Theory Assignment 3: Sample Solutions

Part 1: Hashing

1. Which of the following 3 choices would make the best hash function for an 11-digit account number (assuming a random digit between 0 and 9 is assigned for each of the 11 positions)? Briefly, justify your answer by analyzing and comparing the 3 hash functions given below: $h_i(k)$. We'll use open addressing with 1000 array cells, and we'll use linear probing to resolve collisions. There are 250 keys to hash.

   a) Hash function $h_1(k)$ = floor(sqrt(k)) % 1000

   b) Take the 4 digits in the positions 2, 4, 6, and 8 (offsets start at 0, like a C++ array); call them *a*, *b*, *c*, and *d*, respectively; concatenate them; and then compute $h_2(k)$ = (*abcd*) % 1000.   Note: *abcd* does *not* mean *a***b***c***d*.

   c) $h_3(k)$ = ((the number of zeroes in *k*) + (the number of ones in *k*) + (the number of twos in *k*) + ... + (the number of nines in *k*)) % 1000.

a) h(k) = floor(sqrt(k)) % 1000

This hash function is actually quite good <u>if the keys are generated randomly</u> (i.e., the keys are not generated by the user in ascending order).

I (Ed) wrote a program to deal with integers (this was actually for 9 digits rather than 11 digits, so that I could stick with integers). But here are my observations: for numbers in the range 000,000,000 to 999,999,999 (i.e., 1 billion keys in that 9-digit sequence), we get this distribution among the 1000 hash table (array) cells:

```
Enter the lower bound for the keys to be generated: 0
Enter the upper bound for the keys to be generated: 999999999
Generating 1000000000 sequential keys from 0 to 999999999

The array cells are as follows:
0 to 9: 992031 992096 992160 992224 992288 992352 992416 992480 992544 992608
10 to 19: 992672 992736 992800 992864 992928 992992 993056 993120 993184 993248
20 to 29: 993312 993376 993440 993504 993568 993632 993696 993760 993824 993888
...
980 to 989:990791 990853 990915 990977 991039 991101 991163 991225 991287 991349
990 to 999:991411 991473 991535 991597 991659 991721 991783 991845 991907 991969

min_cell_count = 968657
max_cell_count = 1031776 (a 6.5% difference)


For 2 billion keys:
min_cell_count = 1955580
max_cell_count = 2044845 (a 4.6% difference)


For 4 billion keys:
min_cell_count = 3937059
max_cell_count = 4063296 (a 3.2% difference)
```

The counts seem to be fairly uniform in the hash table, and they're converging to uniformity. Note that 11-digit numbers go up to about 100 billion. My program uses unsigned integers which go up to about 4 billion.

For 4 billion keys, what if we use "mod 100" (i.e., *m* = 100 for the size of the hash table) instead of "mod 1000"? Answer: We get an even tighter fit, so this suggests with 11 digits, we'd get a better fit for 1000 cells.

```
min_cell_count = 39937976
max_cell_count = 40061937 (0.3% difference)
```

A similar convergence trend develops with "mod 10000" (i.e., *m* = 10000 for the size of the hash table).

There is a downside if the keys are generated in ascending order. Ranges of consecutive numbers will hash to the same cell: think about small ascending keys like 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, etc. yielding 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 4, etc. So, our 250 keys in this question could all map to the same hash value. Note that keys 123456789 and (123456789 + 250)—and all numbers in between these two values—map to the same hash value: 11111.

But, the bottom-line is that for randomly generated keys, the hash function is pretty good. Let's see if the other hash functions are better.

b) If the 11-digit keys are randomly generated, we expect the digits in positions 2, 4, 6, and 8 to appear with equal probability. As per the question, we assume that position 0 is the leading digit.

   By the way, notice that we are taking modulus (%) 1000, which means we only take the last 3 digits. Thus, we're only looking at positions 4, 6, and 8—and these positions correspond to *bcd*; but, *a* is ignored.

