Alan Ou
301320642

**Question 1**

1. (512 bytes/sector) x (256 sectors/track) x (10000 tracks/surface) x (2 surfaces/platter) x (8 platters/disk)

   = $2.097152 \times 10^{10}$ bytes x (1 GB / $2^{30}$ bytes) ≈ 19.53125 ≈ **19 GB**

2. (512 bytes/sector) x (256 sectors/track) = **131 072 bytes**

3. $\frac{(512 \text{ bytes/sector}) \text{ x } (256 \text{ sectors/track})}{(4096\ bytes/block)}$ **= 32 blocks**

4. RPM : (1 / 5400) * 60 = 1/90 seconds for one rotation

   36 x (7/256) + 324 x (8/256) = 11.109 degrees
   11.109 / 360 = .0308 rotation
   0.0308 rotation / 90 rotation/sec = **0.3422 ms**

   access time = seek time + rotational delay + transfer time

   = 0 + 0 + **0.3422 ms** = **0.3422 ms**

5. RPM : (1 / 5400) * 60 = 1/90 seconds for one rotation = 11.11111 ms

   access time = seek time + rotational delay + transfer time
   = (10000 tracks) x 0.001ms + 11.11111 ms + 0.3422 ms = **21.4533 ms**

6. RPM : (1 / 5400) * 60 = 1/90 seconds for one rotation = 11.11111 ms

   access time = seek time + rotational delay + transfer time
   = (10000 tracks / 3) x 0.001ms + (11.11111 ms / 2) + 0.3422 ms = **9.2312 ms**

Alan Ou
301320642

**Question 2**

a) **Using the LRU replacement policy**

Page requests: <a b c a d e c b a b>  - marks the least recently used page  * marks a page fault

| Block read | a | b | c | a | d | e | c | b | a | b |
|---|---|---|---|---|---|---|---|---|---|---|
| Frame 0 | -a | a | a | a | a | -a | c | c | -c | c |
| Frame 1 | | b | b | -b | d | d | -d | b | b | b |
| Frame 2 | | | c | c | -c | e | e | -e | a | a |
| Page Fault | * | * | * | | * | * | * | * | * | |

Using the LRU replacement policy, the set <a b c a d e c b a b> will have **8 page faults.**

Page requests: <r s t u r s t u t u w u s w>  - marks the least recently used page  * marks a page fault

| Block read | r | s | t | u | r | s | t | u | t | u | w | u | s | w |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Frame 0 | -r | r | r | u | u | -u | t | t | t | t | -t | t | s | s |
| Frame 1 | | s | s | -s | r | r | -r | u | u | u | u | u | u | -u |
| Frame 2 | | | t | t | -t | s | s | -s | s | s | w | w | -w | w |
| Page Fault | * | * | * | * | * | * | * | * | | | * | | * | |

Using the LRU replacement policy, the set <r s t u r s t u t u w u s w> will have **10 page faults.**

Alan Ou
301320642

**b) Using the MRU replacement policy**

Page requests: <a b c a d e c b a b>  - marks the newest page  * marks a page fault

| Block read | a | b | c | a | d | e | c | b | a | b |
|---|---|---|---|---|---|---|---|---|---|---|
| Frame 0 | -a | a | a | -a | -d | -e | e | e | e | e |
| Frame 1 | | -b | b | b | b | b | b | -b | -a | -b |
| Frame 2 | | | -c | c | c | c | -c | c | c | c |
| Page Fault | * | * | * | | * | * | | | * | * |

Using the MRU replacement policy, the set <a b c a d e c b a b> will have **7 page faults.**

Page requests: <r s t u r s t u t u w u s w>  - marks the newest page  * marks a page fault

| Block read | r | s | t | u | r | s | t | u | t | u | w | u | s | w |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Frame 0 | -r | r | r | r | -r | r | r | r | r | r | r | r | r | r |
| Frame 1 | | -s | s | s | s | -s | -t | t | -t | t | t | t | t | t |
| Frame 2 | | | -t | -u | u | u | u | -u | u | -u | -w | -u | -s | -w |
| Page Fault | * | * | * | * | | | * | | | | * | * | * | * |

