B+ tree: wrapping up for now

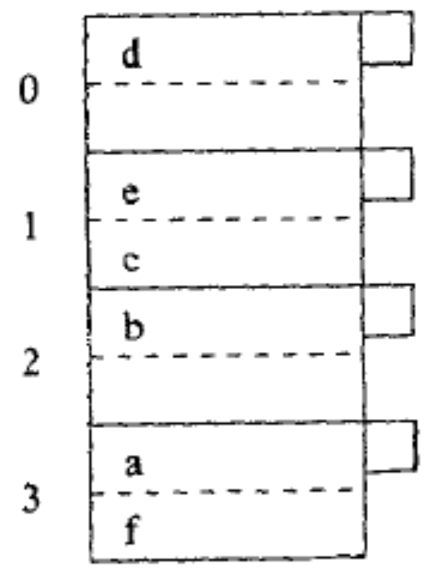
- B+ deletion works by merging nodes to avoid underflow
  - Deletion is also recursive.
  - In practice, however, we can simply delete the key from the leaf...
  - ... as we are often accurate in assuming that databases grow more often than they shrink.
- B+ tree can be used as a sorting algorithm for disk-based sorting.

Hash Tables

- As applied to indexes, these are variants of what you have already seen in other courses:
  - Hash function h(): takes a search key and computes an integer in the range of 0 to B-1
  - Bucket array: array of B linked lists.
  - A <key, value> pair is to be stored in bucket numbered h(key)
- These hash tables must reside on disk (i.e., too large for main memory)
  - Each bucket is, therefore, the size of a disk block
  - Buckets can be extended via overflow blocks
- With a good hash function and number of buckets, we expect behavior comparable to a B+ tree.

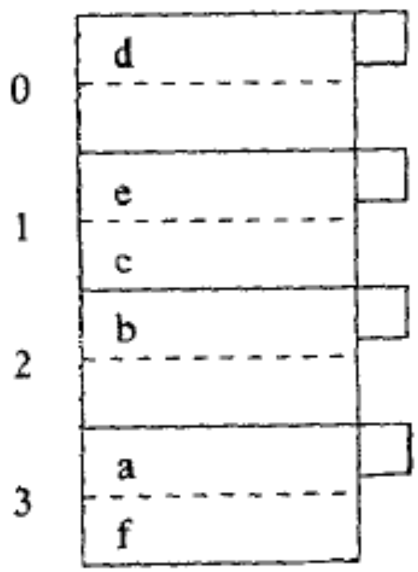
Hash table: example

- Buckets are numbered from 0 to 3
- Keys are letters
- Two <key, value> pairs per bucket.
  - Hash function here is indicated (e.g., h(a)=3, h(b)=2, h(c)=1, etc.)
- Assume some in-memory structure maps from hash number to disk block (bucket)
- Additional information for each block is represented by a small tab.

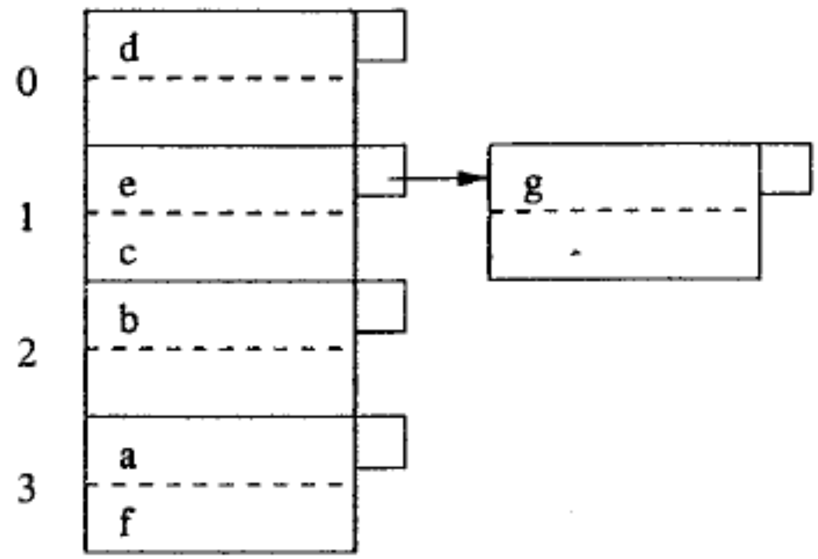


Hash table: insert key/value with key "g"

h("g") = 1

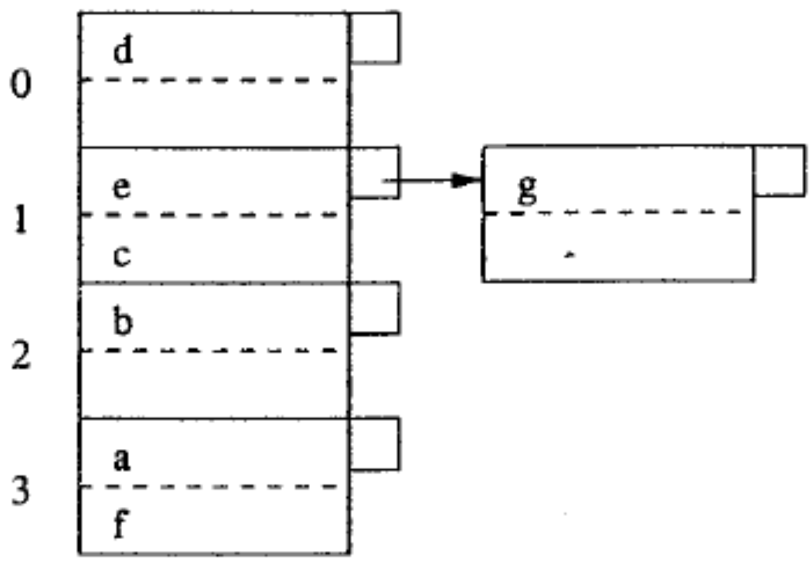


before

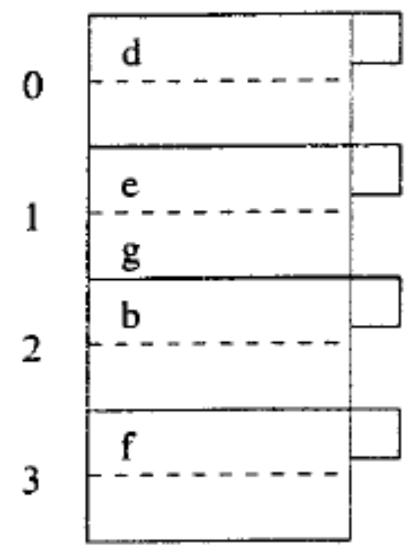


after

Hash table: delete key/value with key "c"



before



after

Static vs. dynamic hash tables

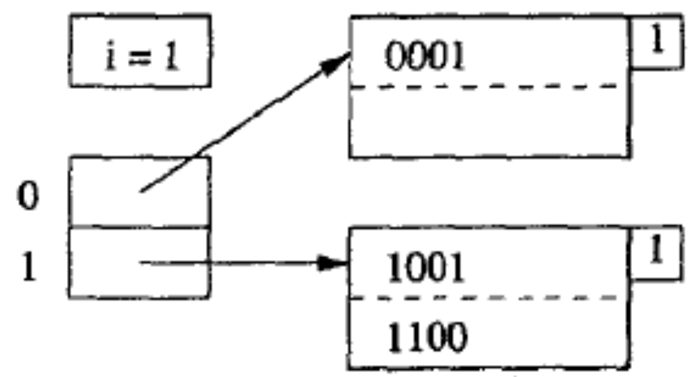
- Where the number of buckets does not change:
  - We have a **static hash table**
- A problem with hash tables is the degeneration of buckets into long linked lists
  - Traversing through the linked list would be one I/O per overflow block
- Recall: want to keep I/O cost low
  - Hence want to keep the block-per-bucket ratio low.
  - Best way to do that: keep an optimal number of buckets at all time
- **Dynamic hash table**:
  - B is allowed to vary so that there is about one block per bucket.

Extensible hash table

- (Sometimes also called **extendible hashing**.)
- A kind of dynamic hash table
  - Indirection (via pointers) used to access buckets
  - Array of pointers to buckets can grow.
  - There need not be a disk block per bucket (i.e., certain buckets can share a block)
  - Hash function **h()** computes a sequence of **k** bits for the key
  - Some buckets use a smaller number of the k bits, some a larger number (i.e., a number **i** for each bucket is maintained where i <= k)

Small extensible hash table

- Hash function produces a sequence of four bits
  - k = 4
- Only one of the bits is used at present
  - Indicated by the box above the bucket pointers where i = 1
  - We use the most significant bits here.
- Relationship between i and k:
  - First bucket holds all keys whose hash value begins with 0
  - Second bucket holds all keys whose hash value begins with 1.



Extensible hash table insertion

- Goal:
  - Insert <K, V> pair into an appropriate bucket...
  - ... yet make sure we add buckets only when necessary ...
  - ... and adjust the number of key bits used to map to the bucket.
- Recall:
  - h(K) is a bit sequence
  - i indicates how many of the first bits of h(K) are used to locate a bucket
  - i is stored with the bucket
- Three cases:
  A. There is room in the bucket for <K, V>.
  B. There is no room in the bucket, and to locate a key-value pair we need fewer bits than i.
  C. There is no room in the bucket, and to locate a key-value pair we need exactly i bits.

Case B

```
k = h(K)
B = bucket_array.forkey(k)
j = identifying_bits(B, k)
```

- There is no room in the bucket, and j < i (where j is stored in that "tab" attached to a bucket).

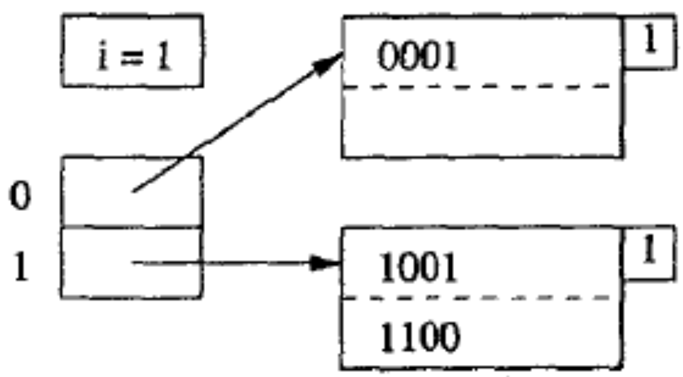
```
B0 = new Bucket()
B1 = new Bucket()

for each item in B:           # Move items from B into either B0 or B1
    if item.key.bit(j+1) == 0:
        B0.insert(item)
        bucket_array.setkey(B0, item.key())   # Bucket array now points B0
    else:
        B1.insert(item)
        bucket_array.setkey(B1, item.key())   # Bucket array now points to B1
```