   If the keys are generated sequentially, then there is a bias to the lower-numbered hash cells, since the first 100 numbers all have the same digits 4, 6, and 8; then the next 100 keys have the same digits 4, 6, and 8; and finally the next 50 keys have the same digits 4, 6, and 8. So, our 250 numbers all map to the same 3 hash values, and those 3 hash values share the same *b* and *c*.

   Nevertheless, if we assume that the numbers are generated randomly, then the hash function is actually quite good.

It has a better distribution than (a). But, (a) isn't that much worse. Since we're generating only 250 keys, either will perform well, and it's possible that for 250 randomly generated keys, (a) will beat (b), or (b) will beat (a)—in terms of uniformity.

c) This is a bad hash function. There are 11 digits, and therefore the sum of the digits of the key corresponding to the given hash function always add up to 11. This means that all the keys collide at 11. Avoid this hash function.

Summary: Hash function (b) is the best choice, but we'll accept either (a) or (b) as being approximately the same (in terms of distribution) for 250 randomly generated keys. Hash function (c) is bad.

2. Using a hash table with *m* = 11 locations and the hash function h(*k*) = *k* % 11, show the hash table that results when the following integers are inserted in this order:

26, 42, 5, 44, 92, 59, 40, 36, 12, 60, 80

… if we assume that collisions are resolved using the following techniques. Note: If your collisions cannot be resolved within *m* tries, write down the failing case.

a) Linear Probing

44	12	80	36	26	5	92	59	40	42	60
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

h(26) = 26 % 11 = 4
h(42) = 42 % 11 = 9
h(5) = 5 % 11 = 5
h(44) = 44 % 11 = 0
h(92) = 92 % 11 = 4 $\Rightarrow$ collision, so go to 5 $\Rightarrow$ 6
h(59) = 59 % 11 = 4 $\Rightarrow$ 5 $\Rightarrow$ 6 $\Rightarrow$ 7
h(40) = 40 % 11 = 7 $\Rightarrow$ 8
h(36) = 36 % 11 = 3
h(12) = 12 % 11 = 1
h(60) = 60 % 11 = 5 $\Rightarrow$ 6 $\Rightarrow$ 7 $\Rightarrow$ 8 $\Rightarrow$ 9 $\Rightarrow$10
h(80) = 80 % 11 = 3 $\Rightarrow$ 4 $\Rightarrow$ 5 $\Rightarrow$ 6 $\Rightarrow$ 7 $\Rightarrow$ 8 $\Rightarrow$ 9 $\Rightarrow$10 $\Rightarrow$ 0 $\Rightarrow$ 1

b) Quadratic Probing

44	12	59	36	26	5	60	40	92	42	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

h(26) = 26 % 11 = 4	
h(42) = 42 % 11 = 9	
h(5) = 5 % 11 = 5	
h(44) = 44 % 11 = 0	
h(92) = 92 % 11 = 4	$\Rightarrow$ collision, so go to: 4 % 1^2 (mod 11) = 5
	$\Rightarrow$ 4 + 2^2 (mod 11) = 8
h(59) = 59 % 11 = 4	$\Rightarrow$ 4 + 1^2 (mod 11) = 5
	$\Rightarrow$ 4 + 2^2 (mod 11) = 8
	$\Rightarrow$ 4 + 3^2 (mod 11) = 2
h(40) = 40 % 11 = 7	
h(36) = 36 % 11 = 3	
h(12) = 12 % 11 = 1	
h(60) = 60 % 11 = 5	$\Rightarrow$ 5 + 1^2 = 6
h(80) = 80 % 11 = 3	$\Rightarrow$ 3 + 1^2 (mod 11) = 4 (collision #1)**—all mod 11**
	$\Rightarrow$ 3 + 2^2 (mod 11) = 7 (collision #2)
	$\Rightarrow$ 3 + 3^2 (mod 11) = 1 (collision #3)
	$\Rightarrow$ 3 + 4^2 (mod 11) = 8 (collision #4)
	$\Rightarrow$ 3 + 5^2 (mod 11) = 6 (collision #5)
	$\Rightarrow$ 3 + 6^2 (mod 11) = 6 (collision #6)
	$\Rightarrow$ 3 + 7^2 (mod 11) = 8 (collision #7)
	$\Rightarrow$ 3 + 8^2 (mod 11) = 1 (collision #8)
	$\Rightarrow$ 3 + 9^2 (mod 11) = 7 (collision #9)
	$\Rightarrow$ 3 + 10^2 = 4 (collision #10)
	$\Rightarrow$ 3 + 11^2 = 3 (collision #11)

