CPSC 131 Data Structures

Instructor: Dr. Doina Bein

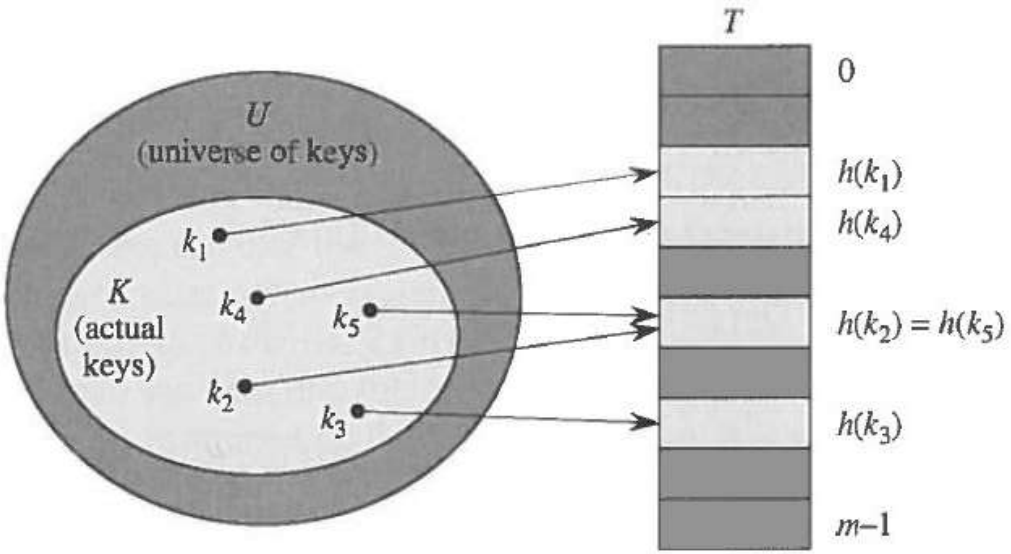
Hash Tables

- A hash table is an ADT that
  - is an *indexed sequence* or *array (Bucket Array)*, i.e
    - a list of items for which the order does matter (sequence)
    - and supports directly addressing (i.e. accessing any item is done in O(1) steps) (indexed)
  - that supports only the dictionary operations Insert, Search and Delete
- Each element has a key, but instead of using the key as the index of the array directly, the index is computed from the key using a so-called *hash function*
- The domain of values for a hash function is a hash table

Hashing

- *Hashing*: If the universe (i.e. the set of possible keys) is large then we store only a relatively small subset of keys, K. Then an element with key k is stored in slot h(k) where *h* is a *hash function*:

$$h: U \rightarrow K$$



element with key k *hashes* to slot h(k)
h(k) is the *hash value* of key k

Figure 11.2 Using a hash function h to map keys to hash-table slots. Keys k_2 and k_5 map to the same slot, so they collide.

Hash Functions

- A *hash function* is any function that can be used to map digital data of arbitrary size to digital data of fixed size, with slight differences in input data producing very big differences in output data
- The hash function is used to map the search key to an *index*; the index gives the place in the hash table where the corresponding record should be stored
- The values returned by a hash function are called *hash values*, *hash codes*, *hash sums*, or simply *hashes*
- Thus, the hash function only hints at the record's location — it tells where one should start looking for it.
- Still, in a half-full table, a good hash function will typically narrow the search down to only one or two entries.

Bucket Array

- Is an array A of size N, in which the element A[k] stores all the (key, value) pairs having the same key k

