Hash Tables 2

Collisions

Dealing with Collisions

- A collision occurs when two different keys are mapped to the same index
  - Collisions may occur even when the hash function is good
- There are two main ways of dealing with collisions
  - Open addressing
  - Separate chaining

Open Addressing

- Idea – when an insertion results in a collision look for an empty array element
  - Start at the index to which the hash function mapped the inserted item
  - Look for a free space in the array following a particular search pattern, known as *probing*
- There are three open addressing schemes
  - Linear probing
  - Quadratic probing
  - Double hashing

Linear Probing

- The hash table is searched sequentially
  - Starting with the original hash location
  - For each time the table is probed (for a free location) add one to the index
    - Search *h*(*search key*) + 1, then *h*(*search key*) + 2, and so on until an available location is found
    - If the sequence of probes reaches the last element of the array, wrap around to *array*[0]
- Linear probing leads to *primary clustering*
  - The table contains groups of consecutively occupied locations
  - These clusters tend to get larger as time goes on
    - Reducing the efficiency of the hash table

Linear Probing Example

- Hash table is size 23
- The hash function, $h = x \; mod \; 23$, where x is the search key value
- The search key values are shown in the table

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58									21	

Linear Probing Example

- Insert 81, $h = 81 \; mod \; 23 = 12$
- Which collides with 58 so use linear probing to find a free space
- First look at 12 + 1, which is free so insert the item at index 13

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	81								21	

Linear Probing Example

- Insert 35, $h = 35 \bmod 23 = 12$
- Which collides with 58 so use linear probing to find a free space
- First look at 12 + 1, which is occupied so look at 12 + 2 and insert the item at index 14

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	81	35							21	

Linear Probing Example

- Insert 60, $h = 60 \textit{ mod } 23 = 14$
- Note that even though the key doesn't hash to 12 it still collides with an item that did
- First look at 14 + 1, which is free

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	81	35	60						21	

Linear Probing Example

- Insert 12, $h = 12 \ mod \ 23 = 12$
- The item will be inserted at index 16
- Notice that primary clustering is beginning to develop, making insertions less efficient

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	81	35	60	12					21	

Searching

- Searching for an item is similar to insertion
- Find 59, $h = 59 \; mod \; 23 = 13$, index 13 does not contain 59, but is occupied
- Use linear probing to find 59 or an empty space
- Conclude that 59 is not in the table

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	81	35	60	12					21	

Quadratic Probing

- Quadratic probing is a refinement of linear probing that prevents primary clustering
  - For each probe, p, add p^2 to the original location index
    - 1st probe: $h(x)+1^2$, 2nd: $h(x)+2^2$, 3rd: $h(x)+3^2$, etc.
- Results in *secondary clustering*
  - The same sequence of probes is used when two different values hash to the same location
  - This delays the collision resolution for those values
- Analysis suggests that secondary clustering is not a significant problem

Quadratic Probing Example

- Hash table is size 23
- The hash function, $h = x \bmod 23$, where x is the search key value
- The search key values are shown in the table

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58									21	

Quadratic Probing Example

- Insert 81, $h = 81 \; mod \; 23 = 12$
- Which collides with 58 so use quadratic probing to find a free space
- First look at $12 + 1^2$, which is free so insert the item at index 13

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	81								21	

Quadratic Probing Example

- Insert 35, $h = 35 \textit{ mod } 23 = 12$
- Which collides with 58
- First look at $12 + 1^2$, which is occupied, then look at $12 + 2^2 = 16$ and insert the item there

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	81			35					21	

Quadratic Probing Example

- Insert 60, $h = 60 \; mod \; 23 = 14$
- The location is free, so insert the item

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	81	60		35					21	

Quadratic Probing Example

- Insert 12, $h = 12 \text{ } mod \text{ } 23 = 12$
- First check index $12 + 1^2$,
- Then $12 + 2^2 = 16$,
- Then $12 + 3^2 = 21$ (which is also occupied),
- Then $12 + 4^2 = 28$, wraps to index 5 which is free

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
					12	29			32			58	81	60		35					21	

Quadratic Probe Chains

- Note that after some time a sequence of probes repeats itself
  - In the preceding example *h*(*key*) = *key* % 23 = 12, resulting in this sequence of probes (table size of 23)
    - 12, 13, 16, 21, 28(5), 37(14), 48(2), 61(15), 76(7), 93(1), 112(20), 133(18), 156(18), 181(20), 208(1), 237(7), ...
- This generally does not cause problems if
  - The data is not significantly skewed,
  - The hash table is large enough (around 2 * the number of items), and
  - The hash function scatters the data evenly across the table

Double Hashing

- In both linear and quadratic probing the probe sequence is independent of the key
- Double hashing produces *key dependent* probe sequences
  - In this scheme a second hash function, h_2, determines the probe sequence
- The second hash function must follow these guidelines
  - $h_2(key) \neq 0$
  - $h_2 \neq h_1$
  - A typical h_2 is $p - (key \bmod p)$ where p is a prime number

