 & 22 & 31 & 44 & 73 \\
\end{array}
\]

How many collisions occur in this case?

\[
h(k,i) = (h(k) + ci + yi^2) \mod m
\]
Open Addressing with Double Hashing

Double Hashing is the ith probe is:

\[ h_i(k) = (h_1(k) + i h_2(k)) \mod m \]

- \( h_1(k) \) is an ordinary hash function that tells where to start the search.
- \( h_2(k) \) is an ordinary hash function that gives offset for subsequent probes.

Notes:
- To make the probe sequence hit all slots in H, we must have \( h_2(k) \) relatively prime to m.
- How many distinct probe sequences are there? \( m^2 \)
  - There are m starting points and each possible \((h_1(k), h_2(k))\) pair yields a distinct probe sequence.
  - Starting point and offset can vary independently.
- Better, but still not uniform.

Double Hashing Example

- \( h_1(k) = k \mod m \)
- \( h_2(k) = k \mod (m - 1) \)
- The ith probe is:

\[ h(k, i) = (h_1(k) + h_2(k) \cdot i) \mod m \]

- \( h_2 \) is an offset to add.

Insert keys: 18 41 22 44 59 32 31 73 (in that order)

<table>
<thead>
<tr>
<th>Key</th>
<th>Probes</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>41</td>
<td>1</td>
</tr>
<tr>
<td>22</td>
<td>2</td>
</tr>
<tr>
<td>44</td>
<td>4</td>
</tr>
<tr>
<td>59</td>
<td>5</td>
</tr>
<tr>
<td>32</td>
<td>6</td>
</tr>
<tr>
<td>31</td>
<td>7</td>
</tr>
<tr>
<td>73</td>
<td>8</td>
</tr>
</tbody>
</table>

Analyzing Open Addressing

Question: How large can the load factor be in a table using open addressing?

Explain your answer.

Analyzing Open Addressing

Assume: uniform hashing (all m! probe sequences equally likely).

- \( \alpha = n/m \) (load factor), so we need \( \alpha \leq 1 \) (table cannot be overfilled).

Theorem: If \( \alpha < 1 \), then the expected number of probes in search is \( \leq 1/(1 - \alpha) \) assuming UH applies.

Proof: In a search when \( \alpha < 1 \) (table not full), some number of probes access occupied slots and the last probe accesses an empty slot or the key being searched for.

Approach 1: Linked Lists

Linked List Implementation (Chaining)
- Insert(x): add x at head of list
- Search(k): start at head and scan list
- Delete(x): start at head, scan list, and then delete if found

Running Time: assume n elements in list
- Search(k): worst-case - element at end of list: n operations
- Average-case - element at middle of list: \( n/2 \) ops
- Best-case - element at head of list: 1 op
- Delete(x): same as searching

We'd like O(1) time for all operations, we have O(n) for two.

Space Usage: O(n) space - very space efficient.
Approach 2: Direct-Addressing

Direct-Address Table: Assume $U = \{0, 1, 2, \ldots, m\}$.
The data structure is an array $T[0..m]$.
- Insert($x$): $T[key[x]] = x$
- Search($k$): return $T[k]$
- Delete($x$): $T[key[x]] = \text{NIL}$

Running Times: (assume $n$ elements in list)
- Insert($x$): $O(1)$ time
- Search($x$): $O(1)$ time
- Delete($x$): $O(1)$ time
Great running time!

Space Usage: (assume $n$ elements in list)
- $O(m)$ space always
- good if $n = \Theta(m)$
- bad if $n << m$

Chaining wins!!

In general, a good hash function with an array of linked lists outperforms the open addressing approach.

However, you may have to tailor the hash function to meet the requirements of your data set by experimentation. No one hash function is good for all data sets.

Chained hash tables remain effective even when the number of table entries $n$ is much higher than the number of slots.

For example, a chained hash table with 1000 slots and 10,000 stored keys (load factor 10) is five to ten times slower than a 10,000-slot table (load factor 1); but still 1000 times faster than a plain sequential list, and possibly even faster than a balanced search tree.

End of Hash Table Lecture