CSE 332: Data Structures and Parallelism

Section 5: B-Trees and Hashing Solutions

0. Implement a B-Tree? Nah, Let's Analyze!

Given the following parameters for a B-Tree with a page size of 256 bytes:

- Key Size = 8 bytes
- Pointer Size = 2 bytes
- Data Size = 14 bytes per record (includes the key)

Assuming that M and L were chosen appropriately, what are M and L? Recall that M is defined as the maximum number of pointers in an internal node, and L is defined as the maximum number of values in a leaf node. Give a numeric answer and a short justification based on two equations using the parameter values above.

Solution:

We start by defining the following variables.

- 1 page on disk is b bytes
- Keys are k bytes
- Pointers are t bytes
- Key/Value pairs are v bytes

We know that the amount of memory used by one leaf node is vL and the amount of memory used by one internal node is $tM + k(M-1)$. We want select values for M and L such that both equations are $\leq b$.

If we solve both equations for M and L, we obtain $M = \left\lfloor \frac{b+k}{t+k} \right\rfloor$ and $L = \left\lfloor \frac{b}{v} \right\rfloor$

Plugging in the given values, we get $M = \left\lfloor \frac{256+8}{2+8} \right\rfloor = 26$ and $L = \left\lfloor \frac{256}{14} \right\rfloor = 18$

1. Oh, B-Trees

Find a tight upper bound on the *worst case runtime* of these operations on a B-tree. Your answers should be in terms of L, M, and n.

(a) Insert a key-value pair

(b) Look up the value of a key

(c) Delete a key-value pair

Solution:

Insertion, Deletion The steps for insert and delete are similar and have the same worst case runtime.

(a) Find the leaf: $\mathcal{O}(\lg(M)\log_M(n))$. (For more details, see the next solution.)

(b) Insert/remove in the leaf – there are L elements, essentially stored in an array: $\mathcal{O}(L)$

(c) Split a leaf/merge neighbors: $\mathcal{O}(L)$

(d) Split/merge parents, in the worst case going up to the root: $\mathcal{O}(M\log_M(n))$

The total cost is then $\lg(M)\log_M(n) + 2L + M\log_M(n)$.

We can simplify this to a worst-case runtime $\mathcal{O}(L + M\log_M(n))$ by combining constants and observing that $M\log_M(n)$ dominates $\lg(M)\log_M(n)$. Note that in the average case, splits for any reasonably-sized B-tree are rare, so we can amortize the work of splitting over many operations.

However, if we're using a B-tree, it's because what concerns us the most is the penalty of disk accesses. In that case, we might find it more useful to look at the worst-case number of disk lookup operations in the B-tree, which is $\mathcal{O}(\log_M(n))$.

Look up (a) We must do a binary search on a node containing M pointers, which takes $\mathcal{O}(\lg(M))$ time, once at each level of the tree.

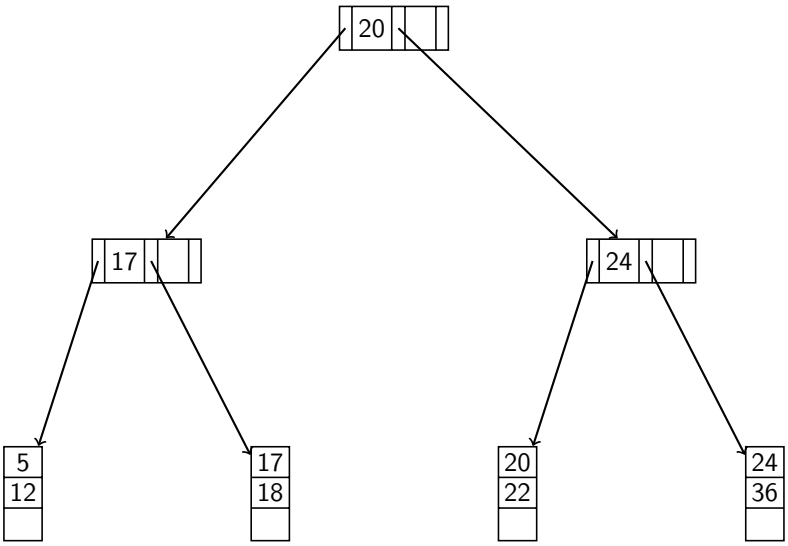
(b) There are $\mathcal{O}(\log_M(n))$ levels.

(c) We must do a binary search on a leaf of L elements, which takes $\mathcal{O}(\lg(L))$ time.

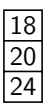
(d) Putting it all together, a tight bound on the runtime is $\mathcal{O}(\lg(M)\log_M(n) + \lg(L))$.

