Extendible Hashing

Hashing gives us good performance for finding records. Unfortunately, the performance may be affected by additions and deletions to the hash table. As the table fills up, collisions become more probable and performance worsens. We need ways to make the hash table less sensitive to additions and deletions, and to extend the hash table as it fills up.

Topic	*Folk & Zoellick*
• Dynamic Files	§ N/A
• Tries	§ 11.2
• Extending the Hash Table	§§ 11.2, 11.3
• Shrinking the Hash Table	§ 11.4
• Dynamic Hashing	§ 11.6.1
• Linear Hashing	§ 11.6.2

Dynamic Files

Dynamic File

A file that changes a lot

- specifically, a file that we add to and delete from frequently

Our lectures in this course have become quite predictable:

1. try to apply some standard algorithm/structures to big files
2. change the algorithm/structures to work with big files
3. find out that the changes are good for static files, but are not so good for dynamic files
4. come up with new algorithms/structures for big/dynamic files

For example:

- B-trees
  1. apply binary search to big sorted files
     - poor performance from jumping around in the file
  2. paged binary trees
     - allow binary searching without the jumping around
  3. good paged binary tree performance requires balanced trees
     - balancing a paged binary tree after an addition or deletion can be expensive
  4. B-trees (or B*-trees) perform as well as paged binary trees for searching, but they're much less sensitive to additions and deletions

Hashing for Dynamic Data

The hashing we've seen requires the size of the hash table to be fixed:

- we can fix the table size small (just slightly bigger than the data requires) to avoid wastage
  - we end up with more collisions
- we can fix the table size large (closer to the size of the keyspace) to avoid collisions
  - we end up with more wastage

Although we can avoid collisions by making the table bigger, it is not feasible to make the table big enough to allow a perfect hashing function.

Hashing for Dynamic Data (cont.)

Fixing the hash table at a reasonable size means that:

1. we *will* have collisions
2. as we add more records, the table fills up and collisions increase
3. as collisions increase, performance worsens (direct access degrades to searching)
4. deleting records does not necessarily make things better
5. eventually, if we add enough records, we may hit the limit of the table or we may *saturate the hash space*
    - have as many records as there are distinct hash values (or even more)

Q: Why do we have to fix the hash table size?

and

Q: Why can’t we have a huge hash space?

A:

Filing for Divorce

We have to fix the size of the hash table because that size is used by the hashing function in calculating the hash values.

We can't have a huge hash space because the bigger the hash space, the more wasted space we have in the hash table.

Both of these problems can be traced to a common root:

the hash values are used directly as addresses in the hash table

If we divorced the hash values from the hash table addresses (RRNs), then we might be able to solve our problems:

- if we're not using hash values as addresses, then the hash function would not need to use the hash table size to keep the values within range
- if the hash values don't need to be within the range of the hash table, we can make the hash space as big as we want
- if the hash function is not dependent on the table size, we don't have to keep the table size constant

Q: That's pretty clever, Einstein, but the whole point of hashing is calculating the reference field (address) directly from the primary key value. How can we get O(1) access now?

Trie This On for Size

Trie

- a data structure
- a tree in which branches (from parent to child) are labelled
  - concatenating the labels along the path from the root to a leaf produces data values
- one of the most unfortunate terminological gaffs in Computer Science history

Some *trie*via:

- the name *trie* comes from the word *retrieval* because *tries* are used to retrieve data. `:-b`
- some sources give the pronunciation as *tree* (like in *retrieval*)
- the pronunciation *try* is probably cited more commonly now, to avoid confusion with *tree*

Q: We've already seen examples of *tries*... where?

A:

Trie Example: A Spell Checker

Q: If I asked you to write a spell checker for English words, how would you do it? Would you take a list of known English words, sort it and do a binary search for every word in my document?

A: I hope not.

```
1. start with a tree with one node
2. for each word W in your sorted list of English words
    a. start at the root of the tree
    b. for each character Ch in the current word
        i. if there is a branch labelled Ch from the
           current node, follow that branch
       ii. otherwise, create a branch labelled Ch from the
           current node and follow that branch
    c. create a branch labelled NULL from the current node
```

Trie Example Continued

Let’s create a spell checker for the following dictionary of words:

```
be, bee, been, beer, beet, bit, bite, bottle, bottom, bud,
bust, busy, but
```

Q: How do you spell check the words in a document?

