Midterm grades

- Grading policy:
  - Programming assignments: 25%
  - Written assignments: 15%
  - Midterm exam: 20%
  - Final exam: 30%
- Weighting for midterm grade:
  - W1+W2: 17%
  - P1, P2: 39%
  - Midterm: 44%
- Midterm grade is only 44% of final grade
  - Still have W3, P3, W4, P4, and final.

Grade Statistics

- Midterm grade distribution:
  - A: 28% B: 25% C: 29% below:17% (includes withdrawls)

- Individual score statistics:

	W1	P1	W2	P2	MT
Total	**48**	**100**	**37**	**100**	**36**
90th %	40	100	37	100	49
75th %	37	100	35	98	47
50th %	34	90	34	88	40
25th %	27	75	33	76	33

percentile

midterm statistics

Column	midterm ()
Points Possible	54
Description	

Statistics	
Count	68
Minimum Value	10.00
Maximum Value	54.00
Range	44.00
Average	38.75
Median	40.00
Standard Deviation	9.58
Variance	91.86

Status Distribution	
Null	18
In Progress	0
Needs Grading	0
Exempt	0

Grade Distribution	
greater than 100	0
90 - 100	9
80 - 89	19
70 - 79	12
60 - 69	11
50 - 59	11
40 - 49	1
30 - 39	3
20 - 29	1
10 - 19	1
0 - 9	0

Hashing: Implementation Issues

EECS 233

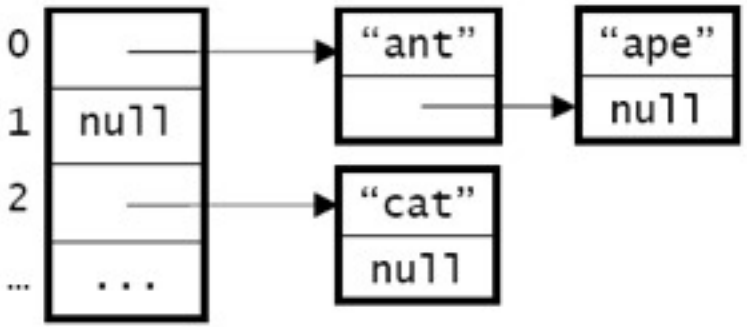
Recap from Previous Lecture

- Hash functions desiderata
- Handling collisions with open addressing
  - Linear probing
  - Quadratic probing
  - Double hashing
  - Need to distinguish between “removed” and “empty” positions

Handling Collisions with Chaining

- If multiple items are assigned the same hash code, we "chain" them together. Each position in the hash table serves as a *bucket* that is able to store multiple data items.

- Two implementations:
  - each bucket is itself an array (or points to an array)
    - ✓ disadvantages: (1) large buckets can waste memory, (2) a bucket may become full; *overflow* occurs when we try to add an item to a full bucket
  - a linked list
    - ✓ disadvantage: memory overhead for the references

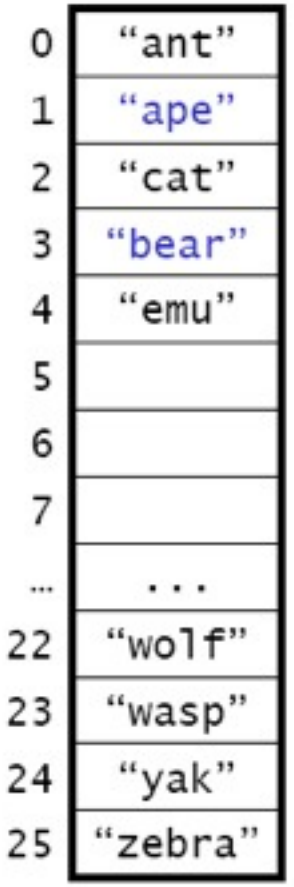


Linear Probing for Open Addressing

- Probe sequence: h(key), h(key) + 1, h(key) + 2, …, wrapping around as necessary.
  - Example:
    - ✓ “ape” (h = 0) would be placed in position 1, because position 0 is already full.
    - ✓ “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
    - ✓ where would “whale” end up?

- Advantage: if there is an open position, linear probing will eventually find it.

- Disadvantage: “clusters” of occupied positions develop
  - Increases the lengths of subsequent probes.
  - As load factor increases, both search and insert times look increasingly linear!