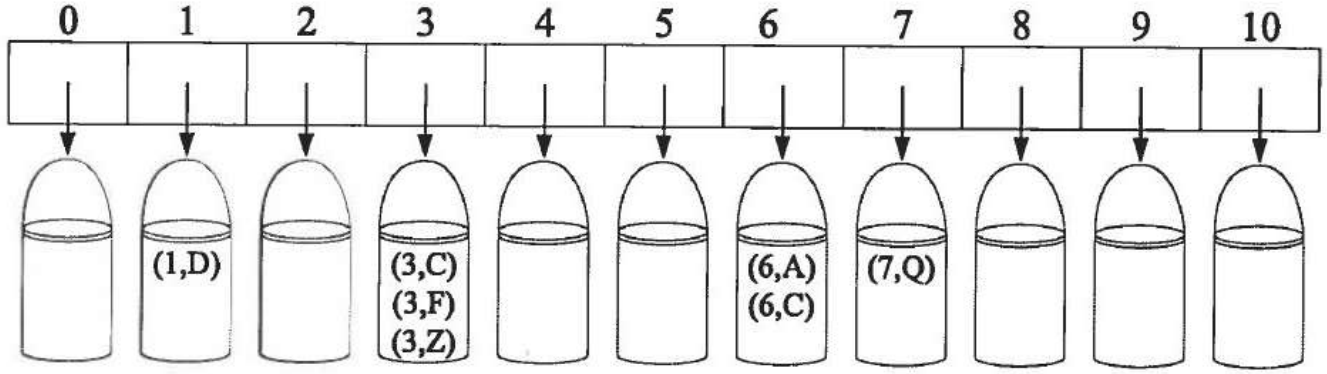


Figure 9.2: A bucket array of size 11 for the entries (1,D), (3,C), (3,F), (3,Z), (6,A), (6,C), and (7,Q).

- It is used in conjunction with a good hash function to store (key, value) pairs using as index the hash value of the key and not the key
- The hash function $h: U \rightarrow [0, 1, \ldots N-1]$
- A hash function is *good* if it encounters the minimum number of collisions

Hash Codes

- A *hash code* is the integer value of a key
- The standard include file `<limits>` contained a templated class `numeric_limits` that stores the number of bits for a base type; given a base type T, `numeric_limits<T>.digits` gives the number of digits used by a variable of type T (e.g. **char**, **int**, **float**)
- Example:

`numeric_limits<int>.digits` displays the number of digits used for type **int**

- For any data type X that is represented using at most as many bits as integer hash codes, we can consider the integer representation of its bits as a hash code for X

- For a data types that need more bits, we can consider the binary representation of an object x of that type as a *k*-tuple $(x_0, x_1, \ldots, x_{k-1})$ of integers and take the hash code to be $\sum_{i=0}^{k-1} x_i$
- Example:
  - For a long integer that has a representation twice as long as type **int**, sum the integer representation of the high-order bits with the integer representation of the low-order bits
  - for a floating point number, we can sum up its mantissa and exponent as long integers, and then apply a hash code for long integers to the result

- Summing will not work for strings
- Take into consideration the position of each x_i by computing the hash code using a nonzero constant $a \neq 0$:

$$\sum_{i=0}^{k-1} \left(x_i \cdot a^{k-1-i}\right)$$

- This is called *polynomial hash code* and can be re-written using Horner's rule as

$$x_{k-1} + a\left(x_{k-2} + a\left(x_{k-3} + \cdots + a\left(x_2 + a(x_1 + ax_0)\right) \ldots\right)\right)$$

- Experimental studies showed that 33, 37, 39 and 41 are good choices for the nonzero constant a when working with character strings that are English words
- For sake of speed, some implementations of the hash code apply the polynomial hash function to only a fraction of the characters in long strings

Compressing Functions

- Since the hash code generally has a large integer value, we use a compression function to map the hash code into an integer in the range [0, N-1].
- The simplest method is to apply the mod operator:

$$h(k) = |k| mod\ N$$

- This is called *division method*
- If N is prime then the distribution of hash values is "spread out"
- If N is not prime, then there is a high likelihood that patterns in the distribution of keys will be repeated in the distribution of hash codes, thereby causing collisions

Collision

- If two different keys hash to the same hash value then we have a *collision*
- There are effective ways to resolve collision, the most popular are *separate chaining* and *open addressing*
- Separate chaining is the most frequently used

Separate Chaining

- All the elements that share the same hash key are placed in a linked list.
- An entry of the hash table is either NULL or contains the pointer to the first element of the linked list.

