CS206

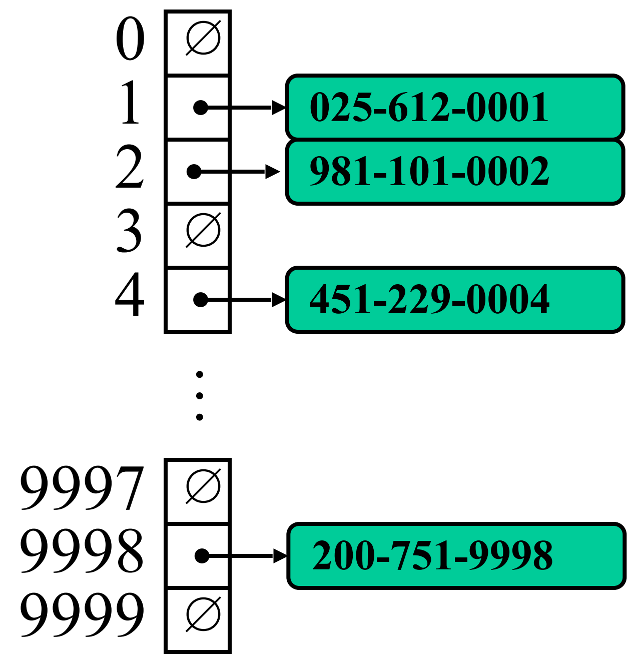
Hash Tables

Hash Functions and Maps

- A hash function h maps a key to integers in a fixed interval $[0, N-1]$
- $h(x) = x \% N$ is such a function for integers
- $h(x)$ is the hash value of key x
- A hash table is an array of size N
  - associated hash function h
  - item (k, v) is stored at index $h(k)$

Example

- A hash table storing entries as (SSN, Name), where SSN is a nine-digit positive integer
- Use an array of size $N = 10000$ and the hash function $h(x) = \text{last 4 digits of } x$



Hash Function

- A hash function is usually specified as the composition of two functions:
  - hash code: $h_1:$ key → integers
  - compression: $h_2:$ integers → $[0, N-1]$
  - $h(x) = h_2(h_1(x))$
- The goal is to "disperse" the keys in an appropriately random way

Hash Codes (h_1)

- Memory address:
  - use the memory address where the keys are stored
  - default hash code for Java objects
- Integer cast: interpret the bits storing the keys as integer – `byte`, `short`, `int` and `float`
- Component sum: partition bits into int components and sum them – `long` and `double`

Hash Codes (h_1)

- Polynomial accumulation: partition bits of key into a sequence of components of fixed length $a_0 a_1 \ldots a_{n-1}$
- Evaluate the polynomial
$p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$
- Strings: the choice of $z = 33$ leads to at most 6 collisions on a set of 50,000 English words

Compression (h_2)

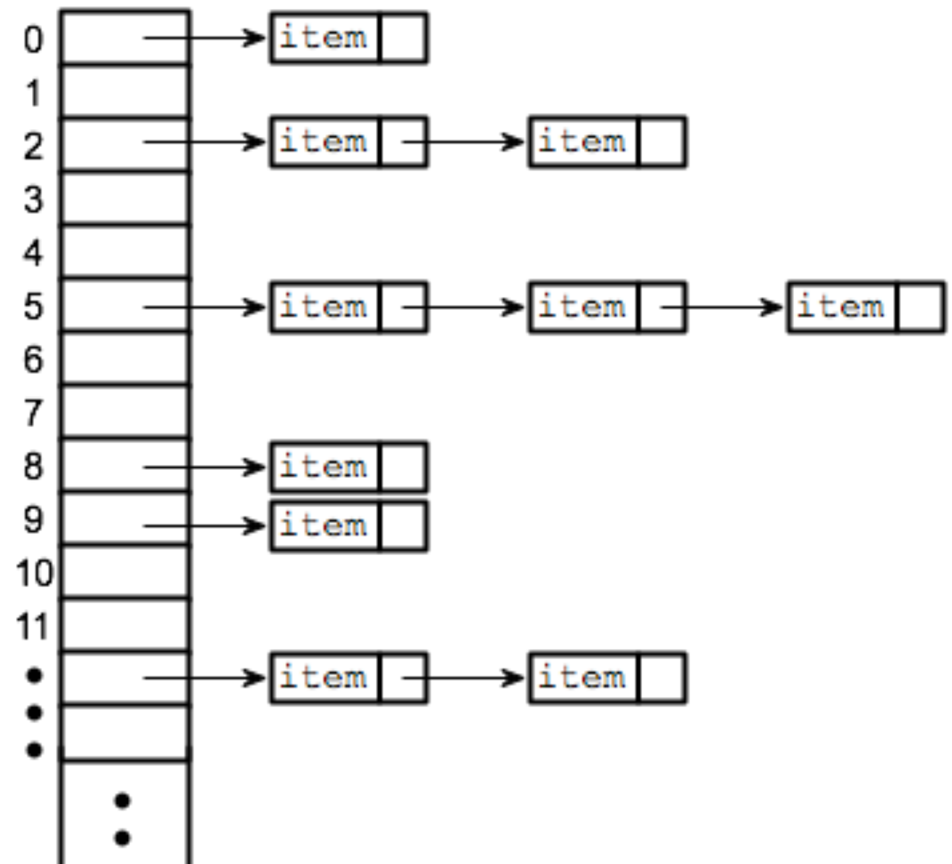
- Division: $h_2(x) = x\%N$
  - N is usually chosen to be a prime
- MAD: $h_2(x) = (ax + b)\%N,\ a\%N \neq 0$
    - multiply: a*x
    - add: +b
    - divide: %N
  - a and b are nonnegative integers
  - a scales the range and b shifts the start

