Hashing

Hash Tables

- We'll discuss the *hash table* ADT which supports only a subset of the operations allowed by binary search trees.
- The implementation of hash tables is called **hashing**.
- Hashing is a technique used for performing insertions, deletions and finds in constant average time (i.e. O(1))
- This data structure, however, is not efficient in operations that require any ordering information among the elements, such as findMin, findMax and printing the entire table in sorted order.

General Idea

- The ideal hash table structure is merely an array of some fixed size, containing the items.
- A stored item needs to have a data member, called ***key***, that will be used in computing the index value for the item.
  - Key could be an *integer*, a *string*, etc
  - e.g. a name or Id that is a part of a large employee structure
- The size of the array is *TableSize*.
- The items that are stored in the hash table are indexed by values from *0* to *TableSize – 1*.
- Each key is mapped into some number in the range 0 to *TableSize – 1.*
- The mapping is called a *hash function*.

Hashing

- A hash function maps a value to an index in the table. The function provides access to an element much like an index provides access to an array element.

- Like a binary search tree, a hash table provides an implementation of the Set and Map interfaces.

- A binary search tree can access data stored by value with $O(\log_2 n)$ average search time.

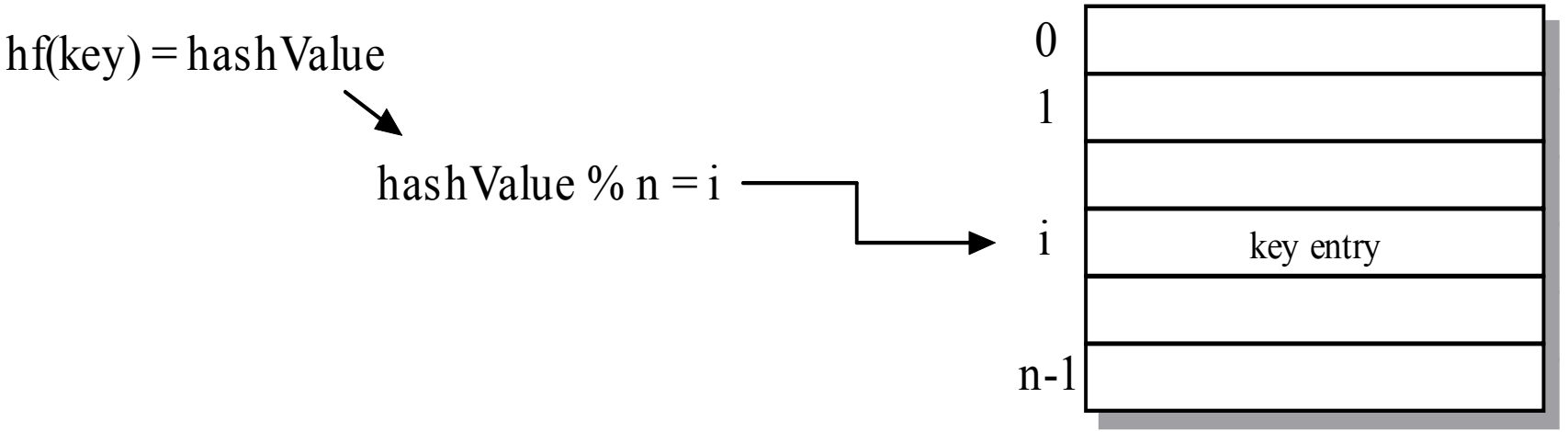
- We would like to design a storage structure that yields O(1) average retrieval time. In this way, access to an item is independent of the number of other items in the collection.

Introduction to Hashing (continued)

- A hash table is an array of references.
  - Associated with the table is a hash function that takes a key as an argument and returns an integer value.
  - By using the remainder after dividing the hash value by the table size, we have a mapping of the key to an index in the table.

