Discussion 7

Hash Indexes & Project 3 Intro

EECS 484

Logistics

- Homework 3 due Nov 3 at 11:55 PM EST
- Project 3 due Nov 10 at 11:55 PM EST
- Midterm exam in the process of grading
  - Please do not discuss the exam
- Today
  - Indexing & Hash index
  - Intro to MongoDB (project 3)
  - MapReduce (project 3)

Indexing

Indexes

- Our database has lots of nice data
  - How do we get the data as fast as possible
    - Indexing!
- Indexes are a type of data structure that allow us to search for, insert, and delete data in a table
  - Take an input k (not necessarily a key for the table, just a search key)
    - Set of columns
    - If search key contains a primary key then we call it the primary index
      - Otherwise it is the secondary index
  - Find data entries k* for each k
    - Can have more than one data entry per search key k

Records

- All data are stored in a unit of space called a page
    - Multiple records can exist in a page
    - Often times needs multiple pages to fit all records in a table
- Pages don’t have to be always full
    - Clustered indexes may keep pages ⅔ full to make inserting and deletion faster

Data Entries k*

- Take search key k and find data entries k* (from the index file) for k
- Alternative 1: Data entry k* is the actual record itself
  - Only 1 k* can be resolved using Alternative 1 (no way to handle duplicates)

  ⇒ or you will have lots of duplicates
- Alternative 2: Data entry k* is (k, rid) where rid is the record ID for the record with the search key k

  search key pointer to the location
- Alternative 3: Data entry k* is (k, list of rids) where each rid is the record ID for the record with the search key k

one entry points to one record, one page contains many records, to fetch one record you need to access the whole page

Clustering

sorted actual data

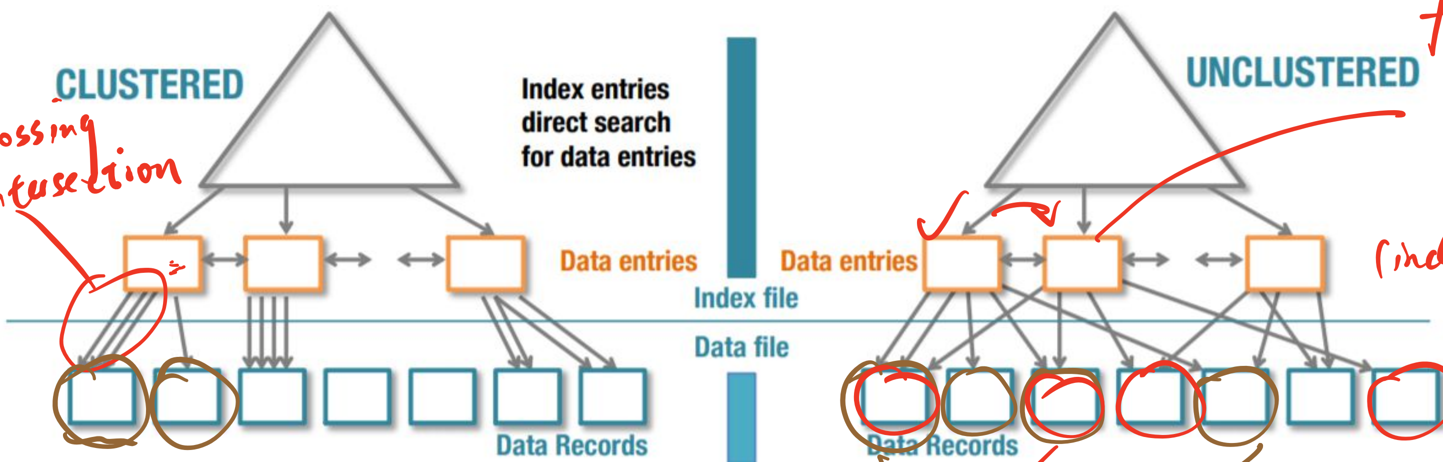
- Clustered index
  - The data records are in the same order (or very close to the same order) as the data entries

    index
    - Alternative 1 implies clustered index - each data entry is the data record
  - Have to read each data record page once only

    (in order): finish reading first page go to next page
- Unclustered index
  - No guaranteed order
  - Have to read each data record page once for each data record we retrieve

first page not in memory

not crossing or intersection



(index)

disk

reading for first data entry: 2 pages

4 pages

read same record multiple times

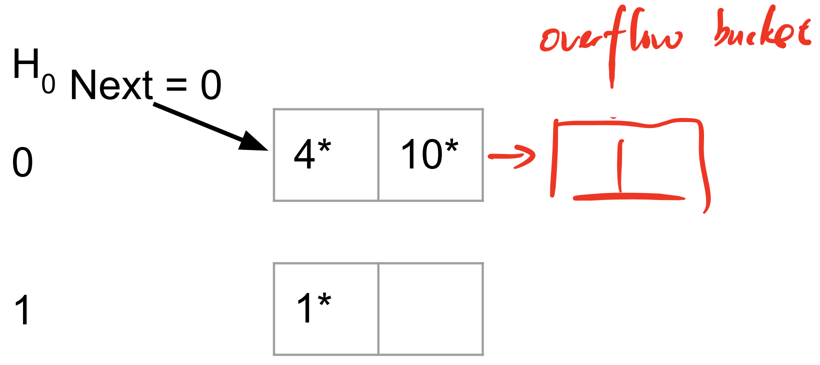
Linear Hashing

Linear Hashing

Take binary transformation and look at last 2 bits =>

N = 2
$H_i(x) = x \pmod{N * 2^i}$
Level = 0

- Family of hash functions
- Split one bucket at a time upon an overflow
- N = **fixed** base number of buckets usually 2
- Level = current level in hash family
- Next = pointer to next bucket to be split
- Split policy: split on overflow



① split when you create new overflow bucket.

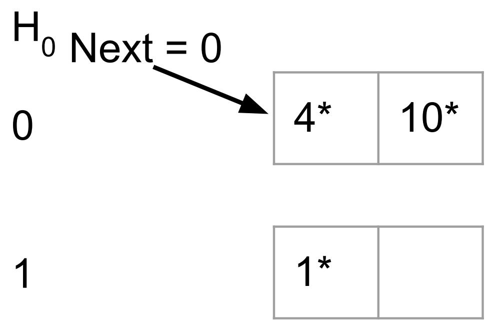
② split when insert into an overflow bucket

H_0: last bit of binary
H_1: last 2 bits

Linear Hashing

$N = 2$
$H_i(x) = x \pmod{N * 2^i}$
Level = 0
$H_0(x) = x \pmod{N * 2^0} = x \pmod 2$

- Family of hash functions
- Split one bucket at a time upon an overflow
- N = **fixed** base number of buckets
- Level = current level in hash family
- Next = pointer to next bucket to be split
- Split policy: split on overflow

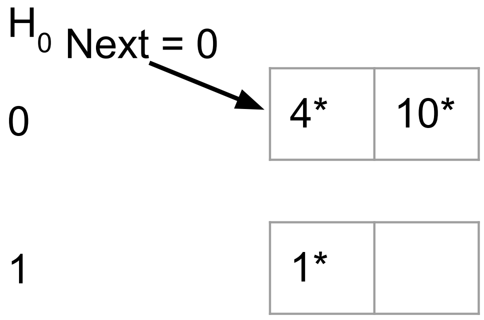


Linear Hashing

$N = 2$
$H_i(x) = x \pmod{N * 2^i}$
Level = 0
$H_0(x) = x \pmod{N * 2^0} = x \pmod 2$

- Family of hash functions
- Split one bucket at a time upon an overflow
- N = **fixed** base number of buckets
- Level = current level in hash family
- Next = pointer to next bucket to be split
- Split policy: split on overflow

- Insert 23*



Linear Hashing

$N = 2$

$H_i(x) = x \pmod{N * 2^i}$

Level = 0

$H_0(x) = x \pmod{N * 2^0} = x \pmod 2$

- Family of hash functions
- Split one bucket at a time upon an overflow
- N = **fixed** base number of buckets
- Level = current level in hash family
- Next = pointer to next bucket to be split
- Split policy: split on overflow

- Insert 23*
  - $H_0(23) = 23 \pmod 2 = 1 \bmod 2 = 0b1$

H_0 Next = 0

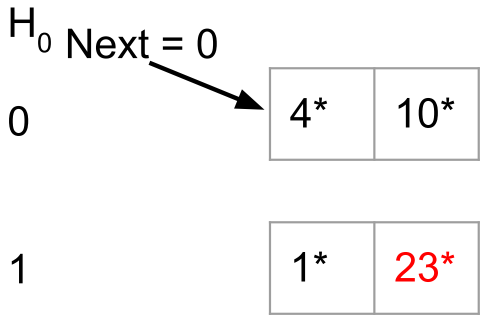
H_0		
0	4*	10*
1	1*	

Linear Hashing

$N = 2$
$H_i(x) = x \pmod{N * 2^i}$
$\text{Level} = 0$
$H_0(x) = x \pmod{N * 2^0} = x \pmod 2$

- Family of hash functions
- Split one bucket at a time upon an overflow
- N = **fixed** base number of buckets
- Level = current level in hash family
- Next = pointer to next bucket to be split
- Split policy: split on overflow