Using the MRU replacement policy, the set <r s t u r s t u t u w u s w> will have **9 page faults.**

Alan Ou
301320642

**c) Using the clock replacement policy. Provided that no page is written on disk and each page is pinned and then unpinned immediately, therefore pin count will always be 0. In addition, as stated in the question, frame pointer starts with frame 0 each time.**

Page requests: <a b c a d e c b a b> * marks a page fault

| Block read | a | b | c | a | d | e | c | b | a | b |
|---|---|---|---|---|---|---|---|---|---|---|
| Frame 0 | a/1 | a/1 | a/1 | a/1 | d/1 | d/0 | d/0 | b/1 | a/1 | a/0 |
| Frame 1 | | b/1 | b/1 | b/1 | b/0 | e/1 | e/1 | e/1 | e/0 | b/1 |
| Frame 2 | | | c/1 | c/1 | c/0 | c/0 | c/1 | c/1 | c/0 | c/0 |
| Page Fault | * | * | * | | * | * | | * | * | * |

Using the MRU replacement policy, the set <a b c a d e c b a b> will have **8 page faults.**

Page requests: <r s t u r s t u t u w u s w> * marks a page fault

| Block read | r | s | t | u | r | s | t | u | t | u | w | u | s | w |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Frame 0 | r/1 | r/1 | r/1 | u/1 | u/0 | s/1 | s/1 | u/1 | u/1 | u/1 | u/0 | u/1 | s/1 | s/1 |
| Frame 1 | | s/1 | s/1 | s/0 | r/1 | r/1 | r/1 | r/0 | r/0 | r/0 | w/1 | w/1 | w/0 | w/1 |
| Frame 2 | | | t/1 | t/0 | t/0 | t/0 | t/1 | t/0 | t/1 | t/1 | t/1 | t/1 | t/0 | t/0 |
| Page Fault | * | * | * | * | * | * | | * | | | * | | * | |

Using the MRU replacement policy, the set <r s t u r s t u t u w u s w> will have **9 page faults.**

**Question 3**

$\frac{1024-24}{12+20+8+10}$ **= 20 records**

Alan Ou
301320642

**Question 4**

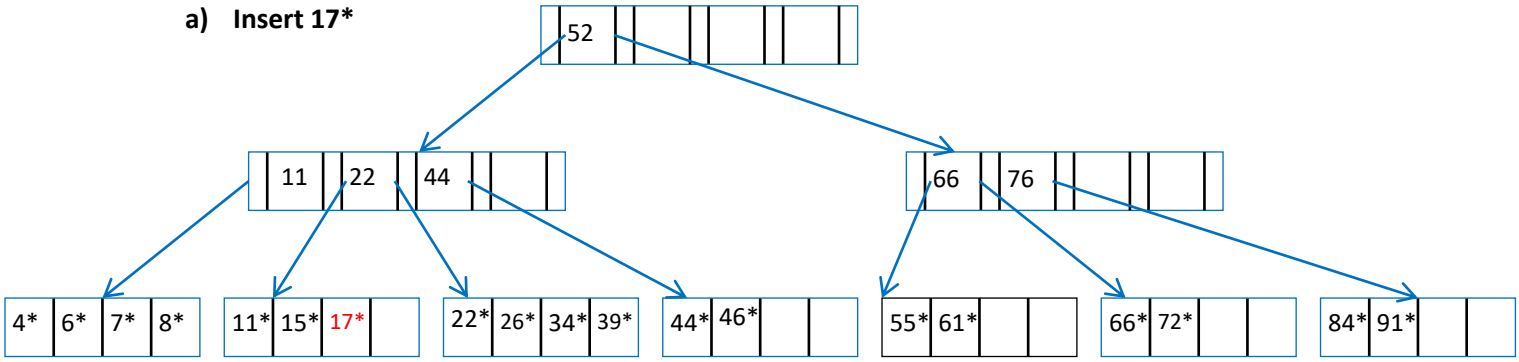
Node with record numbers 77-83, 84-90, 91-97, 98-104, 105-111, 112-118