A:

Q: What is the complexity of spell checking an individual word?

A:

Closure

Let’s bring together the techniques we’ve learned to come up with a better kind of hashing:

1. make the hash table less sensitive to insertions and deletions
    - buckets are 1 cluster big
    - buckets are at least half full
2. buckets can be split when overflowing or merged when underflowing
3. divorce the hash values from the hash table addresses
    - use an index to access hash table buckets
    - a trie of hash values can index the buckets
4. maintain direct access (performance should not depend on N)
    - hash values are not hash table addresses, they are paths in a trie whose leaves point to buckets
    - traversing the trie to find a bucket is linear in the length of the hash values, which is independent of the number of keys

First, the Trie

We’re going to use our hash values to build a trie to index the hash table buckets. Our trie will be a binary trie. Each node will have at most two children, a 0 child and a 1 child. Paths through the trie will correspond to the binary (base two) representation of our hash values.

h(k)
20 = 00010100
21 = 00010101
5 = 00000101
29 = 00011101
13 = 00001101
7 = 00000111
27 = 00011011
28 = 00011100

A Trie-Indexed Hash Table

Let’s say our buckets can hold four records each, and we start off with eight records. To allow for additions and deletions, let’s start with our buckets half full.

20 (10100)
28 (11100)

5 (00101)
21 (10101)

13 (01101)
29 (11101)

27 (11011)
7 (00111)