- Insert 23*
    - $H_0(23) = 23 \pmod 2 = 1 \bmod 2 = 0b1$

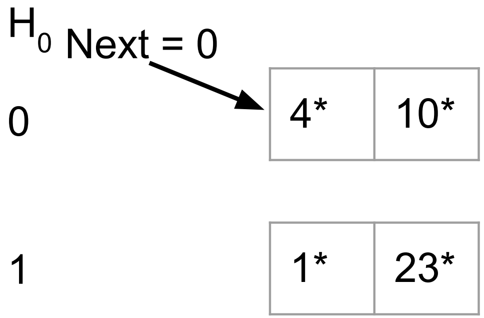


Linear Hashing

$N = 2$
$H_i(x) = x \pmod{N * 2^i}$
$\text{Level} = 0$
$H_0(x) = x \pmod{N * 2^0} = x \pmod 2$

- Family of hash functions
- Split one bucket at a time upon an overflow
- N = **fixed** base number of buckets
- Level = current level in hash family
- Next = pointer to next bucket to be split
- Split policy: split on overflow

- Insert 23*
  - $H_0(23) = 23 \pmod 2 = 1 \bmod 2 = 0b1$
- Done :)

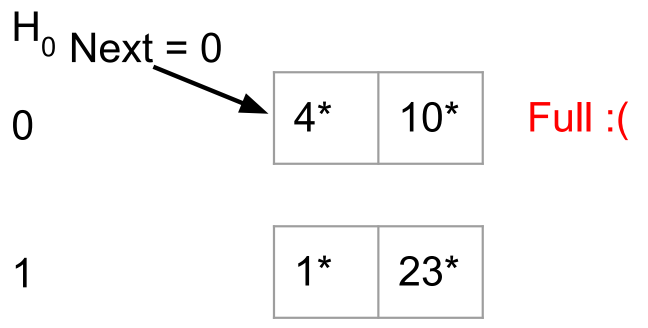


$N = 2$
$H_i(x) = x \pmod{N * 2^i}$
Level = 0
$H_0(x) = x \pmod{N * 2^0} = x \pmod 2$

Linear Hashing

- Family of hash functions
- Split one bucket at a time upon an overflow
- N = **fixed** base number of buckets
- Level = current level in hash family
- Next = pointer to next bucket to be split
- Split policy: split on overflow

- Insert 12*
  - $H_0(12) = 12 \pmod 2) = 0 \bmod 2 = 0b0$

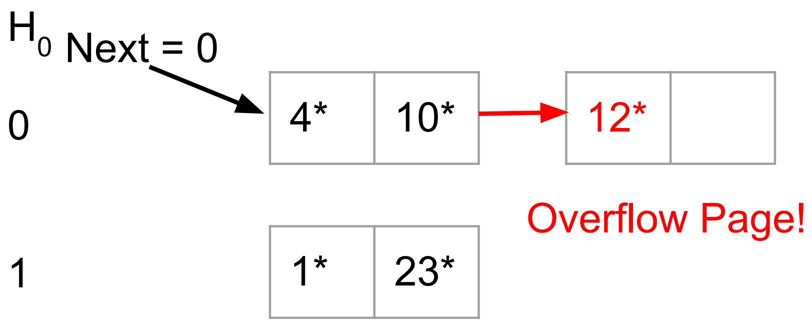


Linear Hashing

$N = 2$
$H_i(x) = x \pmod{N * 2^i}$
Level = 0
$H_0(x) = x \pmod{N * 2^0} = x \pmod 2$

- Family of hash functions
- Split one bucket at a time upon an overflow
- N = **fixed** base number of buckets
- Level = current level in hash family
- Next = pointer to next bucket to be split
- Split policy: split on overflow

- Insert 12*
    - $H_0(12) = 12 \pmod 2 = 0 \bmod 2 = 0b0$

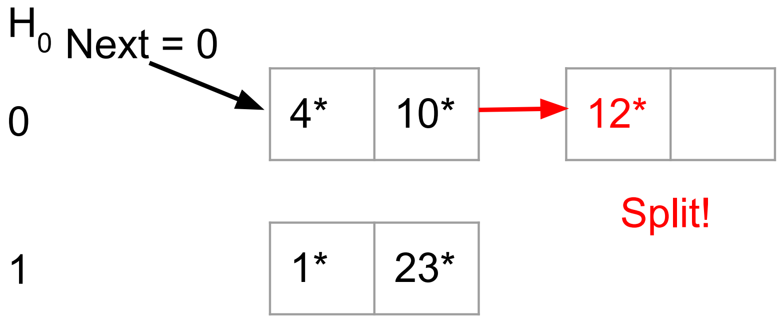


Linear Hashing

$N = 2$
$H_i(x) = x \pmod{N * 2^i}$
Level = 0
$H_0(x) = x \pmod{N * 2^0} = x \pmod 2$

- Family of hash functions
- Split one bucket at a time upon an overflow
- N = **fixed** base number of buckets
- Level = current level in hash family
- Next = pointer to next bucket to be split
- Split policy: split on overflow

- Insert 12*
  - $H_0(12) = 12 \pmod 2 = 0 \bmod 2 = 0b0$

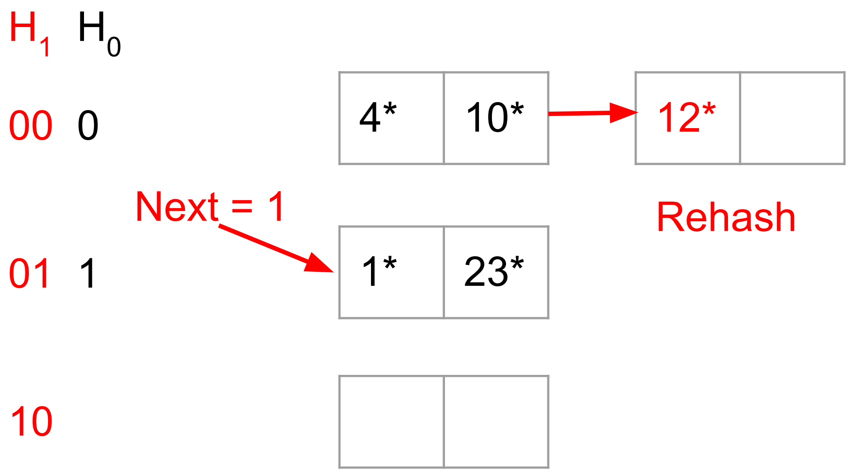


Linear Hashing

$N = 2$
$H_i(x) = x \pmod{N * 2^i}$
Level = 0
$H_0(x) = x \pmod{N * 2^0} = x \pmod 2$
$H_1(x) = x \pmod{N * 2^1} = x \pmod 4$

- Family of hash functions
- Split one bucket at a time upon an overflow
- N = **fixed** base number of buckets
- Level = current level in hash family
- Next = pointer to next bucket to be split
- Split policy: split on overflow

- Insert 12*
    - $H_0(12) = 12 \pmod 2 = 0 \bmod 2 = 0b0$



Linear Hashing

$N = 2$
$H_i(x) = x \ (\mathrm{mod}\ N * 2^i)$
$\mathrm{Level} = 0$
$H_0(x) = x \ (\mathrm{mod}\ N * 2^0) = x \ (\mathrm{mod}\ 2)$
$H_1(x) = x \ (\mathrm{mod}\ N * 2^1) = x \ (\mathrm{mod}\ 4)$

- Family of hash functions
- Split one bucket at a time upon an overflow
- N = **fixed** base number of buckets
- Level = current level in hash family
- Next = pointer to next bucket to be split
- Split policy: split on overflow

- Insert 12*
  - $H_0(12) = 12 \ (\mathrm{mod}\ 2) = 0 \ \mathrm{mod}\ 2 = 0\mathrm{b}0$
- Rehash 4*, 10*, 12*
  - $H_1(4) = 4 \ (\mathrm{mod}\ 4) = 0 \ \mathrm{mod}\ 4 = 0\mathrm{b}00$
  - $H_1(10) = 10 \ (\mathrm{mod}\ 4) = 2 \ \mathrm{mod}\ 4 = 0\mathrm{b}10$
  - $H_1(12) = 12 \ (\mathrm{mod}\ 4) = 0 \ \mathrm{mod}\ 4 = 0\mathrm{b}00$

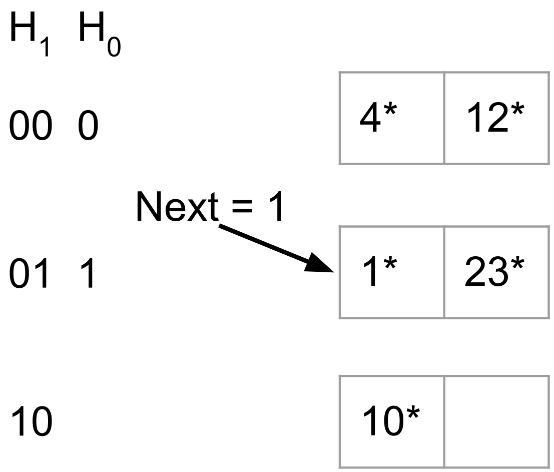
H_1	H_0			
00	0		4*	12*
01	1	Next = 1 →	1*	23*
10			10*	

Linear Hashing

$N = 2$
$H_i(x) = x \pmod{N * 2^i}$
Level = 0
$H_0(x) = x \pmod{N * 2^0} = x \pmod 2$
$H_1(x) = x \pmod{N * 2^1} = x \pmod 4$

