CSE 332: Data Structures and Parallelism

Fall 2022
Richard Anderson
Lecture 11: Hashing (and some B-Tree leftovers)

Announcements

Today

- Finish up B-trees
  - Attempt to clear up some (justified) confusion
  - B-Tree Deletes
- Hashing
  - Arrays for dictionary
  - Key space to array space
  - Dealing with collisions
  - Hash functions
  - Resizing and Load Factors
  - Expected performance

B Tree Structure Properties

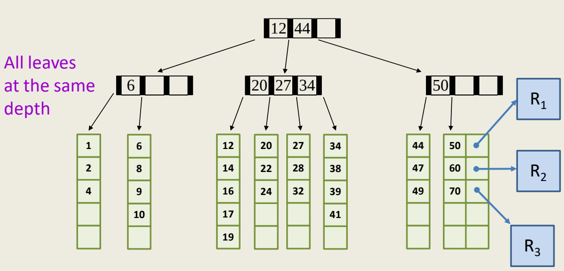
Internal nodes
- store up to M-1 keys
- have between $\lceil M/2 \rceil$ and *M* children
- Store keys and pointers to children

Leaf nodes
- where data is stored
- all at the same depth
- contain between $\lceil L/2 \rceil$ and *L* data items
- Store key-pointer pairs, where the pointer is to the record that stores the data

Root (special case)
- has between 2 and ***M*** children (or root could be a leaf)

B Tree: Example

- B+ Tree with ***M* = 4** (# pointers in internal node)
- and ***L* = 5** (# data items in leaf)



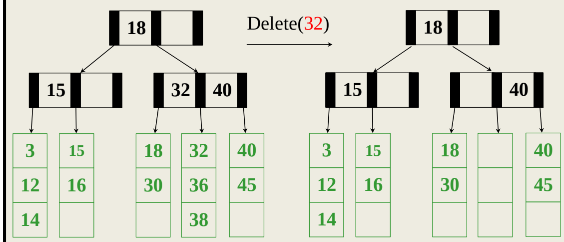
Operations

- Find(K)
  - Return a pointer to the record of K
- Insert(K)
  - Insert key K and return a pointer to the record of K
- Delete(K)
  - Delete key K and associated data

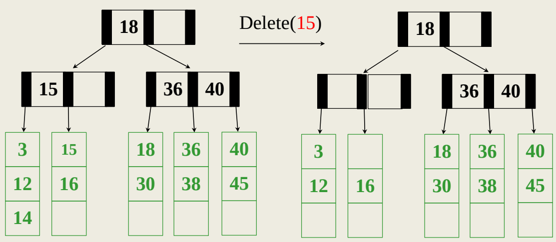
Node sizes

- Internal nodes
  - 4096 bytes, 8 byte keys, 8 byte child pointer
  - M = 256
- Leaves
  - 4096 bytes, 8 byte keys, 8 byte record pointer
  - L = 256