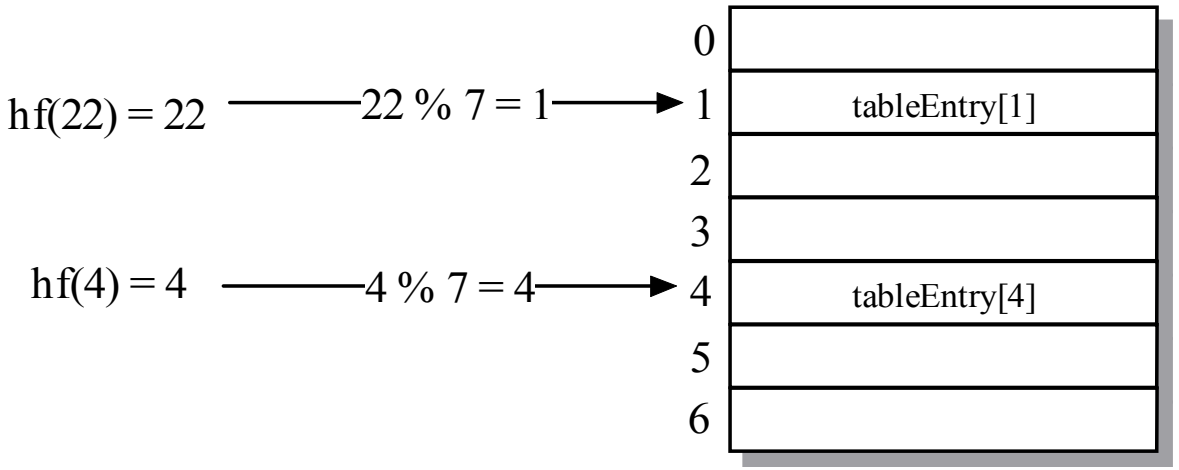
Introduction to Hashing (concluded)

```
Hash Value:          hf(key) = hashValue
HashTable index:     hashValue % n
```



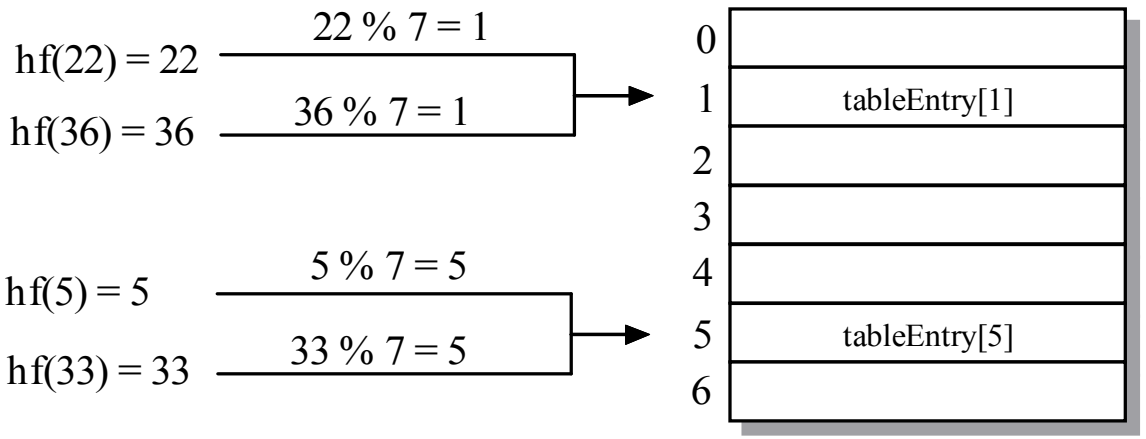
Using a Hash Function

- Consider the hash function
  hf(x) = x, where x is a nonnegative integer (the *identity function*). Assume the table is the array tableEntry with n = 7 elements.



Using a Hash Function (concluded)

- With hash function hf() and table size n, the table index for a key is i = hf(key)%n. Collisions occur for any two keys that differ by a multiple of n.



Designing Hash Functions

- Some general design principles guide the creation of all hash functions.
  - Evaluating a hash function should be efficient.
  - A hash function should produce *uniformly distributed hash values*. This spreads the hash table indices around the table, which helps minimize collisions.

Designing Hash Functions (continued)

- The Java programming language provides a general hashing function with the hashCode() method in the Object superclass.

```
public int hashCode()

{ … }
```

Designing Hash Tables

- When two or more data items hash to the same table index, they cannot occupy the same position in the table.
  - We are left with the option of locating one of the items at another position in the table (*linear probing)* or of redesigning the table to store a sequence of colliding keys at each index (*chaining with separate lists)* .

