Project #2: DGIBox

Computer Algorithm
SE380

Instructor:

Prof. Sungjin Lee (sungjin.lee@dgist.ac.kr)

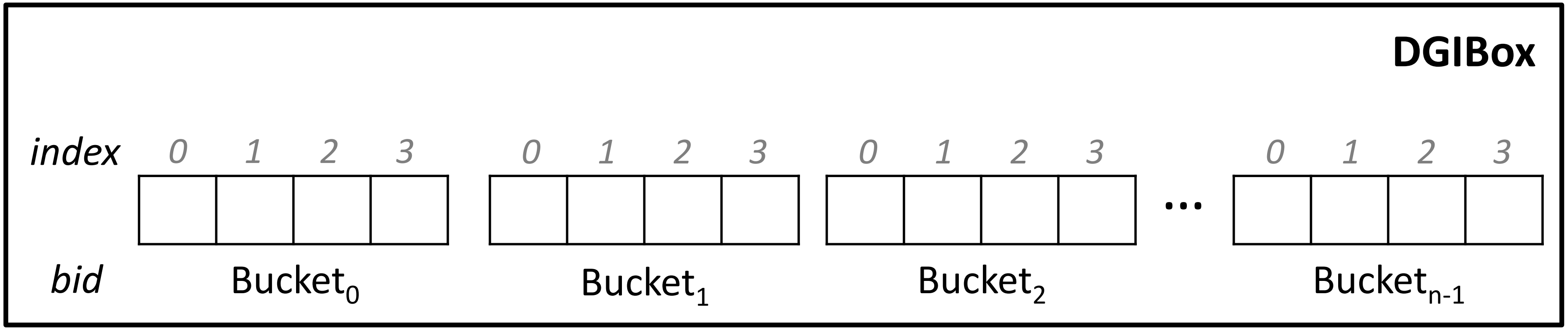
Project #2 Topic

- **Develop an algorithms providing basic operations (write and read) that operate on a magical box, called a DGIBox**

- **Use all of the knowledge you have learned from this class**
  - Sorting
  - Trees – Binary Search Tree, B-Tree, …
  - Hash

What is DGIBox?

- **DGIBox is composed of a large number n of small buckets**
  - Each bucket looks like an array, indexed by an integer number
  - You select any bucket in DGIBox and use it (e.g., inserting a key/value, …)

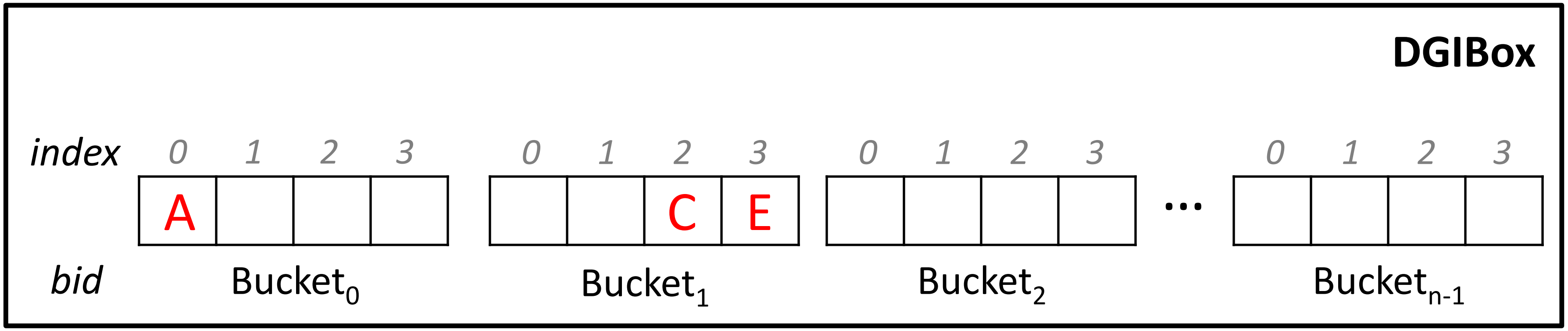


DGIBox with n buckets, each of which can hold 4 items

- Each Bucket provides simple three operations:
  - `Set(bid,index,value)`
  - `value=Get(bid,index)`
  - `Empty(bid)`

How to Use DGIBox?