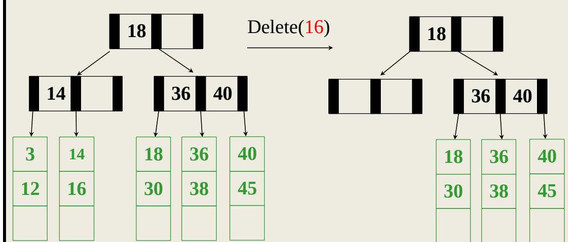
And Now for Deletion…



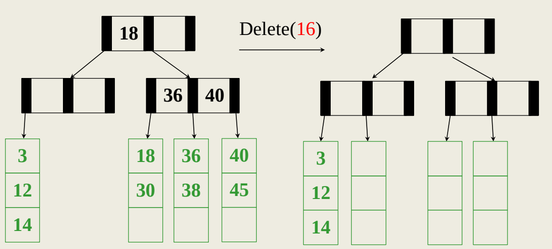
M = 3 L = 3



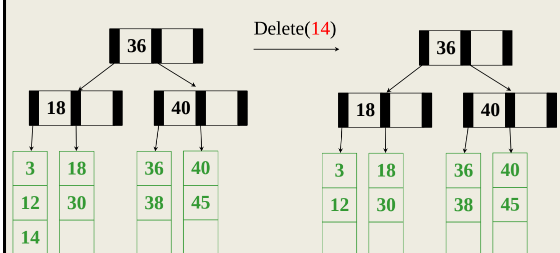
M = 3 L = 3



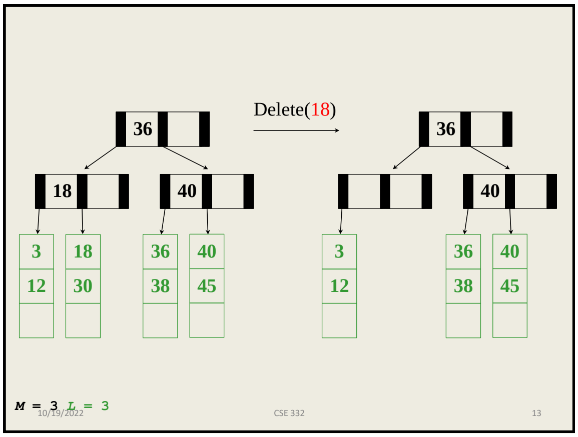
M = 3 L = 3

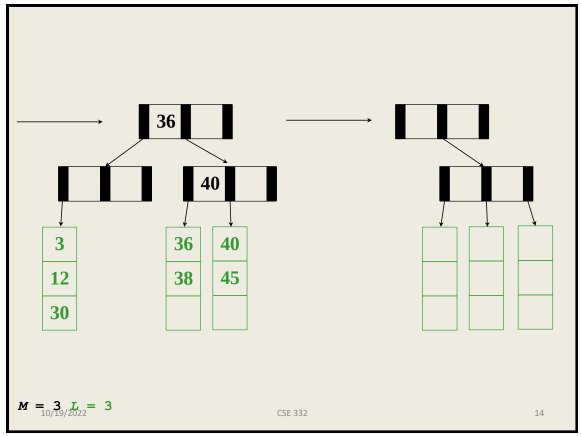


M = 3 L = 3



M = 3 L = 3





Deletion Algorithm

1. Remove the key from its leaf

- 2. If the leaf ends up with fewer than $\lceil L/2 \rceil$ items, **underflow**!
  - Adopt data from a neighbor; update the parent
  - If adopting won't work, delete node and merge with neighbor
  - If the parent ends up with fewer than $\lceil M/2 \rceil$ children, **underflow**!

Deletion Slide Two

- 3. If an internal node ends up with fewer than $\lceil M/2 \rceil$ children, **underflow**!
  - Adopt from a neighbor; update the parent
  - If adoption won't work, merge with neighbor
  - If the parent ends up with fewer than $\lceil M/2 \rceil$ children, **underflow**!

4. If the root ends up with only one child, make the child the new root of the tree

5. Propagate keys up through tree.

This reduces the height of the tree!

Hashing

Dictionary

Array for data lookup

- Store football players by jersey number

10	Uchenna Nwosu
11	Marquise Goodwin
12	
13	Josh Jones
14	DK Metcalf
15	
16	Tyler Lockett
17	
18	
19	Penny Hart
20	
21	Artie Burns
22	
23	Sidney Jones IV
24	Isaiah Dunn

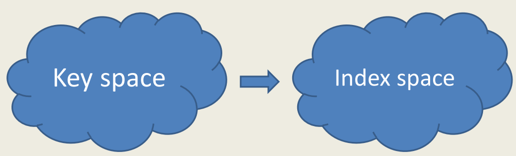
Array for data lookup

- Store students by student ID number

2061129	
2061130	
2061131	
2061132	
2061133	
2061134	
2061135	
2061136	
2061137	
2061138	
2061139	
2061140	Artie Burns
2061141	
2061142	
2061143	

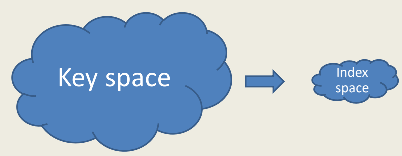
Arrays for dictionaries

- Index by key, O(1) insert and find



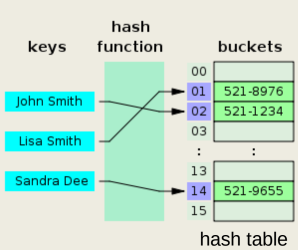
Hashing: Map large keyspace into small index space

- I(K) = hash(K)



Hash Tables

- Map keys to a smaller array called a hash table
  - via a hash function h(K)
  - Find, insert, delete: O(1) on average!