1 + 1 + 1 + 6 = **9 nodes**

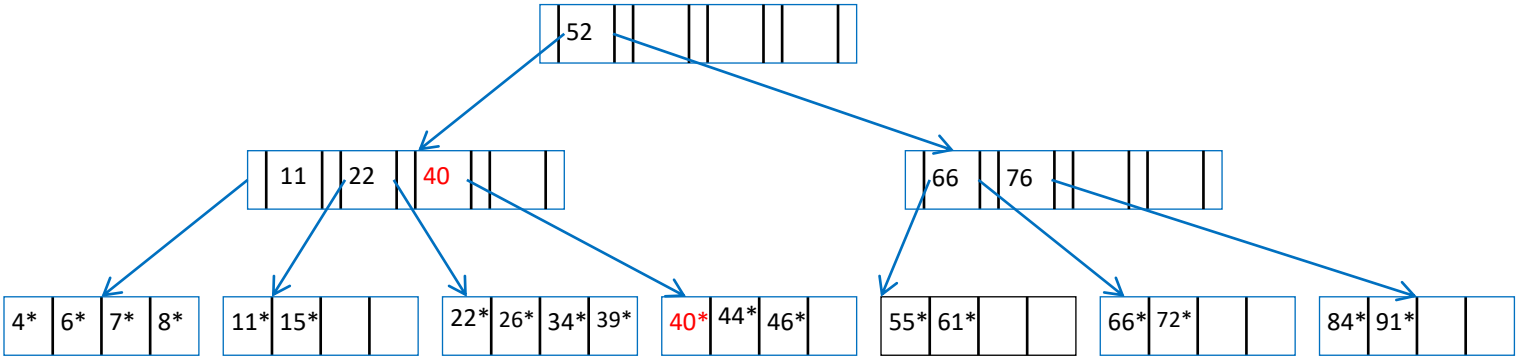
Therefore **9 nodes** must be examined to find records with keys in the range [82,113].