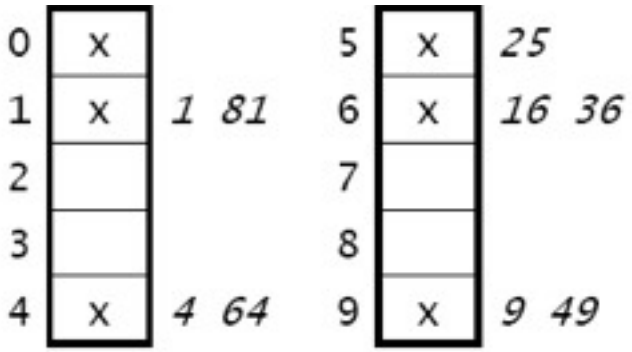
Quadratic Probing for Open Addressing

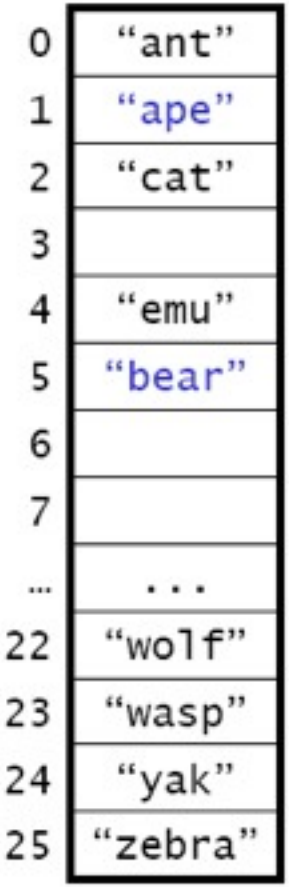
- Probe sequence: h(key), h(key) + 1, h(key) + 4, h(key) + 9, …, wrapping around as necessary.
  - the offsets are perfect squares: h + 1^2, h + 2^2, h + 3^2, …
  - Example:
    - ✓ “ape” (h = 0): try 0, 0 + 1 – open!
    - ✓ “bear” (h = 1): try 1, 1 + 1, 1 + 4 – open!
    - ✓ “whale”?

- Advantage: reduces clustering

- Disadvantage: it may fail to find an existing open position.

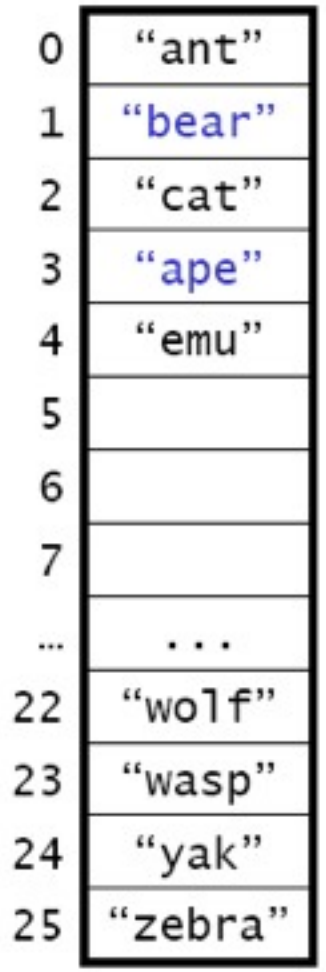
Example:
- table size = 10
- x = occupied
- trying to insert a key with h(key) = 0



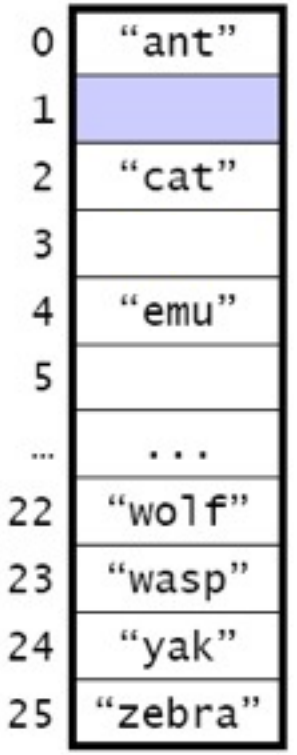


Double Hashing for Open Addressing

- Use two hash functions:
  - h1 computes the hash code
  - h2 computes the increment for probing
  - Probe sequence: h1, h1 + h2, h1 + 2*h2, …
- Example:
  - h1 = our previous function h
  - h2 = number of characters in the string
  - “ape” (h1 = 0, h2 = 3): try 0, 0 + 3 – open!
  - “bear” (h1 = 1, h2 = 4): try 1 – open!
  - “whale”?
- Combines the good features of linear and quadratic probing:
  - reduces clustering
  - Theorem: will find an open position if there is one, provided the table size is a prime number.
  - Disadvantage: the need for two hash functions

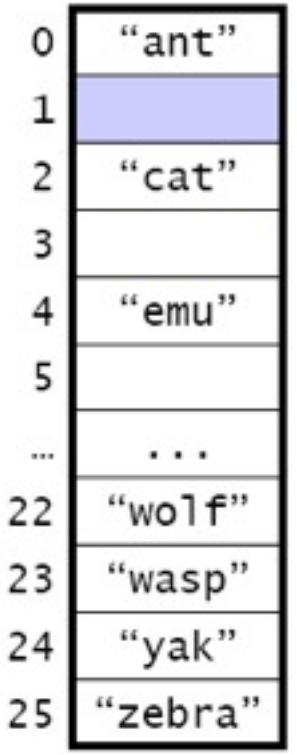


Removing Items with Open Addressing



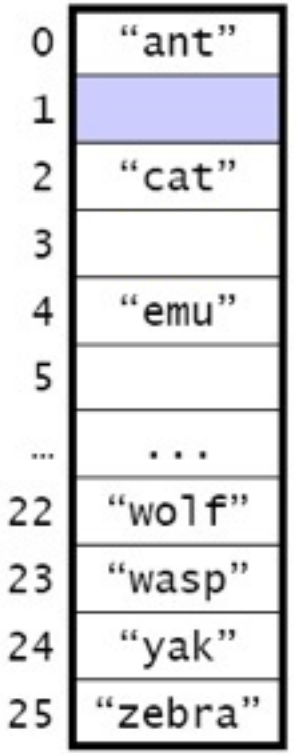
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing



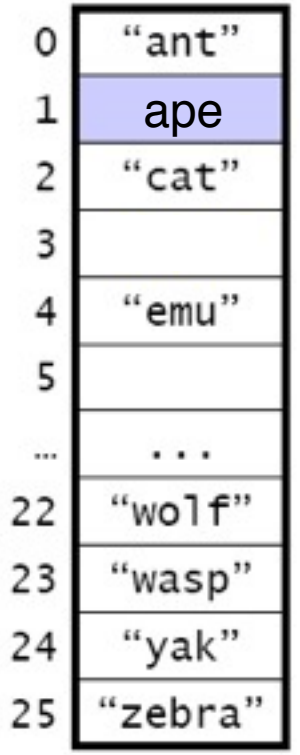
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!