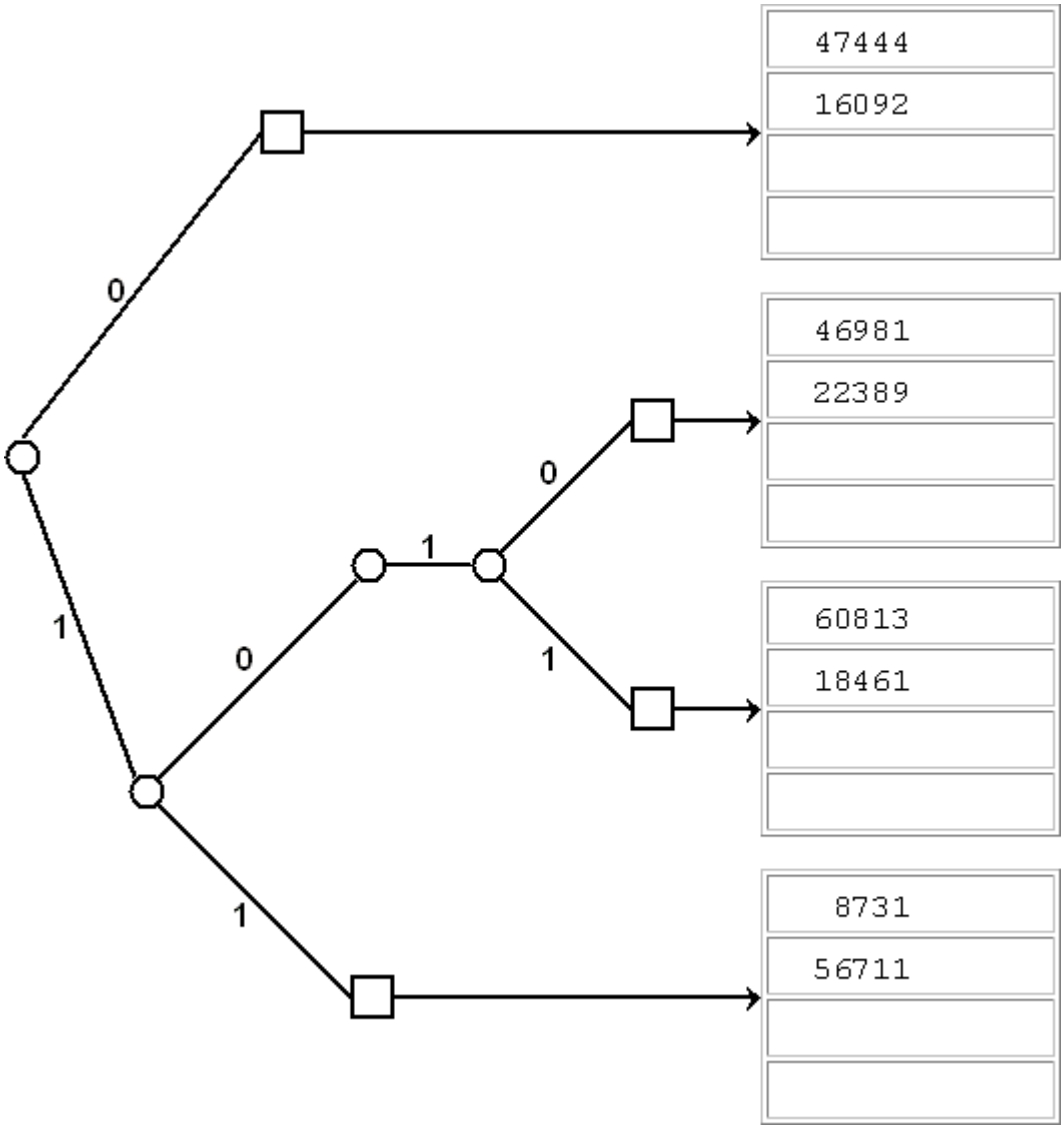
Hash Space vs. Table Size

The previous example shows us that we only need to use as much of the hash value as necessary to identify buckets. To identify our buckets uniquely, we only needed four bits from the hash value (even though our hash values have at least 5 bits of precision).

In general, we can make the hash space as big as we want (using as many bits as we want for the hash values) without affecting the size of the hash table.

h(k)
47444 = 1011100101010100
22389 = 0101011101110101
46981 = 1011011110000101
18461 = 0100100000011101
60813 = 1110110110001101
56711 = 1101110110000111
8731 = 0010001000011011
16092 = 0011111011011100