**Question 5**

**a) Insert 17***

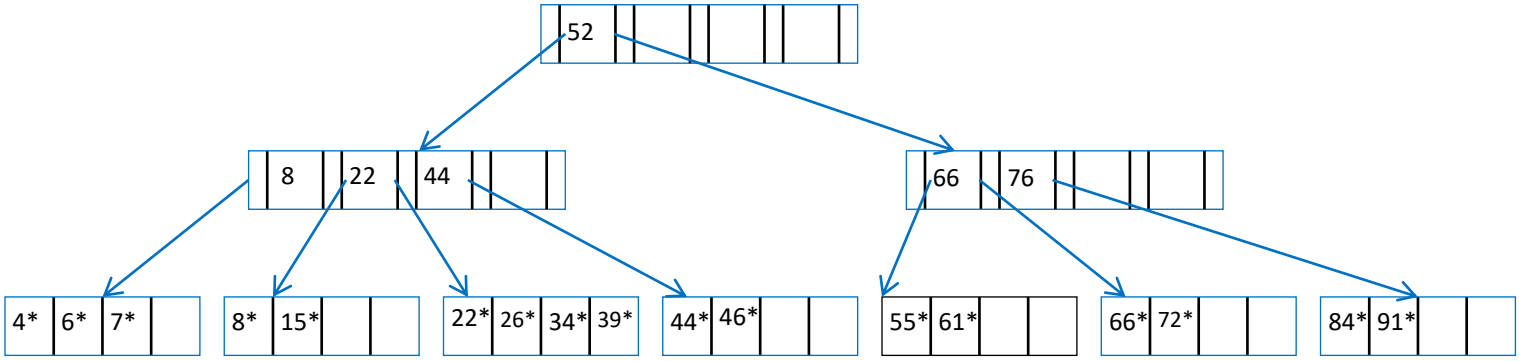



b) Insert 40*

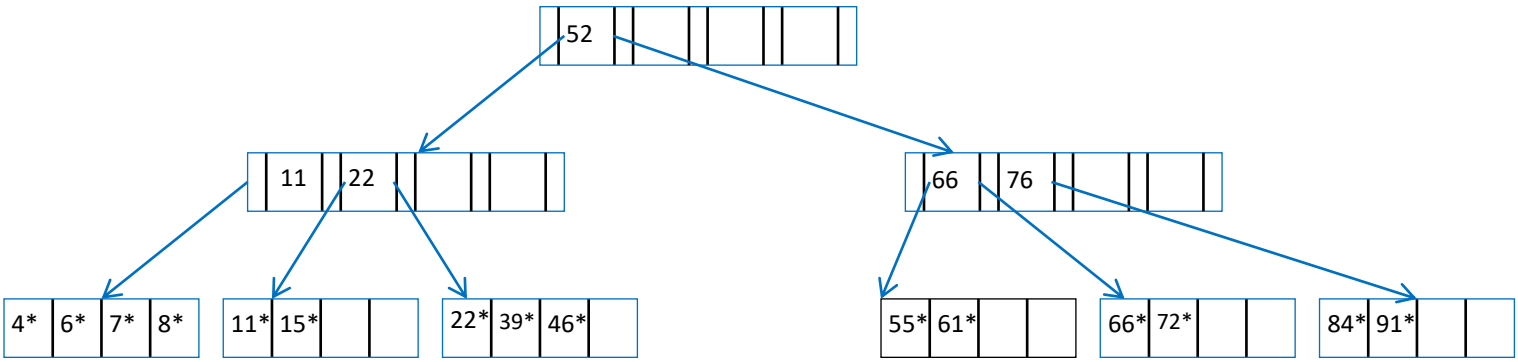


Alan Ou
301320642

c) Delete 11




d) Delete 26, 34 and 44




**Question 6**

a) Maximum number of record pointers = $2(5)(2(5)+1)^4$ = 146 410 records

b) Minimum number of record pointers = $2(\frac{5+1}{2})^4 = 162\ records$

**Question 7**

a) Page size = 4 194 304 bytes  67% full = 2 810 183 bytes

(number of pointers)(pointer size) + (number of pointers -1) key size = page size
16(number of pointers) + 12((number of pointers) - 1) = 2 810 183
number of pointers = 100 364 pointers = 100 363 entries

Number of leaf pages = Ceiling ( 100 000 000 / 100 363 ) = 996.38 = **997 leaf nodes**

b) Height: ceiling($\log_{100364}997$) = 0.599 = 1

Since there are height + 1 levels, there are **2 levels**.

c) Assume that we are using a heap file instead of a B+ tree, the average cost of finding a user in TwitterUsers would be:

Page size = 4 194 304 bytes  Assuming 67% full = 2 810 183 bytes

2 810 183 / 250 bytes each = 11 240 records per page

100 000 000 records / 11 240 records per page = 8896.797 = 8897 pages

B – Number of pages in the file
D – Time needed to read/write one disk page
R – Number of records on a page
C – CPU time needed to process a record – assuming

Search heap = 0.5B (D + RC) = **0.5 (8897 pages) (D + 11 240 C)** = **Cost**
**= 4448.5D + 50 001 140C**

d) **Cost =** $\log_{100364}997$ = **0.5995**

**Question 8**

a) If a university database in a way that student ID numbers only increase, with no reuse of student ID numbers, and graduated students' student ID numbers are never deleted, then eventually, there will be the need to increase the height of the B+ tree.

b) Although reusing student IDs of graduated students or deleting student records after their graduation would best improve the insertion performance, we can seek an alternative approach. One alternative would be to use key compression. Key compression increases fanout, thereby allows more keys to be fit in one page, and thereby reduces height and makes searching more efficient by reducing the number of disk I/Os required to retrieve a data entry. Furthermore, since B+ trees are sorted, insertion and deletion operations require searching the B+ tree beforehand, therefore the increased searching efficiency of key compression also helps with insertion performance.

