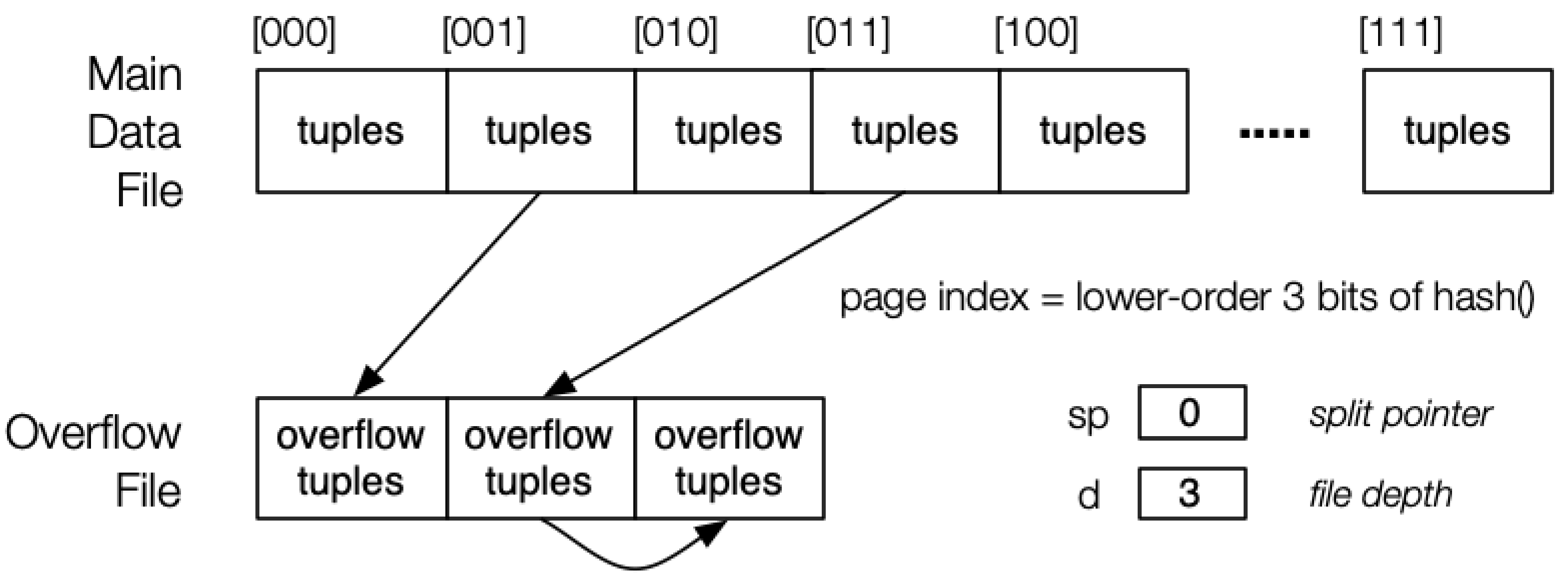
>>

Linear Hashing

❖ Linear Hashing

File organisation:

- file of primary data pages
- file of overflow data pages
- registers called *split pointer* (sp) and *depth* (d)



❖ Linear Hashing (cont)

Linear Hashing uses a systematic method of growing data file ...

- hash function "adapts" to changing address range (via *sp* and *d*)
- systematic splitting controls length of overflow chains