Double Hashing Example

- Hash table is size 23
- The hash function, $h = x \bmod 23$, where x is the search key value
- The second hash function, $h_2 = 5 - (key \bmod 5)$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58									21	

Double Hashing Example

- Insert 81, $h = 81 \mathit{ mod } 23 = 12$
- Which collides with 58 so use h_2 to find the probe sequence value
- $h_2 = 5 – (81 \mathit{ mod } 5) = 4$, so insert at $12 + 4 = 16$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58				81					21	

Double Hashing Example

- Insert 35, $h = 35 \; mod \; 23 = 12$
- Which collides with 58 so use h_2 to find a free space
- $h_2 = 5 - (35 \; mod \; 5) = 5$, so insert at $12 + 5 = 17$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58				81	35				21	

Double Hashing Example

- Insert 60, $h = 60 \; mod \; 23 = 14$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58		60		81	35				21	

Double Hashing Example

- Insert 83, h = 83 mod 23 = 14
- h_2 = 5 – (83 mod 5) = 2, so insert at 14 + 2 = 16, which is occupied
- The second probe increments the insertion point by 2 again, so insert at 16 + 2 = 18

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58		60		81	35	83			21	

Removals and Open Addressing

- Removals add complexity to hash tables
  - It is easy to find and remove a particular item
  - But what happens when you want to search for some other item?
  - The recently empty space may make a probe sequence terminate prematurely
- One solution is to mark a table location as either empty, occupied or removed
  - Locations in the *removed* state can be re-used as items are inserted

Removal Example

- Array elements are marked as empty, occupied or removed
- The hash function is $h = x \bmod 23$, use linear probing
- Remove 60

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	36		49	81					21	
empty	empty	empty	empty	empty	empty	occupied	empty	empty	occupied	empty	empty	occupied	occupied	removed	occupied	occupied	empty	empty	empty	empty	occupied	empty

Removal Example

- Array elements are marked as empty, occupied or removed
- The hash function is $h = x \bmod 23$, use linear probing
- Remove 60
- Search for 81: $81 \bmod 23 = 12$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
						29			32			58	36		49	81					21	
empty	empty	empty	empty	empty	empty	occupied	empty	empty	occupied	empty	empty	occupied	occupied	removed	occupied	occupied	empty	empty	empty	empty	occupied	empty

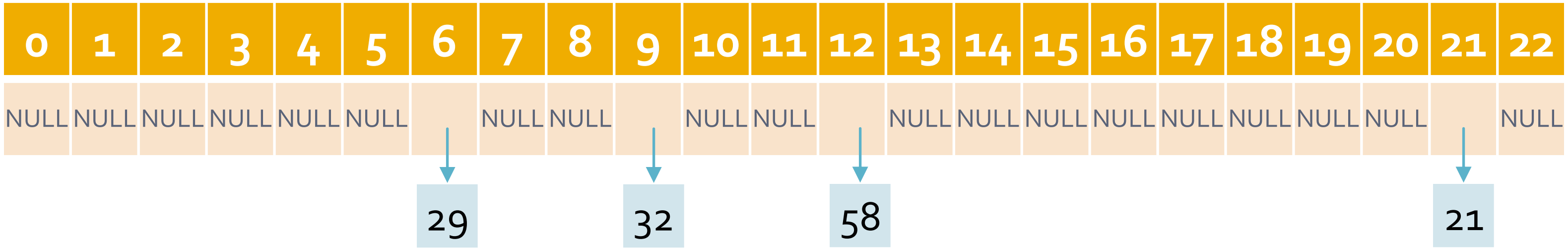
removed flag so continue search

Separate Chaining

- Separate chaining takes a different approach to collisions
- Each entry in the hash table is a pointer to a linked list
  - If a collision occurs the new item is added to the end of the list at the appropriate location
- Performance degrades less rapidly using separate chaining
  - But each search or insert requires an additional operation to access the list

