Paul Elder, Shibo Xing
pye1@pitt.edu, shx26@pitt.edu

1.

If we want to eliminate the lost update problem and also not execute the transactions in strict modem, we can simply overwrite all older before images on commit. On the basis of ACA, the before image of a variable is only updated when a transaction which writes to this variable is committed. During the transaction, for every write that occurs, a list of changed values and their locations can be saved, and upon commit, these values can be taken to overwrite all other instances of the before images.

If we have transactions T1 and T2 with the following history
initial before images: x = 1, y = 0

T1	T2	before image
w1(x, 2);		x = 1, y = 0
	w2(x, 8);	
	w2(y, 9);	
	c2;	x = 8, y = 9
w1(y, 3);		
a1		x = 8, y = 9

Thus, when T1 aborts, x and y are recovered using the before images, T2's updates are not lost. Also, T2 is permitted to overwrite the uncommitted data by T1, thus this execution is not in strict mode.

2a.

h0(x) = x mod 4, h1(x) = x mod 8

Bucket #	0	1 (B_split)	2	3	4 (B_last)
	4		2	3	
			6	7	
			14		
Overflow:			18		

h0(x) = x mod 4, h1(x) = x mod 8

Bucket #	0	1	2 (B_split)	3	4	5 (B_last)
	4	9	2	3		
			6	7		
			14	19		
Overflow:			18	11		

2b.

table after splits under no overflow buckets:
s = 4, h0(x) = x mod 4, h1(x) = x mod 8

Bucket #	0	1 (B_split)	2	3	4 (B_last)
			2	3	4
			6	7	
			14		

Bucket #	0	1	2 (B_split)	3	4	5 (B_last)
			2	3	4	
			6	7		
			14			

Bucket #	0	1	2	3 (B_split)	4	5	6 (B_last)
			2	3	4		6
			18	7			14

s = 8, h0(x) = x mod 8, h1(x) = x mod 16

Bucket #	0 (B_split)	1	2	3	4	5	6	7 (B_last)
		9	2	3	4		6	7
			18	19			14	
				11				

In terms of **insertion** with no collisions, linear bucket with overflow buckets has an amortized efficiency of O(1). This is the same with linear hashing without overflow bucket, since no duplicate hash values occur and the split pointer is not moved.

During insertions with collisions, if the overflow bucket is far ahead from the split point, overflow bucket will delay the split, thus insertion is still O(1). A few split operations will trigger the freeing of an overflow bucket, causing an O(k) operation of rehashing where k is the size of the overflow bucket.
If not using overflow buckets, many insertions might cause an O(S) split time, where S is the hash key value. Split time is O(S) since the filled bucket has to constantly wait for the split pointer to reach its index, especially when multiples of a value are inserted over

and over. Thus, the amortized insertion efficiency with overflow buckets is definitely superior to the that without overflow buckets.

While **Searching,** linear hashing without overflow bucket has a superior runtime since every queried data is guaranteed to be one of the records in some bucket. If we consider the number of records as a constant number, then this operation is O(1). Linear hashing with overflow buckets on the other hand, might cause a probing operation through the overflow buckets list. This might result in a linear runtime if significant data are stored in overflow buckets.

When performing **deletion without compaction,** hashing without overflow buckets will result in O(1) runtime on average, the same as searching. If compaction is not performed, then deleting any specific elements will not free up any buckets. If overflow buckets are present, deletion will perform probing through the overflow buckets, achieving the same runtime as searching.

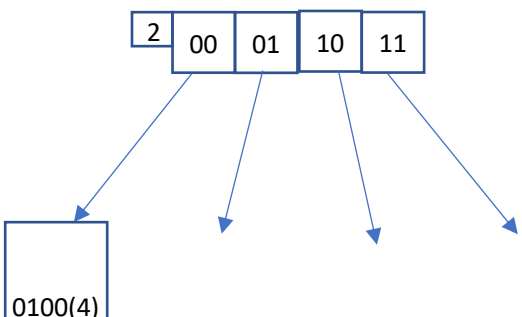
When performing **deletion with compaction,** hashing without overflow bucket is O(1) on average, some deletion may free a bucket and move the split and last pointer but these operations don't contribute much to the runtime. If overflow buckets are present, fewer deletion will free buckets if long lists of overflow buckets exists. Thus deletion with compaction still result in a linear runtime in the worst case.

The **split frequency** of linear hashing without overflow buckets is higher than linear hashing with buckets. If without overflow buckets, split pointer will have to keep moving until it reaches the index where the collision happens. If overflow buckets are used, then it is guaranteed that only one split operation will be performed upon a collision. Thus, more splits happen for linear hashing without overflow buckets than with overflow buckets.

The i**nternal fragmentation** of linear hashing without overflow buckets is more than with overflow buckets. As discussed in previous paragraphs, some collisions cause the split pointer to move to their indices. During this process, the split pointer skips over a lot of buckets, leading to a long expansion of the bucket list which has many empty buckets. If overflow buckets are used, only one split will happen per collision so not too many empty buckets will be created.

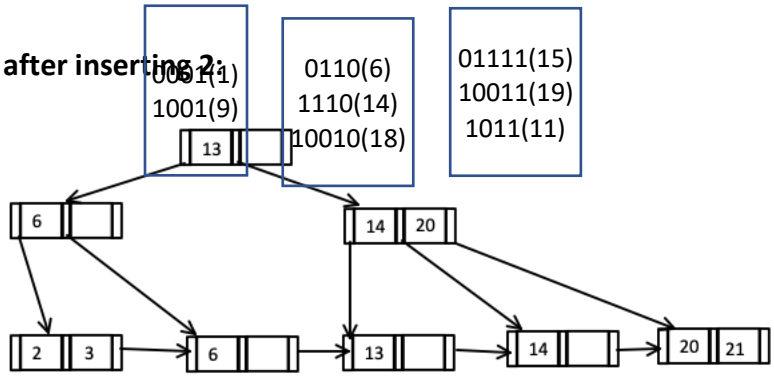
The **number of pointers** in linear hashing without overflow buckets is less than that with overflow buckets since overflow buckets of each bucket is a linked-list of buckets. Every newly overflow bucket will increase the number of pointer by 1.