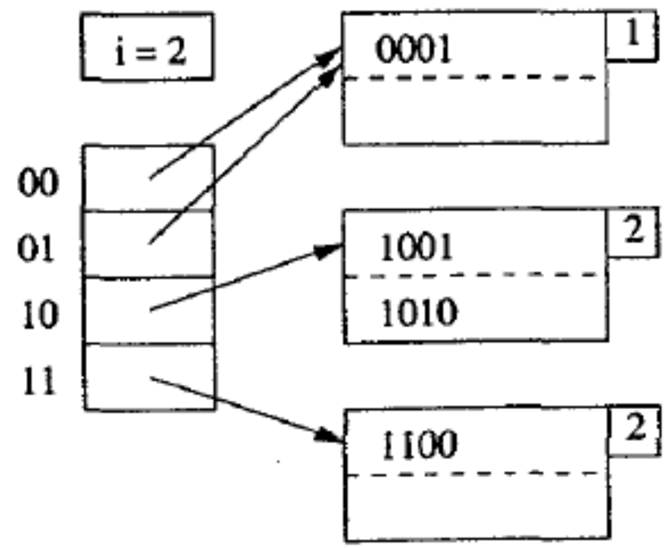
Case C

- There is no room in the bucket, and to locate a key-value pair we need exactly **i** bits.
- In all of our examples so far, we haven't shown how this global value of **i** is changed.
  - A change was sneaked in
- With this case, we must increment **i**, possibly several times
  - Incrementing **i** means doubling the number of bucket pointers (from 2^i to 2^{i+1})
  - Must update pointers to buckets as some pointer pairs still lead to the same bucket.
- We only split a bucket when we have enough bucket pointers to that case B takes over.

Example: Insert <K, V> where h(K) = 1010

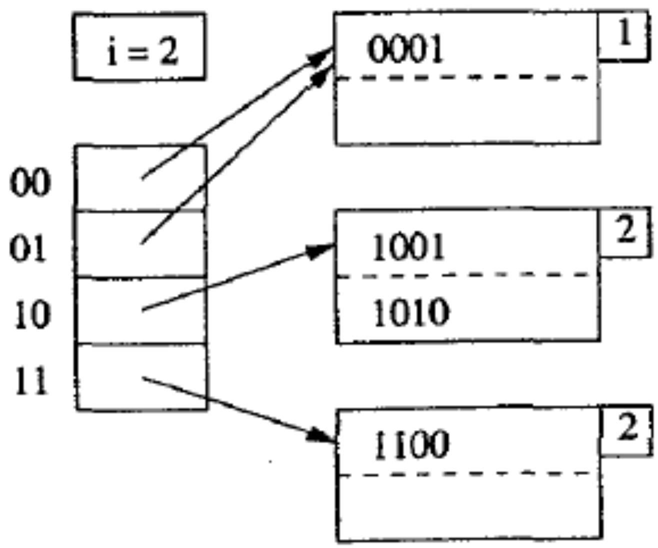


before

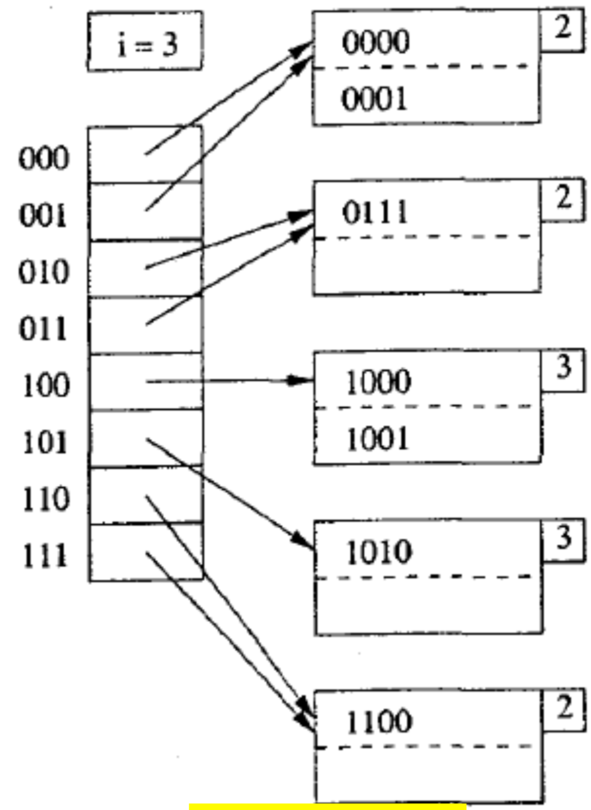


after

Insert pairs with keys 0000, 0111, 1000



before



after