- **Example**



DGIBox with n buckets, each of which can hold 4 items

```
Set(0, 0, 'A');
Get(0, 0); /* return 'A' */
Set(1, 2, 'C');
Get(0, 1); /* return nothing */
Set(1, 3, 'E');
Get(1, 2); /* return 'C' */
Empty (1);
Get(1, 3); /* return nothing */
```

Ok… What are Its Unique Properties?

- For each bucket, only append-only insertion (increasing order) is allowed

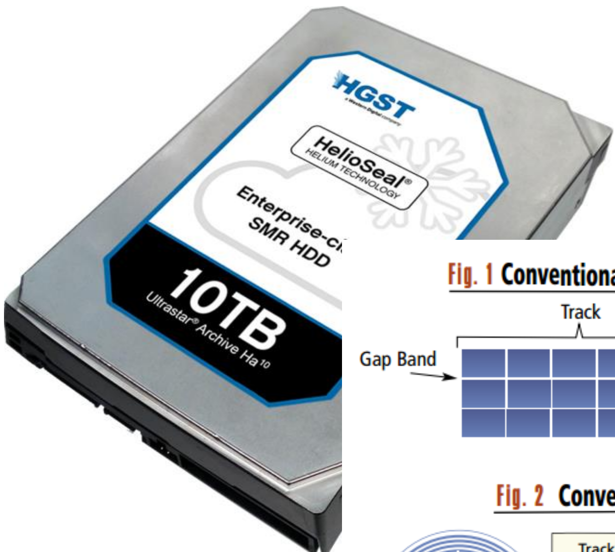
```
Set(0, 1, 'A');
Set(0, 2, 'B');
Set(0, 3, 'C');
Set(0, 0, 'D'); /* Not Allowed */
```

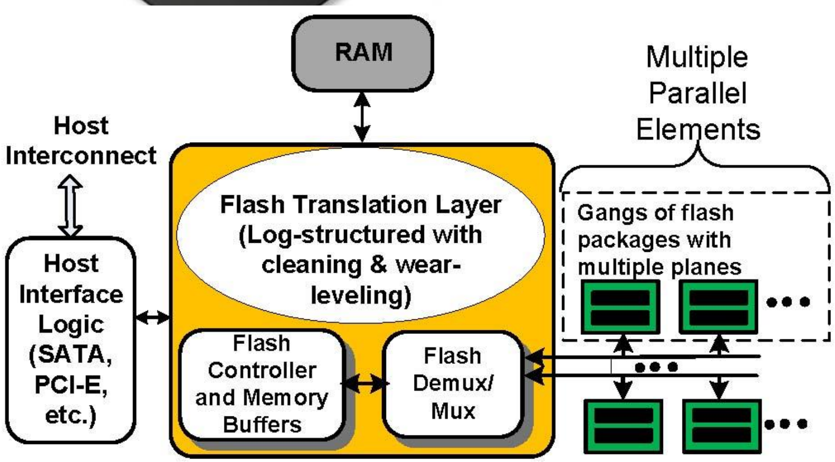
- For each bucket, overwrites are prohibited

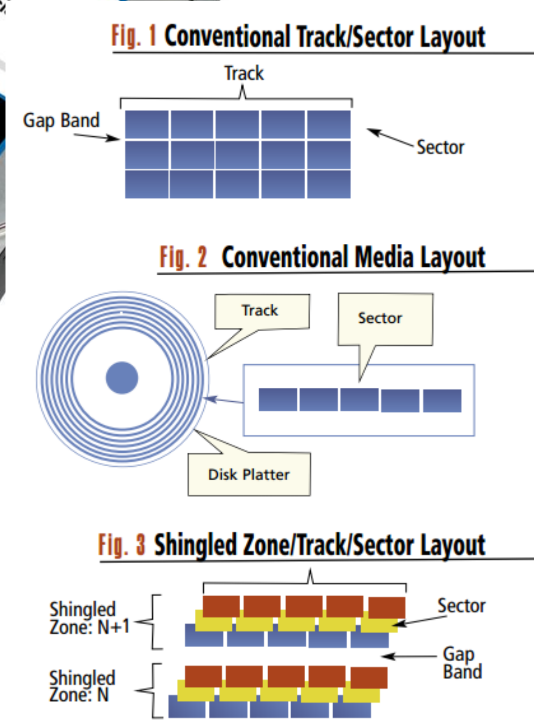
```
Set(0, 1, 'A');
Set(0, 2, 'B');
Set(0, 3, 'C');
Set(0, 3, 'D'); /* Not Allowed */
```