3.

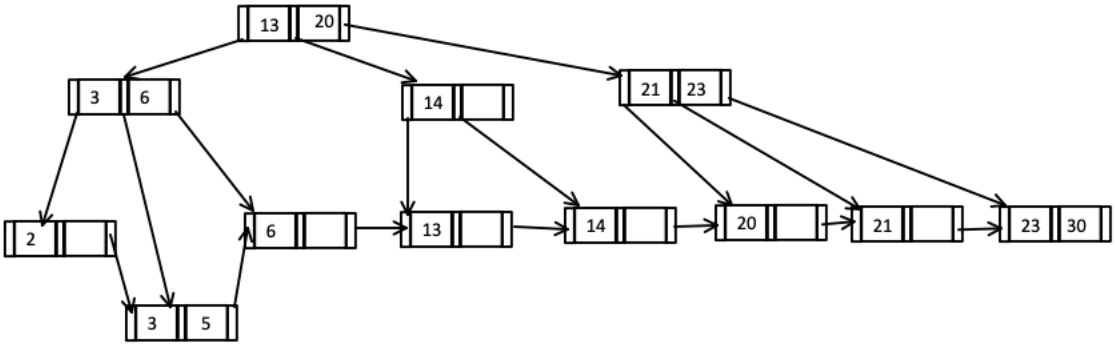


4.

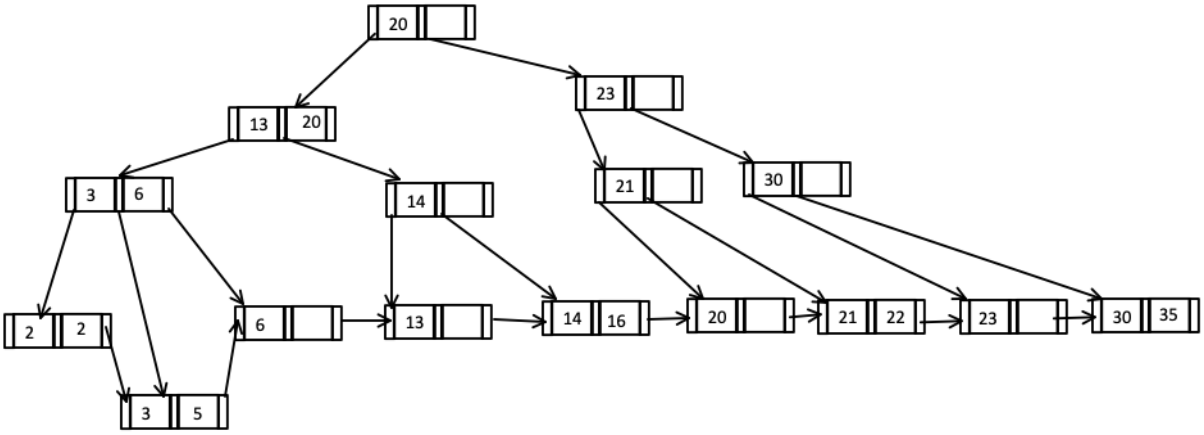
after inserting 2:



after inserting 30:



after inserting all the keys:



5. With spinning disks, slotted pages work well because the head can pick up the whole record as the disk with the record spins by. For SSDs, we have random access with automatic caching via cache lines. Therefore, PAX pages will work better, since we can pick up one field for multiple records into the cache with one read. This means that searching for a key will be faster as well, thanks to the spacial locality.

6. Head starts on track 12. Without loss of generality, assume that the head starts on sector 0. That is, the sector numbers declared in the following order can trivially be incremented by 4 modulo 64, but it is easier to reason with 0, 31, and 63 than it is to reason with 4, 35, and 3. Assume instant reading and writing as long as head is above target sector.

TIME	ACTION
0ms	
	6ms to read half of track into the buffer.
6ms	
	1.5 x 2 = 3ms to move the head to track 14, concurrent with spinning the disk another 6ms to get back to sector 0.
12ms	
	6ms to write the buffer from sector 0 to sector 31. Head is now at sector 31.
18ms	
	3ms to move the head back to track 12, concurrent with spinning the disk 12ms to get back to sector 31.
30ms	
	6ms to read half of the track into the buffer. Head is now at sector 0.
36ms	
	3ms to move the head to track 14, concurrent with spinning the disk 6ms to get back to sector 32.
42ms	
	6ms to write the buffer from sector 32 to sector 63.
48ms (total)	

7a. Frequent, random accesses to a small file

Since the access is frequent, these should be stored on tracks near the center of the seek range, to minimize the necessary seek travel time.

7b. Sequential scans of a large file

These should be placed on an outer track. The higher data transfer speed can be taken advantage of, and since the file is large and the scan is sequential, there wouldn't be that much competition for seeking.

7c. Random accesses to a large file via an index

These should be stored on tracks near the center of the seek range, on adjacent tracks. The important part is that it is stored on adjacent tracks. This will help decrease the time waiting for the disk spinning, for random access.

7d. Sequential scans of a small file.

Similar to the sequential scans of a large file, sequential scanning of small files can take advantage of the higher data transfer speeds of the outer tracks. Since these files are small however, they might have more competition from other processes to seek elsewhere, in which case the outermost track would take a seek time penalty. Thus the outer tracks, but not **too** outer, would be the best.