2. B-Trees

(a) For the following B-Tree, delete $17, 12, 22, 5, 36$



Solution:



(b) Given the following parameters for a B-Tree with $M = 11$ and $L = 8$

- Key Size = 10 bytes
- Pointer Size = 2 bytes
- Data Size = 16 bytes per record (includes the key)

Assuming that M and L were chosen appropriately, what is the likely page size on the machine where this implementation will be deployed? Give a numeric answer and a short justification based on two equations using the parameter values above.

Solution:

We use the following two equations to find M and L to fit as best as possible in the page size, where:

- 1 page on disk is p bytes
- Keys are k bytes
- Pointers are t bytes
- Key/Value pairs are v bytes

$M = \left\lfloor \frac{p+k}{t+k} \right\rfloor$ and $L = \lfloor \frac{p}{v} \rfloor$

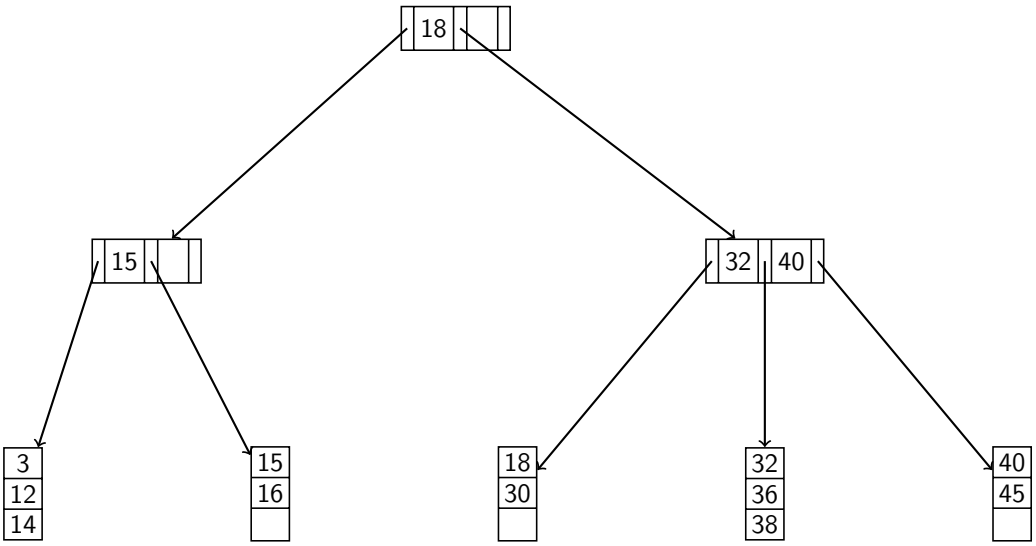
Plugging in the given values, we get:

$M = \left\lfloor \frac{p+10}{2+10} \right\rfloor$ and $L = \lfloor \frac{p}{16} \rfloor$

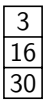
And solving for p gives us an answer of 128 bytes.

3. It's Fun to B-Trees!

(a) For the following B-Tree, delete $45, 14, 15, 36, 32, 18, 38, 40, 12$



Solution:



4. Hash... Browns?

For the following scenarios, insert the following elements in this order: 7, 9, 48, 8, 37, 57. For each table, TableSize = 10, and you should use the primary hash function $h(k) = k1 \bmod m0$. If an item cannot be inserted into the table, please indicate this and continue inserting the remaining values.

(a) Linear Probing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Solution:

0	8
1	37
2	57
3	
4	
5	
6	
7	7
8	48
9	9

(b) Quadratic Probing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Solution:

0	
1	37
2	8
3	
4	
5	
6	57
7	7
8	48
9	9

(c) Separate chaining hash table - Use a linked list for each bucket. Order elements within buckets in any way you wish.

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Solution:

0	
1	
2	
3	
4	
5	
6	
7	57 $\rightarrow$37 $\rightarrow$7
8	8 $\rightarrow$48
9	9

5. Double Double Toil and Trouble

(a) Describe double hashing.

Solution:

The first hash function determines the original location where we should try to place the item. If there is a collision, then the second hash function is used to determine the probing step distance as 1*h2(key), 2*h2(key), 3*h2(key) etc. away from the original location.

(b) List 2 cons of quadratic probing and describe how one of those is fixed by using double hashing.

Solution:

In quadratic probing, 1) if the table is more than half full (load factor = 0.5) then you are not guaranteed to be able to find a location to place the item, 2) suffers from secondary clustering (items that initially hash to the same location resolve the collision identically).

Assuming a good second hash function is used, double hashing does not suffer from 1). Assuming a good second hash function is used, double hashing avoids secondary clustering because items that initially hash to the same location resolve the collision differently, which decreases the likelihood that two elements will hash to the same index after initial collision.