DGIBox Abstracts Modern Storage

- Flash-based SSDs and SMR drives







Example

User Input:
pairs of key and value

```
write(key=0, value=A)
write(key=1, value=B)
write(key=11, value=C)
write(key=10, value=D)
write(key=9, value=E)
write(key=0, value=A')
write(key=1, value=B')
write(key=2, value=F)
write(key=3, value=G)
write(key=4, value=H)
write(key=5, value=I)
write(key=6, value=J)
```

Your Algorithm:
Map keys to buckets

```
set(0,0,A)
set(0,1,B)
set(0,2,C)
set(0,3,D)
set(1,0,E)
set(1,1,A')
set(1,2,B')
set(1,3,F)
set(2,0,G)
set(2,1,H)
set(2,2,I)
set(2,3,J)
```

Bucket_0
A
B
C
D

Bucket_1
E
A’
B’
F

Bucket_2
G
H
I
J

Example (Cont.)

User Input:
pairs of key and value

```
write(key=0, value=A)
write(key=1, value=B)
write(key=11, value=C)
write(key=10, value=D)
write(key=9, value=E)
write(key=0, value=A')
write(key=1, value=B')
write(key=2, value=F)
write(key=3, value=G)
write(key=4, value=H)
write(key=5, value=I)
write(key=6, value=J)
read(key=0)
read(key=1)
```

Your Algorithm:
Map keys to buckets

```
set(0,0,A)
set(0,1,B)
set(0,2,C)
set(0,3,D)
set(1,0,E)
set(1,1,A')
set(1,2,B')
set(1,3,F)
set(2,0,G)
set(2,1,H)
set(2,2,I)
set(2,3,J)
get(??,??)
get(??,??)
```

Dictionary
(in your algorithm)

0	**(1,1)**
1	**(1,2)**
2	**(1,3)**
3	**(2,0)**
4	**(2,1)**
5	**(2,2)**
6	**(2,3)**
7	
8	
9	**(1,0)**
10	**(0,3)**
11	**(0,2)**

Bucket_0

A
B
C
D

Bucket_1

E
A’
B’
F

Bucket_2

G
H
I
J

Example (Cont.)

User Input:
pairs of key and value

```
write(key=0, value=A)
write(key=1, value=B)
write(key=11, value=C)
write(key=10, value=D)
write(key=9, value=E)
write(key=0, value=A')
write(key=1, value=B')
write(key=2, value=F)
write(key=3, value=G)
write(key=4, value=H)
write(key=5, value=I)
write(key=6, value=J)
read(key=0)
read(key=1)
```

Your Algorithm:
Map keys to buckets

```
set(0,0,A)
set(0,1,B)
set(0,2,C)
set(0,3,D)
set(1,0,E)
set(1,1,A')
set(1,2,B')
set(1,3,F)
set(2,0,G)
set(2,1,H)
set(2,2,I)
set(2,3,J)
get(1,1)
get(1,2)
```

Dictionary
(in your algorithm)

0	**(1,1)**
1	**(1,2)**
2	**(1,3)**
3	**(2,0)**
4	**(2,1)**
5	**(2,2)**
6	**(2,3)**
7	
8	
9	**(1,0)**
10	**(0,3)**
11	**(0,2)**

Bucket_0

A
B
C
D

Bucket_1

E
A'
B'
F

Bucket_2

G
H
I
J

You can get locations of keys 0 and 1
by referring to the dictionary

Example (Cont.)