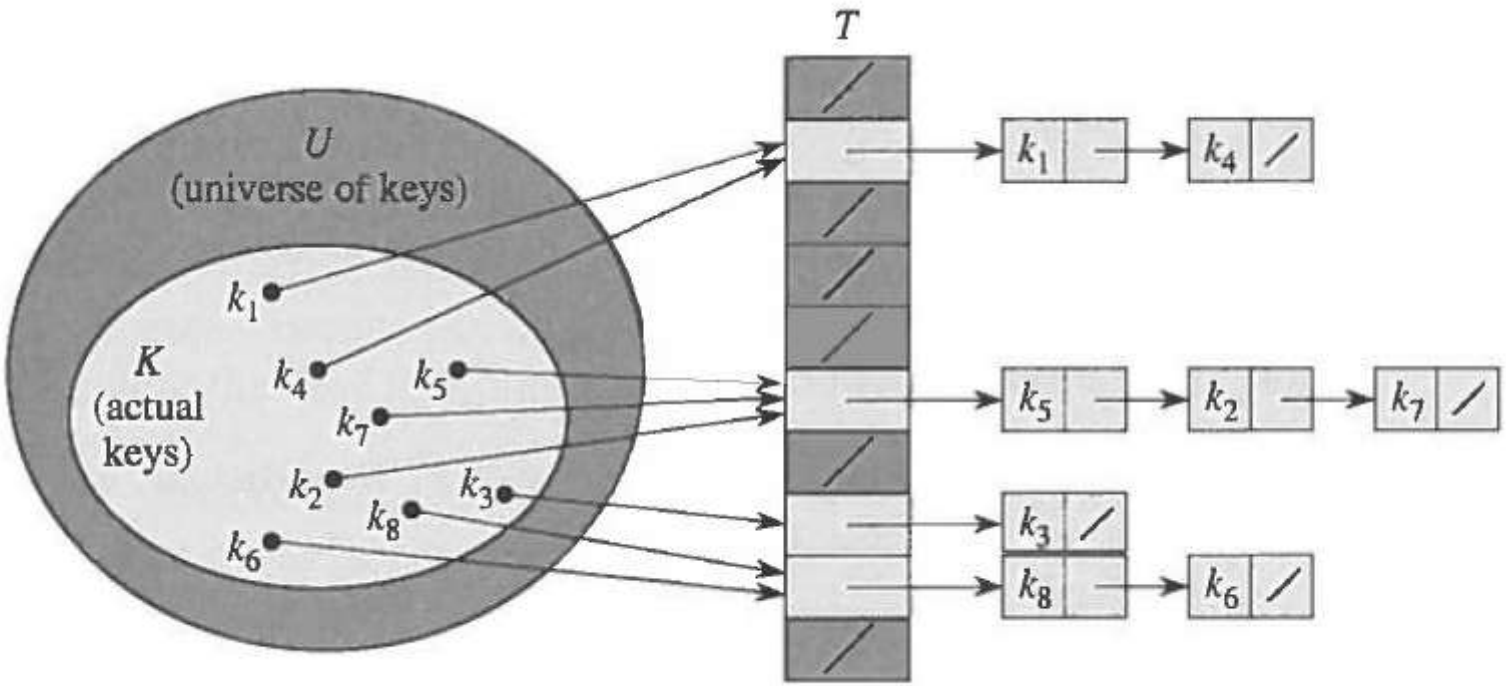


Figure 11.3 Collision resolution by chaining. Each hash-table slot $T[j]$ contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$.

Open Addressing

- All elements are stored in the hash table itself: each entry contains either an element of the dynamic set or NULL.
- There are no lists and no elements stored outside the table, as there are in chaining.
- A hash table can fill-up so that no further insertions can be made
- Let *n* be the number of elements stored in the hash table and *N* be the number of slots. We define the load factor of the hash table to be $\alpha = {}^{n}/_{N}$
- The load factor α can never exceed 1.
- We could store the linked lists for chaining inside the hash table, in the unused hash-table slots
- Instead of using pointers, we compute the sequence of slots to be examined.

- By not using pointers, we save memory that is used to increase the number of slots in the hash table.
- Insertion: we successively examine, or probe, the hash table until we find an empty slot in which we put the key.
- The sequence of indices to be explored is not $0,1,\ldots,N-1$ which requires $\Theta(\mathrm{N})$ search time but depends on the key being inserted.
- We extend the hash function to include the probe number (starting at 0) as a second input value:

$$h: U \times \{0,1,2,\cdots,N-1\} \rightarrow \{0,1,2,\cdots,N-1\}$$

- We require that for every key k, the probe sequence $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ be a permutation of $\langle 0,1,2,\cdots,N-1\rangle$ so that every hash-table position is eventually considered as a slot for a new key as the table fills up.