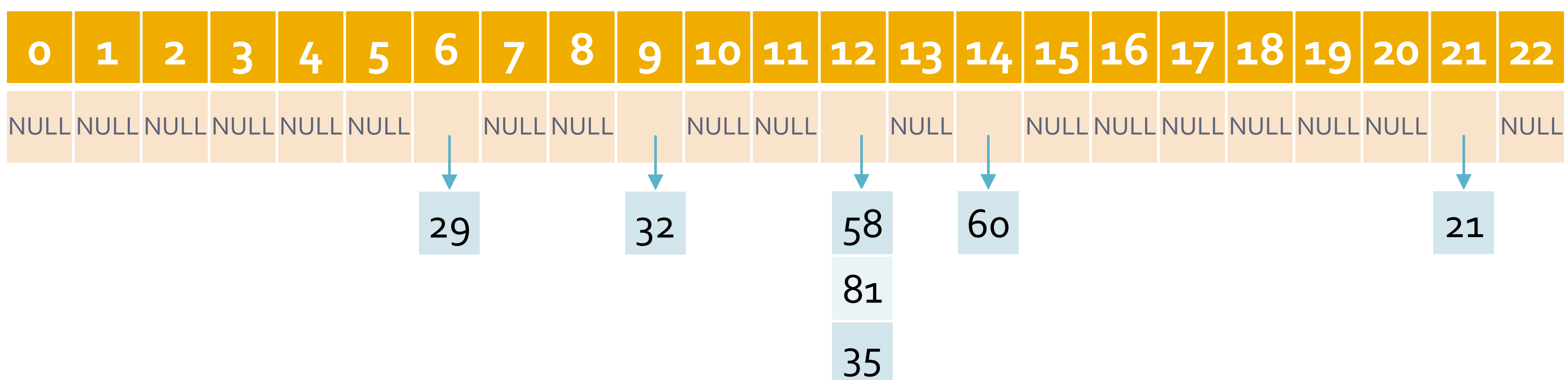
Separate Chaining Example

- Hash table is size 23
- The hash function, $h = x \bmod 23$
- Each table entry consists of a pointer to a linked list



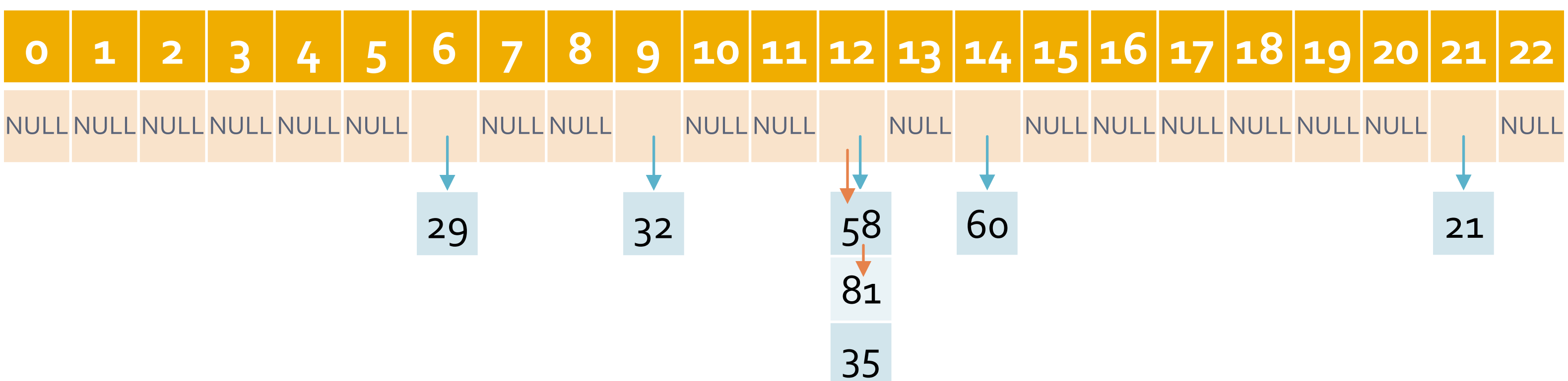
Separate Chaining Example

- Hash table is size 23, $h = x \bmod 23$
- Insert 81: 81 *mod* 23 = 12
- Insert 60: 60 mod 23 = 14
- Insert 35: 35 mod 23 = 12



Separate Chaining Example

- Hash table is size 23, $h = x \bmod 23$
- Insert 81: 81 *mod* 23 = 12
- Insert 60: 60 mod 23 = 14
- Insert 35: 35 mod 23 = 12
- Search for 81



Efficiency

Hash Table Efficiency

- When analyzing the efficiency of hashing it is necessary to consider *load factor*, α
  - α = *number of items / table size*
  - As the table fills, α increases, and the chance of a collision occurring also increases
    - Performance decreases as α increases
  - Unsuccessful searches make more comparisons
    - An unsuccessful search only ends when a free element is found
- It is important to base the table size on the largest possible number of items
  - The table size should be selected so that α does not exceed 2/3

Average Comparisons

- Linear probing
  - When $\alpha = 2/3$ unsuccessful searches require 5 comparisons, and
  - Successful searches require 2 comparisons
- Quadratic probing and double hashing
  - When $\alpha = 2/3$ unsuccessful searches require 3 comparisons
  - Successful searches require 2 comparisons
- Separate chaining
  - The lists have to be traversed until the target is found
  - α comparisons for an unsuccessful search
    - Where α is the average size of the linked lists
  - $1 + \alpha / 2$ comparisons for a successful search

Hash Table Discussion

- If α is less than ½, open addressing and separate chaining give similar performance
  - As α increases, separate chaining performs better than open addressing
  - However, separate chaining increases storage overhead for the linked list pointers
- It is important to note that in the worst case hash table performance can be poor
  - That is, if the hash function does not evenly distribute data across the table