Hash Space vs. Table Size II



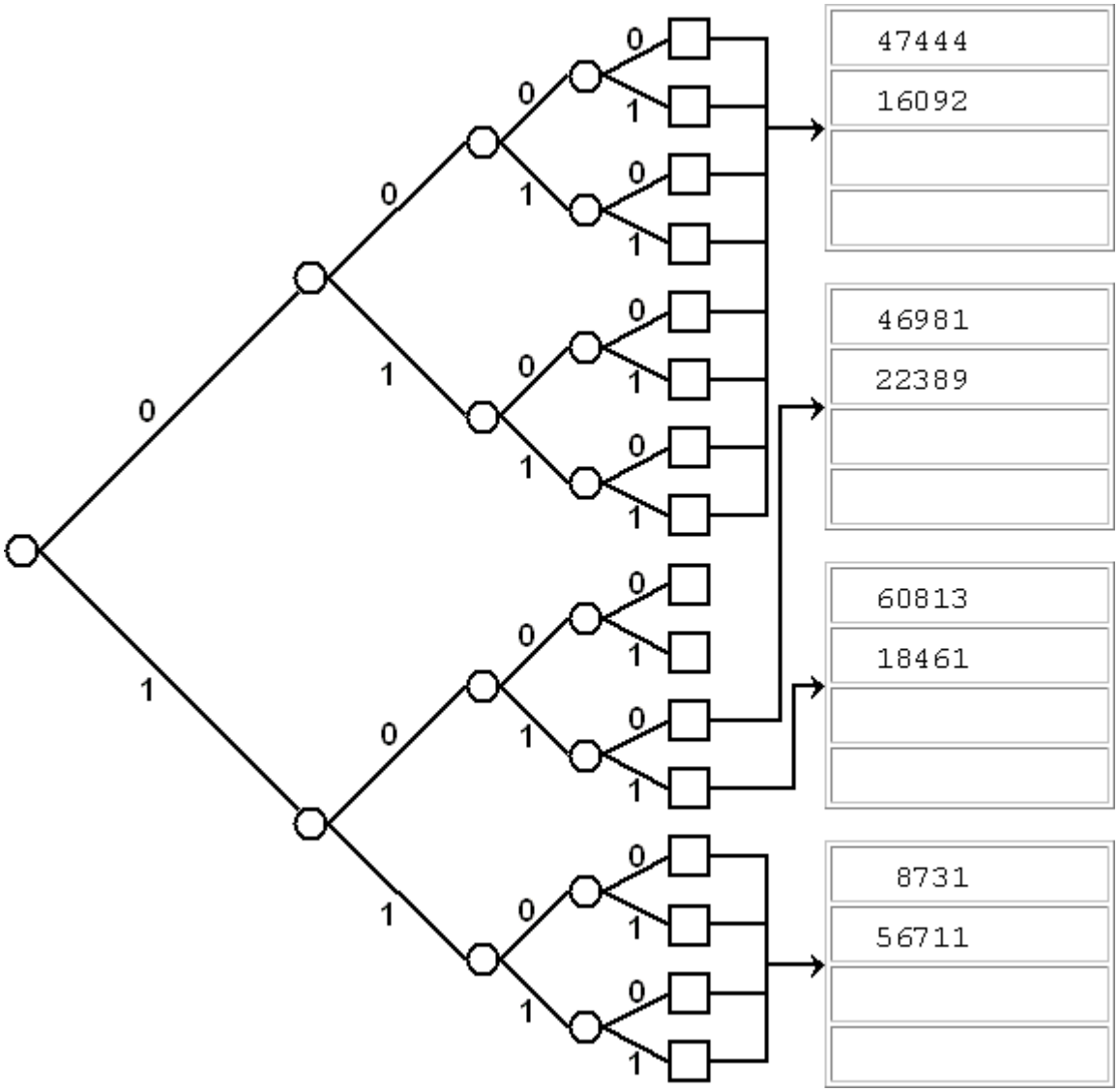
Growing the Trie

In our example we needed only four bits worth of hash value to index our hash table using a trie. That is, the depth of the trie was four (plus pointers to the hash table buckets). But observe that

1. if we have good (random-like) distribution among the values, our trie will be shallower and more complete
2. if we complete the trie, we will be able to add buckets without affecting the depth of the trie
3. given (1), completing the trie shouldn't be *too* big a waste of space

□

Still Growing the Trie



Q: What is the complexity of finding a particular bucket?

A:

Q: What else can we do with a complete trie?

A:

Cutting Down the Trie

Since our trie is a binary tree, the complete trie can be represented efficiently using an array. This form of the trie is called a *directory*.

0000	1
0001	1
0010	1
0011	1
0100	1
0101	1
0110	1
0111	1
1000	0
1001	0
1010	2
1011	3
1100	4
1101	4
1110	4
1111	4

1

52 (110100)
28 (011100)

2

5 (000101)
21 (010101)

3

13 (001101)
61 (111101)

4

27 (011011)
39 (100111)

Q: What is the complexity of finding a particular bucket?

A:

Dynamic Trie-Indexed Hash Tables

We now have a way of accessing buckets in a hash table in constant time (with respect to the number of records/buckets). But we had that with plain old static hashing. It remains to show that our new indexed hash table allows us to add records, delete records, and to grow and shrink the table without affecting our access efficiency.

Let's add some records to our hash table. We'll add records

r_9, r_{10}, r_{11} and r_{12}

having keys

k_9, k_{10}, k_{11} and k_{12}.

When we apply our hashing function to the keys, we get the following values:
H(k_9) = 26 (011010),
H(k_{10}) = 31 (011111),
H(k_{11}) = 53 (110101) and
H(k_{12}) = 6 (000110).

0000	1
0001	1
0010	1
0011	1
0100	1
0101	1
0110	1
0111	1
1000	0
1001	0
1010	2
1011	3
1100	4
1101	4
1110	4
1111	4

1
H(k1) = 52 (110100)
H(k8) = 28 (011100)

2
H(k3) = 5 (000101)
H(k2) = 21 (010101)

3
H(k5) = 13 (001101)
H(k4) = 61 (111101)

4
H(k7) = 27 (011011)
H(k6) = 39 (100111)

Extending the Hash Table (Splitting Buckets)

That was too easy. Let's add some more records: H(k_{13}) = 37 (100101) and H(k_{14}) = 8 (001000)