Hashing

- Open-bucket Hash tables store only one element per position
  - Collision resolution is required
  - Single step collision resolution potentially produces long contiguous sequences of unoccupied locations: problem not so great if different step size applied; Make sure step size is not exact divisor of table size
- Closed-bucket Hash tables may store one or more elements per position

Closed Bucket v Open Bucket

- Use Closed Bucket for unbounded sets and maps
- Closed Bucket Tables nearly always preferable as use dynamic allocation
- Use Open Bucket Tables where memory is scarce or number of entries is bounded

Hash Tables

- Closed bucket hash table
  - Collection of data elements associated with each bucket
- Open bucket hash table
  - Single data element stored in each bucket

Closed Bucket Hash Table

Hfn(keyval1) ⇨ 2

Hfn(keyval2) ⇨ n-1

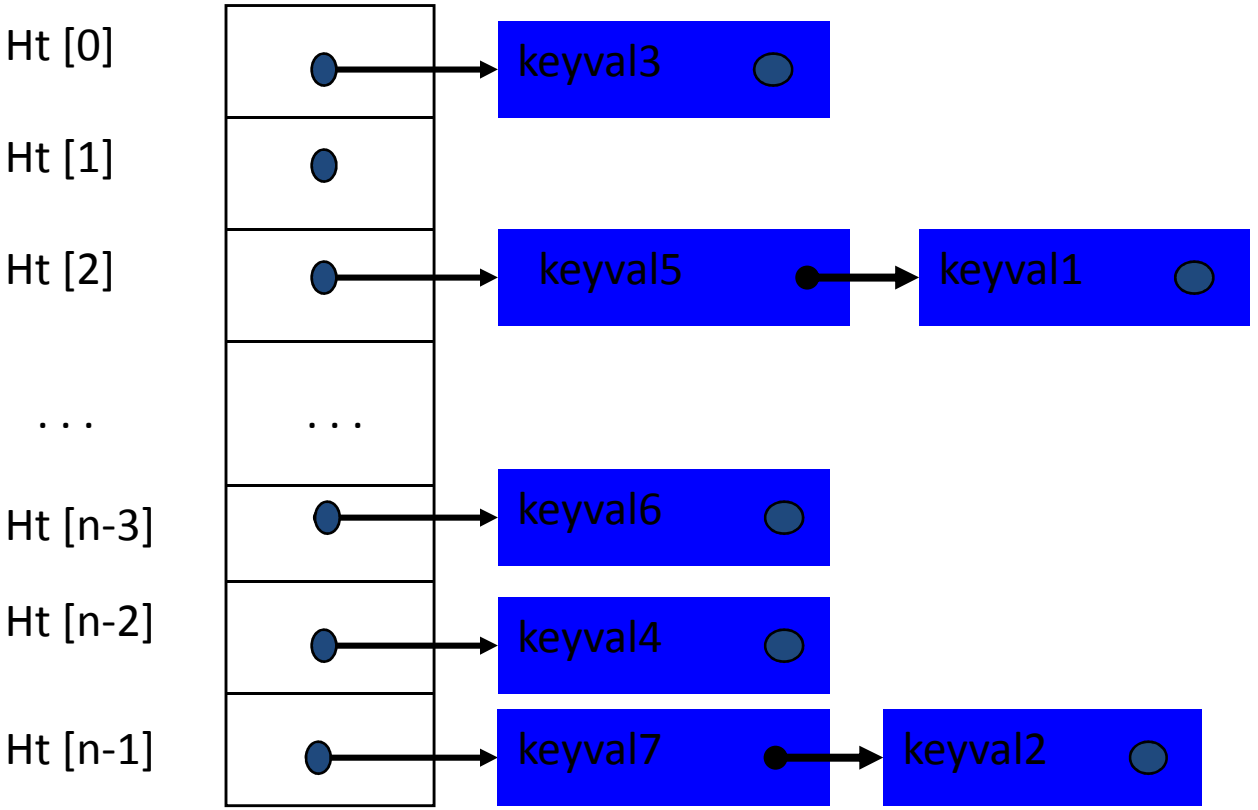
Hfn(keyval3) ⇨ 0

Hfn(keyval4) ⇨ n-2

Hfn(keyval5) ⇨ 2

Hfn(keyval6) ⇨ n-3

Hfn(keyval7) ⇨ n-1



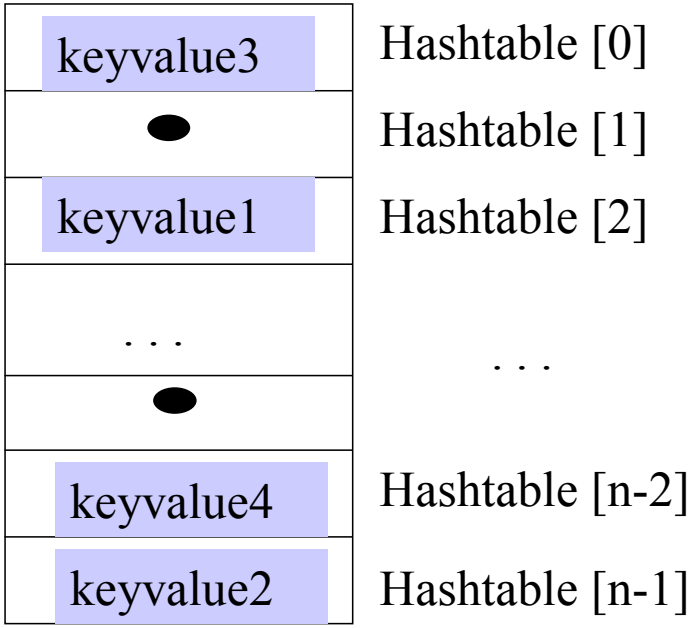
Open Bucket Hash Tables

- Each bucket holds at most 1 item
- Implementation as array of element type