- **Insertion**: insert the element with the key `k` into the hash table

```
HASH-INSERT(T,k)
i = 0
repeat
   j = h(k,i)
   if T[j] == NULL
     then T[j] = element with key k
          return j
     else i ← i + 1
  until i == N
Error "hash table overflow"
```

- **Searching**: for the element with key `k` by probing the same sequence of slots that the insertion algorithm examined when key `k` was inserted.

```
HASH-SEARCH(T,k)
i = 0
repeat
   j = h(k,i)
   if T[j] == k
     then return j
     i = i + 1
  until T[j] == NULL or i == N
return NULL
```

- **Deletion**: we cannot simply delete the slot by marking it NULL, otherwise it will be impossible to retrieve any key `k` during whose insertion we had probed slot `i` and found it occupied
  - One solution: mark the slot by AVAILABLE instead of NULL and modify procedure HASH-INSERT to treat AVAILABLE as NULL. Procedure HASH-SEARCH remains unchanged.
  - Search time is no longer dependent on the load factor α
- To compute the probe sequences: linear probing, quadratic probing and double hashing.

Type of cells in a hash table

- Three types:
  - Occupied: the cell has an item
  - An *empty-since-start cell* has been empty since the hash table was created.
  - An *empty-after-removal cell* had an item (or successive items) that were deleted sometimes in the past and that caused the cell now to be empty.
- The distinction between the empty-since-start and empty-after-removal is important for search: during searching we stop only when we find empty-since-start cells, and we continue searching when we find empty-after-removal cell
- There is no distinction between the empty-since-start and empty-after-removal for insertion: we insert the item in that cell in either case

Linear Probing

- Given an ordinary hash function $h'\colon U \to \{0, 1, 2, \cdots, N-1\}$ and an element with the key k to be inserted in the hash table, in linear probing the first slot probed is T[h'(k)]
- We next probe T[h'(k)+1], T[h'(k)+2], and so on.
- Then we wrap around to slots T[0], T[1], …, T[h'(k)-1].
- Uses the hash function:

$h(k,p) = (h'(k) + p) mod\ N$ for $p = 0, 1, \ldots, N-1$.

- Because the initial probe determines the entire probe sequence, there are only N distinct probe sequences.
- Problem: ***primary clustering***. Long runs of occupied slots build up, increasing the average search time.

Searching with Linear Probing

- Consider a hash table A of size N that uses linear probing
- find(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty-since-start cell is found, or
    - N cells have been unsuccessfully probed

```
Algorithm find(k)
    i ← h(k)
    p ← 0
    repeat
        c ← A[i]
        if c = ∅-since-start
            return false
        else if c.key () = k
            return c.value()
        else
            i ← (i + 1) mod N
        p ← p + 1
    until p = N
    return false
```

Inserts using linear probing

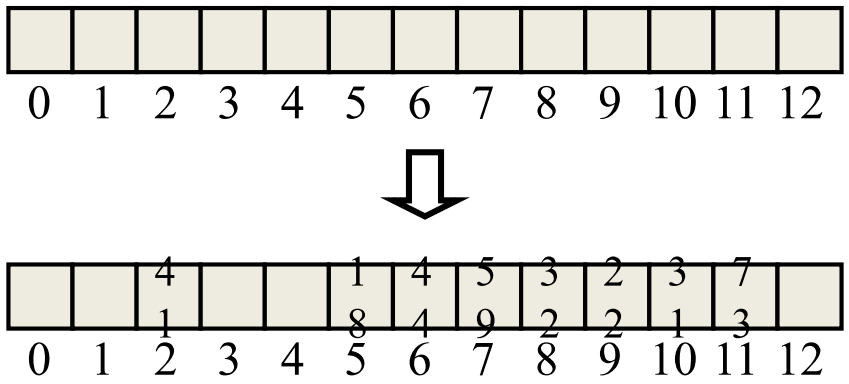
- Use the item's key to determine the initial bucket
- Linearly probes (or checks) each bucket, and inserts the item in the next empty bucket (the empty kind doesn't matter).
- If the probing reaches the last bucket, the probing continues at bucket 0.
- The insert algorithm returns true if the item was inserted, and returns false if all buckets are occupied.

```
Algorithm insert(k)
    i ← h(k)
    p ← 0
    repeat
      c ← A[i]
      if c = ∅
            A[i] = k
            return true
       else
            i ← (i + 1) mod N
                  p ← p + 1
      until       p = N
    return false
```