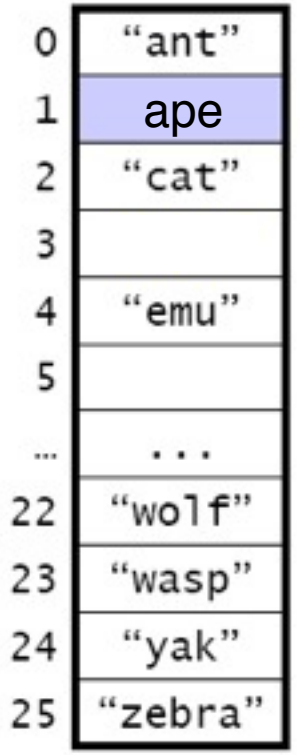
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!



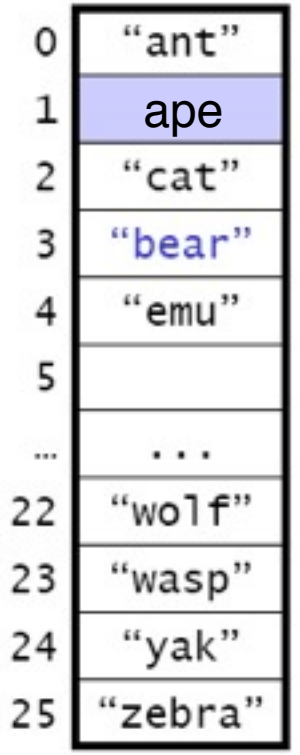
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!



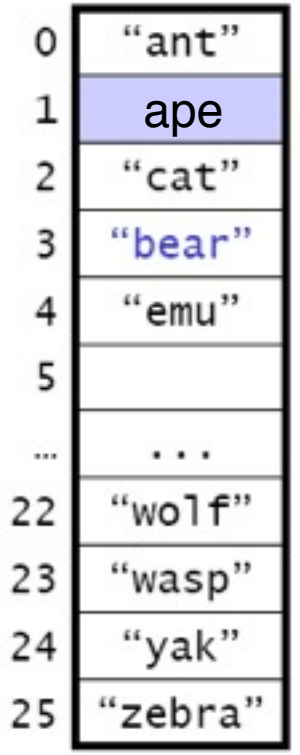
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!



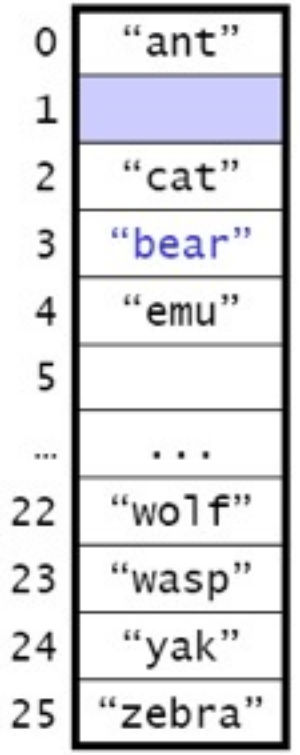
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove “ape”



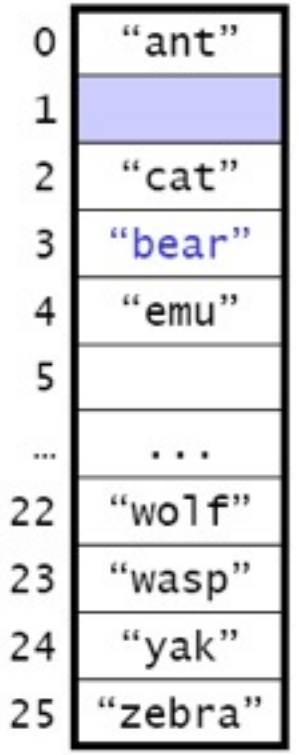
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove “ape”



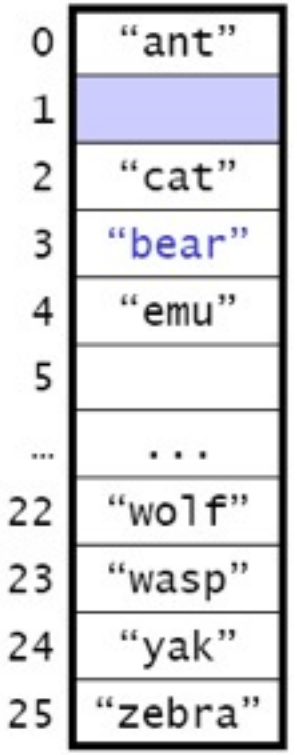
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove “ape”
  - search for “ape”: try 0, 0 + 1 – no item