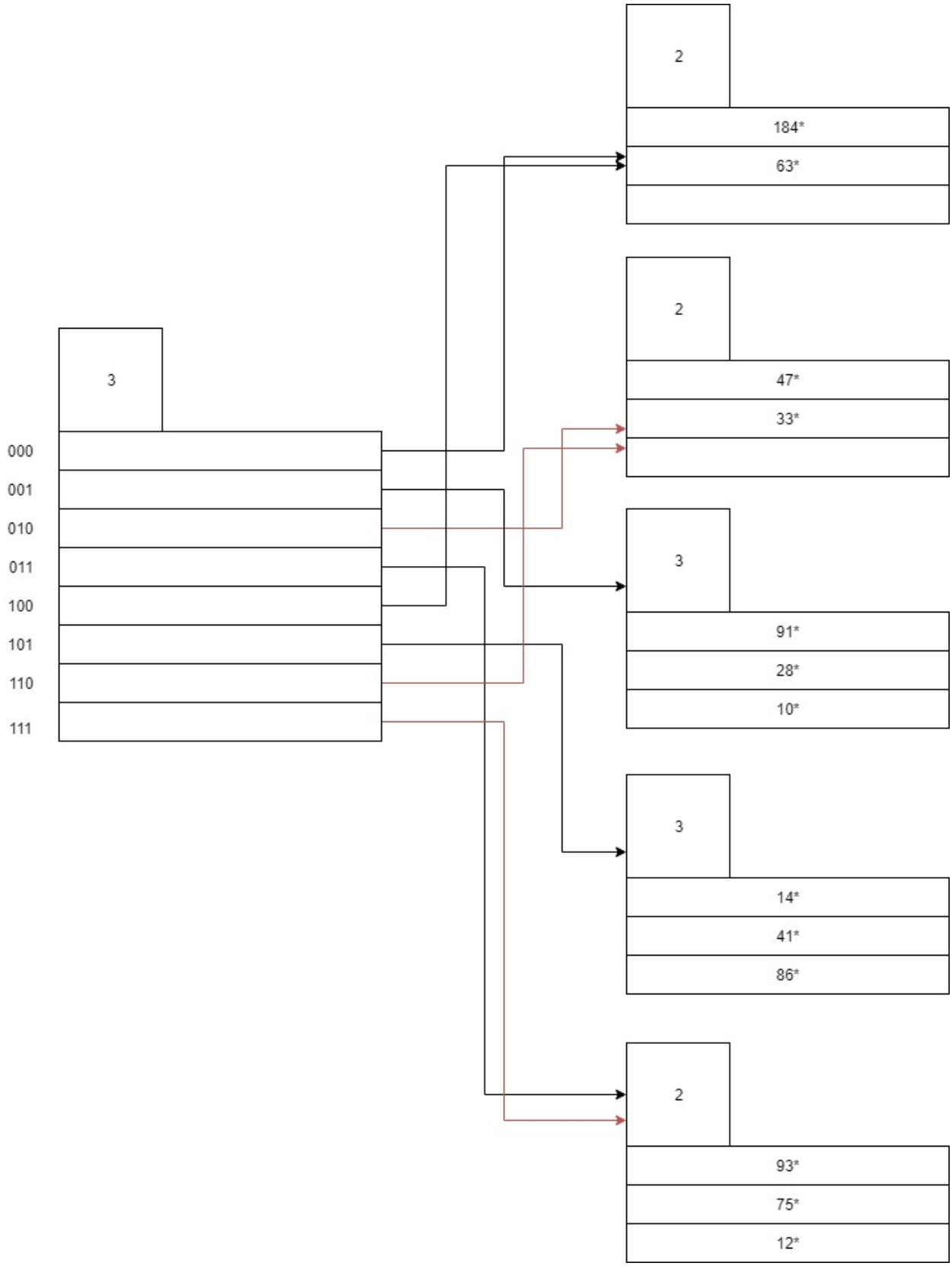
When inserting a data entry, we can use a hash function to compress and shorten the key values. By shortening the key values, we can store more keys within a page, and therefore keep the height low. By keeping the height low, there will be reduced number of disk I/Os, and thereby increased insertion performance.