Conclusion: It looks like it loops forever; so, we would have to resize the table since key 80 will not fit into the table. We'll stop here, as per the instructions.

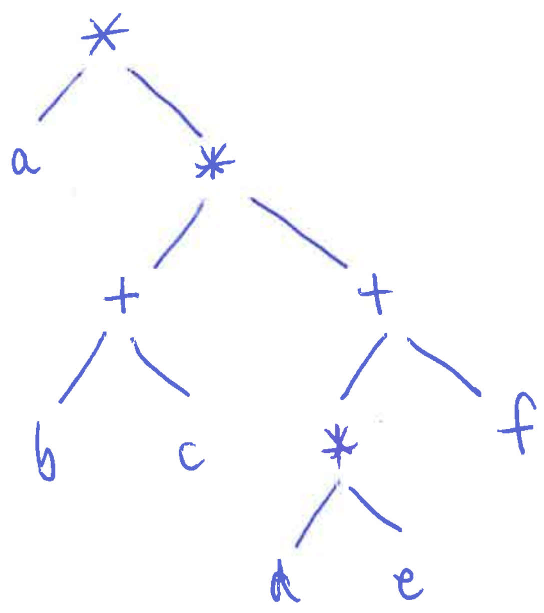
c) **Double Hashing** with the following secondary hash function:

$h_2(x) = (2x) \% 11$, if this expression is non-zero
$h_2(x) = 1$, if the expression in the previous line is zero

44	92	12	36	26	5	59	40	80	42	60
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

h(26) = 26 % 11 = 4
h(42) = 42 % 11 = 9
h(5) = 5 % 11 = 5
h(44) = 44 % 11 = 0
h(92) = 92 % 11 = 4 ⇒ collision, so go to: h(92) + 1*h_2(92)—**all mod 11**
= 4 + 1(184 % 11) = (4 + 1*8) % 11 = 1
h(59) = 59 % 11 = 4 ⇒ 4 + 1(118 % 11) = 4 + 1(8) % 11 = 1
⇒ 4 + 2(8) = 20 % 11 = 9
⇒ 4 + 3(8) = 28 % 11 = 6
h(40) = 40 % 11 = 7
h(36) = 36 % 11 = 3
h(12) = 12 % 11 = 1 ⇒ 1 + 1(24 % 11) = 1 + 1(2) % 11 = 3
⇒ 1 + 2(2) = 5
⇒ 1 + 3(2) = 7
⇒ 1 + 4(2) = 9
⇒ 1 + 5(2) = 11 % 11 = 0
⇒ 1 + 6(2) = 13 % 11 = 2
h(60) = 60 % 11 = 5 ⇒ 5 + 1(120 % 11) = (5 + 10) % 11= 4
⇒ 5 + 2(10) = 25 % 11 = 3
⇒ 5 + 3(10) = 35 % 11 = 2
⇒ 5 + 4(10) = 45 % 11 = 1
⇒ 5 + 5(10) = 55 % 11 = 5
⇒ 5 + 6(10) = 65 % 11 = 10
h(80) = 80 % 11 = 3 ⇒ 3 + 1(160 % 11) = 3 + 1(6) = 9
⇒ 3 + 2(6) = 15 % 11 = 4
⇒ 3 + 3(6) = 21 % 11 = 10
⇒ 3 + 4(6) = 27 % 11 = 5
⇒ 3 + 5(6) = 33 % 11 = 0
⇒ 3 + 6(6) = 39 % 11 = 6
⇒ 3 + 7(6) = 45 % 11 = 1
⇒ 3 + 8(6) = 51 % 11 = 7
⇒ 3 + 9(6) = 57 % 11 = 2
⇒ 3 + 10(6) = 63 % 11 = 8