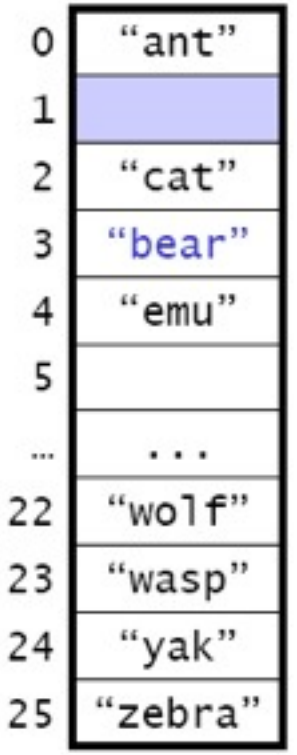
Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove “ape”
  - search for “ape”: try 0, 0 + 1 – no item
  - search for “bear”: try 1 – no item,



Removing Items with Open Addressing

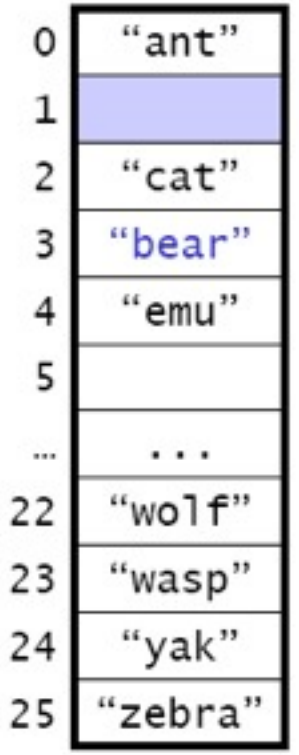
- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove “ape”
  - search for “ape”: try 0, 0 + 1 – no item
  - search for “bear”: try 1 – no item,
  - but “bear” is further down in the table!



Removing Items with Open Addressing

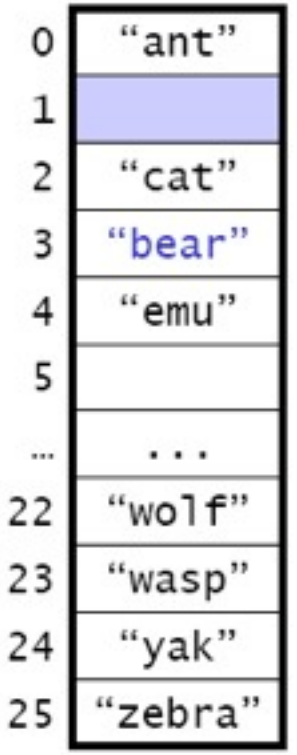
- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove “ape”
  - search for “ape”: try 0, 0 + 1 – no item
  - search for “bear”: try 1 – no item,
  - but “bear” is further down in the table!

  - Cannot tell if it is not in the table



Removing Items with Open Addressing

- Consider the following scenario:
  - using linear probing
  - insert “ape” (h = 0): try 0, 0 + 1 – open!
  - insert “bear” (h = 1): try 1, 1 + 1, 1 + 2 – open!
  - remove “ape”
  - search for “ape”: try 0, 0 + 1 – no item
  - search for “bear”: try 1 – no item,
  - but “bear” is further down in the table!
  - Cannot tell if it is not in the table



- Any difference with quadratic probing or double hashing?

Implementation of Hash Tables

- A simple hash table with open addressing.

```
public class HashTable {
        private class Entry {
                private String key;
                private boolean removed;
                private String etymology;
                …
        }
        private Entry[] table;
        private int tableSize;
        ...
}
```

- for an empty position, table[i] will equal null
- for a removed position, table[i] will refer to an Entry object whose *removed* field equals true
- for an occupied position, table[i] will refer to an Entry object whose *removed* field equals false

Constructors

- Initializing the hash table

```
public HashTable(int size) {
        table = new Entry[size];
        tableSize = size;
}
```

- Initializing an entry (before insertion)

```
private Entry(String key, String etymology) {
        this.key = key;
        this.etymology = etymology;
        removed = false;
        …
}
```

Finding An Open Position

- Using double hashing

```
private int probe(String key) {
        int i = h1(key); // first hash function
        int j = h2(key); // second hash function

        // keep probing while the current position is occupied (non-empty and non-
           removed)
        while ( ?                                                    )

              ?

        return i;
}
```

- Does it always terminate?

Finding An Open Position

- Using double hashing

```
private int probe(String key) {
        int i = h1(key); // first hash function
        int j = h2(key); // second hash function

        // keep probing while the current position is occupied (non-empty and non-
            removed)
        while (   table[i] != null && table[i].removed==false   )

                ?

        return i;
}
```

- Does it always terminate?

Finding An Open Position

- Using double hashing

```
private int probe(String key) {
        int i = h1(key); // first hash function
        int j = h2(key); // second hash function

        // keep probing while the current position is occupied (non-empty and non-
            removed)
        while (   table[i] != null && table[i].removed==false   )

                    i = (i + j) % tableSize;

        return i;
}
```

- Does it always terminate?

