Week 6

Theory

B+trees

- The keys in leaf nodes are copies of keys from the data file. These keys are distributed among the leaves in sorted order, from left to right.
- At the root, there are at least two used pointers.[2] All pointers point to B-tree blocks at the level below.
- At a leaf, the last pointer points to the next leaf block to the right, i.e., to the block with the next higher keys. Among the other n pointers in a leaf block, at least $\lfloor (n+1)/2 \rfloor$ of these pointers are used and point to data records; unused pointers are null and do not point anywhere. The ith pointer, if it is used, points to a record with the ith key.

- At an interior node, all $n+1$ pointers can be used to point to B-tree blocks at the next lower level. At least $\lceil (n+1)/2 \rceil$ of them are actually used (but if the node is the root, then we require only that at least 2 be used, regardless of how large n is). If j pointers are used, then there will be $j-1$ keys, say $K_1, K_2, \ldots, K_{j-1}$. The first pointer points to a part of the B-tree where some of the records with keys less than K_1 will be found. The second pointer goes to that part of the tree where all records with keys that are at least K_1, but less than K_2 will be found, and so on. Finally, the jth pointer gets us to the part of the B-tree where some of the records with keys greater than or equal to K_{j-1} are found. Note that some records with keys far below K_1 or far above K_{j-1} may not be reachable from this block at all, but will be reached via another block at the same level.
- All used pointers and their keys appear at the beginning of the block, with the exception of the $(n+1)$st pointer in a leaf, which points to the next leaf.

Extensible hashing

1. There is a level of indirection for the buckets. That is, an array of pointers to blocks represents the buckets, instead of the array holding the data blocks themselves.

2. The array of pointers can grow. Its length is always a power of 2, so in a growing step the number of buckets doubles.

3. However, there does not have to be a data block for each bucket; certain buckets can share a block if the total number of records in those buckets can fit in the block.

4. The hash function h computes for each key a sequence of k bits for some large k, say 32. However, the bucket numbers will at all times use some smaller number of bits, say i bits, from the beginning or end of this sequence. The bucket array will have 2^i entries when i is the number of bits used.

Insertion:

1. If $j < i$, then nothing needs to be done to the bucket array. We:
   (a) Split block B into two.
   (b) Distribute records in B to the two blocks, based on the value of their $(j+1)$st bit — records whose key has 0 in that bit stay in B and those with 1 there go to the new block.
   (c) Put $j+1$ in each block's "nub" (header) to indicate the number of bits used to determine membership.
   (d) Adjust the pointers in the bucket array so entries that formerly pointed to B now point either to B or the new block, depending on their $(j+1)$st bit.

Note that splitting block B may not solve the problem, since by chance all the records of B may go into one of the two blocks into which it was split. If so, we need to repeat the process on the overfull block, using the next higher value of j and the block that is still overfull.

2. If $j = i$, then we must first increment i by 1. We double the length of the bucket array, so it now has 2^{i+1} entries. Suppose w is a sequence of i bits indexing one of the entries in the previous bucket array. In the new bucket array, the entries indexed by both $w0$ and $w1$ (i.e., the two numbers derived from w by extending it with 0 or 1) each point to the same block that the w entry used to point to. That is, the two new entries share the block, and the block itself does not change. Membership in the block is still determined by whatever number of bits was previously used. Finally, we proceed to split block B as in case 1. Since i is now greater than j, that case applies.

Linear hashing:

- The number of buckets n is always chosen so the average number of records per bucket is a fixed fraction, say 80%, of the number of records that fill one block.
- Since blocks cannot always be split, overflow blocks are permitted, although the average number of overflow blocks per bucket will be much less than 1.
- The number of bits used to number the entries of the bucket array is $\lceil \log_2 n \rceil$, where n is the current number of buckets. These bits are always taken from the *right* (low-order) end of the bit sequence that is produced by the hash function.
- Suppose i bits of the hash function are being used to number array entries, and a record with key K is intended for bucket $a_1 a_2 \cdots a_i$; that is, $a_1 a_2 \cdots a_i$ are the last i bits of $h(K)$. Then let $a_1 a_2 \cdots a_i$ be m, treated as an i-bit binary integer. If $m < n$, then the bucket numbered m exists, and we place the record in that bucket. If $n \leq m < 2^i$, then the bucket m does not yet exist, so we place the record in bucket $m - 2^{i-1}$, that is, the bucket we would get if we changed a_1 (which must be 1) to 0.

Insertion:

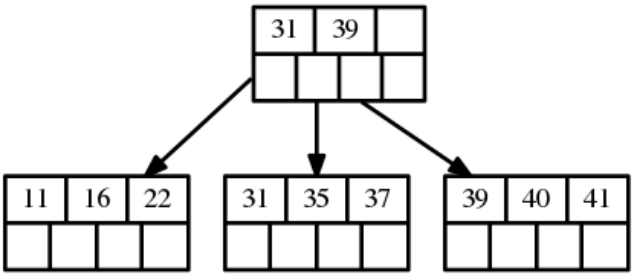
When we insert a new record, we determine its bucket by the algorithm outlined in Section 3.7. We compute $h(K)$, where K is the key of the record, and we use the i bits at the end of bit sequence $h(K)$ as the bucket number, m. If $m < n$, we put the record in bucket m, and if $m \geq n$, we put the record in bucket $m - 2^{i-1}$. If there is no room in the designated bucket, then we create an overflow block, add it to the chain for that bucket, and put the record there.

Each time we insert, we compare the current number of records r with the threshold ratio of r/n, and if the ratio is too high, we add the next bucket to the table. Note that the bucket we add bears no relationship to the bucket into which the insertion occurs! If the binary representation of the number of the bucket we add is $1a_2 \cdots a_i$, then we split the bucket numbered $0a_2 \cdots a_i$, putting records into one or the other bucket, depending on their last i bits. Note that all these records will have hash values that end in $a_2 \cdots a_i$, and only the ith bit from the right end will vary.

The last important detail is what happens when n exceeds 2^i. Then, i is incremented by 1. Technically, all the bucket numbers get an additional 0 in front of their bit sequences, but there is no need to make any physical change, since these bit sequences, interpreted as integers, remain the same.

Old Exam1 task 4

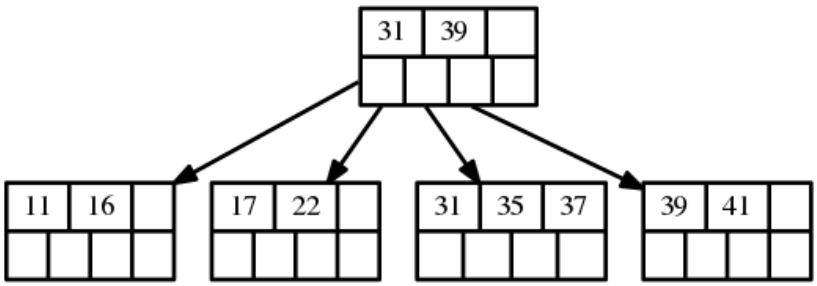
Inserting 40… there are enough space so np:



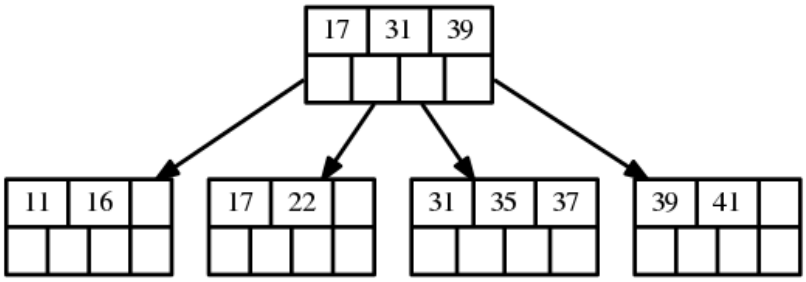
Inserting 17:

Should be in the lower left… need to split this:

Leaving the first $\left\lceil \frac{n+1}{2} \right\rceil = 2$

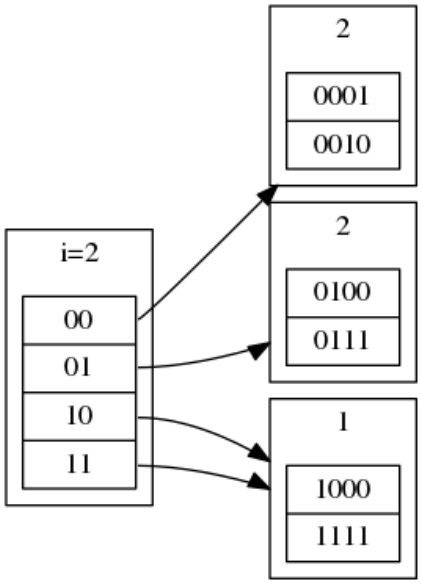


Parent gets updated:

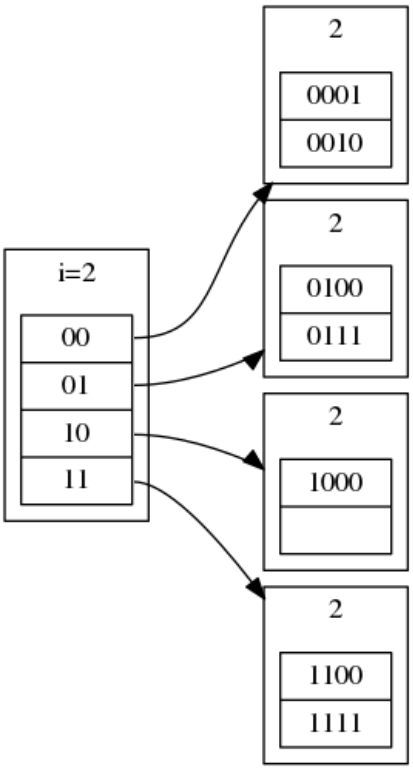


Extensible hashing:

Original:



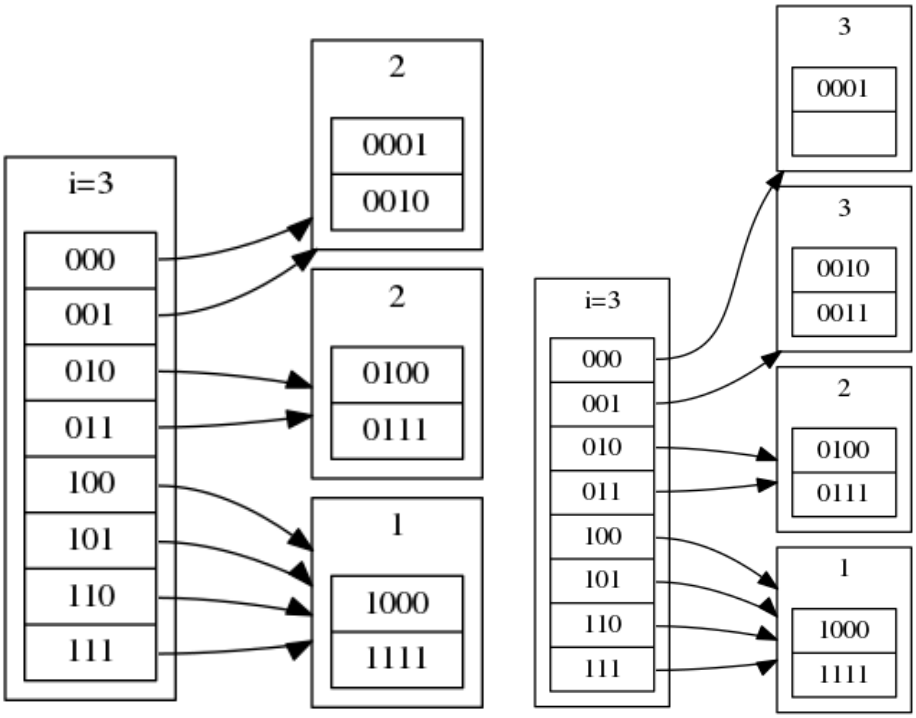
Inserting 1100, splitting j<i:



Inserting 0011 into original table:

Should be in the first bucket… no more space, i=j:

- Increment i
- Update pointers
- Insert new shit



Linear hashing:

$i = 2$ $n = 3$ $r = 4$		00	0100
		01	1011
			1101
		10	1010

Inserting np.

Looking at the ratio $\frac{r}{n}$

(n buckets, r records)

Obs should take the capacity of the buckets into account:

$$\frac{r}{n \cdot f} = \frac{5}{3 \cdot 2} = 0{,}83$$

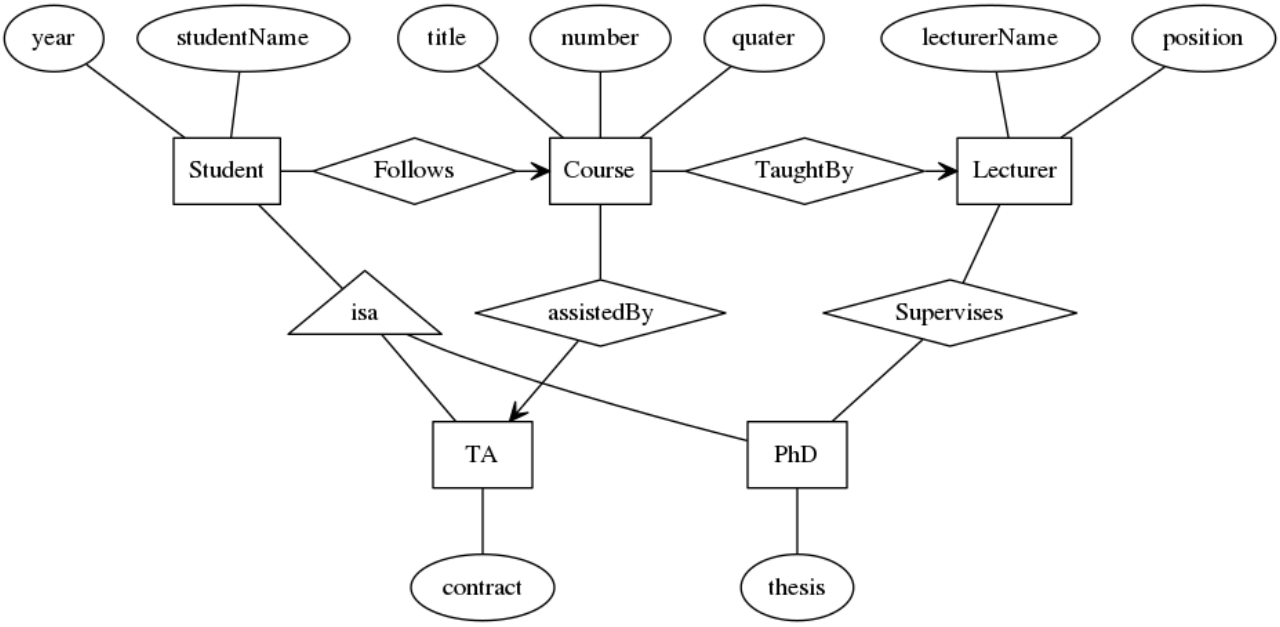
Which is larger than 0.80 so we should add new bucket:

$i = 2$ $n = 4$ $r = 5$		00	0100
		01	
			1101
		10	1010
			0010
		11	1011

Checking if $n > 2^i$ it is not, we are fine…

Old Exam2 task 1

E/R -diagram:



Conversion using E/R-style:

Student(name, year)

TA(name, year, contract)

PhD(name, year, thesis)

Course(title, name, number)

Lecturer(name, position)

Follows(name, year, number)

TaughtBy(name, number)

assistsIn(name, year, number)

supervises(phdName, lecturerName, year)

Old Exam2 task 2

Relation	**FD's**	**Keys**	**BCNF Violations**	**BCNF Decomposition**
$R(A,B,C,D,E)$	$A \rightarrow B$ $AD \rightarrow C$ $B \rightarrow E$ $E \rightarrow A$	AD ED BD	$A \rightarrow B$ $B \rightarrow E$ $E \rightarrow A$ $A \rightarrow E$ $B \rightarrow A$ $E \rightarrow B$	Decompose on: $A \rightarrow B$ $R1(A,B,E)$ $R2(A,C,D)$
$R1(A,B,E)$	$A \rightarrow B$ $B \rightarrow E$ $E \rightarrow A$	A B E	None	Done
$R2(A,C,D)$	$AC \rightarrow D$	AC	None	Done

Meaning we are ending up with:

$$R1(A,B,E)$$

$$R2(A,C,D)$$

Relation	**FD's**	**Keys**	**3NF Violations**	**3NF Decomposition**
$R(A,B,C,D,E)$	$A \rightarrow B$ $AD \rightarrow C$ $B \rightarrow E$ $E \rightarrow A$	AD ED BD	None all rhs part of key	Done

Old Exam2 task 3

a. …

```
CREATE TABLE Cars (
        Name        VARCHAR(20),
        color       VARCHAR(20),
        price       INT CHECK(price >= 0)
);
```

b. …

```
SELECT name, count(*)
FROM Salesmen, Sale, Person
WHERE
        salesmanid = id
        AND
        salesmanid = person.id
GROUP by Sale.id;
```

c. …

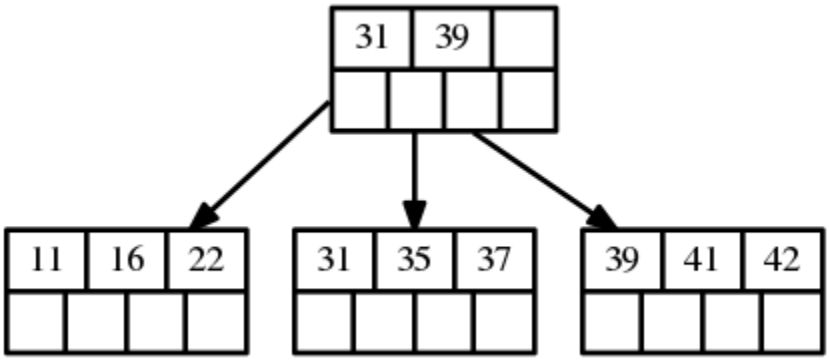
```
UPDATE Cars
SET price = price*0.90
WHERE name in (SELECT name FROM Fords);
```

d. …

```
SELECT p1.name, p2.name
FROM
        ((Customers JOIN Sale ON Customers.id = Sale.customerid) as p
        JOIN
        Cars ON car.name = p.carname) as p1, p1 as p2
WHERE
        p1.customerid < p2.customerid AND p1.color == p2.color;
```

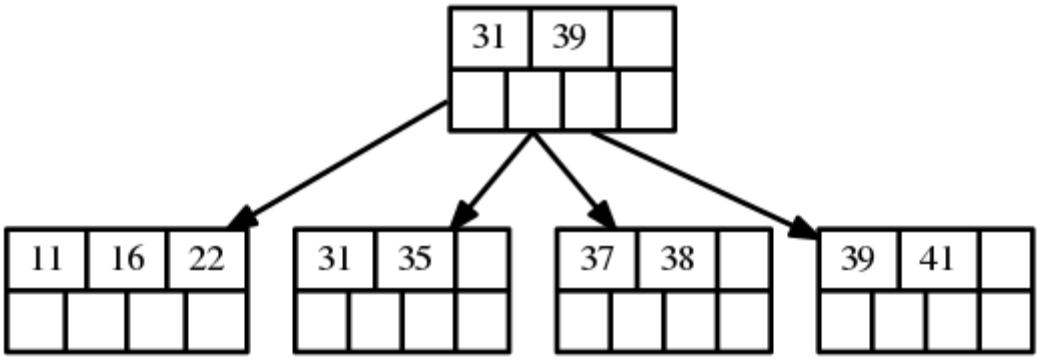
Old Exam2 task 4

Inserting 42 there are enough space so np:

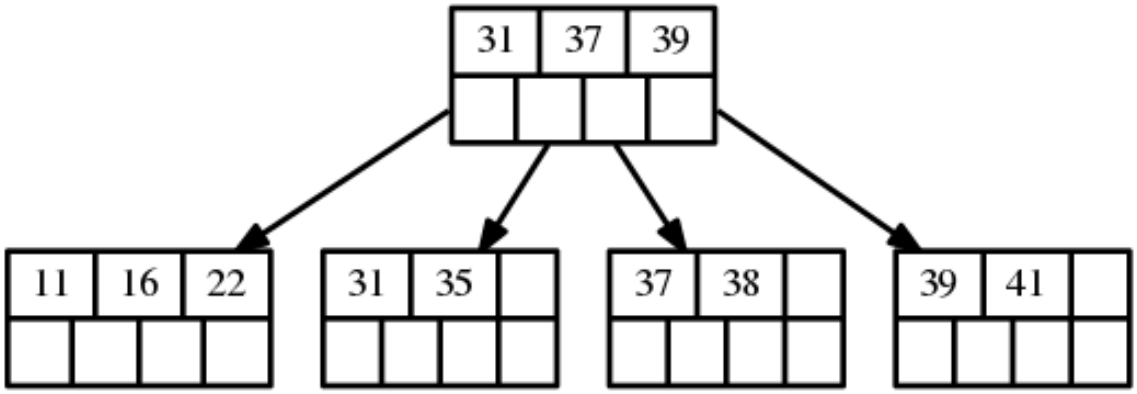


Inserting 38 into original tree:

Should be in the middle node, no more space, splitting it:

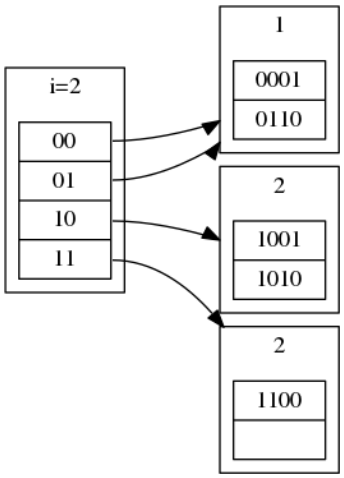


Updating parent:



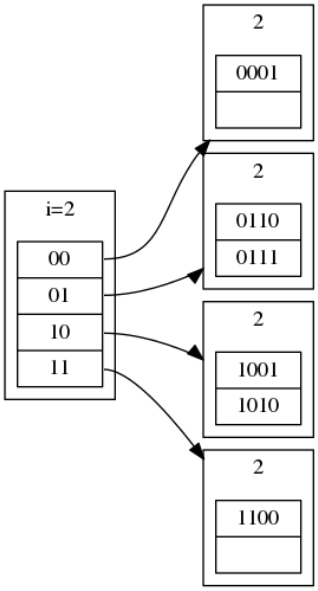
Extensible hashing:

Original:



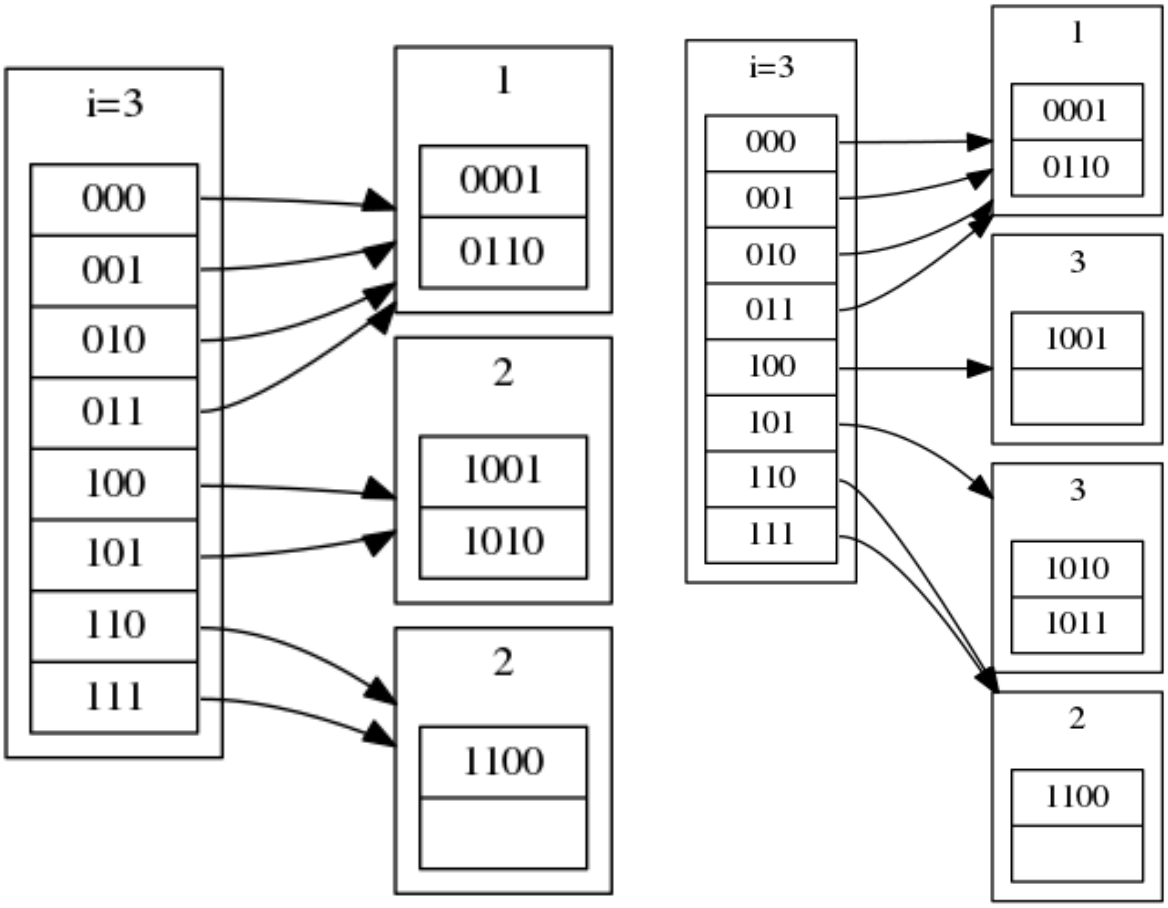
Inserting 0111:

Splitting the first node, incrementin j:



Inserting 1011 into original table:

- Not more space
- I=j
- Increment I
- Reorganize
- Split and insert


i=3
000
001
010
011
100
101
110
111
1
0001
0110
2
1001
1010
2
1100
i=3
000
001
010
011
100
101
110
111
1
0001
0110
3
1001
3
1010
1011
2
1100

Linear hashing:

Inserting 0001.

$i = 2$ $n = 3$ $r = 4$		00	0000
		01	0011 0111
		10	1010

Not any more space… inserting using linear chaining

Looking at the ration $\frac{r}{n}$

(n buckets, r records)

Obs should take the capacity of the buckets into account:

$$\frac{r}{n \cdot f} = \frac{5}{3 \cdot 2} = 0{,}83$$

Which is larger than 0.80 so we should add new bucket, and redistribute:

$i = 2$ $n = 4$ $r = 5$		00	0000
		01	0001
		10	1010
		11	0011 0111

Checking if $n > 2^i$ it is not, we are fine…