```
String[] hashTable = new String[TABLESIZE];
```

- Needs collision resolution strategy
  - This means that array must be treated as cyclic

Open Bucket Hash Table



One element per bucket

Collision Resolution

- If, when an element is inserted, it hashes to the same value as an already inserted element, then we have a collision and need to resolve it.
  - E.g. For *TableSize* = 17, the keys 18 and 35 hash to the same value
  - 18 mod 17 = 1 and 35 mod 17 = 1

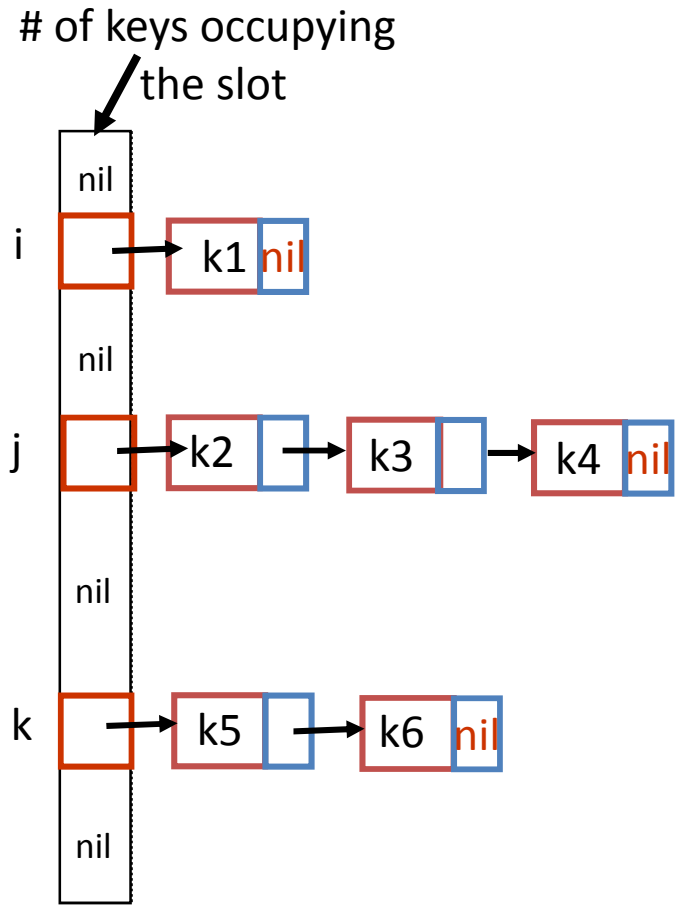
- There are several methods for dealing with this:
  - **Separate chaining**
  - **Open addressing**
    - Linear Probing
    - Quadratic Probing
    - Double Hashing

Separate Chaining

- The idea is to keep a list of all elements that hash to the same value.
  - The array elements are pointers to the first nodes of the lists.
  - A new item is inserted to the front of the list.
- Advantages:
  - Better space utilization for large items.
  - Simple collision handling: searching linked list.
  - Overflow: we can store more items than the hash table size.
  - Deletion is quick and easy: deletion from the linked list.