- Family of hash functions
- Split one bucket at a time upon an overflow
- N = **fixed** base number of buckets
- Level = current level in hash family
- Next = pointer to next bucket to be split
- Split policy: split on overflow

- Insert 12*
  - $H_0(12) = 12 \pmod 2) = 0 \bmod 2 = 0b0$
- Done :)

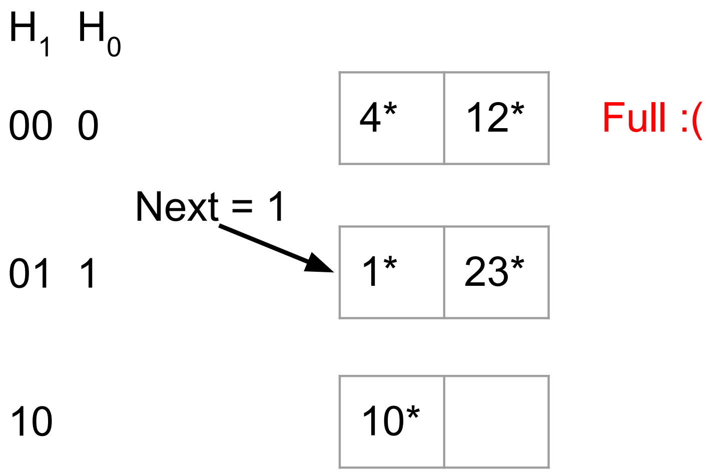


Linear Hashing

$N = 2$
$H_i(x) = x \pmod{N * 2^i}$
$Level = 0$
$H_0(x) = x \pmod{N * 2^0} = x \pmod 2$
$H_1(x) = x \pmod{N * 2^1} = x \pmod 4$

- Family of hash functions
- Split one bucket at a time upon an overflow
- N = **fixed** base number of buckets
- Level = current level in hash family
- Next = pointer to next bucket to be split
- Split policy: split on overflow

- Insert 8*
  - $H_0(8) = 8 \pmod 2 = 0 \bmod 2 = 0b0$
  - $H_1(8) = 8 \pmod 4 = 0 \bmod 4 = 0b00$

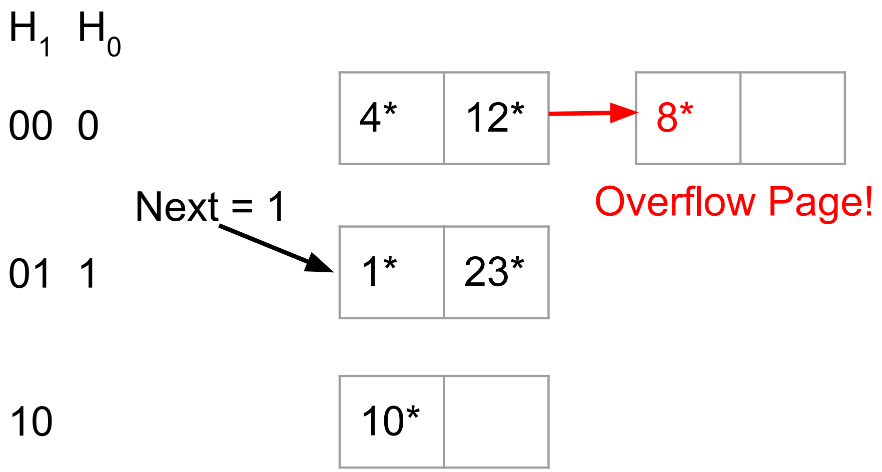


Linear Hashing

$N = 2$
$H_i(x) = x \pmod{N * 2^i}$
$\text{Level} = 0$
$H_0(x) = x \pmod{N * 2^0} = x \pmod{2}$
$H_1(x) = x \pmod{N * 2^1} = x \pmod{4}$

- Family of hash functions
- Split one bucket at a time upon an overflow
- N = **fixed** base number of buckets
- Level = current level in hash family
- Next = pointer to next bucket to be split
- Split policy: split on overflow

- Insert 8*
  - $H_0(8) = 8 \pmod{2} = 0 \bmod 2 = 0b0$
  - $H_1(8) = 8 \pmod{4} = 0 \bmod 4 = 0b00$

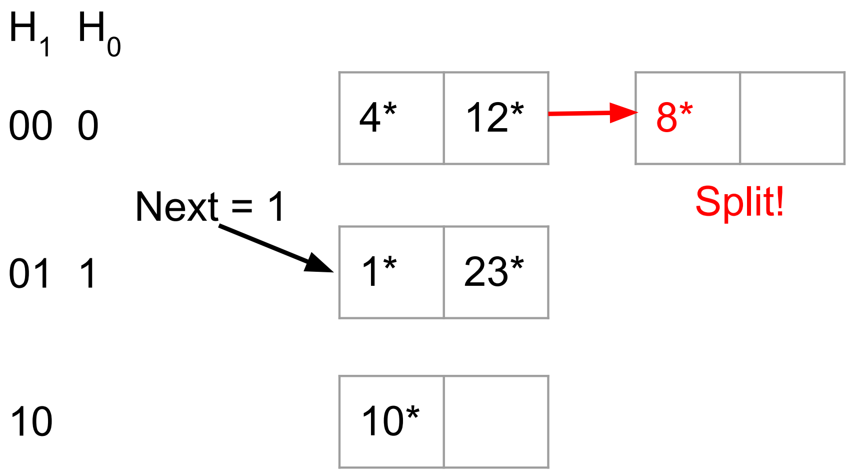


Linear Hashing

$N = 2$
$H_i(x) = x \pmod{N * 2^i}$
Level = 0
$H_0(x) = x \pmod{N * 2^0} = x \pmod 2$
$H_1(x) = x \pmod{N * 2^1} = x \pmod 4$

- Family of hash functions
- Split one bucket at a time upon an overflow
- N = **fixed** base number of buckets
- Level = current level in hash family
- Next = pointer to next bucket to be split
- Split policy: split on overflow

- Insert 8*
  - $H_0(8) = 8 \pmod 2 = 0 \bmod 2 = 0b0$
  - $H_1(8) = 8 \pmod 4 = 0 \bmod 4 = 0b00$

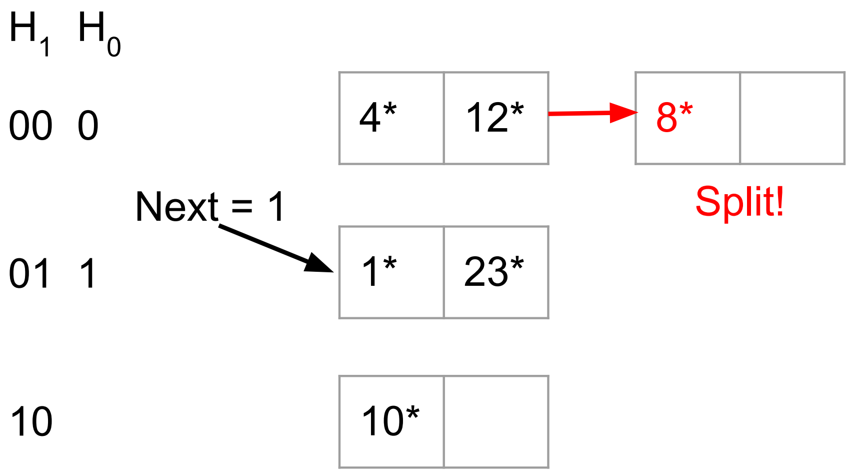


Linear Hashing

$N = 2$
$H_i(x) = x \pmod{N * 2^i}$
Level = 0
$H_0(x) = x \pmod{N * 2^0} = x \pmod 2$
$H_1(x) = x \pmod{N * 2^1} = x \pmod 4$

- Family of hash functions
- Split one bucket at a time upon an overflow
- N = **fixed** base number of buckets
- Level = current level in hash family
- Next = pointer to next bucket to be split
- Split policy: split on overflow

- Insert 8*
  - $H_0(8) = 8 \pmod 2 = 0 \bmod 2 = 0b0$
  - $H_1(8) = 8 \pmod 4 = 0 \bmod 4 = 0b00$
  - Remember we split the next node always even though it might not be overflow



Linear Hashing

$N = 2$

$H_i(x) = x \pmod{N * 2^i}$

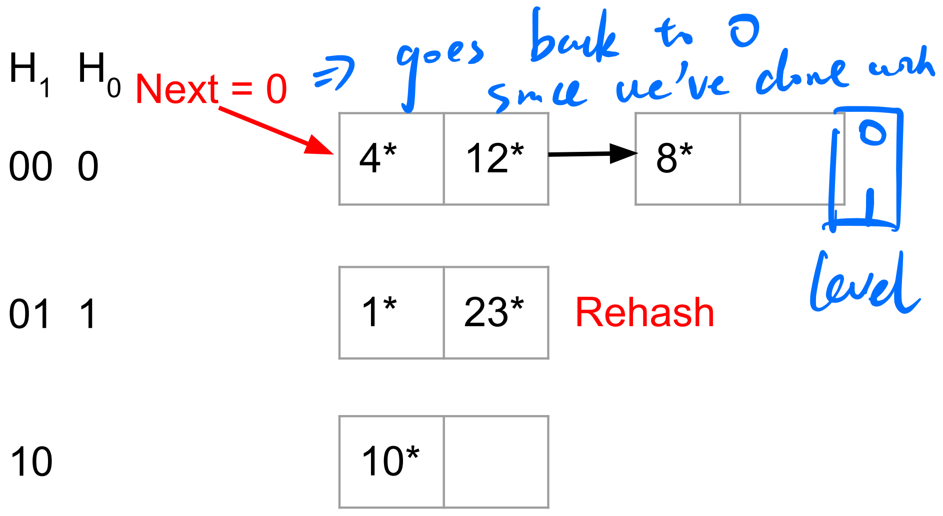
Level = 1

$H_0(x) = x \pmod{N * 2^0} = x \pmod 2$

$H_1(x) = x \pmod{N * 2^1} = x \pmod 4$

- Family of hash functions
- Split one bucket at a time upon an overflow
- N = **fixed** base number of buckets
- Level = current level in hash family
- Next = pointer to next bucket to be split
- Split policy: split on overflow

- Insert 8*
  - $H_0(8) = 8 \pmod 2) = 0 \bmod 2 = 0b0$
  - $H_1(8) = 8 \pmod 4) = 0 \bmod 4 = 0b00$
  - Remember we split the next node always even though it might not be overflow

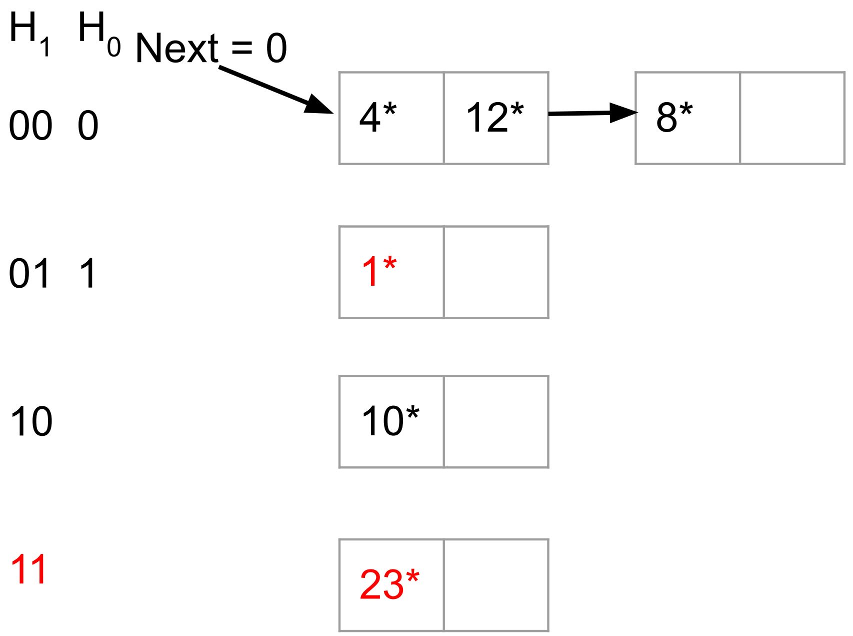


Linear Hashing

$N = 2$
$H_i(x) = x \pmod{N * 2^i}$
Level = 1
$H_0(x) = x \pmod{N * 2^0} = x \pmod 2$
$H_1(x) = x \pmod{N * 2^1} = x \pmod 4$

- Family of hash functions
- Split one bucket at a time upon an overflow
- N = **fixed** base number of buckets
- Level = current level in hash family
- Next = pointer to next bucket to be split
- Split policy: split on overflow

- Insert 8*
  - $H_0(8) = 8 \pmod 2 = 0 \bmod 2 = 0b0$
  - $H_1(8) = 8 \pmod 4 = 0 \bmod 4 = 0b00$
  - $H_1(1) = 1 \pmod 4 = 1 \bmod 4 = 0b01$
  - $H_1(23) = 23 \pmod 4 = 3 \bmod 4 = 0b11$

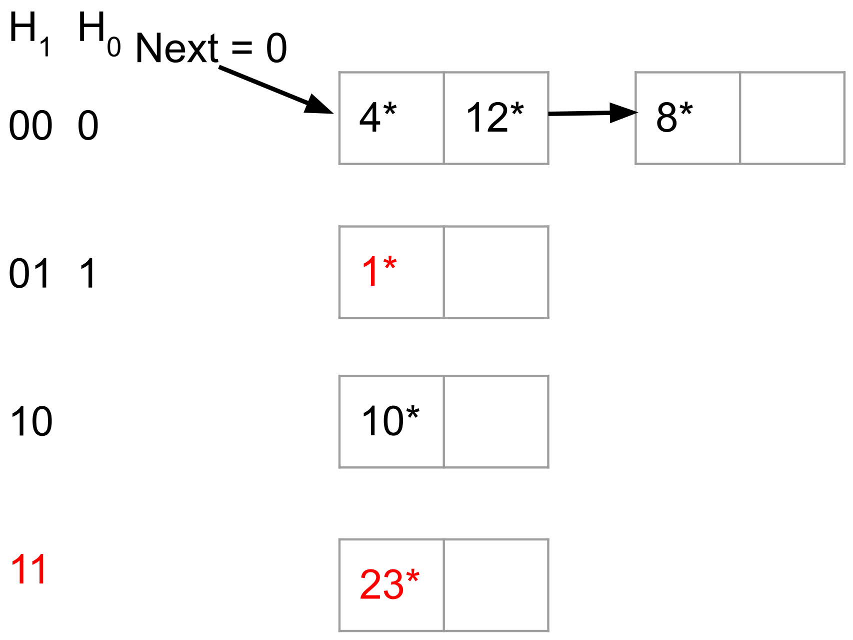


Linear Hashing

$N = 2$
$H_i(x) = x \pmod{N * 2^i}$
Level = 1
$H_0(x) = x \pmod{N * 2^0} = x \pmod 2$
$H_1(x) = x \pmod{N * 2^1} = x \pmod 4$

- Family of hash functions
- Split one bucket at a time upon an overflow
- N = **fixed** base number of buckets
- Level = current level in hash family
- Next = pointer to next bucket to be split
- Split policy: split on overflow

- Insert 8*
  - $H_0(8) = 8 \pmod 2 = 0 \bmod 2 = 0b0$
  - $H_1(8) = 8 \pmod 4 = 0 \bmod 4 = 0b00$
  - $H_1(1) = 1 \pmod 4 = 1 \bmod 4 = 0b01$
  - $H_1(23) = 23 \pmod 4 = 3 \bmod 4 = 0b11$
  - Still have overflow page; that's ok

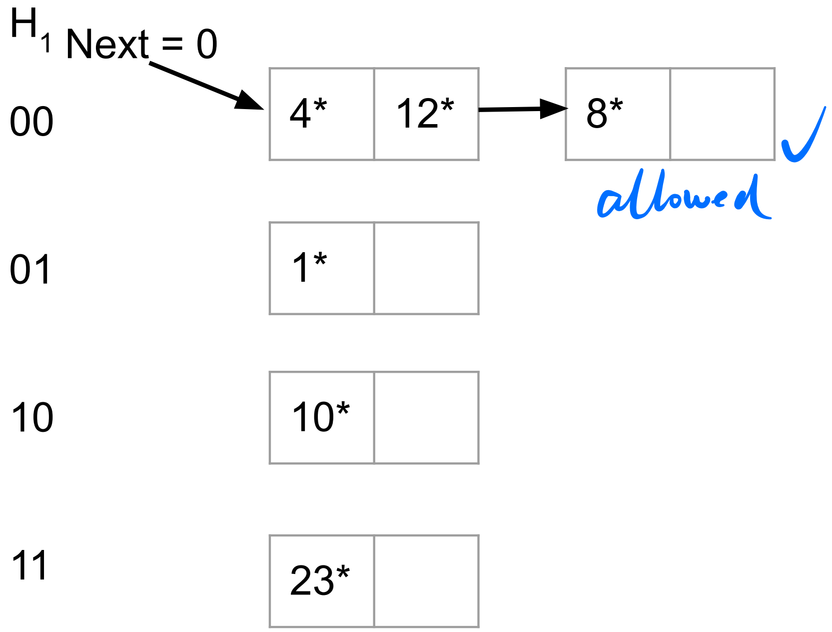


Linear Hashing

$N = 2$
$H_i(x) = x \pmod{N * 2^i}$
$Level = 1$
$H_0(x) = x \pmod{N * 2^0} = x \pmod 2$
$H_1(x) = x \pmod{N * 2^1} = x \pmod 4$

- Family of hash functions
- Split one bucket at a time upon an overflow
- N = **fixed** base number of buckets
- Level = current level in hash family
- Next = pointer to next bucket to be split
- Split policy: split on overflow

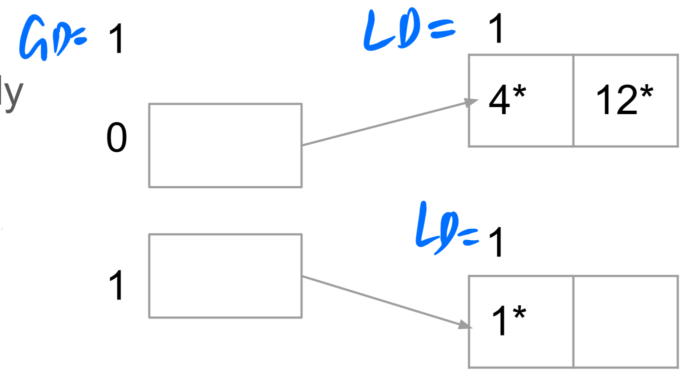
- Insert 8*
  - $H_0(8) = 8 \pmod 2 = 0 \bmod 2 = 0b0$
  - $H_1(8) = 8 \pmod 4 = 0 \bmod 4 = 0b00$
  - $H_1(1) = 1 \pmod 4 = 1 \bmod 4 = 0b01$
  - $H_1(23) = 23 \pmod 4 = 3 \bmod 4 = 0b11$
  - Still have overflow page; that's ok
  - Done :)



Extendible Hashing

Extendible Hashing

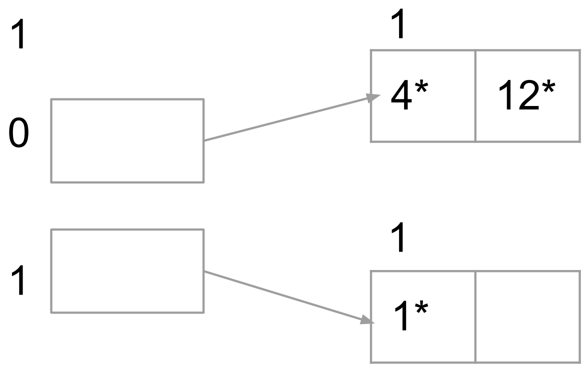
- Use directory of pointers
  - Split on overflow
  - Once we’re out of room in directory, double size
  - Global depth = number of bits considered globally
  - Local depth = number of bits considered locally



GD ≥ LD

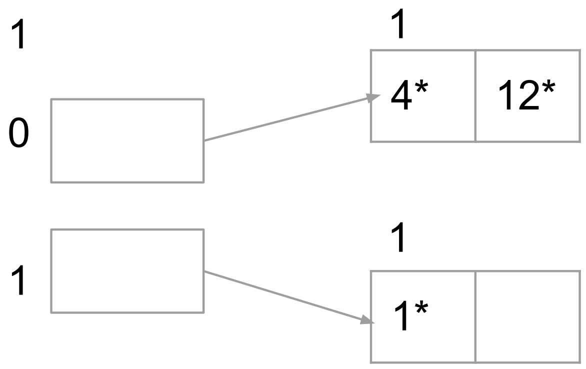
Extendible Hashing

- Use directory of pointers
  - Split on overflow
  - Once we're out of room in directory, double size
  - Global depth = number of bits considered globally
  - Local depth = number of bits considered locally
- Insert 5*



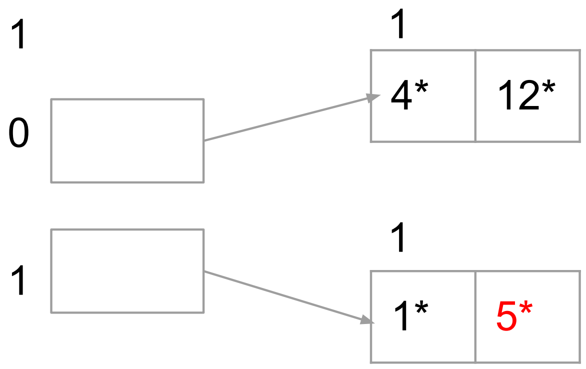
Extendible Hashing

- Use directory of pointers
  - Split on overflow
  - Once we're out of room in directory, double size
  - Global depth = number of bits considered globally
  - Local depth = number of bits considered locally
- Insert 5*
  - 5=0b101



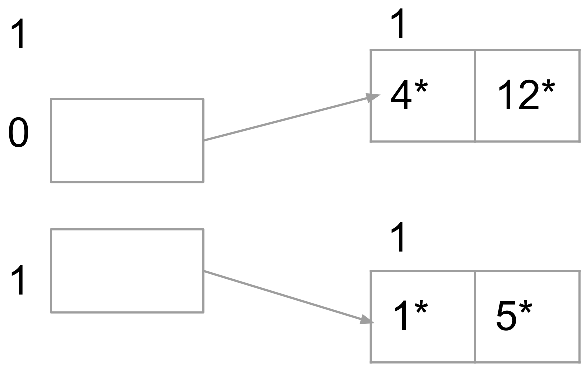
Extendible Hashing

- Use directory of pointers
  - Split on overflow
  - Once we're out of room in directory, double size
  - Global depth = number of bits considered globally
  - Local depth = number of bits considered locally
- Insert 5*
  - 5=0b10<u>1</u>



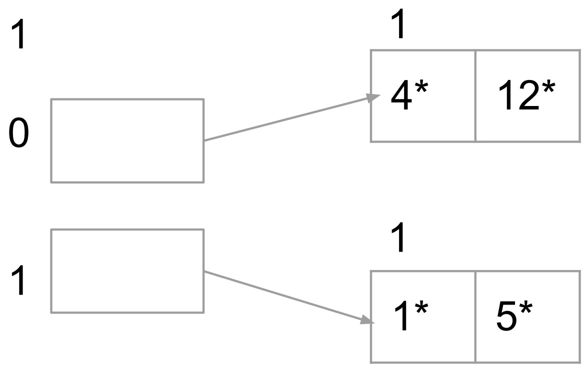
Extendible Hashing

- Use directory of pointers
  - Split on overflow
  - Once we’re out of room in directory, double size
  - Global depth = number of bits considered globally
  - Local depth = number of bits considered locally
- Insert 5*
  - 5=0b10<u>1</u>
  - Done :)



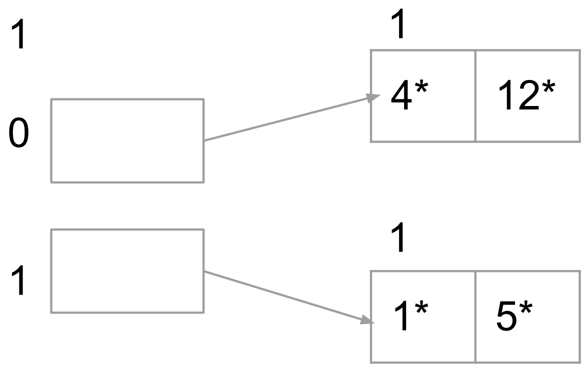
Extendible Hashing

- Use directory of pointers
  - Split on overflow
  - Once we're out of room in directory, double size
  - Global depth = number of bits considered globally
  - Local depth = number of bits considered locally
- Insert 10*



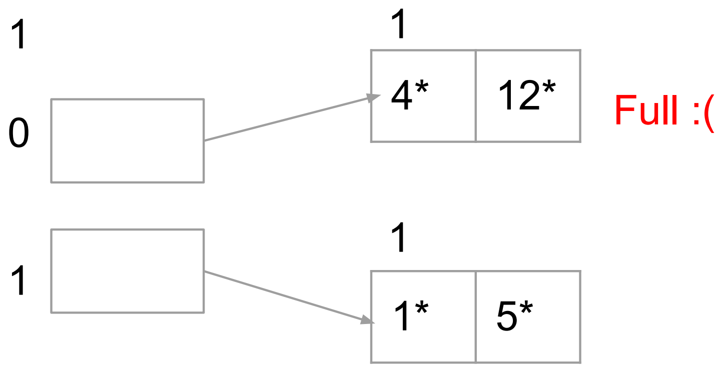
Extendible Hashing

- Use directory of pointers
  - Split on overflow
  - Once we’re out of room in directory, double size
  - Global depth = number of bits considered globally
  - Local depth = number of bits considered locally
- Insert 10*
  - 10=0b1010



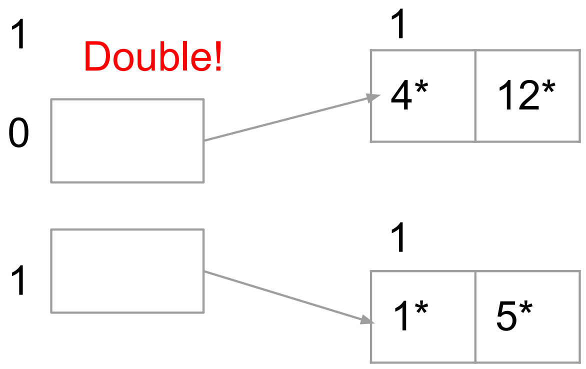
Extendible Hashing

- Use directory of pointers
  - Split on overflow
  - Once we're out of room in directory, double size
  - Global depth = number of bits considered globally
  - Local depth = number of bits considered locally
- Insert 10*
  - 10=0b1010



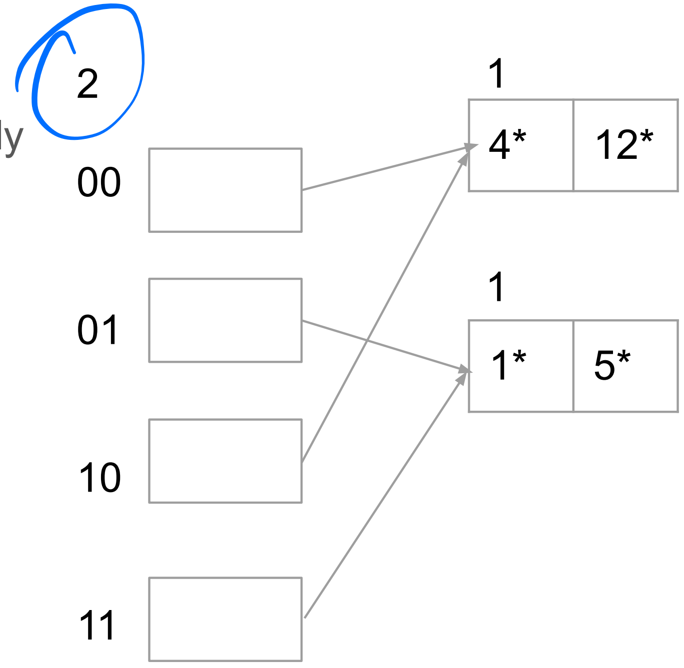
Extendible Hashing

- Use directory of pointers
  - Split on overflow
  - Once we're out of room in directory, double size
  - Global depth = number of bits considered globally
  - Local depth = number of bits considered locally
- Insert 10*
  - 10=0b1010



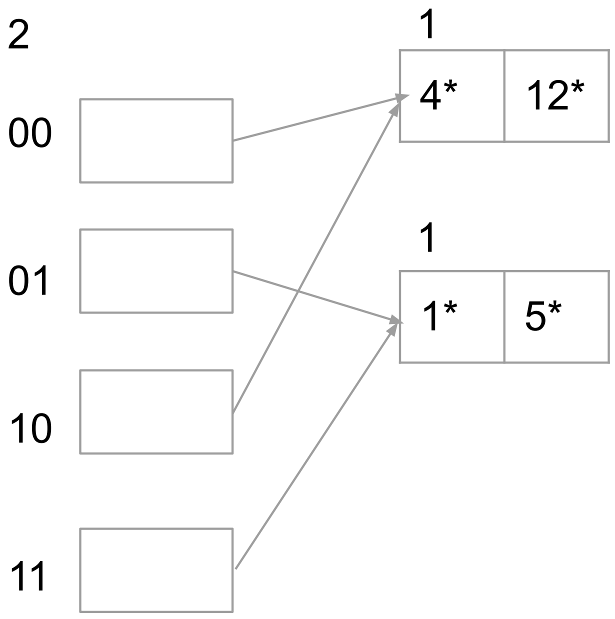
Extendible Hashing

- Use directory of pointers
  - Split on overflow
  - Once we're out of room in directory, double size
  - Global depth = number of bits considered globally
  - Local depth = number of bits considered locally
- Insert 10*
  - 10=0b1010
  - New directories point to "Split image"



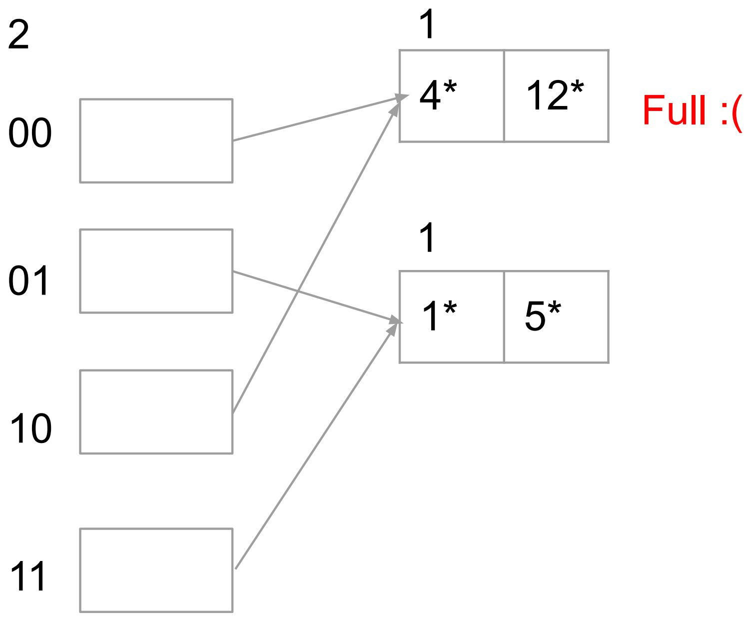
Extendible Hashing

- Use directory of pointers
  - Split on overflow
  - Once we’re out of room in directory, double size
  - Global depth = number of bits considered globally
  - Local depth = number of bits considered locally
- Insert 10*
  - 10=0b1010
  - New directories point to “Split image”
  - Try to insert again



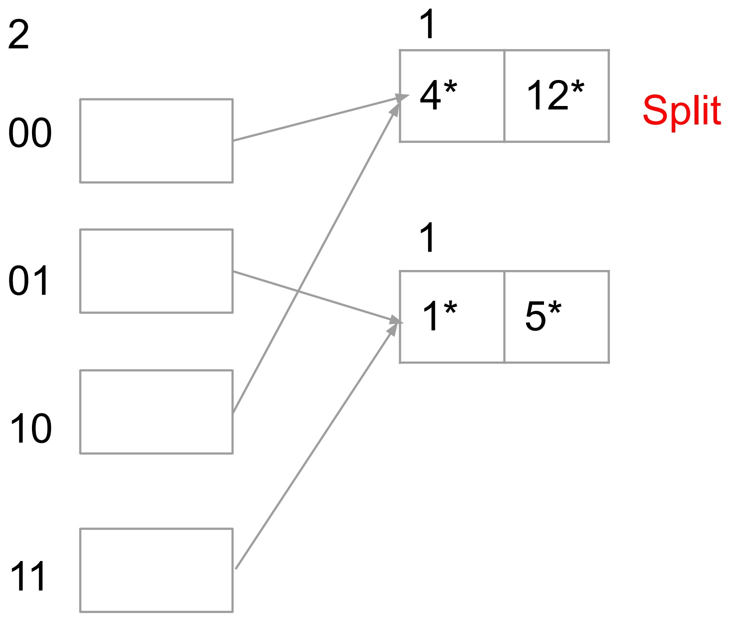
Extendible Hashing

- Use directory of pointers
  - Split on overflow
  - Once we’re out of room in directory, double size
  - Global depth = number of bits considered globally
  - Local depth = number of bits considered locally
- Insert 10*
  - 10=0b1010
  - New directories point to “Split image”
  - Try to insert again



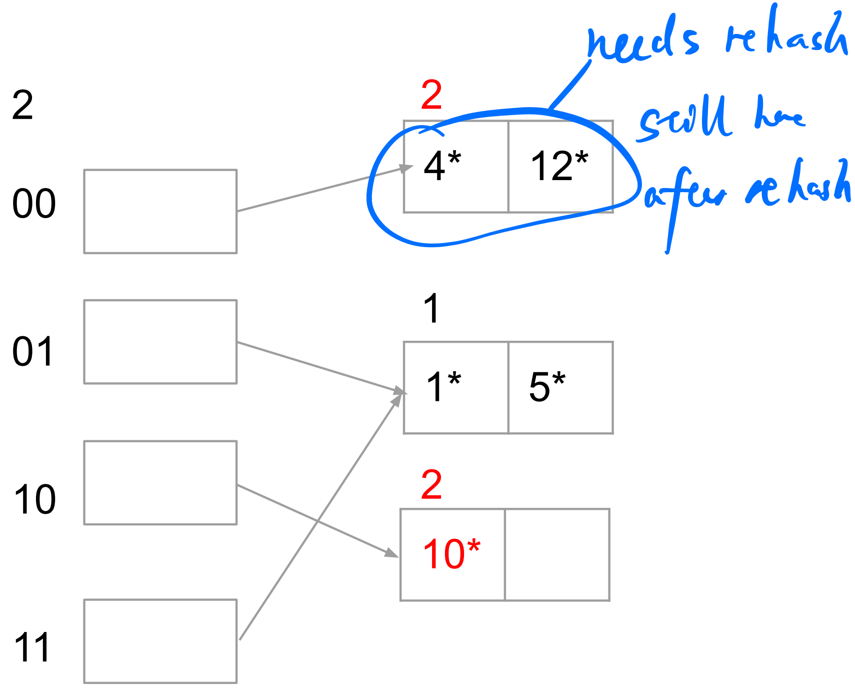
Extendible Hashing

- Use directory of pointers
  - Split on overflow
  - Once we're out of room in directory, double size
  - Global depth = number of bits considered globally
  - Local depth = number of bits considered locally
- Insert 10*
  - 10=0b1010
  - New directories point to "Split image"
  - Try to insert again



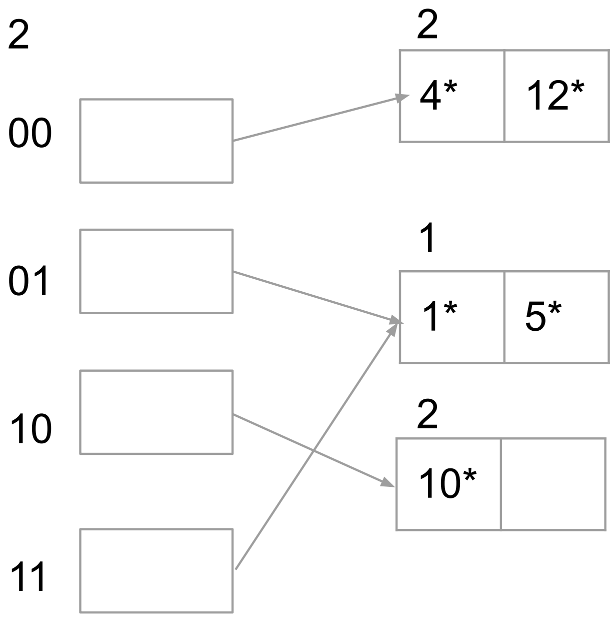
Extendible Hashing

- Use directory of pointers
  - Split on overflow
  - Once we're out of room in directory, double size
  - Global depth = number of bits considered globally
  - Local depth = number of bits considered locally
- Insert 10*
  - 10=0b1010
  - New directories point to "Split image"
  - Try to insert again