Finding An Open Position: Infinite Loop Protection

```
private int probe(String key) {
        int i = h1(key); // first hash function
        int j = h2(key); // second hash function
        int iterations = 0;

        // keep probing until we get an empty or removed position
        while (table[i] != null && table[i].removed==false) {
                i = (i + j) % tableSize;
                iterations++;
                if (iterations > tableSize) return -1;
        }

        return i;
}
```

Finding the Position of A Key

- Different from probe()
  - Returns position of the key, not an available position.

```
private int findKey(String key) {
        int i = h1(key); // first hash function
        int j = h2(key); // second hash function
        int iterations = 0;

        // keep probing while the entry is not empty

        while ( ?                                          ) {
                // return if key is found, otherwise continue
                ?


                i = (i + j) % tableSize;
                iterations++;
                if (iterations > tableSize) return -1;
        }

        return -1;
}
```

Finding the Position of A Key

- Different from probe()
  - Returns position of the key, not an available position.

```
private int findKey(String key) {
        int i = h1(key); // first hash function
        int j = h2(key); // second hash function
        int iterations = 0;

        // keep probing while the entry is not empty

        while (   table[i] != null                          ) {
                // return if key is found, otherwise continue
                ?


                i = (i + j) % tableSize;
                iterations++;
                if (iterations > tableSize) return -1;
        }

        return -1;
}
```

Finding the Position of A Key

- Different from probe()
  - Returns position of the key, not an available position.

```
private int findKey(String key) {
        int i = h1(key); // first hash function
        int j = h2(key); // second hash function
        int iterations = 0;

        // keep probing while the entry is not empty

        while (   table[i] != null                              ) {
              // return if key is found, otherwise continue
             if (table[i].key == key)
                 return i;

              i = (i + j) % tableSize;
              iterations++;
              if (iterations > tableSize) return -1;
        }

        return -1;
}
```

Finding the Position of A Key

- Different from probe()
  - Returns position of the key, not an available position.

```
private int findKey(String key) {
        int i = h1(key); // first hash function
        int j = h2(key); // second hash function
        int iterations = 0;

        // keep probing while the entry is not empty

        while (   table[i] != null                          ) {
                // return if key is found, otherwise continue
                ?


                i = (i + j) % tableSize;
                iterations++;
                if (iterations > tableSize) return -1;
        }

        return -1;
}
```

Finding the Position of A Key

- Different from probe()
  - Returns position of the key, not an available position.

```
private int findKey(String key) {
        int i = h1(key); // first hash function
        int j = h2(key); // second hash function
        int iterations = 0;

        // keep probing while the entry is not empty

        while (   table[i] != null                              ) {
              // return if key is found, otherwise continue
             if (table[i].removed==false && table[i].key.equals(key))
                 return i;

              i = (i + j) % tableSize;
              iterations++;
              if (iterations > tableSize) return -1;
        }

        return -1;
}
```

Search() Method

- Search for the entry with the key, and return the Etymology (in other implementation, we may return other fields instead)

```
public String search(String key) {
        int i = findKey(key);
        if (i == -1)
                return null;
        else
                return table[i].etymology ;
}
```

- It calls the helper method findKey() to locate the position of the key.

Remove() Method

- Search the hash table and delete the key if found

```
public void remove(String key) {
        int i = findKey(key);
        if (i == -1)
                return;
        table[i].removed = true;
}
```

- It also uses findKey().

Insertion

- We begin by probing for the key to find an empty position.
- Several cases:
  - Encountered a removed position while probing
    - ✓ put the (key, value) pair in the removed position that we encountered while probing for the key.
  - Reached an empty position
    - ✓ put the (key, value) pair in the empty position
  - No removed position or empty position encountered
    - ✓ Overflow: throw an exception

Insert() Method

- If no empty or removed position is available, report error; otherwise, insert the new entry

```
public void insert(String key, String etymology) {
        int i = probe(key);
        if (i == -1)
                throw exception;
        else
                ?
}
```

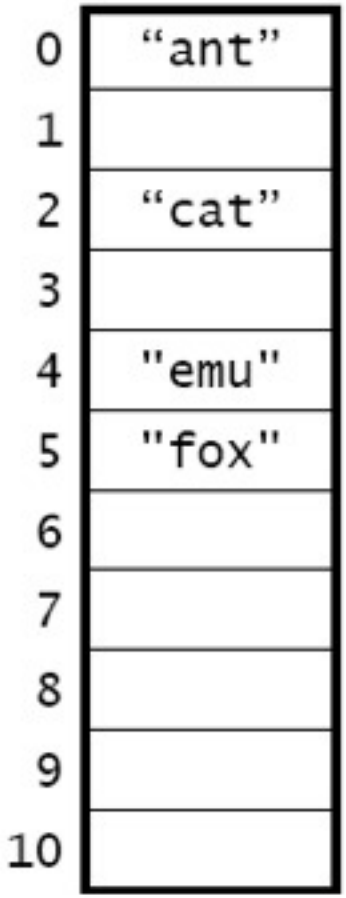
Insert() Method

- If no empty or removed position is available, report error; otherwise, insert the new entry

```
public void insert(String key, String etymology) {
        int i = probe(key);
        if (i == -1)
              throw exception;
        else
              table[i].key = key;
              table[i].etymology = etymology;
}
```

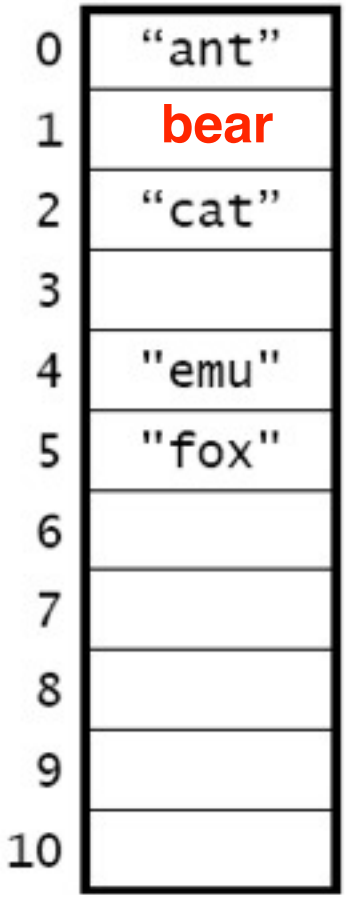
Tracing the Methods

- Start with the hash table at right with:
  - double hashing
  - our earlier hash functions h1 and h2
    - ✓ h1: <first character> -'a'
    - ✓ h2: length
- Perform the following operations:
  - insert "bear"
  - insert "bison"
  - insert "cow"
  - remove "emu"
  - search "eel"
  - insert "bee"