User Input:
pairs of key and value

```
write(key=0, value=A)
write(key=1, value=B)
write(key=11, value=C)
write(key=10, value=D)
write(key=9, value=E)
write(key=0, value=A')
write(key=1, value=B')
write(key=2, value=F)
write(key=3, value=G)
write(key=4, value=H)
write(key=5, value=I)
write(key=6, value=J)
read(key=0)
read(key=1)
write(key=7, value=K)
write(key=8, value=L)
```

Your Algorithm:
Map keys to buckets

```
set(0,0,A)
set(0,1,B)
set(0,2,C)
set(0,3,D)
set(1,0,E)
set(1,1,A')
set(1,2,B')
set(1,3,F)
set(2,0,G)
set(2,1,H)
set(2,2,I)
set(2,3,J)
get(1,1)
get(1,2)
no free
space!!!
```

Dictionary
(in your algorithm)

0	**(1,1)**
1	**(1,2)**
2	**(1,3)**
3	**(2,0)**
4	**(2,1)**
5	**(2,2)**
6	**(2,3)**
7	
8	
9	**(1,0)**
10	**(0,3)**
11	**(0,2)**

Bucket$_0$

A
B
C
D

Bucket$_1$

E
A'
B'
F

Bucket$_2$

G
H
I
J

Example (Cont.)

We actually have enough space to store values of 7 and 8
Bucket_0 contains old data that are not valid now

User Input:
pairs of key and value

```
write(key=0, value=A)
write(key=1, value=B)
write(key=11, value=C)
write(key=10, value=D)
write(key=9, value=E)
write(key=0, value=A')
write(key=1, value=B')
write(key=2, value=F)
write(key=3, value=G)
write(key=4, value=H)
write(key=5, value=I)
write(key=6, value=J)
read(key=0)
read(key=1)
write(key=7, value=K)
write(key=8, value=L)
```

Your Algorithm:
Map keys to buckets

```
set(0,0,A)
set(0,1,B)
set(0,2,C)
set(0,3,D)
set(1,0,E)
set(1,1,A')
set(1,2,B')
set(1,3,F)
set(2,0,G)
set(2,1,H)
set(2,2,I)
set(2,3,J)
get(1,1)
get(1,2)
no free
space!!!
```

Dictionary
(in your algorithm)

0	**(1,1)**
1	**(1,2)**
2	**(1,3)**
3	**(2,0)**
4	**(2,1)**
5	**(2,2)**
6	**(2,3)**
7	
8	
9	**(1,0)**
10	**(0,3)**
11	**(0,2)**