3. a)



Verify with a preorder visitation:

* a * + b c + * d e f ✓

b) Inorder visitation:

a * b + c * d * e + f

which when appropriately parenthesized yields:

a * (b + c) * ((d * e) + f)

4.) The only pairs that up to 16 are:

1 + 15 ← pick exactly one of these to begin our worst-case scenario

3 + 13 ← pick one

5 + 11 ← " "

7 + 9 ← " "

8 ← pick it
10 ← pick it

So, at this point, we have 6 numbers in our list — none of which pair up to yield 16.

The very next number you pick will pair up. Whatever number it is ("pigeon"), it will have to go into 1 of the 4 pair-case "pigeonholes" to add up to 16.

this will be the 7th one

∴ 7 unique numbers are needed to guarantee...

5.) There are only 20 integers between 1 and 100 that are divisible by 5: {5, 10, ..., 100}. So, we can pick 100 - 20 = 80 other numbers first. Then, the very next number we pick (the 81st one has to be divisible by 5). Answer: 81.

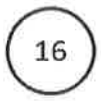
Part 4: AVL Trees

6. Draw the AVL tree *after each key is inserted.* When necessary, draw the re-balanced AVL tree and declare which rotation was used for rebalancing. For a double-rotation, draw the AVL tree after the first rotation in addition to the final result.

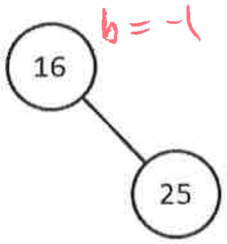
   - Insert the following nodes (one at a time) into an initially empty AVL tree:

16, 25, 29, 7, 12

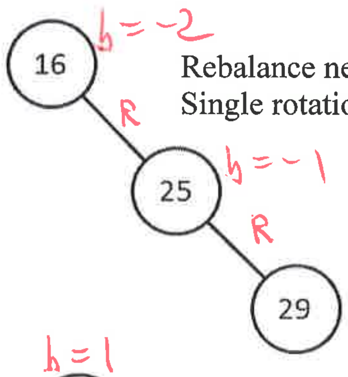
insert(16):



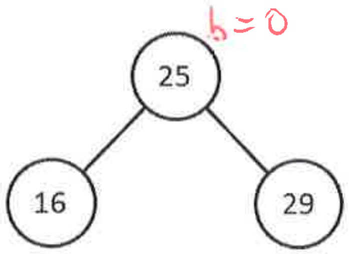
insert(25):



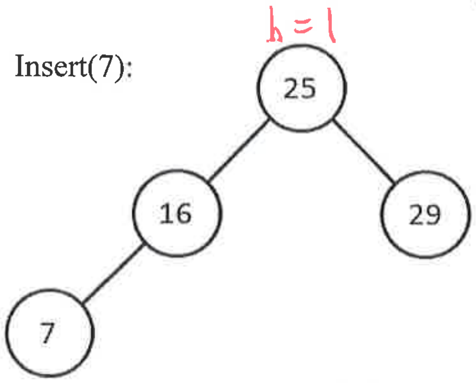
insert: 29:



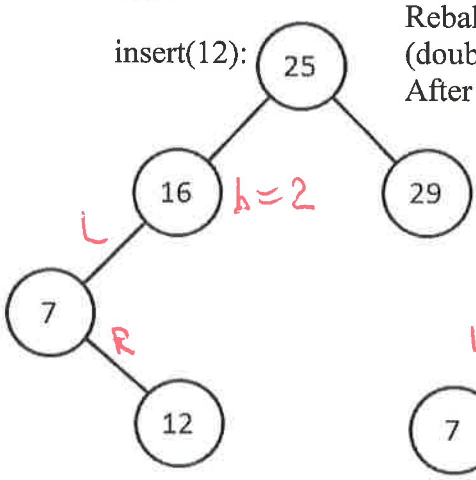
Rebalance needed
Single rotation (rotateLeft):



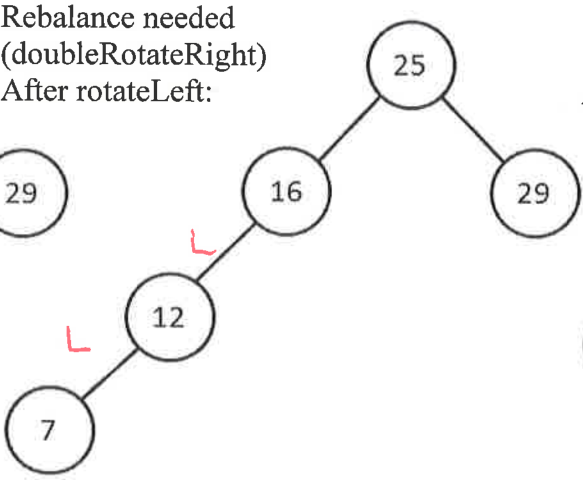
Insert(7):



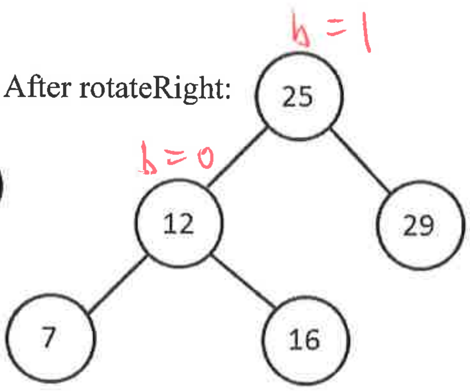
insert(12):



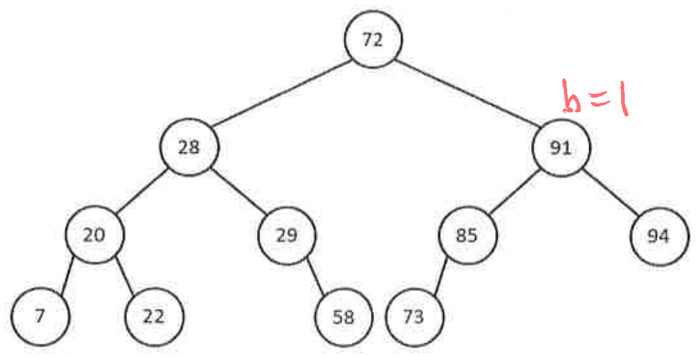
Rebalance needed
(doubleRotateRight)
After rotateLeft:



After rotateRight:

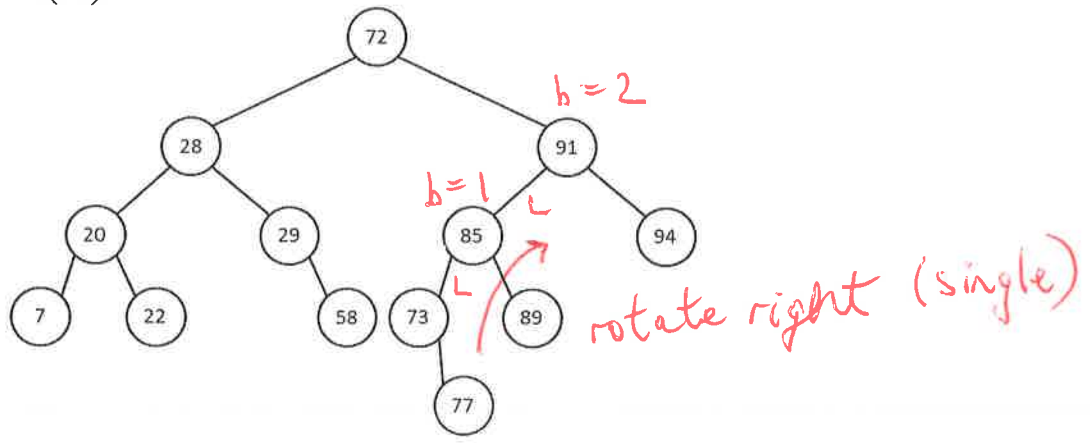


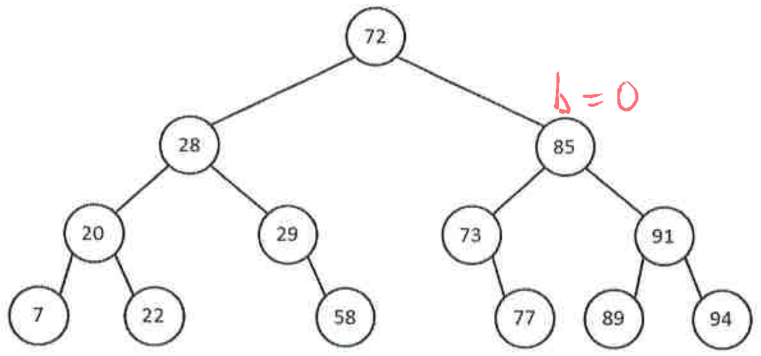
7. Repeat the same process as in Question 6, but assume the AVL already contains the nodes shown below. Insert 89, and then 77.



insert(89) then insert(77):

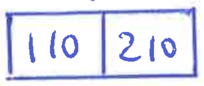
↑
still
balanced



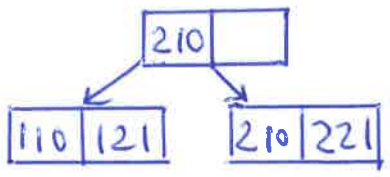


8.

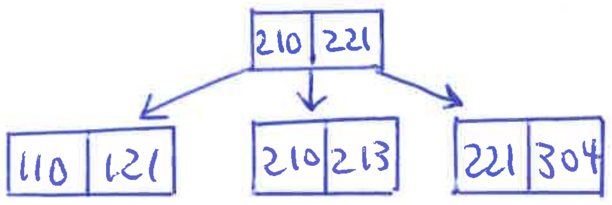
$\Downarrow$ 110, 210



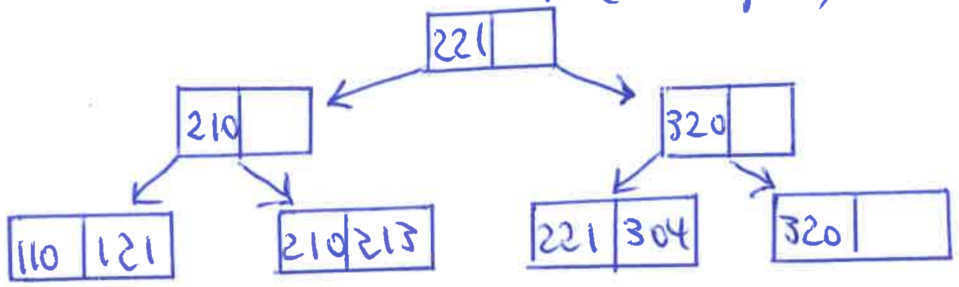
$\Downarrow$ 121 (causes split), 221 can be added right after that