Tracing the Methods

- Start with the hash table at right with:
  - double hashing
  - our earlier hash functions h1 and h2
    - ✓ h1: <first character> -'a'
    - ✓ h2: length
- Perform the following operations:
  - insert "bear"
  - insert "bison"
  - insert "cow"
  - remove "emu"
  - search "eel"
  - insert "bee"



Tracing the Methods

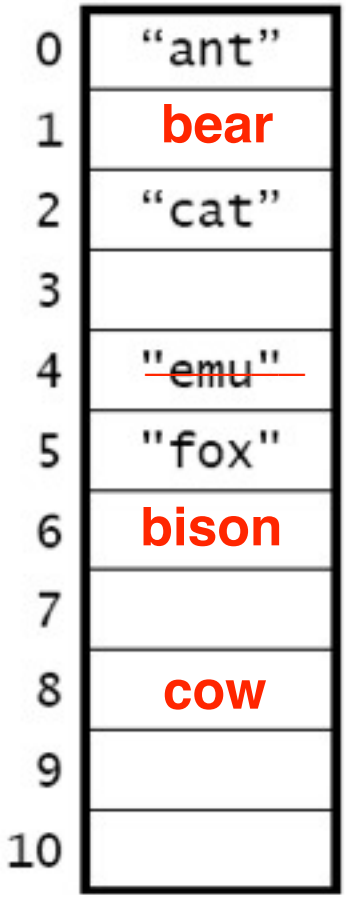
- Start with the hash table at right with:
  - double hashing
  - our earlier hash functions h1 and h2
    - ✓ h1: <first character> -'a'
    - ✓ h2: length
- Perform the following operations:
  - insert "bear"
  - insert "bison"
  - insert "cow"
  - remove "emu"
  - search "eel"
  - insert "bee"

Tracing the Methods

- Start with the hash table at right with:
  - double hashing
  - our earlier hash functions h1 and h2
    - ✓ h1: <first character> -'a'
    - ✓ h2: length
- Perform the following operations:
  - insert "bear"
  - insert "bison"
  - insert "cow"
  - remove "emu"
  - search "eel"
  - insert "bee"

Tracing the Methods

- Start with the hash table at right with:
  - double hashing
  - our earlier hash functions h1 and h2
    - ✓ h1: <first character> -'a'
    - ✓ h2: length
- Perform the following operations:
  - insert "bear"
  - insert "bison"
  - insert "cow"
  - remove "emu"
  - search "eel"
  - insert "bee"



Tracing the Methods

- Start with the hash table at right with:
  - double hashing
  - our earlier hash functions h1 and h2
    - ✓ h1: <first character> -'a'
    - ✓ h2: length
- Perform the following operations:
  - insert "bear"
  - insert "bison"
  - insert "cow"
  - remove "emu"
  - search "eel"
  - insert "bee"

Issues and Questions

- As the hash table is used, more positions will be marked removed
  - e.g., 90% of the entries are marked as *removed*, and 10% are marked as *occupied*
  - does it affect the running time of successful/unsuccessful search?
- If the hash table gets full or almost full, this slows down the insertion
  - What is the expected running time of insertion (in number of iterations in probe()), if the hash table is 90% full?

Hash Table Performance May Degrade

- When many entries are “removed”, search must continue
  - If the searched key is in the table, it is likely it will be found after few iterations
  - However, if the key is not in the table at all, search continues until reaching an “empty” entry, or after N iterations

- When most entries are occupied, insertion method may not be able to probe an open position quickly
  - the *load factor* of a hash table is the ratio of the number of keys to the table size.
  - The expected number of iterations for load factor λ
    - ✓ Obviously grows with λ

Rehashing

- What to do in the first case?
  - Here the load factor is not high at all, but that many entries are not utilized (marked as “removed”).
    - ✓ the load factor can be easily tracked.
  - We can reorganize the hash table by rehashing the entries: Create another hash table, and move every entry to the new table
- What to do in the second case?
  - Here the problem is high load factor
  - We should expand the hash table: create another hash table of doubled size, and move the entries to the new table
- What is the running time of rehashing? Does it affect the overall running of hash table operations?
  - May not happen very often, e.g., once every O(N) insertions
  - But when it happens, may take some running time

Rehash() Method (Expanding)

- Let us still consider the example hash table:

```
public class HashTable {
        private class Entry {
                private String key;
                private String etymology;
                private boolean removed;
        }
        private Entry[ ] table;
        private int tableSize;
        ...
        public void rehash( ) {
                int oldSize = tableSize;
                Entry[ ] oldTable = table;

                tableSize = nextPrime(2 * oldSize);
                table = new Entry[tableSize];
                for(i = 0; i < oldSize; i++)
                        if ( ? )
                                ?
        }
}
```

- We define rehash() as public so the user can decide when to rehash

Rehash() Method (Expanding)