Collision

- A hash function does not guarantee one-to-one mapping – no hash function does
- When more than one key hashes to the same index, we have a “collision”
- Handling collisions
  - Separate Chaining
  - Open Addressing
    - Linear Probing
    - Quadratic probing
    - Double Hashing

Separate chaining

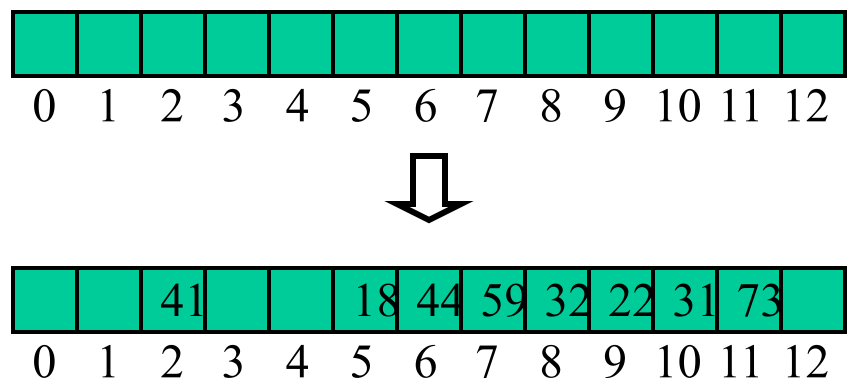
- LoadFactor ==> $\alpha = n/N$
  - n=itemcount
  - N=tablesize
- Given good hash and α<1 then put/get run constant time
- Bad hash / high α issues



Open Addressing and Probing

- Colliding item is put in a different cell (referred to as open addressing)
- Linear probing: place the colliding item in the next (circularly) available table cell
- Colliding items cluster together – future collisions to cause a longer sequence of probes

- Example: $h(x) = x\%13$
- insert 18, 41, 22, 44, 59, 32, 31, 73



Probing Distance

- Given a hash value $h(x)$, linear probing generates $h(x),\ h(x)+1, h(x)+2, \ldots$
- Primary clustering – the bigger the cluster gets, the faster it grows
- Quadratic probing – $h(x),\ h(x)+1, h(x)+4,\ h(x)+9, \ldots$
- Quadratic probing leads to secondary clustering, more subtle, not as dramatic, but still systematic

Double Hashing

- Interval between probes is fixed but computed by a second hash function
- Use a secondary hash function $d(k)$ to handle collisions by placing an item in the first available cell of the series
  $i + jd(k)\%N,\ 0 \leq j \leq N - 1$
- $d(k) \neq 0$
- N must be prime
- $d(k) = q - k\%q, q < N, q$ is prime

Example

- Double hashing:
  - $N = 13$
  - $h(k) = k\%13$
  - $d(k) = 7 - k\%7$
- Insert 18, 41, 22, 44, 59, 32, 31, 73

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		

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

⇩

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

Performance Analysis

- In the worst case, searches, insertions and removals take $O(n)$ time
  - when all the keys collide
- The load factor α affects the performance of a hash table
  - expected number of probes for an insertion with open addressing is $\frac{1}{1-\alpha}$
- Expected time of all operations is $O(1)$ provided α is not close to 1