Alan Ou
301320642

**Question 9**

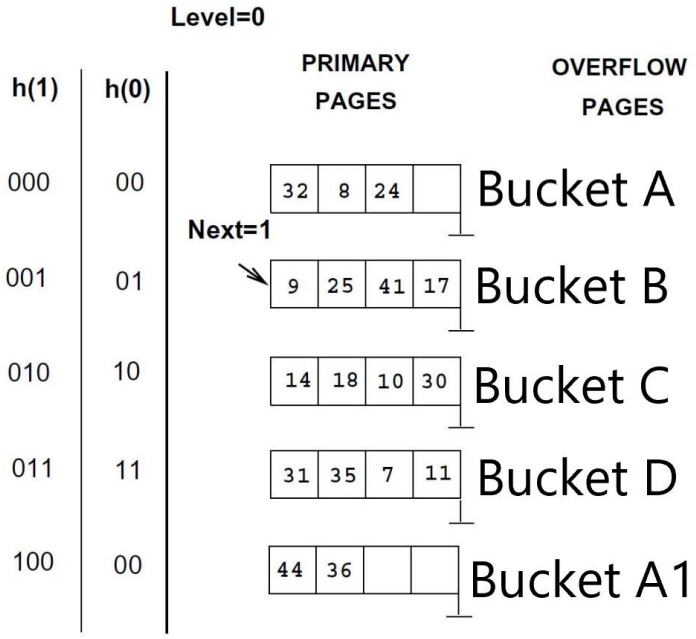
h(93) = 93 mod 9 = 3 = 000011
h(91) = 91 mod 9 = 1 = 000001
h(184) = 184 mod 9 = 4 = 000100
h(63) = 63 mod 9 = 0 = 000000
h(28) = 28 mod 9 = 1 = 000001
h(47) = 47 mod 9 = 2 = 000010
h(75) = 75 mod 9 = 3 = 000011
h(14) = 14 mod 9 = 5 = 000101
h(41) = 41 mod 9 = 5 = 000101
h(86) = 86 mod 9 = 5 = 000101
h(10) = 10 mod 9 = 1 = 000001
h(33) = 33 mod 9 = 6 = 000110
h(12) = 12 mod 9 = 3 = 000011

Alan Ou
301320642



Alan Ou
301320642

**Question 10**



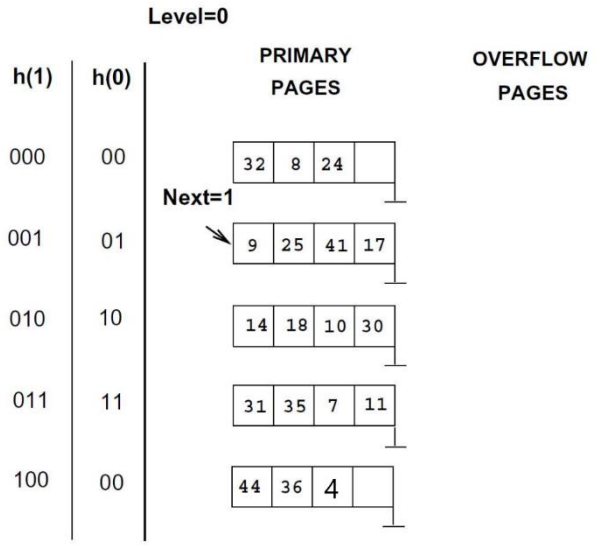

a) Since there is no list of key values, any key value to the most right of each bucket could be the last entry. For example, 24 could be the last entry in the sequence 9, 25, 42, 17, 14, 18, 10, 30, 31, 35, 7, 11, 44, 36, 32, 8, 24. The last insertion of the key value 24 caused a split. A deletion of the last entry 24 would cause a merge back with the original bucket.

b) The last insertion could have been the split causing key entry because bucket A and bucket A1 add up to five key entries. The insertion of either 36 or 24 would have caused a splitting of bucket A.