- Let us still consider the example hash table:

```
public class HashTable {
        private class Entry {
                private String key;
                private String etymology;
                private boolean removed;
        }
        private Entry[ ] table;
        private int tableSize;
        ...
        public void rehash( ) {
                int oldSize = tableSize;
                Entry[ ] oldTable = table;

                tableSize = nextPrime(2 * oldSize);
                table = new Entry[tableSize];
                for(i = 0; i < oldSize; i++)
                        if (oldTable[i] != null && oldTable[i].removed == false)
                            insert( oldTable[i].key, oldTable[i].etymology );
        }
}
```

- We define rehash() as public so the user can decide when to rehash

Efficiency of Hash Tables

- In the best case, search and insertion are *O*(1).
- In the worst case, search and insertion are linear.
  - open addressing: *O*(m), where m = the size of the hash table
  - separate chaining: *O*(n), where n = the number of keys
- With a good choice of hash function and table size, the time complexity is generally better than *O*(log n) and approaches *O*(1).
- As the load factor increases, performance tends to decrease.
  - Linear probing: try to keep the load factor < ½
  - separate chaining: try to keep the load factor < 1
- Time-space tradeoff: bigger tables tend to have better performance, but they use up more memory.

Limitations of Hash Tables

- It can be hard to come up with a good hash function for a particular data set.
  - The choice of hash functions may depend on the characteristics of the data set.
  - Complexity of hash functions
- The items are not ordered by key. As a result, we can't easily
  - Print the contents in sorted order
  - Solve the selection problem - get the k-th largest item
- We *can* do all of these things with many tree structures.

Implementation of Hash Function

Hash Functions Desiderata Revisited

- Example: String as key (e.g., Webster)
  - keys = character strings composed of lower-case letters
  - hash function:
    - ✓ h(key) = (the byte sum of all characters) mode table-size
    - ✓ example: h("cat") = ('a' + 'c' + 't') mod 100000
    - ✓ But: assume words are mostly up to 20 character-long
      - Max sum is 127*20 = 2540; any larger table is of no use!
    - ✓ But: permutations are not distinguished
      - h("cat") = h("act")

- Requirements for good hash functions:
  - Full table size utilization
  - Even ("uniform") key mapping throughout the table
    - ✓ Utilizing known key distribution in the objects
    - ✓ Making hash function "random" - the distribution of keys is independent of distribution of indexes to which they map

Hash Functions Desiderata Revisited

- Example: String as key (e.g., Webster)
  - keys = character strings composed of lower-case letters
  - hash function:
    - ✓ h(key) = (the byte sum of all characters) mode table-size
    - ✓ example: h("cat") = ('a' + 'c' + 't') mod 100000
    - ✓ But: assume words are mostly up to 20 character-long
      - Max sum is 127*20 = 2540; any larger table is of no use!
    - ✓ But: permutations are not distinguished
      - h("cat") = h("act")

- Requirements for good hash functions:
  - Full table size utilization
  - Even ("uniform") key mapping throughout the table
    - ✓ Utilizing known key distribution in the objects
    - ✓ Making hash function "random" - the distribution of keys is independent of distribution of indexes to which they map

    Using the entire key to compute the hash value

Hash Functions Desiderata Revisited

- Example: String as key (e.g., Webster)
  - keys = character strings composed of lower-case letters
  - hash function:
    - ✓ h(key) = (the byte sum of all characters) mode table-size
    - ✓ example: h("cat") = ('a' + 'c' + 't') mod 100000
    - ✓ But: assume words are mostly up to 20 character-long
      - Max sum is 127*20 = 2540; any larger table is of no use!
    - ✓ But: permutations are not distinguished
      - h("cat") = h("act")

- Requirements for good hash functions:
  - Full table size utilization
  - Even ("uniform") key mapping throughout the table
    - ✓ Utilizing known key distribution in the objects
    - ✓ Making hash function "random" - the distribution of keys is independent of distribution of indexes to which they map

    Using the entire key to compute the hash value

  Must be efficient to compute!

Hash Functions for Strings

- Bad example: the sum of the character codes
  - Max sum is only 127 * max length

- A better example: a weighted sum of the character codes
  - $h_b = (a_0 b^{n-1} + a_1 b^{n-2} + \ldots + a_{n-2} b^1 + a_{n-1}) \bmod (\text{tableSize})$
    - ✓ a_i is encoding of the i-th character,
    - ✓ b is a constant
  - All characters contribute
  - Contribution of a character depends on its position

- Examples for b = 31:
  - $h_b(\text{“table”}) = 116*31^4 + 97*31^3 + 98*31^2 + 108*31 + 101 = 110115790$
  - $h_b(\text{“eat”}) = 101*31^2 + 97*31 + 116 = 100184$
  - $h_b(\text{“tea”}) = 116*31^2 + 101*31 + 97 = 114704$

- Java uses this hash function with b = 31 in the hashCode() method of the String class.

Hash Functions for Strings

- Bad example: the sum of the character codes
  - Max sum is only 127 * max length

- A better example: a weighted sum of the character codes
  - $h_b = (a_0 b^{n-1} + a_1 b^{n-2} + \ldots + a_{n-2} b^1 + a_{n-1}) \bmod (\text{tableSize})$
    - ✓ a_i is encoding of the i-th character,
    - ✓ b is a constant
  - All characters contribute
  - Contribution of a character depends on its position

- Examples for b = 31:
  - $h_b(\text{“table”}) = 116*31^4 + 97*31^3 + 98*31^2 + 108*31 + 101 = 110115790$
  - $h_b(\text{“eat”}) = 101*31^2 + 97*31 + 116 = 100184$
  - $h_b(\text{“tea”}) = 116*31^2 + 101*31 + 97 = 114704$

- Java uses this hash function with b = 31 in the hashCode() method of the String class.