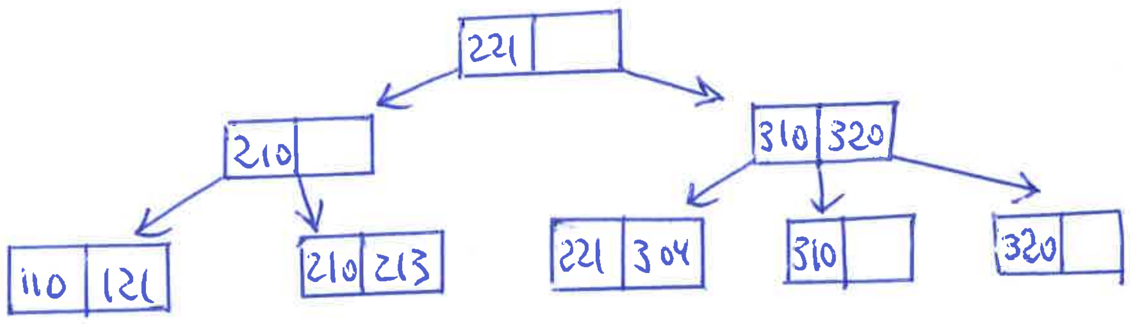
$\Downarrow$ 213 (causes split), 304



$\Downarrow$ 320 (causes split and increases the height)



$\Downarrow$ 310 (causes the final split)



9) Each data entry (D.E.) is made up of the key & its value (pointer):

$$\underset{\text{key}}{56 \text{ bytes}} + \underset{\text{value}}{8 \text{ bytes}} = 64 \frac{\text{bytes}}{\text{DE}}$$

a) max # of DE's per leaf page?

$$\left\lfloor \frac{4096 - 3(8)\frac{\text{bytes}}{\text{leaf page}}}{64 \frac{\text{bytes}}{\text{DE}}} \right\rfloor = \left\lfloor \frac{4072 \frac{\cancel{\text{bytes}}}{\text{leaf page}}}{64 \frac{\cancel{\text{bytes}}}{\text{DE}}} \right\rfloor$$

$$= 63 \frac{\text{DE}}{\text{leaf page}} \Big\} \text{ use } \underline{\underline{62}} \text{ for an even \#}$$

How many leaf pages do we need to hold 200,000 DE's?

$$\left\lceil \frac{200{,}000 \text{ DE}}{62 \frac{\text{DE}}{\text{leaf page}}} \right\rceil = \underline{3226 \text{ leaf pages}}$$

9 cont.

b) How many parents do we need to support 3226 leaf pages?

↙ I.E. (internal data entries)

The index entries (key + pointer pairs) in an internal node are the same size as for the leaf pages.

As per the course notes (Slide 10), we need one extra pointer field (for the last child node) — and we still have the 3 parent, left sibling, and right sibling pointers.

They still fit in 4096 bytes because $62\text{ I.E.}\left(64\ \frac{\text{bytes}}{\text{I.E.}}\right) + 8\text{ bytes} + 3(8\text{ bytes}) = 4000\text{ bytes} \leq 4096\text{ bytes}$.

∴ # of parents needed:

$$\left\lceil \frac{3226\text{ leaf pages}}{(62+1)\ \frac{\text{leaf pages}}{\text{parent page}}} \right\rceil = \underline{52\text{ parent pages}}$$

9 b) cont.

How many parent pages do we need to support 52 children?

$$\left\lceil \frac{52 \text{ children}}{(62+1)\ \frac{\text{children}}{\text{parent page}}} \right\rceil = 1 \text{ root page}$$

↑
stop here

Answer summary:

$$\text{internal pages}\begin{cases} 1 \text{ root page (with 51 keys)} \\ 52 \text{ level-1 pages} \end{cases}$$

$$\text{leaf pages}\ \big\{\ 3226 \text{ leaf pages}$$

{ filled to capacity where possible

- each node is $\geq$ half full except for the root