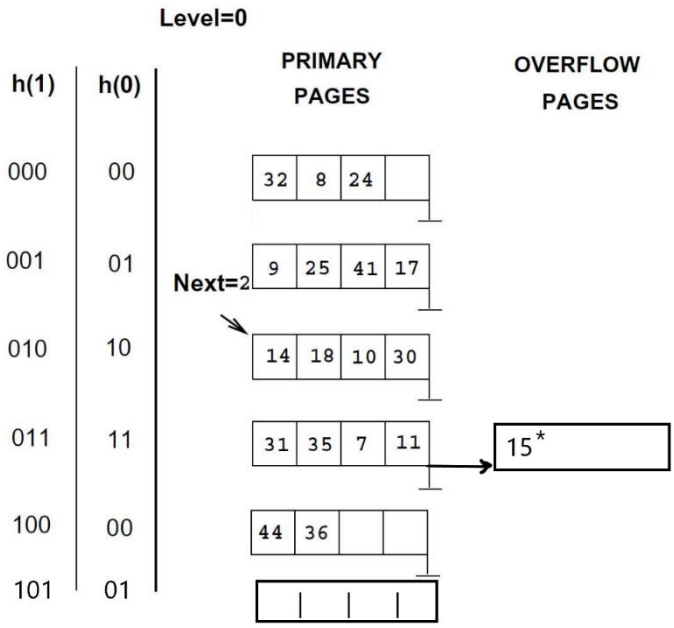
c) Since there have not been any deletions, and the last entry whose insertion into the index caused a split, the inserted entry must have been in bucket A or A1 because after splitting, there must be room for new entries. But since bucket B, C, and D are all full, the entry must be in bucket A or A1.

Alan Ou
301320642

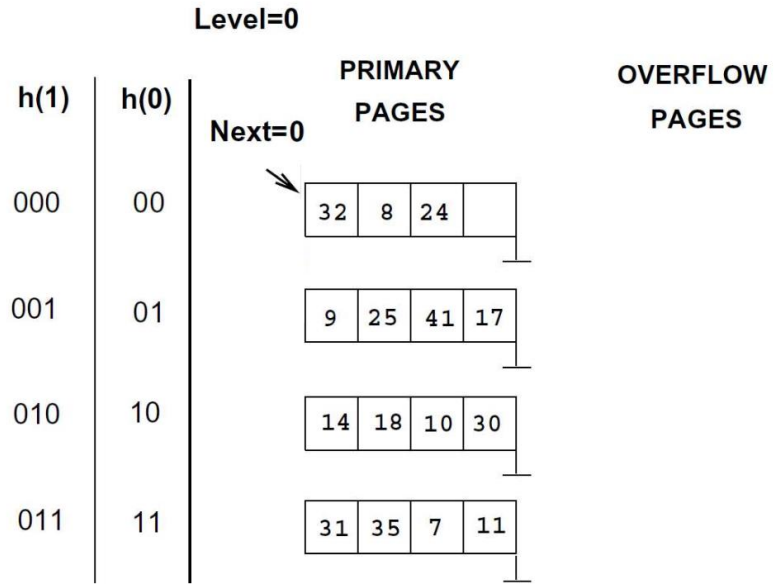
d)




e)



f)




g) The following is the minimal list of entries required to lead to a bucket with two overflow pages:

3, 27, 59, 123, 251

The insertion of 3 would cause the first split and Next to move to 2, while the insertion of 251 would cause the second split and Next to move to 3, pointing at the bucket we are currently inserting to. If we only want two overflow buckets, then we can insert at most 3 more entries to a maximum total of 8 entries before another split happens on this bucket, redistributing the entries.

However, if there is no maximum number of overflow buckets, then by $\sum_{i=3}^{n} 011 + 10^{i}$ , we can cause the directory to double, and eventually delay the splitting, causing long overflow pages.

**Question 11**

a) No, this approach is not efficient because all students within the same department will be placed in the same bucket. By using the student ID number as the search key, through the hash function h(StudentID) = StudentID mod 1000, all student within the engineering department would have the search key 101, and so on. This would result in poor performance because when we do a search, we must search through all pages in the bucket, in other words, this means that we must search through all students IDs within the department.

b) A solution to this problem would simply be to find a better hashing function. One simple approach would be to choose a modulo number other than 1000, preferably a prime number, such as **h(StudentID) = StudentID mod 997**. By using this hash function, the student records would be more evenly distributed among the buckets.