Separate Chaining

- Each hash table cell holds a pointer to a linked list of records with same hash value (i, j, k in figure)
- Collision: Insert item into linked list
- To Find an item: compute hash value, then do Find on linked list
- Can use a linked-list for Find/Insert/Delete in linked list
- Can also use BSTs: O(log N) time instead of O(N). But lists are usually small – not worth the overhead of BSTs

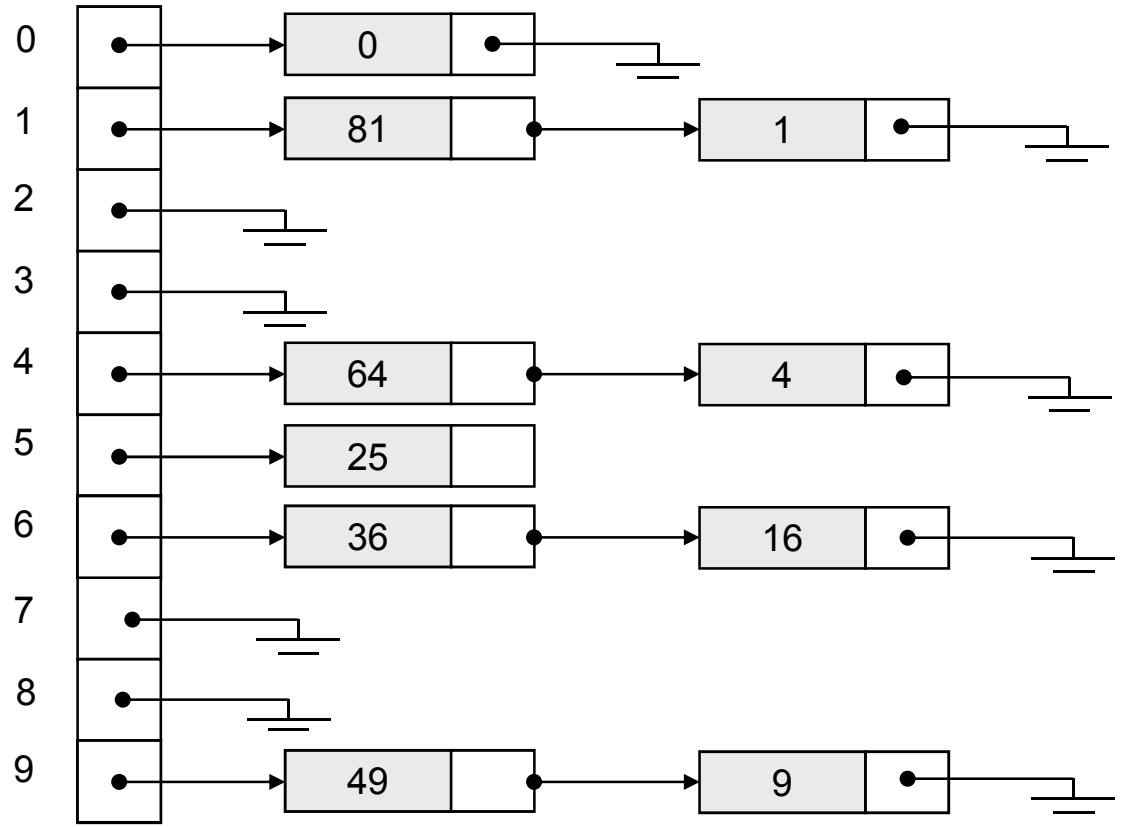


* **Hash(k1) = i**
* **Hash(k2)=Hash(k3)=Hash(k4) = j**
* **Hash(k5)=Hash(k6)=k**