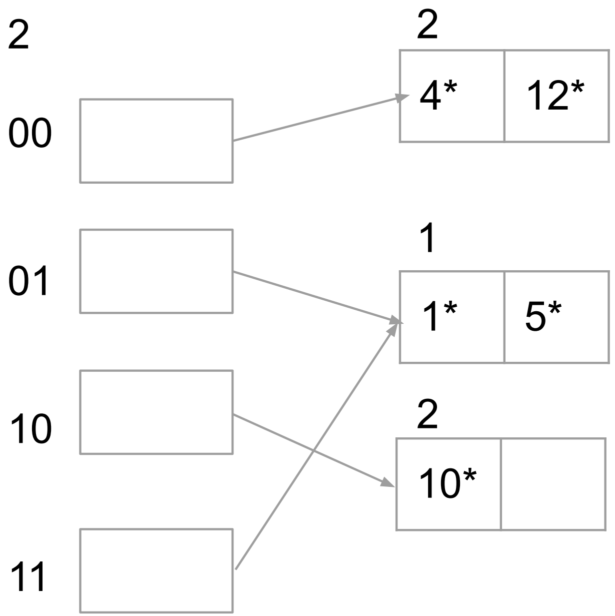
Extendible Hashing

- Use directory of pointers
  - Split on overflow
  - Once we're out of room in directory, double size
  - Global depth = number of bits considered globally
  - Local depth = number of bits considered locally
- Insert 10*
  - 10=0b1010
  - New directories point to "Split image"
  - Try to insert again
  - Done :)



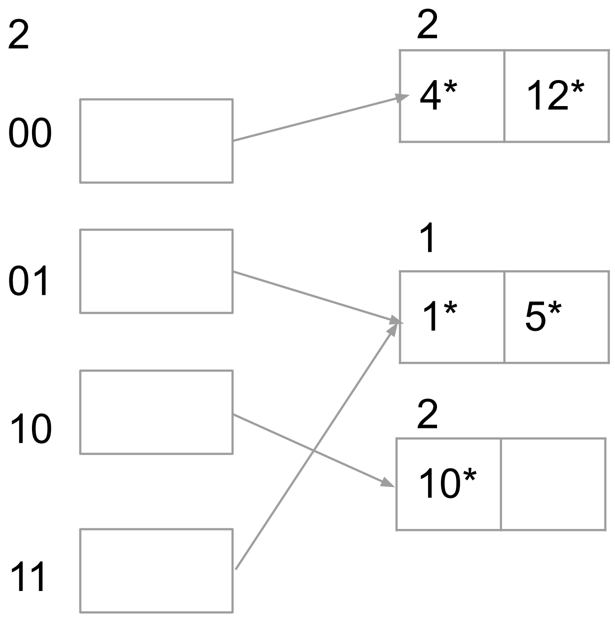
Extendible Hashing

- Use directory of pointers
  - Split on overflow
  - Once we're out of room in directory, double size
  - Global depth = number of bits considered globally
  - Local depth = number of bits considered locally
- Insert 7*



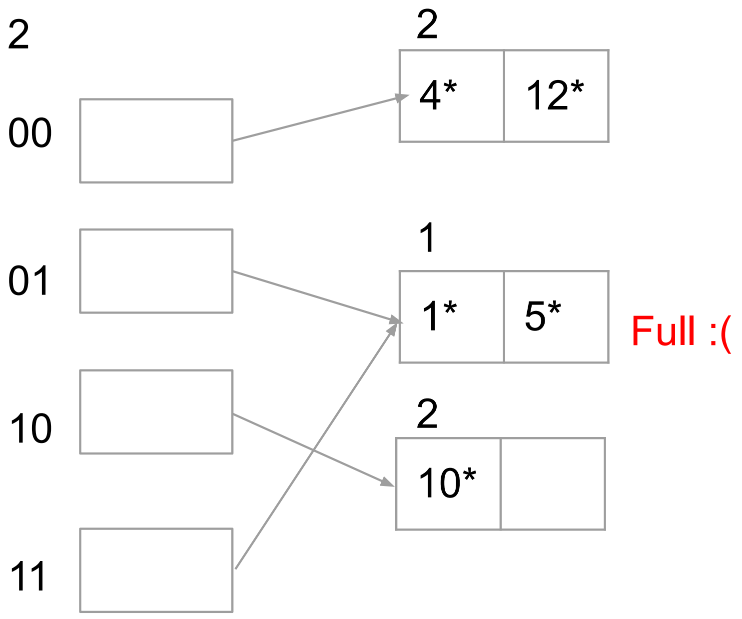
Extendible Hashing

- Use directory of pointers
  - Split on overflow
  - Once we’re out of room in directory, double size
  - Global depth = number of bits considered globally
  - Local depth = number of bits considered locally
- Insert 7*
  - 7=0b111



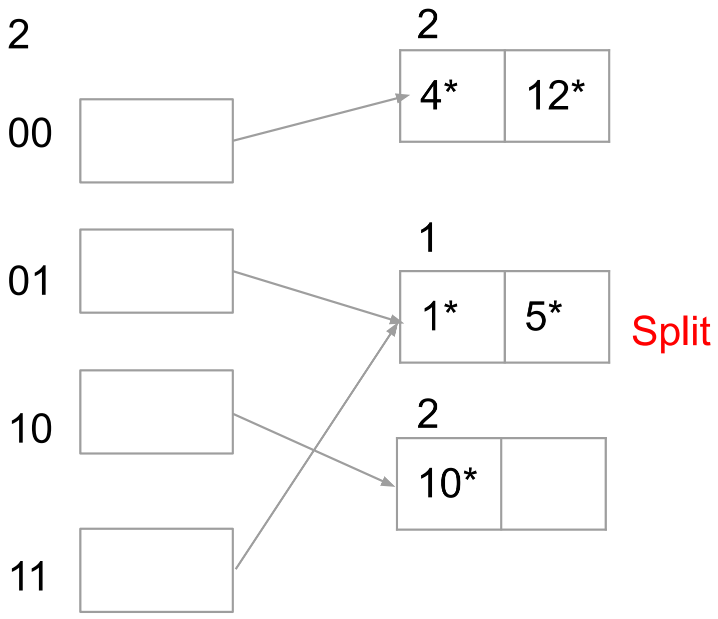
Extendible Hashing

- Use directory of pointers
  - Split on overflow
  - Once we're out of room in directory, double size
  - Global depth = number of bits considered globally
  - Local depth = number of bits considered locally
- Insert 7*
  - 7=0b111



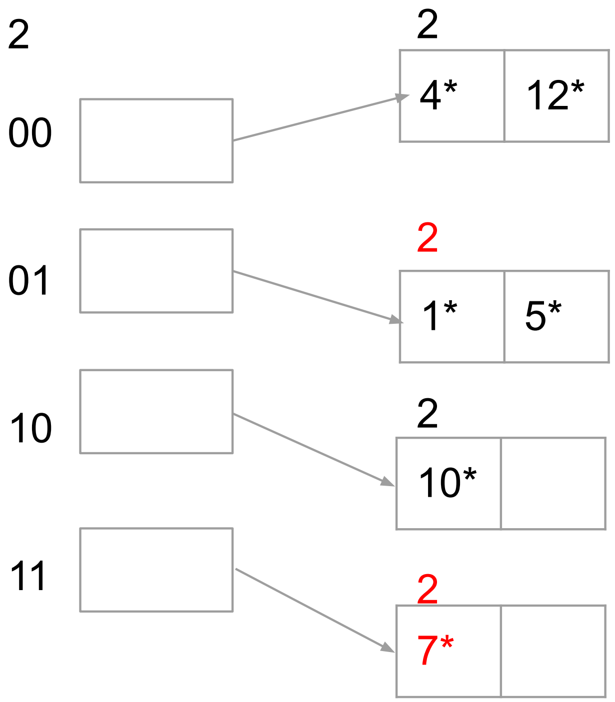
Extendible Hashing

- Use directory of pointers
  - Split on overflow
  - Once we're out of room in directory, double size
  - Global depth = number of bits considered globally
  - Local depth = number of bits considered locally
- Insert 7*
  - 7=0b111



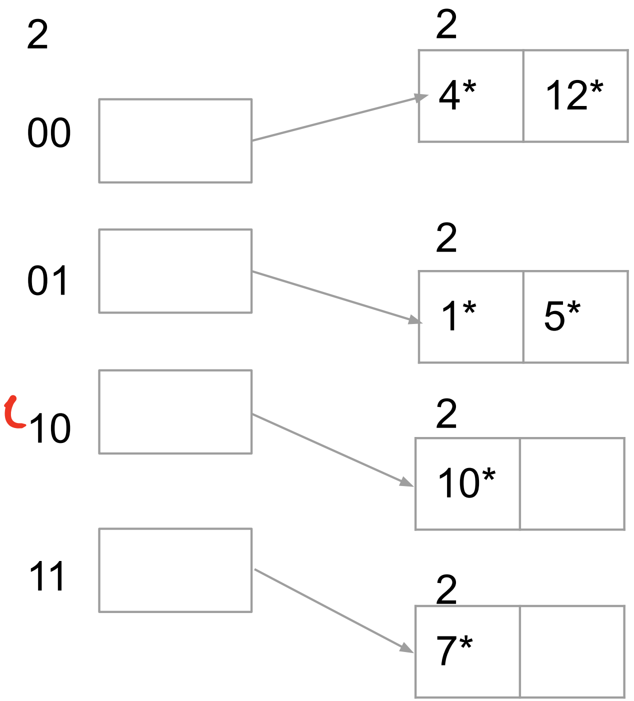
Extendible Hashing

- Use directory of pointers
  - Split on overflow
  - Once we're out of room in directory, double size
  - Global depth = number of bits considered globally
  - Local depth = number of bits considered locally
- Insert 7*
  - 7=0b111



Extendible Hashing

- Use directory of pointers
  - Split on overflow
  - Once we're out of room in directory, double size
  - Global depth = number of bits considered globally
  - Local depth = number of bits considered locally
- Insert 7*
  - 7=0b111
  - Done :)



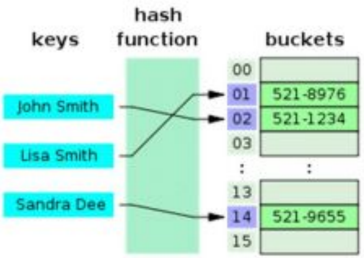
☆ what if there are still overflow page after split? ⇒ keep splitting

Hashing Overview

- Directory size can double in extendible hashing
  - Linear hashing only adds one bucket at a time
  - Global depth >= local depth always
  - # pointers to any specific bucket = 2^{GD-LD}
  - Overflow pages only in rare cases (unavoidable collisions)

e.g. duplicates

- Linear Hashing only adds one bucket at a time
  - Better memory usage
  - Doesn't avoid overflow pages in many cases
    - Over time we minimize overflow pages

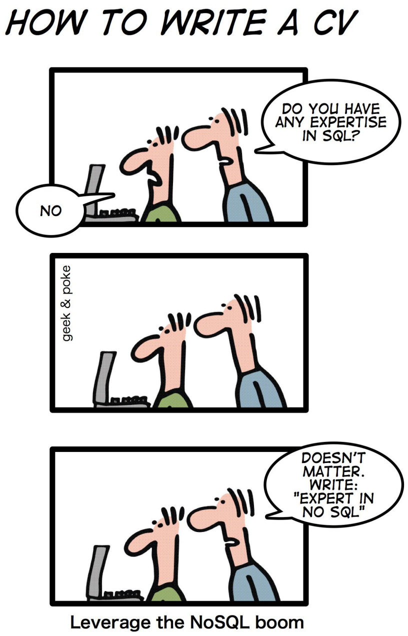


Hash browns < hash tables

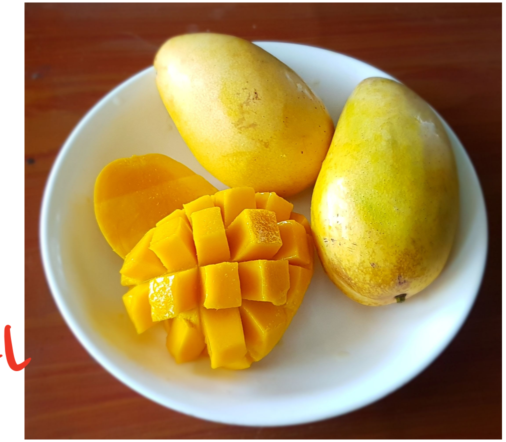
MongoDB and NoSQL

NoSQL

- Not Only SQL!
  - Non relational databases
  - Use very different data structures compared to traditional relational databases
- Reading: https://www.mongodb.com/nosql-explained
- Different data models used by different distributions
- Data in a common set doesn’t need to adhere to a schema



MongoDB

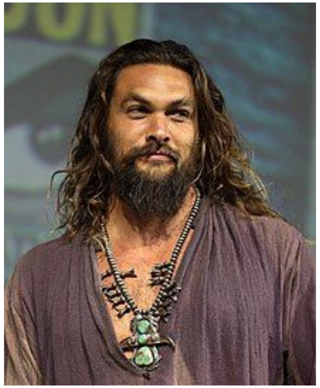


- Instead of rows we have documents
  - Fields and values
    - I.E “Age”: “7”
- Instead of relations/tables we have collections = table in SQL
- Use Javascript instead of SQL to interact
- Data is in JSON
- https://docs.mongodb.com/manual/reference/sql-comparison/

JSON

- JavaScript Object notation
- {Key: Value}
  - var data = {"Name": "Alice", "Major": CS", "University": "UofM", "Hobby": "Beating MSU"};
  - Can have nested key values as well:
    - {"Location": {"City": "Ann Arbor", "State": "Michigan", "Country": "USA"}};
- Retrieve data by data["Name"]
  - Returns "Alice"
- JSON Objects are not ordered

nested key-value pair

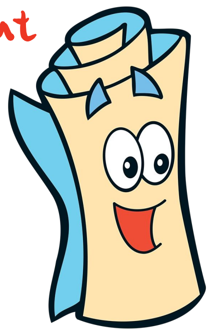


note: JSON has nothing to do with Jason Momoa, but again, they sound similar

MapReduce

MapReduce

- Programming paradigm to calculate aggregate analytics processing large amount of data in parallel
- Designed to be highly parallelized, which works great when we have a large set of data and many processors
  - Historically performed on Hadoop (distributed storage) framework
- Often performs worse compared to an equivalent serial algorithm working on a single machine
  - But we will use it anyway with MongoDB in P3, to get practice with the paradigm



MapReduce

- Needed for queries 7 and 8
- Mapper
  - Run on each record
  - Emits (a key and a value) intermediate key-value
- Reducer
  - Run on each mapped object
  - Combine values in aggregate that share key to get value of interest
  - **It's possible to have multiple calls to reduce!**
- Finalizer
  - Specific to MongoDB
  - Performs any last calculations before returning in final form

grouping :
key [x, y, z]

MapReduce

First names

Steve		{Steve: 1}		{Steve: 1}		{Steve: 1}
Alice		{Alice: 1}		{Alice: 1}		{Alice: 3}
Bob	Map	{Bob: 1}	Reduce	{Bob: 2}	Reduce	{Bob: 2}
Bob		{Bob: 1}		{Eve: 1}		{Eve: 1}
Eve		{Eve: 1}		{Alice: 2}		
Alice		{Alice: 1}				
Alice		{Alice: 1}				

take care by seperate machine

group by key

Project 3 Intro

Project 3

- Part A: Java code to export database to JSON
    - Need to perform this on CAEN just like in Project 2
- Part B: MongoDB Queries
    - You can run entirely on your local machine (requires installing MongoDB)
        - We won't be able to help with specific installation issues though
    - Use the eecs484 server through CAEN
        - Set up your project on CAEN and edit the Makefile as specified in the spec
        - Run commands which will connect to a MongoDB server setup for you on the eecs484 server
    - Write the queries!
        - Lots of helpful references in the spec to various MongoDB documentation

General Tips

- Try to focus on getting the solution correct rather than doing it in the fanciest way
  - Yes you can use an aggregate, group, out pipeline or you can iterate a couple times
  - No efficiency tests
  - No private tests
  - Most important thing: make sure you understand how and why your code works
- Take the time to get familiar with Javascript
  - Documentation will be your best friend
- Query 5 is the hardest
  - Has a similar concept to how you dealt with the friends relation in Query 6 on Project 2
    - Completely different code but similar work around needed

Query 7

key is month

- Find number of users born in each month using map reduce
- Mapper
  - Given access to a user (this) what can we return that will give us useful information?
  - Emit Tuple: (key, value)
  - Hint: Think about a way to mark that this user was born in this month
- Reducer
  - Given access to a set of values that correspond to a key, how do we combine these values?
  - Return value
    - Should match the same time of value output by the mapper
  - Hint: The value output at the end of this method should be the number of users (that we know of) born in the month denoted by key
- Finalizer
  - Get the final answers for each key ready to return
  - No change needed for query 7 :)

Query 8

city as the key

- Find average friend count per city
- Mapper
  - Given access to a user (this) what can we return that will give us useful information?
  - Emit Tuple: (key, value)
  - Hint: Value can be a tuple in it of itself!

☆ don't calculate avg in reducer

- Reducer
  - Given access to a set of values that correspond to a key, how do we combine these values?
  - Return value - should match the same type of value output by the mapper
  - Hint: The reducer can be called multiple times during execution
    - We can take intermediate sums, but not intermediate averages

calculate here

- Finalizer
  - Get the final answers for each key ready to return
  - Small change needed for query 8
  - Hint: We can't compute an average in the reducer, but we can in the finalizer

- Find average of {5, 3, 9, 5, 13} given that they all map to the same key
  - Incorrect solution:
    - Reducer: Find average of 5 and 3 (4)
    - Reducer: Find average of 9 and 5 and 13 (9)
    - Reducer: Find the average of the averages (gives us 6.5 when the real answer is 7)
  - Correct solution
    - Reducer: sums 5 and 3, and outputs 8
    - Reducer: sums 9 and 5 and 13, and outputs 27
    - Reducer: sums 8 and 27, and outputs 35
    - Finalizer: computes the average 35/5 to output 7 as the average

Get started on HW4 and Project 3!