Open Addressing vs Chaining

- Probing is significantly faster in practice
- locality of references – much faster to access a series of elements in an array than to follow the same number of pointers in a linked list
- Efficient probing requires soft/lazy deletions – tombstoning, why?
- May require “graveyard defragmenting”

Probing Tradeoffs

- Linear probing – best cache performance but most sensitive to clustering
- Double hashing – poor cache performance but exhibits virtually no clustering
- Quadratic – inbetween
- As load factor approaches 100%, number of probes rises dramatically
- Even with good hash functions, keep load factor 80% or below (50% is typical)
- Other open addressing methods besides probing

Good Hash Function

- is critical to performance
- A poor hash function can lead to poor performance even at very low load factor
- It is easy to unintentionally write a hash function that leads to severe clustering
- Testing your hash function is paramount

Performance of Hashtable

	Hash Expected	Hash Worst
search		
insert		
remove		
min/max		

	Unsorted array	Sorted array	Unsorted list	Sorted list	BST balanced	Hash Expected
search						
insert						
remove						
min/max						

Hashtable vs Array

- A hashtable is an unsorted array with a fast search – $O(1)$ expected
- An array is more memory efficient, but slower for searching
- If your data has natural indexing – a way to assign an ID/unique integer to each entry, then you are better off using an array. You have a hash function with 1-to-1 mapping and guaranteed no collisions

Hashtable Size

- Should be a prime
- twice the size of max number of keys
- or 1.3 times if n is very large
- 1/1.333 = 75% load factor
- Keep track of load factor and expand (rehash) the hash table when necessary

Midterm 2 review

Show all the steps for a merge sort when sorting the following list of integers

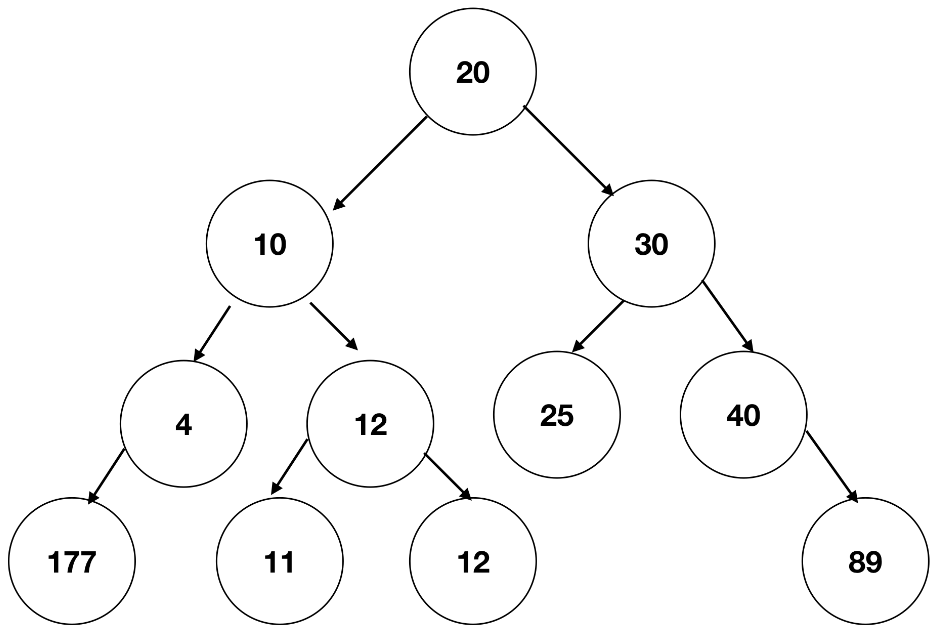
1719, 166, 569, 346, 1993, 1522, 726, 585, 1747, 956, 1512, 1909, 917, 1476, 1755

Show every recursive step for a poll operation on an array based max heap that contains the following data. (Shown in order within the array. That is 14 is in position 0, 13 in position 1, etc)

14, 13, 11, 9, 12, 5, 10, 4, 3, 7, 8, 0, 2, 6, 1

Alternately phrased, show the contents of the array every time the array changes.

Give the preorder traversals for this tree
Give the postorder traversals for this tree
Give a breadth-first traversal of this tree.



Write a recursive method that takes an array of integers and rearranges them so that all odd integers appear before all even integers