Advantage: does not require auxiliary storage for a directory

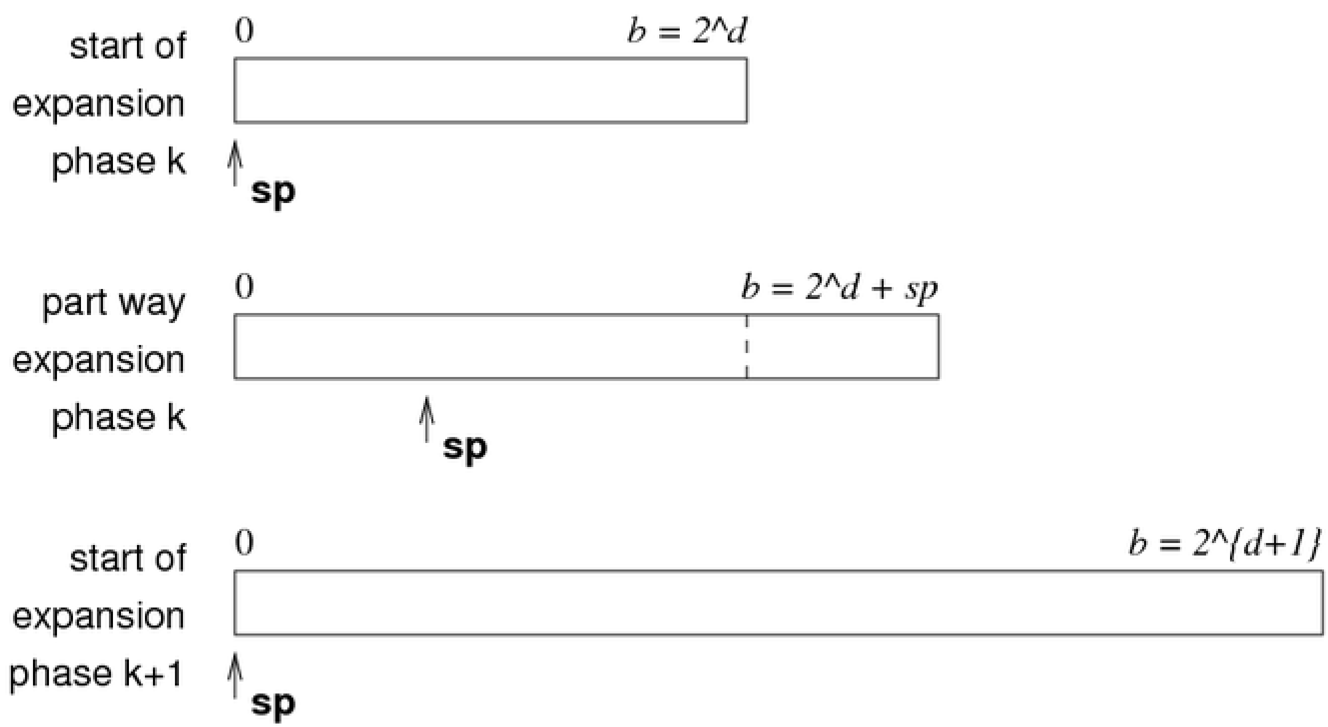
Disadvantage: requires overflow pages (don't split on full pages)

<< ∧ >>

❖ Linear Hashing (cont)

File grows linearly (one page at a time, at regular intervals).

Has "phases" of expansion; over each phase, *b* doubles.



❖ Selection with Lin.Hashing

If $b=2^d$, the file behaves exactly like standard hashing.

Use d bits of hash to compute page address.

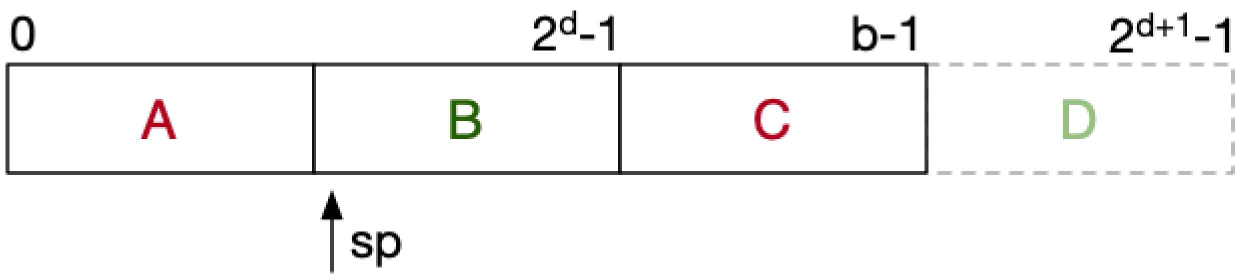
```
// select * from R where k = val
h = hash(val);
P = bits(d,h);  // lower-order bits
for each tuple t in page P
         and its overflow pages {
    if (t.k == val) return t;
}
```

Average $Cost_{one}$ = $1+Ov$

<< ^ >>

❖ Selection with Lin.Hashing (cont)

If b != 2^d, treat different parts of the file differently.



Parts *A* and *C* are treated as if part of a file of size 2^{d+1}.

Part *B* is treated as if part of a file of size 2^d.

Part *D* does not yet exist (tuples in *B* may eventually move into it).