Example

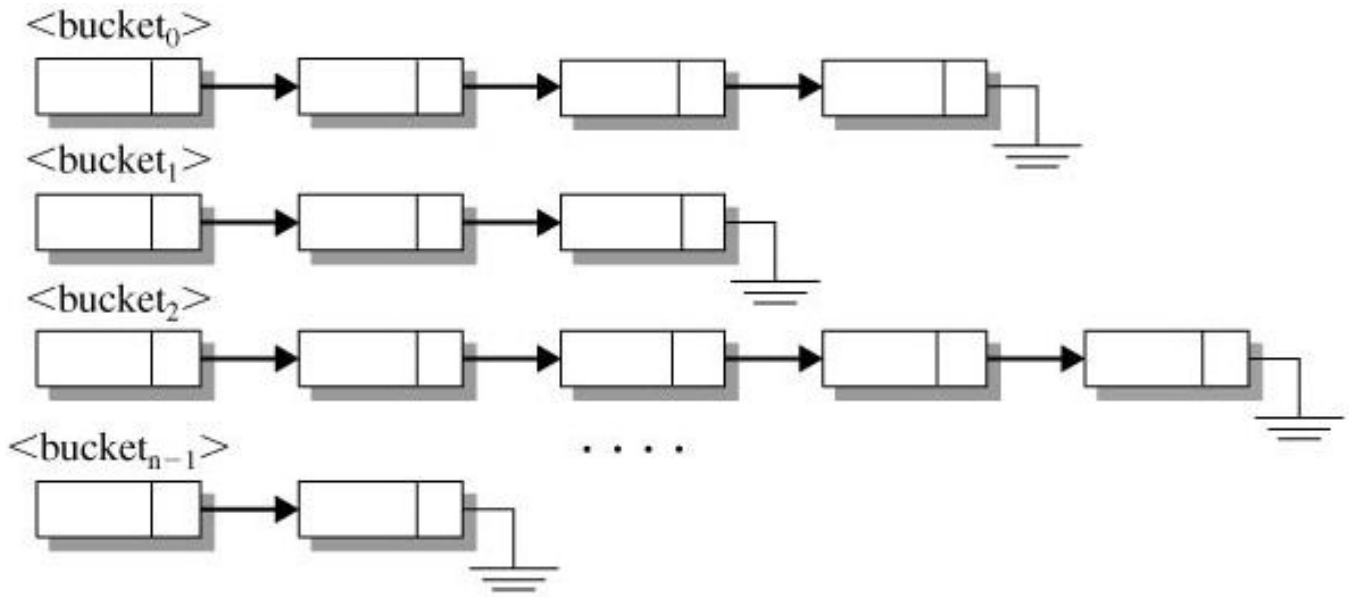
Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

hash(key) = key % 10.



Chaining with Separate Lists

- Chaining with separate lists defines the hash table as an indexed sequence of linked lists. Each list, called a *bucket,* holds a set of items that hash to the same table location.



Chaining with Separate Lists (concluded)

- Chaining with separate lists is generally faster than linear probing since chaining only searches items that hash to the same table location.
- With linear probing, the number of table entries is limited to the table size, whereas the linked lists used in chaining grow as necessary.
- To delete an element, just erase it from the associated list.

Collision Resolution by Open Addressing

- Linked lists can take up a lot of space…
- Open addressing (or probing): When collision occurs, try alternative cells in the array until an empty cell is found
- Given an item X, try cells $h_0(X)$, $h_1(X)$, $h_2(X)$, …, $h_i(X)$
- **$h_i(X) = (Hash(X) + F(i)) \bmod TableSize$**
- Define F(0) = 0
- F is the collision resolution function. **Three possibilities**:
  - Linear: **F(i) = i**
  - Quadratic: **F(i) = i^2**
  - Double Hashing: **$F(i) = i * Hash_2(X)$**

Open Addressing I: Linear Probing

- **Main Idea**: When collision occurs, scan down the array one cell at a time looking for an empty cell

- hi(X) = (Hash(X) + i) mod *TableSize* (i = 0, 1, 2, …)

- Compute hash value and increment until free cell is found

- **In-Class Example**: Insert {18, 19, 20, 29, 30, 31} into empty hash table with *TableSize* = 10 using:
  - (a) separate chaining
  - (b) linear probing

Open Addressing II: Quadratic Probing

- Main Idea: Spread out the search for an empty slot Increment by i^2 instead of I
- **hi(X) = (Hash(X) + i^2) mod *TableSize*** (i = 0, 1, 2, …)
  - No primary clustering but secondary clustering possible

- Example 1: Insert {18, 19, 20, 29, 30, 31} into empty hash table with *TableSize* = 10

- Example 2: Insert {1, 2, 5, 10, 17} with *TableSize* = 16

Open Addressing III: Double Hashing

- Idea: Spread out the search for an empty slot by using a second hash function
  - No primary or secondary clustering

- **hi(X) = (Hash(X) + i*Hash$_2$(X)) mod *TableSize*** for i = 0, 1, 2, …
- **E.g. Hash$_2$(X) = R – (X mod R)**
  - R is a prime smaller than *TableSize*

- Try this example: Insert {18, 19, 20, 29, 30, 31} into empty hash table with *TableSize* = 10 and R = 7

- No clustering but slower than quadratic probing due to Hash2

Hash Table Performance

- The worst case linear probe or chaining with separate lists occurs when all data items hash to the same table location. If the table contains n elements, the search time is O(n), no better than that for the sequential search.