- Example:

  $h(x, p) = x \bmod 13 + p$

  Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



Remove using linear probing

- Use the item's key to determine the initial bucket
- Linearly probes (or checks) each bucket until either a matching item is found, an empty-since-start bucket is found, or all buckets have been probed.
- If the item is found, the item is removed, and the bucket is marked as empty-after-removal.
- Returns true if the item was found and removed, and returns false if the item was not found.

```
Algorithm remove(k)
    i ← h(k)
    p ← 0
    repeat
      c ← A[i]
      if c = ∅-since-start
            return false
       else if c.key () = k
             c = ∅-after-removal
            return true
      else
            i ← (i + 1) mod N
                  p ← p + 1
      until       p = N
    return false
```

Quadratic Probing

- Given an ordinary hash function $h'\colon U \to \{0, 1, 2, \cdots, N-1\}$ and an element with the key `k` to be inserted in the hash table, in quadratic probing the first slot probed is `T[h'(k)]`
- Later positions probed are offset by amounts that depend in a quadratic manner on the probe number i
- Uses the hash function:

$h(k,p) = (h'(k) + c_1 p + c_2 p^2) mod\ N$ for for $p = 0, 1, \ldots, N-1$, c_1 and c_2 are non-zero constants

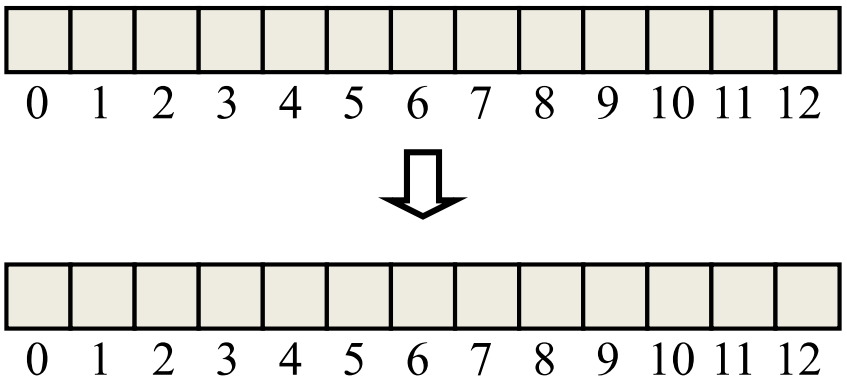
- Because the initial probe determines the entire probe sequence, there are only `m` distinct probe sequences.
- Much better than linear probing, but suffers from ***secondary clustering***: if two keys k_1 and k_2 have the same initial probe position, then their probe sequences are the same since $h(k_1, 0) = h(k_2, 0)$ implies that $h(k_1, p) = h(k_2, p)$

- Example:

  $h(x, p) = (x \bmod 13 + c1*p + c2* p*p) \bmod 13$

  Consider c1 = c2 = 1

  Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



Searching with Quadratic Probing

- Consider a hash table A of size N that uses quadratic probing
- find(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty-since-start cell is found, or
    - N cells have been unsuccessfully probed

```
Algorithm find(k)
    i ← h(k)
    p ← 0
    repeat
        c ← A[i]
        if c = ∅-since-start
            return false
        else if c.key () = k
            return c.value()
        else
            i ← (h(k)+c1*p+c2*p*p)
mod N
        p ← p + 1
    until p = N
    return false
```

Inserts using quadratic probing

- Use the item's key to determine the initial bucket
- Quadratically probes (or checks) each bucket, and inserts the item in the next empty bucket (the empty kind doesn't matter).
- If the probing reaches the value of N, then return false.
- The insert algorithm returns true if the item was inserted, and returns false if all searched buckets are occupied.
- Please note that in some cases, same buckets may be searched multiple times and some buckets may not be searched at all

```
Algorithm insert(k)
    i ← h(k)
    p ← 0
    repeat
      c ← A[i]
      if c = ∅
      A[i] = k
      return true
      else
      i ← (h(k)+c1*p+c2*p*p) mod N
            p ← p + 1
      until       p = N
    return false
```

Remove using quadratic probing

- Use the item's key to determine the initial bucket
- Quadratically probes (or checks) each bucket until either a matching item is found, an empty-since-start bucket is found, or probing reaches the value of N.
- If the item is found, the item is removed, and the bucket is marked as empty-after-removal.
- Returns true if the item was found and removed, and returns false if the item was not found.