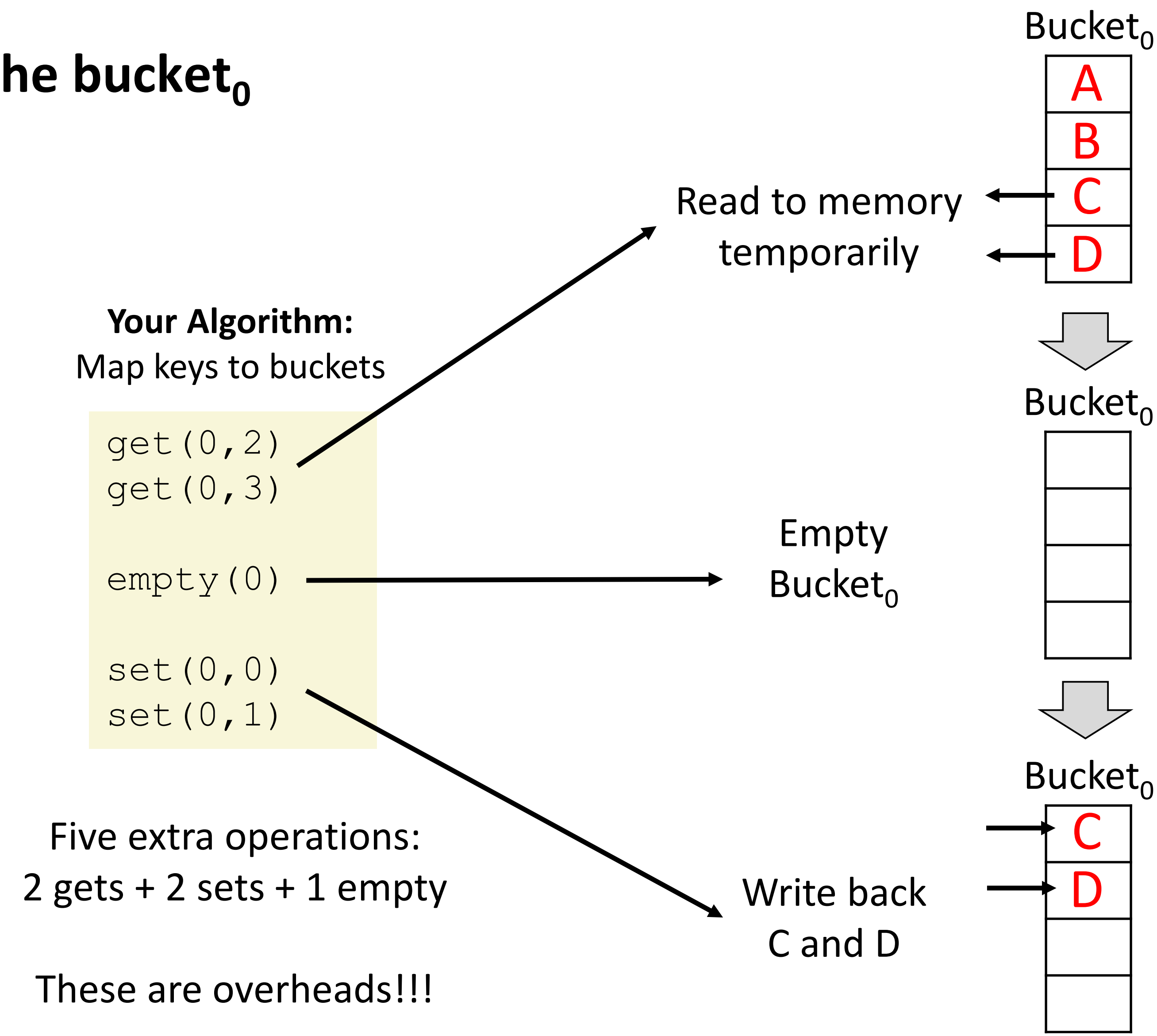
Bucket_0
A
B
C
D

Bucket_1
E
A'
B'
F

Bucket_2
G
H
I
J

Example (Cont.) – Cleaning

- **Cleaning the bucket$_0$**



Example (Cont.)

User Input:
pairs of key and value

```
write(key=0, value=A)
write(key=1, value=B)
write(key=11, value=C)
write(key=10, value=D)
write(key=9, value=E)
write(key=0, value=A')
write(key=1, value=B')
write(key=2, value=F)
write(key=3, value=G)
write(key=4, value=H)
write(key=5, value=I)
write(key=6, value=J)
read(key=0)
read(key=1)
write(key=7, value=K)
write(key=8, value=L)
```

Your Algorithm:
Map keys to buckets

```
set(0,0,A)
set(0,1,B)
set(0,2,C)
set(0,3,D)
set(1,0,E)
set(1,1,A')
set(1,2,B')
set(1,3,F)
set(2,0,G)
set(2,1,H)
set(2,2,I)
set(2,3,J)
get(1,1)
get(1,2)
no free
space!!!
```

Dictionary
(in your algorithm)

0	(1,1)
1	(1,2)
2	(1,3)
3	(2,0)
4	(2,1)
5	(2,2)
6	(2,3)
7	
8	
9	(1,0)
10	(0,1)
11	(0,0)

Bucket$_0$

C
D

Bucket$_1$

E
A’
B’
F

Bucket$_2$

G
H
I
J

Example (Cont.)