Simple Integer Hash Functions

- key space K = integers
- TableSize = 10

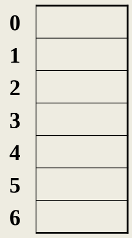
- h(K) =

- **Insert**: 7, 18, 41, 34

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Simple Integer Hash Functions

- key space K = integers
- TableSize = 7
- h(K) = K mod 7
- **Insert**: 7, 18, 41, 34



Aside: Properties of Mod

To keep hashed values within the size of the table, we will generally do:

h(K) = function(K) mod TableSize

(In the previous examples, function(K) = K.)

Useful properties of mod:

(a + b) mod c = [(a mod c) + (b mod c)] mod c

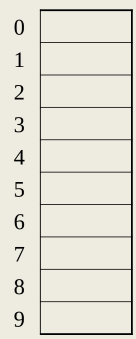
(a b) mod c = [(a mod c) (b mod c)] mod c

a mod c = b mod c → (a – b) mod c = 0

Collision Resolutions

- Separate Chaining
- Open Addressing

Separate Chaining



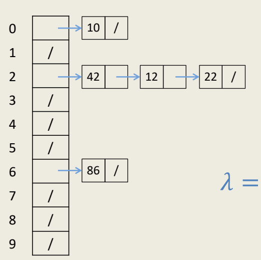
Insert:
10
22
107
12
42

All keys that map to the same hash value are kept in a list (or "bucket").

Analysis of Separate Chaining

The load factor, λ, of a hash table is $\lambda = \frac{N}{TableSize}$

λ = average # of elements per bucket



$\lambda =$

Analysis of Separate Chaining

The load factor, λ, of a hash table is $\lambda = \frac{N}{TableSize}$

λ = average # of elems per bucket

Average cost of:

- Unsuccessful find?
- Successful find?
- Insert?

Alternative: Use Empty Space in the Table

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Insert:
38
19
8
109
10

Try h(K).
If full, try h(K)+1.
If full, try h(K)+2.
If full, try h(K)+3.
Etc…

Open Addressing

After a collision, try "next" spot. If there's another collision, try another, etc.

Finding the next available spot is called **probing**:

0^{th} probe = h(k) % TableSize
1^{th} probe = (h(k) + f(1)) % TableSize
2^{th} probe = (h(k) + f(2)) % TableSize
. . .
i^{th} probe = (h(k) + f(i)) % TableSize

f(i) is the probing function. We'll look at a few…

Linear Probing

f(i) = i

- Probe sequence:
  - 0^{th} probe = h(K) % TableSize
  - 1^{th} probe = (h(K) + 1) % TableSize
  - 2^{th} probe = (h(K) + 2) % TableSize
  - . . .
  - i^{th} probe = (h(K) + i) % TableSize

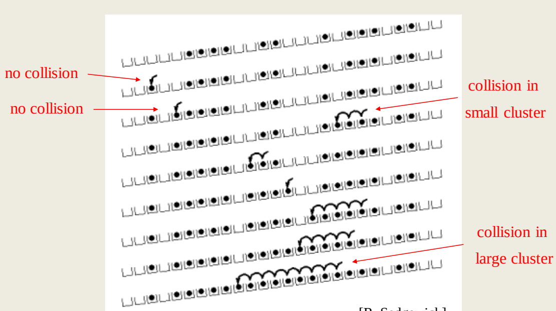
Linear Probing

0	8
1	109
2	10
3	
4	
5	
6	
7	
8	38
9	19

Insert:
38
19
8
109
10

Try h(K)
If full, try h(K)+1.
If full, try h(K)+2.
If full, try h(K)+3.
Etc…

Linear Probing – Clustering



[R. Sedgewick]