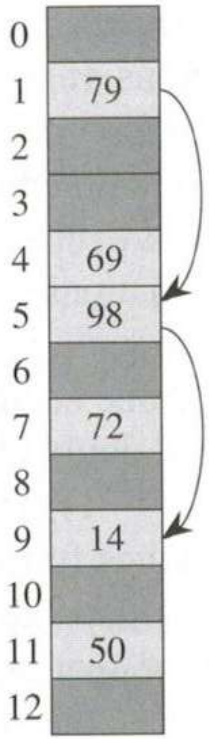
```
Algorithm remove(k)
    i ← h(k)
    p ← 0
    repeat
      c ← A[i]
      if c = ∅-since-start
      return false
      else if c.key () = k
      c = ∅-after-removal
      return true
      else
      i ← (h(k)+c1*p+c2*p*p) mod N
            p ← p + 1
      until       p = N
    return false
```

Double Hashing

- Uses a hash function of the form

$h(k,i) = (h_1(k) + i \cdot h_2(k)) mod\ N$ where h_1 and h_2 are auxiliary hash functions.

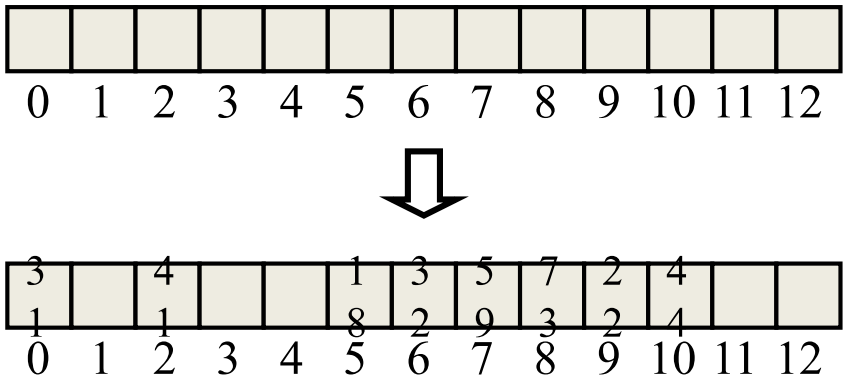
- Initially $T[h_1(k)]$ is probed
- Successive probe positions are offset from previous positions by the amount $h_2(k)$ modulo N. Example:



- Hash table of size 13 with $h_1(k) = k\ mod\ 13$ and $h_2(k) = 1 + (k\ mod\ 11)$.
- Key 14 needs to be inserted: $h(14,0) = h_1(14) = 1$, but slot 1 is occupied.
- $h(14,1) = h_1(14) + 1 \cdot h_2(14) = 1 + 4 = 5$ but slot 5 is occupied.
- $h(14,2) = h_1(14) + 2 \cdot h_2(14) = 1 + 2 \cdot 4 = 9$ and slot 9 is free
- Key 14 is inserted into the empty slot 9

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - k \bmod 7$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		



Direct Hashing

- Uses the item's key as the bucket index.
- Example: If the key is 937, the index is 937.
- A hash table with a direct hash function is called *a direct access table*.
- Given a *key*, a direct access table search algorithm returns the item at index *key* if the bucket is not empty, and returns null (indicating item not found) if empty.
- Advantage: no collisions
- Limitations:
  - All keys must be non-negative integers, but for some applications keys may be negative.
  - The hash table's size equals the largest key value plus 1, which may be very large.

Performance of Hashing

- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time; the worst case occurs when all the keys inserted into the map collide
- The load factor $\alpha = n / N$ affects the performance of a hash table
- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is $1 / (1 - \alpha)$
- The expected running time of search, insert, delete in a hash table is $O(1)$
- In practice, hashing is very fast provided the load factor is not close to 100%
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches

Rehashing

- Hash Table may get full
  - No more insertions possible
- Hash table may get too full
  - Insertions, deletions, search take longer time
- Solution: Rehash
  - Build another table that is twice as big and has a new hash function
  - Move all elements from smaller table to bigger table
- Cost of Rehashing = O(N)
  - But happens only when table is close to full
  - Close to full = table is X percent full, where X is a tunable parameter