User Input:
pairs of key and value

```
write(key=0, value=A)
write(key=1, value=B)
write(key=11, value=C)
write(key=10, value=D)
write(key=9, value=E)
write(key=0, value=A')
write(key=1, value=B')
write(key=2, value=F)
write(key=3, value=G)
write(key=4, value=H)
write(key=5, value=I)
write(key=6, value=J)
read(key=0)
read(key=1)
write(key=7, value=K)
write(key=8, value=L)
```

Your Algorithm:
Map keys to buckets

```
set(0,0,A)
set(0,1,B)
set(0,2,C)
set(0,3,D)
set(1,0,E)
set(1,1,A')
set(1,2,B')
set(1,3,F)
set(2,0,G)
set(2,1,H)
set(2,2,I)
set(2,3,J)
get(1,1)
get(1,2)
no free
space!!!
```

Dictionary
(in your algorithm)

0	**(1,1)**
1	**(1,2)**
2	**(1,3)**
3	**(2,0)**
4	**(2,1)**
5	**(2,2)**
6	**(2,3)**
7	**(0,2)**
8	**(0,3)**
9	**(1,0)**
10	**(0,1)**
11	**(0,0)**

Bucket$_0$

C
D
K
L

Bucket$_1$

E
A’
B’
F

Bucket$_2$

G
H
I
J

Example (Cont.) – Summary

- **Operations that are performed by your algorithm**
  - **Input ops:** 14 writes and 2 reads
  - **Extra ops:** 2 writes + 2 reads + 1 empty
  - **Total:** 16 writes, 4 reads, and 1 empty

- **Analysis**
  - **The cost of a write:** (16 writes + 2 reads + 1 empty) / (14 writes) = $\Theta(1.36)$
    - Assume that the costs of a write, a read, and an empty are the same
  - **The cost of a read:** 1 read = $\Theta(1)$

What is the Problem with the Example

- **The previous solution requires lots of DRAM**
  - Suppose that DGIBox has 2^23 buckets whose sizes are 256 and the size of a table entry is 4 bytes
  - The number of entries in the dictionary is 2^23 x 2^9 = 2^32
  - The amount of DRAM required for the dictionary is

2^32 x 4 bytes = 16 GB!!!

- **How to reduce the memory requirement?**
  - Hash?

Example with Hash

User Input:
pairs of key and value

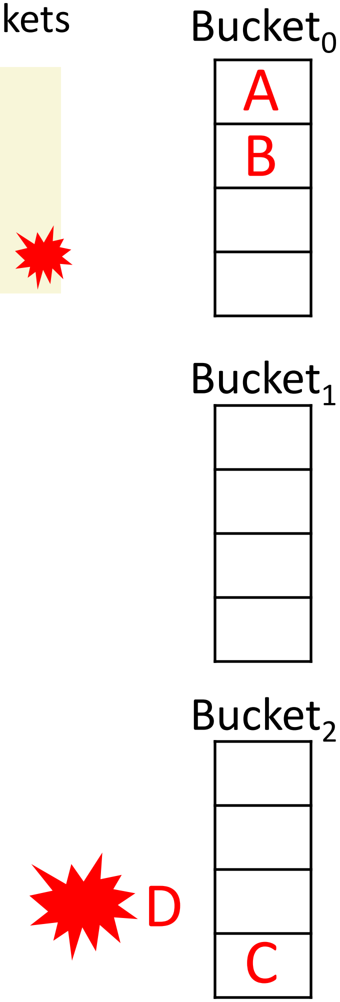
```
write(key=0, value=A)
write(key=1, value=B)
write(key=11, value=C)
write(key=10, value=D)
write(key=9, value=E)
write(key=0, value=A')
write(key=1, value=B')
write(key=2, value=F)
write(key=3, value=G)
write(key=4, value=H)
write(key=5, value=I)
write(key=6, value=J)
read(key=0)
read(key=1)
write(key=7, value=K)
write(key=8, value=L)
```

Hash Function:
Division

```
 0/4=0,  0%4=0
 1/4=0,  1%4=1
11/4=2, 11%4=3
10/4=2, 10%4=2
```

Your Algorithm:
Map keys to buckets

```
set(0,0,A)
set(0,1,B)
set(2,3,C)
set(2,2,D)
```



It violates the property of DGIBox

Example with Hash (Cont.)

- **Cleaning the bucket$_2$**

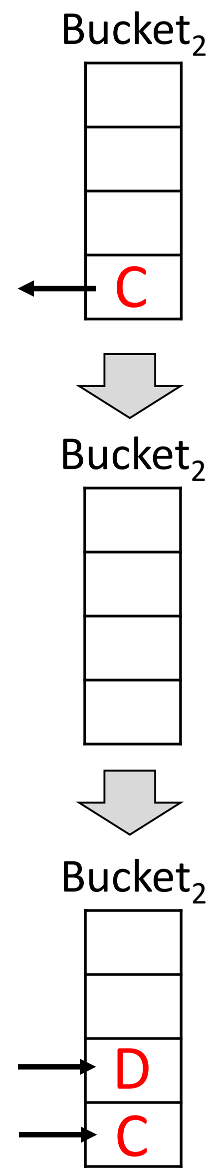
Your Algorithm:
Map keys to buckets

```
get(2,3)

empty(2)

set(2,2)
set(2,3)
```

Read to memory temporarily

Empty Bucket$_2$

Write back C with D



Five extra operations:
1 get + 1 sets + 1 empty

These are overheads!!!

Example with Hash (Cont.)

User Input: pairs of key and value	**Hash Function:** Division	**Your Algorithm:** Map keys to buckets
`write(key=0, value=A)`	`0/4=0, 0%4=0`	`set(0,0,A)`
`write(key=1, value=B)`	`1/4=0, 1%4=1`	`set(0,1,B)`
`write(key=11, value=C)`	`11/4=2,11%4=3`	`set(2,3,C)`
`write(key=10, value=D)`	`10/4=2,10%4=2`	`set(2,2,D)`
`write(key=9, value=E)`	`9/4=2, 9%4=1`	`set(2,1,E)`
`write(key=0, value=A')`	`0/4=0, 0%4=0`	`set(0,0,A')`
`write(key=1, value=B')`	`1/4=0, 1%4=1`	`set(0,1,B')`
`write(key=2, value=F)`	`2/4=0, 2%4=2`	`set(0,2,F)`
`write(key=3, value=G)`	`3/4=0, 3%4=3`	`set(0,3,G)`
`write(key=4, value=H)`	`4/4=1, 4%4=0`	`set(1,0,H)`
`write(key=5, value=I)`	`5/4=1, 5%4=1`	`set(1,1,I)`
`write(key=6, value=J)`	`6/4=1, 6%4=2`	`set(1,2,J)`
`read(key=0)`	`0/4=0, 0%4=0`	`get(0,0)`
`read(key=1)`	`1/4=0, 1%4=1`	`get(0,1)`
`write(key=7, value=K)`	`7/4=1, 7%4=3`	`set(1,3,K)`
`write(key=8, value=L)`	`8/4=2, 8%4=0`	`set(2,0,L)`

$Bucket_0$

A'
B'
F
G

$Bucket_1$

H
I
J
K

$Bucket_2$

L
E
D
C

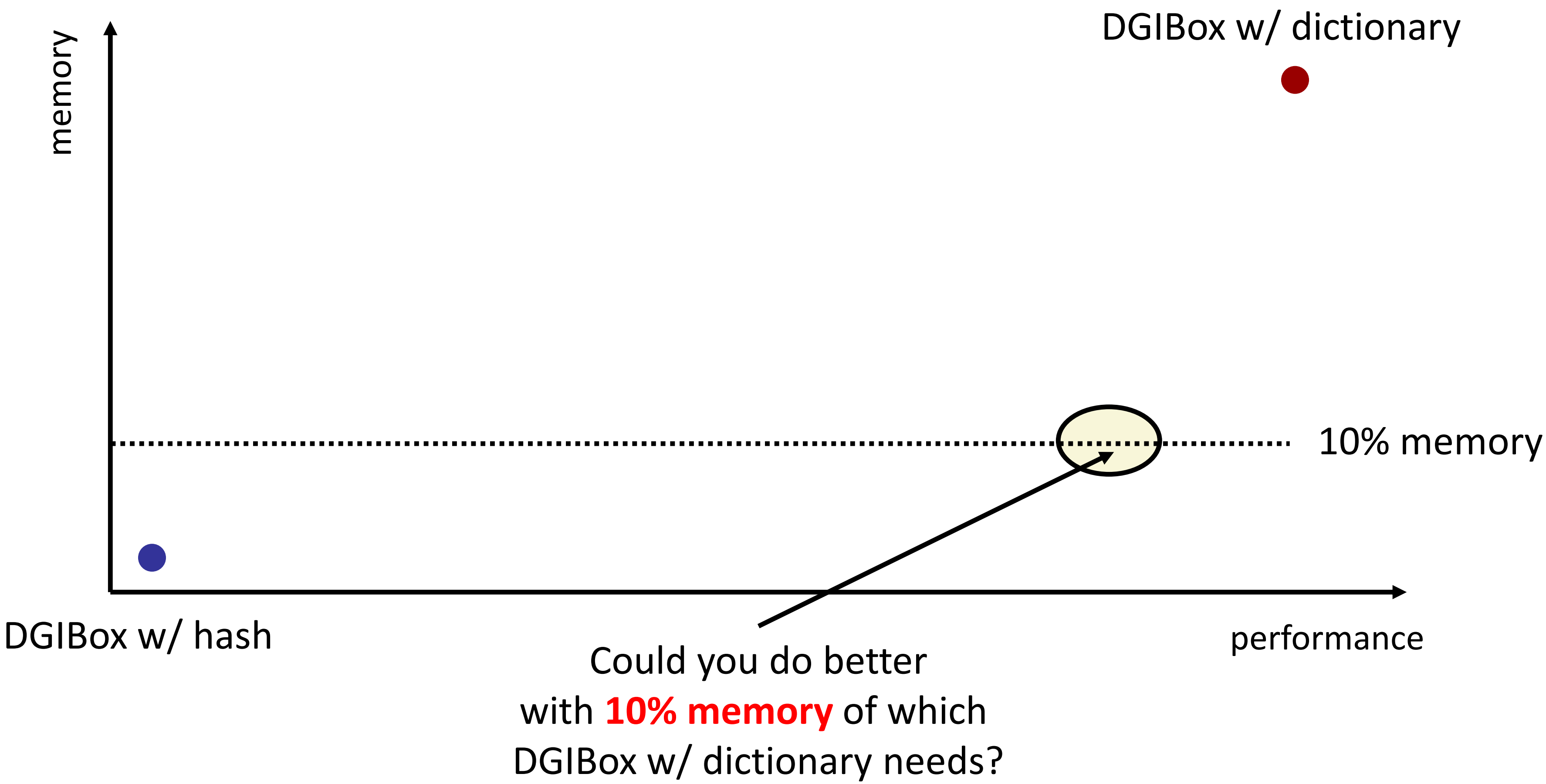
Example (Cont.) – Summary

- **DGIBox with Hash requires zero memory for the dictionary,**

- **But, operations that are performed**
  - **Input ops:** 14 writes and 2 reads
  - **Extra ops:** 8 writes + 8 reads + 5 empty
  - **Total:** 22 writes, 10 reads, and 5 empty

- **Analysis**
  - **The cost of a write:** (22 writes + 10 reads + 5 empty) / (14 writes) = $\Theta(2.64)$
    - Assume that the costs of a write, a read, and an empty are the same
  - **The cost of a read:** 1 read = $\Theta(1)$

Trade-off Between Cost and Memory



Grading

- **Develop algorithms providing basic operations (write and read) that operate on DGIBox**

- **Specific Implementation Topics**
  1. DGIBox w/ Hash: **10%**
  2. DGIBox w/ dictionary: **20%**
  3. DGIBox w/ your own algorithm: **40%**
     1. Correctly works with limited memory (10% of dictionary): **10%**
     2. Performance: **30%**
        – It is a race!
  4. Design Report: **30%**
     1. It would be good if you formulate complexity of your algorithm

Rules

- **Development Tool & Languages**
  - Language: C/C++ (No Java and Python!)
  - Microsoft Visual Studio
- **What to submit?**
  - Source codes that include all your algorithms
  - Design report
- **Where?**
  - To TA via email
- **Until when?**
  - Until 11:59pm on November 15th
    - Double column, **3 pages**, 10 font size, No title page
    - Template & Sample: Uploaded in BlackBoard

Cheating?

- F Grade

- TA will check up all your codes manually and using cheating detection tools