Alan Ou
301320642

**Question 12**

**Workload I:**
250,000 queries:
select * from R where s < ?

1. Attributes the index should cover: s
2. Clustered or unclustered: clustered
3. Hash-based or tree-based index: tree-based

20,000 queries:
select * from R where t = ?

1. Attributes the index should cover: t
2. Clustered or unclustered: unclustered
3. Hash-based or tree-based index: hash-based

There should be two indexes created for this workload: a clustered tree-based indexing for the 250 000 queries since tree-based indexing is best for range selections, and a unclustered, hash-based index for the 20,000 queries because hash-based indexing is faster for equality searches, which this query has.

For the clustered tree-based indexing covering 250 000 queries, the index should only cover the attribute <s> because using the index, we can use it to receive tuples that satisfy s < ?. The index should be clustered because all records with s < ? will be clustered together allowing for fewer disk I/Os.

For the unclustered hash-based indexing covering 20 000 queries, the index should only cover the attribute <t> because using the index, we can use it to receive tuples that satisfy t = ?. The index should not be clustered because this query (20 000 queries) is only a portion of the query containing 250 000 queries, and since there can only be one clustered index per table, the clustering should go to the index that supports the most queries.

**Workload II:**
250,000 queries:
select s, u from R where u < ?

1. Attributes the index should cover: u
2. Clustered or unclustered: clustered
3. Hash-based or tree-based index: tree-based

25,000 queries:
select * from R where s < ?

Since hash-based indexing is only best for equality selections, and also since the majority of the queries (250 000 queries) are not equality selections, a tree-based indexing technique would best improve performance.

Furthermore, since queries must be a prefix subset of the index, the index should only cover the attribute <u> because using the index, we can use it to receive tuples that satisfy u < ?, the majority of the queries. Considering that only 10% of the queries will be used to find s < ?, that we already used a clustered index, that only one clustered index is allowed per table, and that hash based indexing are only for equality searches, there should not be any indexing on s because if we choose to use a unclustered index, there could be in the worst case, one I/O per qualifying record, making the operation more expensive than sequential scanning.

In addition, the index should be clustered because all records with u < ? will be clustered together allowing for fewer disk I/Os.

Alan Ou
301320642

**Workload III:**
250,000 queries:
select * from R where u < ? and w = ?

1. Attributes the index should cover: w, u
2. Clustered or unclustered: clustered
3. Hash-based or tree-based index: tree-based

25,000 queries:
select * from R where s = ?

1. Attributes the index should cover: s
2. Clustered or unclustered: unclustered
3. Hash-based or tree-based index: hash-based

2,500 queries:
select * from R where t = ?

There should be two indexes created for this workload: a clustered tree-based indexing for the 250 000 queries since tree-based indexing is best for range selections and equality searches, and a unclustered, hash-based index for the 25 000 queries because hash-based indexing is faster for equality searches, which this query has.

For the clustered tree-based indexing covering 250 000 queries, the index should cover the attributes <w, u> because using the index, we can use it to receive tuples that first satisfy w = ?, then within those tuples, records that satisfy u < ?. The index should be clustered because all records with w =? and u < ? will be clustered together allowing for fewer disk I/Os.

For the unclustered hash-based indexing covering 25 000 queries, the index should only cover the attribute <s> because using the index, we can use it to receive tuples that satisfy s = ?. The index should not be clustered because this query (25 000 queries) is only a portion of the query containing 250 000 queries, and since there can only be one clustered index per table, the clustering should go to the index that supports the most queries.

Since there is a limit of two indexes allowed per workload, the query with the least number of queries (2 500 queries) will not be provided with an index.

**Extra Credit Question**

A column-oriented database is as its name suggests, stores data tables by column instead of by row. A row-oriented database stores tables by rows. The advantages of using a column-oriented data are:

- They are quicker in processing Queries that involve only a few columns/attributes
- And therefore is also quicker at processing aggregation queries such as sum, average, etc.