Hash Table Performance

- A good hash function provides a uniform distribution of hash values.
- Hash table performance is measured by using the load factor $\lambda = n/m$, where n is the number of elements in the hash table and m is the number of buckets.
  - For linear probe, $0 \leq \lambda \leq 1$.
  - For chaining with separate lists, it is possible that $\lambda > 1$.

Load Factor

- The *load factor* is the number of elements in a table divided by the table size.
- In a closed bucket hash table too high a load factor means buckets likely to have more than 2 entries
- Too low a load factor means high space consumption.
- Choose a prime number of buckets to avoid problems with patterns in keys

Hash Table Performance (continued)

- Assume that the hash function uniformly distributes indices around the hash table.
  - We can expect $\lambda = n/m$ elements in each bucket.
    - On the average, an unsuccessful search makes λ comparisons before arriving at the end of a list and returning failure.
    - Mathematical analysis shows that the average number of probes for a successful search is approximately $1 + \lambda/2$.

Hash Tables vs Search Trees

- Hash Tables are good if you would only like to perform ONLY Insert/Delete/Find
- Hash Tables are not good if you would also like to get a sorted ordering of the keys
  - Keys are stored arbitrarily in the Hash Table
- Hash Tables use more space than search trees
  - Our rule of thumb: Time versus space tradeoff
- Search Trees support sort/predecessor/successor/min/max operations, which cannot be supported by a Hash Table

Summary

- A hash table is based on an array.
- The range of key values is usually greater than the size of the array.
- A key value is hashed to an array index by a hash function.
- An English-language dictionary is a typical example of a database that can be efficiently handled with a hash table.
- The hashing of a key to an already filled array cell is called a collision.
- Collisions can be handled in two major ways: open addressing and separate chaining.
- In open addressing, data items that hash to a full array cell are placed in another cell in the array.
- In separate chaining, each array element consists of a linked list. All data items hashing to a given array index are inserted in that list.
- We discussed three kinds of open addressing: linear probing, quadratic probing, and double hashing.
- In linear probing the step size is always 1, so if x is the array index calculated by the hash function, the probe goes to x, x+1, x+2, x+3, and so on.
- The number of such steps required to find a specified item is called the probe length.
- In linear probing, contiguous sequences of filled cells appear. These are called primary clusters, and they reduce performance.

In quadratic probing the offset from x is the square of the step number, so the probe goes to x, x+1, x+4, x+9, x+16, and so on.

• Quadratic probing eliminates primary clustering, but suffers from the less severe secondary clustering.

• Secondary clustering occurs because all the keys that hash to the same value follow the same sequence of steps during a probe.

• All keys that hash to the same value follow the same probe sequence because the step size does not depend on the key, but only on the hash value.

• In double hashing the step size depends on the key, and is obtained from a secondary hash function.

•

If the secondary hash function returns a value s in double hashing, the probe goes to x, x+s, x+2s, x+3s, x+4s, and so on, where s depends on the key, but remains constant during the probe.

• The load factor is the ratio of data items in a hash table to the array size.

• The maximum load factor in open addressing should be around 0.5. For double hashing at this load factor, searches will have an average probe length of 2.

- Search times go to infinity as load factors approach 1.0 in open addressing.
- It's crucial that an open-addressing hash table does not become too full.
- A load factor of 1.0 is appropriate for separate chaining.

- At this load factor a successful search has an average probe length of 1.5, and an unsuccessful search, 2.0.

- Probe lengths in separate chaining increase linearly with load factor.

- A string can be hashed by multiplying each character by a different power of a constant, adding the products, and using the modulo (%) operator to reduce the result to the size of the hash table.

- To avoid overflow, the modulo operator can be applied at each step in the process, if the polynomial is expressed using Horner's method.

- Hash table sizes should generally be prime numbers. This is especially important in quadratic probing and separate chaining.

- Hash tables can be used for external storage. One way to do this is to have the elements in the hash table contain disk-file block numbers.