0000	1
0001	1
0010	1
0011	1
0100	1
0101	1
0110	1
0111	1
1000	0
1001	0
1010	2
1011	3
1100	4
1101	4
1110	4
1111	4

1
```
H(k1)  = 52 (110100)
H(k8)  = 28 (011100)
H(k9)  = 26 (011010)
H(k12) =  6 (000110)
```

2
```
H(k3)  =  5 (000101)
H(k2)  = 21 (010101)
H(k11) = 53 (110101)
```

3
```
H(k5)  = 13 (001101)
H(k4)  = 61 (111101)
```

4
```
H(k7)  = 27 (011011)
H(k6)  = 39 (100111)
H(k10) = 31 (011111)
```

5

Extending the Address Space

That was still pretty easy.
Let's add another record:
H(k_{15}) = 21 (010101)

Q: ?

0000	1
0001	1
0010	1
0011	1
0100	5
0101	5
0110	5
0111	5
1000	0
1001	0
1010	2
1011	3
1100	4
1101	4
1110	4
1111	4

1
```
H(k1)  = 52 (110100)
H(k8)  = 28 (011100)
H(k14) =  8 (001000)
```

2
```
H(k3)  =  5 (000101)
H(k2)  = 21 (010101)
H(k11) = 53 (110101)
H(k13) = 37 (100101)
```

3
```
H(k5)  = 13 (001101)
H(k4)  = 61 (111101)
```

4
```
H(k7)  = 27 (011011)
H(k6)  = 39 (100111)
H(k10) = 31 (011111)
```

5
```
H(k9)  = 26 (011010)
H(k12) =  6 (000110)
```

Hash Space > Address Space

We can't just split bucket 2 because all of the records in bucket 2 are accessed through the same entry in the index. They all have the same last 4 bits.

But because we divorced the hash space from the address space (hash values are not used as addresses in the hash table), we can make the hash space as big as we want. By making the hash space much larger than the address space of the hash table, we can use more bits of the hash values as needed as the hash table grows.

00000	1
00001	1
00010	1
00011	1
00100	1
00101	1
00110	1
00111	1
01000	5
01001	5
01010	5
01011	5
01100	5
01101	5
01110	5
01111	5
10000	0
10001	0
10010	0
10011	0
10100	2
10101	6
10110	3
10111	3
11000	4
11001	4
11010	4
11011	4
11100	4
11101	4
11110	4
11111	4

1
```
H(k1)  = 52 (110100)
H(k8)  = 28 (011100)
H(k14) =  8 (001000)
```

2
```
H(k3)  =  5 (000101)
H(k13) = 37 (100101)
```

3
```
H(k5)  = 13 (001101)
H(k4)  = 61 (111101)
```

4
```
H(k7)  = 27 (011011)
H(k6)  = 39 (100111)
H(k10) = 31 (011111)
```

5
```
H(k9)  = 26 (011010)
H(k12) =  6 (000110)
```

6
```
H(k2)  = 21 (010101)
H(k11) = 53 (110101)
H(k15) = 21 (010101)
```

Deleting Records

Let’s go back to an older verion of the hash table.

Now we’re going to delete a couple of records from our hash table. Why not delete r_2 and r_5.

There’s no reason to have two buckets (2 and 3) with only one record each. We should probably combine them into one bucket.

0000	1
0001	1
0010	1
0011	1
0100	1
0101	1
0110	1
0111	1
1000	0
1001	0
1010	2
1011	3
1100	4
1101	4
1110	4
1111	4

1
H(k1) = 52 (110100)
H(k8) = 28 (011100)

2
H(k3) = 5 (000101)
H(k2) = 21 (010101)

3
H(k5) = 13 (001101)
H(k4) = 61 (111101)

4
H(k7) = 27 (011011)
H(k6) = 39 (100111)

Combining Buckets

0000	1
0001	1
0010	1
0011	1
0100	1
0101	1
0110	1
0111	1
1000	0
1001	0
1010	2
1011	2
1100	4
1101	4
1110	4
1111	4

1

H(k1) = 52 (110100)
H(k8) = 28 (011100)

2

H(k3) = 5 (000101)
H(k4) = 61 (111101)

4

H(k7) = 27 (011011)
H(k6) = 39 (100111)

Q: Does combining buckets cause the hash table to shrink?

