EXTENDIBLE HASHING I

Contents of today's lecture:

- What is extendible hashing.
- Insertions in extendible hashing.

Reference: Folk, Zoellick and Riccardi, File Structures, 1998. Sections 12.1,12.2,12.3(overview only).

What is extendible hashing ?

- It is an approach that tries to make hashing **dynamic**, i.e. to allow insertions and deletions to occur without resulting in poor performance after many of these operations.
  Why this is not the case for ordinary hashing?
- Extendible hashing combines two ingredients:
  hashing and **tries**.
  (tries are digital trees like the one used in Lempel-Ziv)
- Keys are placed into buckets, which are independent parts of a file in disk.
  Keys having a hashing address with the same prefix share the same bucket.
  A trie is used for fast access to the buckets. It uses a prefix of the hashing address in order to locate the desired bucket.

Tries and buckets

Consider the following grouping of keys into buckets, depending on the prefix of their hash addresses:

bucket:	this bucket contains keys with hash address with prefix:
A	01
B	10
C	11

Drawing of the trie that provides and index to buckets:

```
                         +-------+
          *------------> |   A   |
    0                    +-------+


   *
                         +-------+
    1       0  *-------> |   B   |
                         +-------+
         *
                         +-------+
            1  *-------> |   C   |
                         +-------+
```

Note: You need to connect the parents to the children in the drawing of the trie above.

Directory structure and buckets

Representing the trie as a tree would take too much space. Instead, do the following:

1) Extend the tree to a complete binary tree:

```
            0  *----+
                    +--> +--------+
         *               |   A    |
    0               +--> +--------+
            1  *----+
   *
                         +--------+
    1       0  *-------> |   B    |
                         +--------+
         *
                         +--------+
            1  *-------> |   C    |
                         +--------+
```

2) Flatten the trie up into an array:

```
    +------+      +--------+
 00 |      |------>|   A    |
    +------+   +-->+--------+
 01 |      |---+
    +------+      +--------+
 10 |      |------>|   B    |
    +------+      +--------+
 11 |      |--+
    +------+  |   +--------+
              +--->|   C    |
                  +--------+
```

How to search in extendible hashing ?

Searching for a key:

- Calculate the hash address of the key (note that no table size is specified, so we don't take "mod").
- Check how many bits are used in the directory (2 bits in the previous example). Call i this number of bits.
- Take the least significative i bits of the hash address (in reverse order). This gives an index of the directory.
- Using this index, go to the directory and find the bucket address where the record might be.

What makes it extendible?

So far we have not discussed how this approach can be dynamic, making the table expand or shrink as records are added or deleted. The dynamic aspects are handled by two mechanisms:

- Insertions and bucket splitting.
- Deletions and bucket combination.

Bucket splitting to handle overflow

Extendible hashing solves bucket overflow by splitting the bucket into two and if necessary increasing the directory size.
When the directory size increases it doubles its size a certain number of times.

Example:

To simplify matters, let us assume the keys are numbers and the hash function returns the number itself.
For instance, $h(20) = 20$.
Bucket size: 2

numbers	1	2	3	5	6	7
binary representation:	0001	0010	0011	0101	0110	0111

```
   +------+        +-------+
00 |      |------->| 2, 6  |
   +------+    +-->+-------+
01 |      |----+
   +------+        +-------+
10 |      |------->| 1, 5  |
   +------+        +-------+
11 |      |--+
   +------+  |     +-------+
             +---->| 3, 7  |
                   +-------+
```

```
   +-----+       +-------+
00 |     |------>| 2, 6  |
   +-----+    +-->+-------+
01 |     |---+
   +-----+       +-------+
10 |     |------>| 1, 5  |
   +-----+       +-------+
11 |     |--+
   +-----+  |    +-------+
            +--->| 3, 7  |
                 +-------+
```

Insert key 4 into the above structure:

- Since `4` has binary representation 0100, it must go into the first bucket. Since it is full, it will get split.
- The splitting will separate `4` from `2,6` since the last two bits of `4` in reverse order are `00` and the ones of `2,6` are `01`.
- the directory is prepared to accommodate the splitting, so no doubling is needed.

```
                 +-------+
            +--->|   4   |
   +-----+  |    +-------+
00 |     |--+    +-------+
   +-----+    +-->| 2, 6  |
01 |     |---+   +-------+
   +-----+       +-------+
10 |     |------>| 1, 5  |
   +-----+       +-------+
11 |     |--+
   +-----+  |    +-------+
            +--->| 3, 7  |
                 +-------+
```

Bucket splitting with increase in directory size

```
   +-----+       +-------+
00 |     |------>|  2, 6 |
   +-----+   +-->+-------+
01 |     |---+
   +-----+       +-------+
10 |     |------>|  1, 5 |
   +-----+       +-------+
11 |     |--+
   +-----+  |    +-------+
            +--->|  3, 7 |
                 +-------+
```

Insert key `9` into the above structure:

- Since `9` has binary representation 1001, it must go into the bucket indexed by 10. Since it is full, it will get split.
- The splitting will separate `1,9` from `5` since the last two bits of `1,9` in reverse order are `100` and the ones of `5` are `101`.
- The directory is not prepared to accommodate the splitting, so doubling is needed. The doubling in the directory size will add an extra bit to the directory index.

Result of the addition of key 9 as described above:

```
    +-----+
000 |     |---+
    +-----+   +-->+-------+
001 |     |------>|  2, 6 |
    +-----+   +-->+-------+
010 |     |---|
    +-----+   |
011 |     |---+   +-------+
    +-----+    +->| 1, 9  |
100 |     |----+  +-------+
    +-----+       +-------+
101 |     |------>|   5   |
    +-----+       +-------+
110 |     |------>+-------+
    +-----+   +-->|  3,7  |
111 |     |---+   +-------+
    +-----+
```

Insertion Algorithm

- Calculate the hash function for the key and the key address in the current directory.
- Follow the directory address to find the bucket that should receive the key.
- Insert into the bucket, splitting it if necessary.
- If splitting took place, calculate i: the number of bits necessary to differentiate keys within this bucket. Double the directory as many times as needed to create a directory indexed by i bits.

Note: This algorithm does not work if such an i does not exist. That is, if there are too many keys with the same hash address ("too many" here meaning more than the size of a bucket).

Why?

Which are possible fixes?

Practice exercise

Insert the following keys, into an empty extendible hashing structure:

2,10,7,3,5,16,15,9

Show the structure after each insertion.

All the intermediate steps will be performed in lecture.
Final solution for your checking only:

```
000 and 001 point to bucket with key 16
010 and 011 point to bucket with keys 2,10
100 and 101 point to bucket with key 5
110 points to bucket with key 3
111 points to bucket with keys 7,15
```