Analysis of Linear Probing

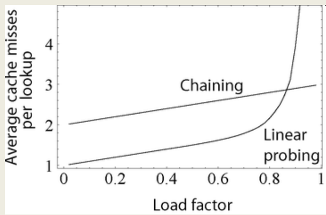
- For *any* $\lambda < 1$, linear probing *will* find an empty slot
- Expected # of probes (for large table sizes)
  - unsuccessful search:

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

  - successful search:

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$$

- Linear probing suffers from ***primary clustering***
- Performance quickly degrades for $\lambda > 1/2$



Quadratic Probing

Less likely to encounter Primary Clustering

$f(i) = i^2$

- Probe sequence:
  - 0^{th} probe = h(K) % TableSize
  - 1^{th} probe = (h(K) + 1) % TableSize
  - 2^{th} probe = (h(K) + 4) % TableSize
  - 3^{th} probe = (h(K) + 9) % TableSize
  - . . .
  - i^{th} probe = (h(K) + i^2) % TableSize

Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Insert:
89
18
49
58
79

Another Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	

TableSize = 7
h(K) = K % 7

insert(76) 76 % 7 =6
insert(40) 40 % 7 =5
insert(48) 48 % 7 =6
insert(5) 5 % 7 =5
insert(55) 55 % 7 =6
insert(47) 47 % 7 =5

Quadratic Probing: Properties

- For *any* $\lambda < \frac{1}{2}$, quadratic probing will find an empty slot; for bigger λ, quadratic probing *may* find a slot.
- Quadratic probing does not suffer from *primary* clustering: keys hashing to the same *area* is ok
- But what about keys that hash to the same *slot*?
  - ***Secondary Clustering!***

Double Hashing

Idea: given two different (good) hash functions h(K) and g(K), it is unlikely for two keys to collide with both of them.

So…let's try probing with a second hash function:

$f(i) = i * g(K)$

- Probe sequence:
  - 0^{th} probe = h(K) % TableSize
  - 1^{th} probe = (h(K) + g(K)) % TableSize
  - 2^{th} probe = (h(K) + 2*g(K)) % TableSize
  - 3^{th} probe = (h(K) + 3*g(K)) % TableSize
  - . . .
  - i^{th} probe = (h(K) + i*g(K)) % TableSize

Double Hashing Example

0	
1	
2	
3	
4	
5	
6	

TableSize = 7

h(K) = K % 7

g(K) = 5 – (K % 5)

Insert(76) 76 % 7 = 6 and 5 - 76 % 5 =

Insert(93) 93 % 7 = 2 and 5 - 93 % 5 =

Insert(40) 40 % 7 = 5 and 5 - 40 % 5 =

Insert(47) 47 % 7 = 5 and 5 - 47 % 5 =

Insert(10) 10 % 7 = 3 and 5 - 10 % 5 =

Insert(55) 55 % 7 = 6 and 5 - 55 % 5 =

Deletion in Separate Chaining

How do we delete an element with separate chaining?

Deletion in Open Addressing

h(k) = k % 7

Linear probing

0	
1	
2	16
3	23
4	59
5	
6	76

Delete(23)

Find(59)

Insert(30)

Need to keep track of deleted items... leave a “marker”

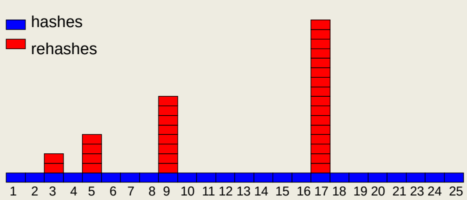
Rehashing

When the table gets too full, create a bigger table (usually 2x as large) and hash all the items from the original table into the new table.

- When to rehash?
  - Separate chaining: full ($\lambda = 1$)
  - Open addressing: half full ($\lambda = 0.5$)
  - When an insertion fails
  - Some other threshold
- Cost of a single rehashing?

Rehashing Picture

- Starting with table of size 2, double when load factor > 1.



Amortized Analysis of Rehashing

- Cost of inserting n keys is < 3n
- suppose $2^k + 1 \leq n \leq 2^{k+1}$
  - Hashes = n
  - Rehashes = $2 + 2^2 + \ldots + 2^k = 2^{k+1} - 2$
  - Total = $n + 2^{k+1} - 2 < 3n$
- Example
  - n = 33, Total = 33 + 64 – 2 = 95 < 99