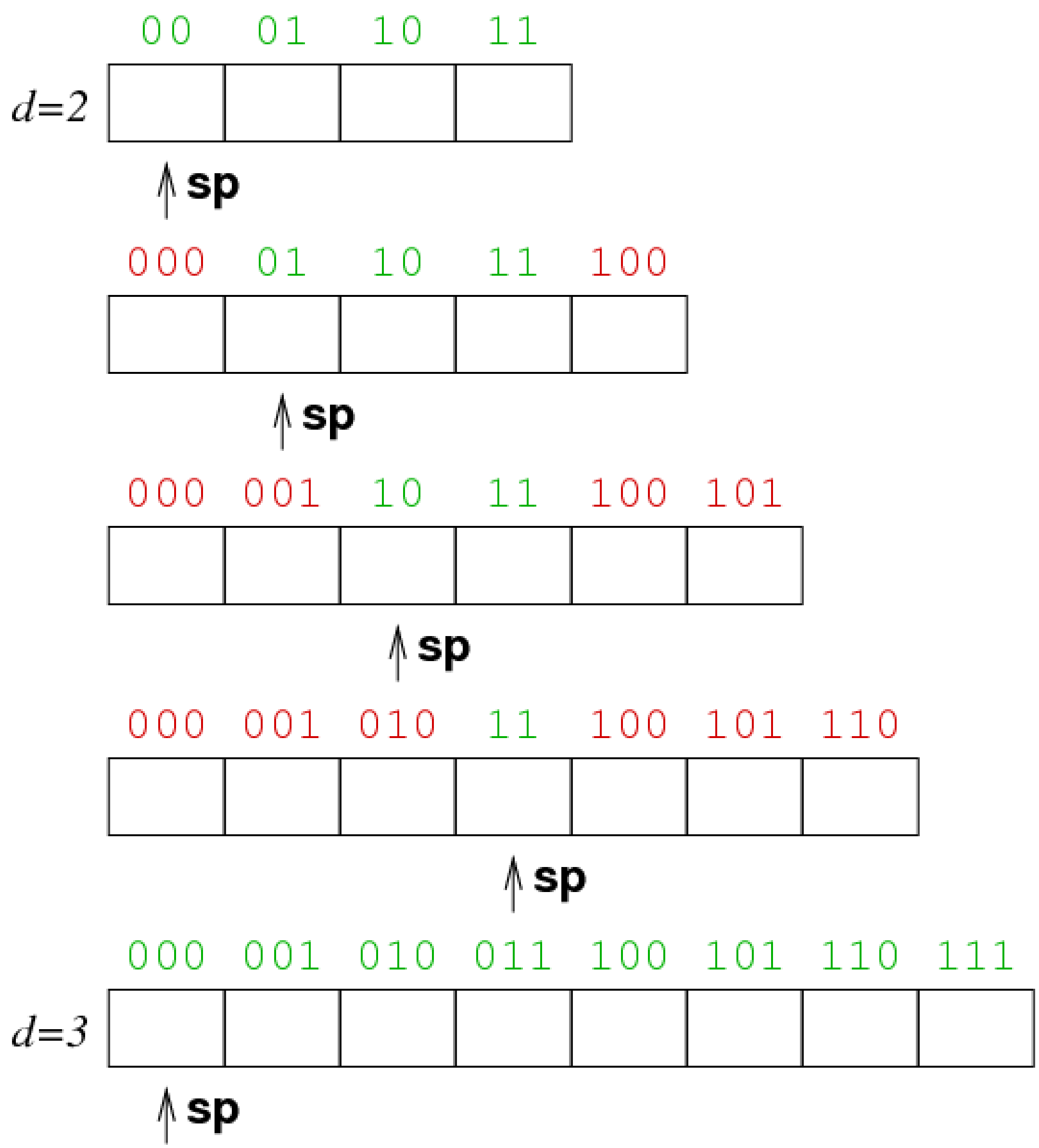
❖ Selection with Lin.Hashing (cont)

Modified search algorithm:

```
// select * from R where k = val
h = hash(val);
pid = bits(d,h);
if (pid < sp) { pid = bits(d+1,h); }
P = getPage(f, pid)
for each tuple t in page P
         and its overflow pages {
    if (t.k == val) return R;
}
```

<< ^ >>

❖ File Expansion with Lin.Hashing



<< ∧ >>

❖ Insertion with Lin.Hashing

Abstract view:

```
pid = bits(d,hash(val));
if (pid < sp) pid = bits(d+1,hash(val));
// bucket P = page P + its overflow pages
P = getPage(f,pid)
for each page Q in bucket P {
    if (space in Q) {
        insert tuple into Q
        break
    }
}
if (no insertion) {
    add new ovflow page to bucket P
    insert tuple into new page
}
if (need to split) {
    partition tuples from bucket sp
          into buckets sp and sp+2^d
    sp++;
```

```
        if (sp == 2^d) { d++; sp = 0; }
    }
```

❖ Splitting

How to decide that we "need to split"?