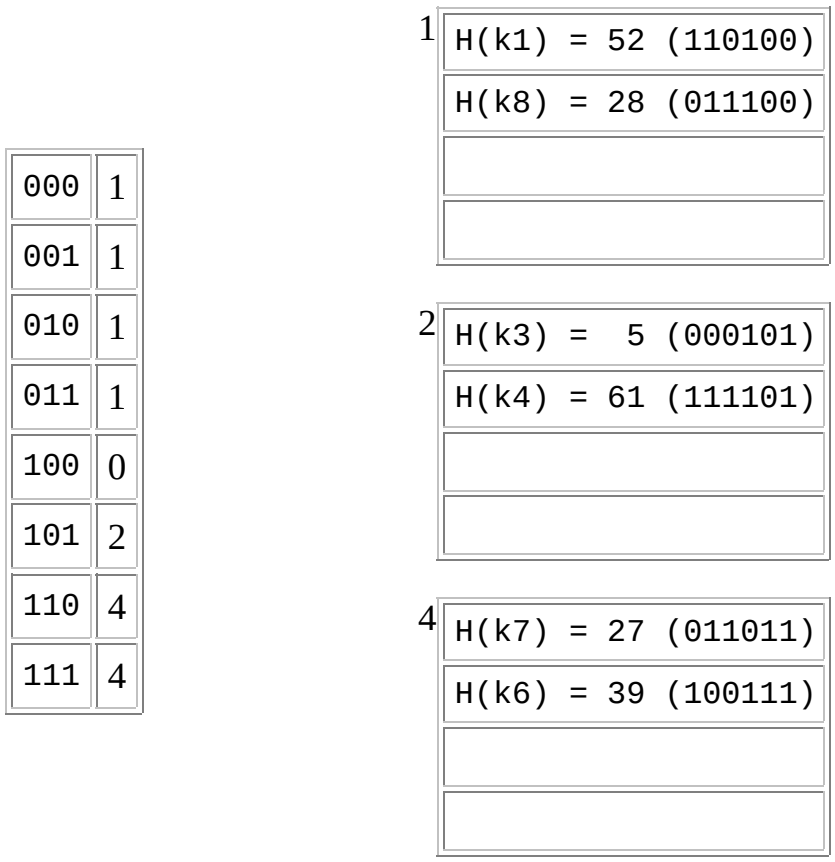
A:

Q: We're keeping 4 bits to index our buckets. Do we really need 4 bits in this index?

A:

Shrinking the Address Space

We no longer need 4 bits to index the hash table, so we can shrink our directory to 3 bits:



A Closer Look at Combining Buckets

The whole deletion/comination/shrinkage process seemed to easy to be real. Is it always that easy?

Q: Buckets 2 and 3 combined nicely. Could we have combined any two buckets?

A:

Q: As soon as we combined buckets 2 and 3, we were able to shrink the address space of the table. Does combining buckets always allow us to shrink the address space?

A:

A Closer Closer Look at Combining Buckets

Being able to shrink the address space of the hash table is a desirable property. When we combine two buckets into one, we want to do it in such a way that the new combined bucket can be indexed with fewer bits than were needed to index the individual buckets before combining.

0000	1
0001	1
0010	1
0011	1
0100	2
0101	2
0110	3
0111	9
1000	4
1001	4
1010	5
1011	6
1100	7
1101	7
1110	8
1111	8

Q: Which pairs of buckets could be combined (assuming each bucket in the pair is less than half full)?

A:

Buddy Buckets

We will restrict combination of buckets to *buddy buckets*.

buddy buckets

two buckets whose indeces differ only in the least significant bit.

We can find buddy buckets by going through the directory sequentially, looking for indeces that differ only in the least significant bit that point to two different buckets.

In the previous example, buckets 3 and 9 are buddies, and buckets 5 and 6 are buddies.

Who’s Your Buddy?

Q: Can a "non-leaf" bucket (a bucket pointed to by more than one directory entry) have a buddy?

A:

Q: What happens if we delete all of the records in a bucket that has no buddy?

A:

Q: What does it mean if no buckets have buddies?

A:

Q: When should we look for buddy buckets?

A: