# Hashing

CSE 2011

Winter 2011

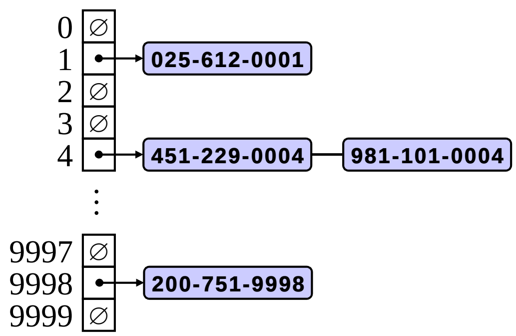


## Hashing

- BST, AVL trees: $O(\log N)$ for insertion, deletions and searches.
- Hashing is a technique used for performing insertion, deletions and searches in *constant* *average* time.
- Finding min, finding max, printing the whole collection in sorted order in linear time are *not* supported.
- A hash table data structure consists of:
  - Hash function ***h***
  - Array of size ***N*** (bucket array)

# Example

- We design a hash table for a dictionary storing items (SIN, Name), where SIN (social insurance number) is a ten-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function $h(x) = x \bmod N$
- We use chaining to handle collisions
- Assuming *integer keys*, how do we map keys to hash table entries?




3

# Hash Functions and Hash Tables

- A hash function ***h*** maps keys of a given type to integers in a fixed interval [0, ***N*** - 1]
- Example:
  $h(x) = x \bmod N$
  is a hash function for integer keys
- The integer ***h***(***x***) is called the hash value of key ***x***
- The goal of a hash function is to uniformly disperse keys in the range [0, ***N*** - 1]

- A hash table for a given key type consists of
  - Hash function ***h***
  - Array of size ***N***
- A collision occurs when two keys in the dictionary have the same hash value.
- Collision handing schemes:
  - Chaining: colliding items are stored in a sequence
  - Open addressing: the colliding item is placed in a different cell of the table

4

## Design Issues

- Hash function
  - For integer keys (compression functions)
  - For strings
- Collision handling
  - Separate chaining
  - Probing (open addressing)
    - Linear probing
    - Quadratic probing
    - Double hashing
- Table size (should be a prime number)

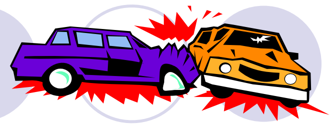
5

## Hash Functions

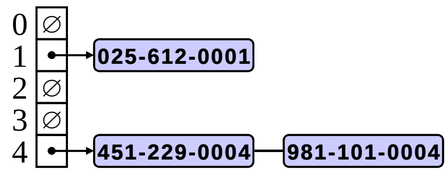
- Division:
  - $h_2(y) = y \bmod N$
  - The size $N$ of the hash table is usually chosen to be a prime number to minimize the number of collisions
  - The reason has to do with number theory and is beyond the scope of this course
- Multiply, Add and Divide (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - $a$ and $b$ are nonnegative integers such that
    $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value $b$

6

# Collision Handling



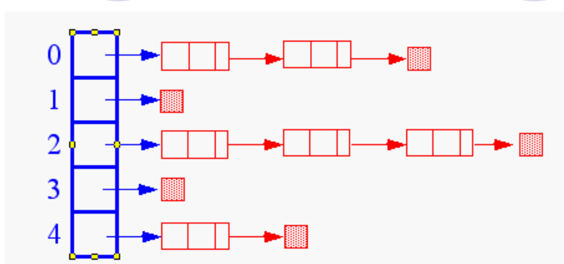
- Collisions occur when different elements are mapped to the same cell
- **Separate Chaining**: let each cell in the table point to a linked list of entries that map there




- Separate chaining is simple, but requires additional memory outside the table



# Separate Chaining




- Use ***chaining*** to set up ***lists*** of items with same index
- The ***expected*** search/insertion/removal time is O(n/N), provided that the indices are uniformly distributed
  - N = hash table size
  - n = number of elements in the table
- If n = O(N), the expected running time is O(1)

## Load Factor – Separate Chaining

- Define the load factor $\lambda = n/N$
  - n = number of elements in the hash table
  - N = hash table size (prime number)

- To obtain best performance with separate chaining, ensure $\lambda \leq 1$.

- As we add more elements to the hash table, $\lambda$ goes up $\Rightarrow$ rehashing (allocate a bigger table, define a new hash function, and copy the elements to the new array).

9

## Collision Handling

- Separate chaining
- Probing (open addressing)
  - Linear probing
  - Quadratic probing
  - Double hashing

10

# Linear Probing

- Linear probing handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a “probe”
- Colliding items lump together; future collisions will cause a longer sequence of probes

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order
  - Remove 44, 32, 73, 31

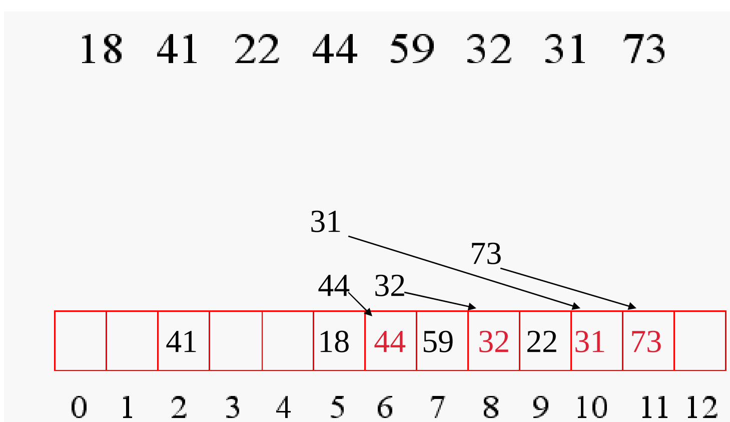
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | |

⇩

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 41 | | | 18 | 44 | 59 | 32 | 22 | 31 | 73 | |



# Linear Probing Example



## Search with Linear Probing

- Consider a hash table $A$ that uses linear probing
- $get(k)$
  - 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 cell is found, or
    - $N$ cells have been unsuccessfully probed

```
Algorithm get(k)
   i ← h(k)
   p ← 0
   repeat
      c ← A[i]
      if c == ∅
         return NULL
      else if c.key () = k
         return c.element()
      else
         i ← (i + 1) mod N
         p ← p + 1
   until p = N
   return NULL
```

13

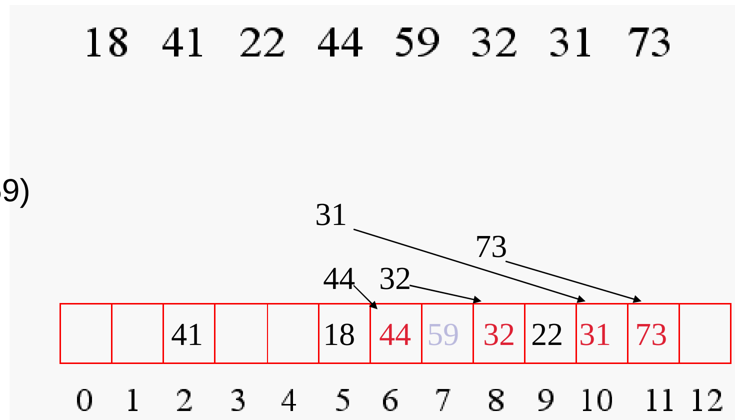
## Removal and Insertion with Probing

- *remove(k)*
  - Call *get*(*k*) to get the element.
  - Should we set the now empty cell to NULL?
    - No. It would mess up the search procedure. See example on the next slide.
  - Return the element.
- A cell has three states:
  - null: brand new, never used. *get*(*x*) stops when a null cell is reached.
  - in use: currently used.
  - available: previously used, now available but unused. *get*(*x*) continues the search when encountering an available cell.
    - Example of available cells: key has value -1.

14

# Example with *remove*(*k*)

*remove*(59)
*get*(31)






# Linear Probing: Removal and Insertion

- To handle insertions and deletions, we marked the deleted cells as “available” instead of null.
- *remove*(***k***)
  - We search for a cell with key ***k***
  - If such an item is found, we mark the cell as “available” and we return the element.
  - Else, we return ***NULL***
- *put*(***k, e***)
  If table is not full, we start at cell ***h***(***k***). If this cell is occupied:
  - We probe consecutive cells until a cell ***i*** is found that is either null or marked as “available”.
  - We store item (***k, e***) in cell ***i***

## Load Factor – Linear Probing

- Define the load factor $\lambda = n/N$
  - n = number of elements in the hash table
  - N = hash table size (prime number)

- To obtain best performance with linear probing, ensure that $\lambda \leq 0.5$.

- As we add more elements to the hash table, $\lambda$ goes up $\Rightarrow$ rehashing (allocate a bigger table, define a new hash function, and copy the elements to the new array).

17

## Next time …

- Probing (open addressing)
  - Linear probing
  - Quadratic probing
  - Double hashing
- Rehashing
- Hash functions for strings

- For a brief comparison of hash tables and self-balancing binary search trees (such as AVL trees), see http://en.wikipedia.org/wiki/Associative_array#Efficient_representations

18