Two approaches to triggering a split:

- split every time a tuple is inserted into full page
- split when load factor reaches threshold (every *k* inserts)

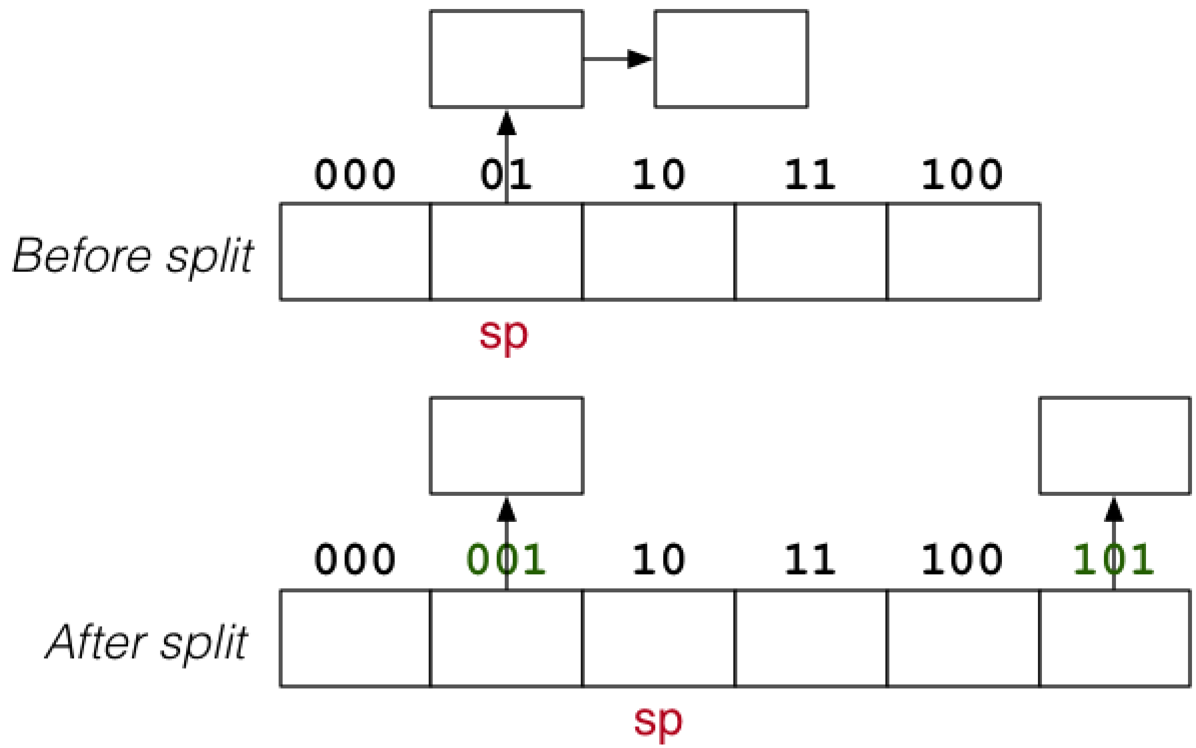
Note: always split page *sp*, even if not full or "current"

Systematic splitting like this ...

- eventually reduces length of every overflow chain
- helps to maintain short average overflow chain length

❖ Splitting (cont)

Splitting process for page *sp*=**`01`**:



❖ Splitting (cont)

Splitting algorithm:

```
// partition tuples between two buckets
newp = sp + 2^d; oldp = sp;
for all tuples t in P[oldp] and its overflows {
    p = bits(d+1,hash(t.k));
    if (p == newp)
        add tuple t to bucket[newp]
    else
        add tuple t to bucket[oldp]
}
sp++;
if (sp == 2^d) { d++; sp = 0; }
```

❖ Insertion Cost

If no split required, cost same as for standard hashing:

$Cost_{insert}$: Best: $1_r + 1_w$, Avg: $(1+Ov)_r + 1_w$, Worst: $(1+max(Ov))_r + 2_w$

If split occurs, incur $Cost_{insert}$ plus cost of splitting:

- read page *sp* (plus all of its overflow pages)
- write page *sp* (and its new overflow pages)
- write page $sp+2^d$ (and its new overflow pages)

On average, $Cost_{split} = (1+Ov)_r + (2+Ov)_w$

❖ Deletion with Lin.Hashing

Deletion is similar to ordinary static hash file.

But might wish to contract file when enough tuples removed.

Rationale: r shrinks, b stays large $\Rightarrow$ wasted space.

Method:

- remove last bucket in data file (contracts linearly).
- merge tuples from bucket with its buddy page (using d-1 hash bits)

Produced: 10 Mar 2021