What happens if we use a table size of 31?

Hash Functions for Strings

Hash Functions for Strings

- How to calculate h_b(supercalifragilisticexpialidocious)?
  - ‘s’ b^{33} + ‘u’ b^{32} + … + ‘s’.
  - A lot of multiplications!

Hash Functions for Strings

- How to calculate h_b(supercalifragilisticexpialidocious)?
  - ‘s’ b^{33} + ‘u’ b^{32} + … + ‘s’.
  - A lot of multiplications!
- Better way: Horner’s method
  - $a_0 b^{n-1} + a_1 b^{n-2} + \ldots + a_{n-2} b^1 + a_{n-1} = (\ldots((a_0 b + a_1)b + a_2)b + \ldots + a_{n-2})b + a_{n-1}$
  - example: $101*31^2 + 97*31 + 116 = ((101*31 + 97)*31 + 116$

Hash Functions for Strings

- How to calculate h_b(supercalifragilisticexpialidocious)?
  - ‘s’ b^{33} + ‘u’ b^{32} + … + ‘s’.
  - A lot of multiplications!

- Better way: Horner’s method
  - $a_0 b^{n-1} + a_1 b^{n-2} + \ldots + a_{n-2} b^1 + a_{n-1} = (\ldots((a_0 b + a_1) b + a_2) b + \ldots + a_{n-2}) b + a_{n-1}$
  - example: $101*31^2 + 97*31 + 116 = ((101*31 + 97)*31 + 116$

```
int hash = 0;
for (int i = 0; i < s.length(); i++)
        hash = hash * b + s.charAt(i);
```

Hash Functions for Strings

- How to calculate h_b(supercalifragilisticexpialidocious)?
  - ‘s’ b^{33} + ‘u’ b^{32} + … + ‘s’.
  - A lot of multiplications!

- Better way: Horner’s method
  - $a_0 b^{n-1} + a_1 b^{n-2} + \ldots + a_{n-2} b^1 + a_{n-1} = (\ldots((a_0 b + a_1)b + a_2)b + \ldots + a_{n-2})b + a_{n-1}$
  - example: $101*31^2 + 97*31 + 116 = ((101*31 + 97)*31 + 116$

```
int hash = 0;
for (int i = 0; i < s.length(); i++)
    hash = hash * b + s.charAt(i);
```

- For b=31, we have more efficient ways to “multiply by 31”
  - n << i shifts the binary representation of n left by i places
  - n << i = n * 2^i
  - n * 31 = n * (32 – 1) = (n * 32) – n = (n << 5) – n
  - example: n = 100 = 0000000001100100_2
    - ✓ 100 << 5 = 0000110010000000_2 = 3200
    - ✓ 100 * 31 = 3200 – 100 = 3100

External Hashing?

External Hashing?

- Recall search trees (binary trees/AVL trees) => B-trees
  - Each tree node holds multiple key ranges

External Hashing?

- Recall search trees (binary trees/AVL trees) => B-trees
  - Each tree node holds multiple key ranges
- What if the hash table is too large to be in main memory?

External Hashing?

- Recall search trees (binary trees/AVL trees) => B-trees
  - Each tree node holds multiple key ranges
- What if the hash table is too large to be in main memory?

External Hashing?

- Recall search trees (binary trees/AVL trees) => B-trees
  - Each tree node holds multiple key ranges
- What if the hash table is too large to be in main memory?
- How about just using disk addresses instead of memory addresses?
  - Hash table = hash file
  - Slot position = offset in the hash file
  - References in separate chaining = file addresses

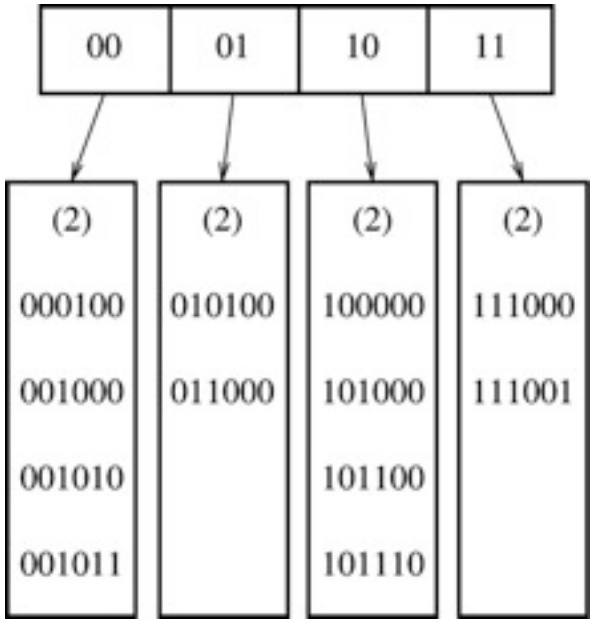
External Hashing?

- Recall search trees (binary trees/AVL trees) => B-trees
  - Each tree node holds multiple key ranges
- What if the hash table is too large to be in main memory?
- How about just using disk addresses instead of memory addresses?
  - Hash table = hash file
  - Slot position = offset in the hash file
  - References in separate chaining = file addresses
- Issues:
  - Worst time access - O(N) random disk accesses
  - Hash table reorganization - O(N) random disk accesses

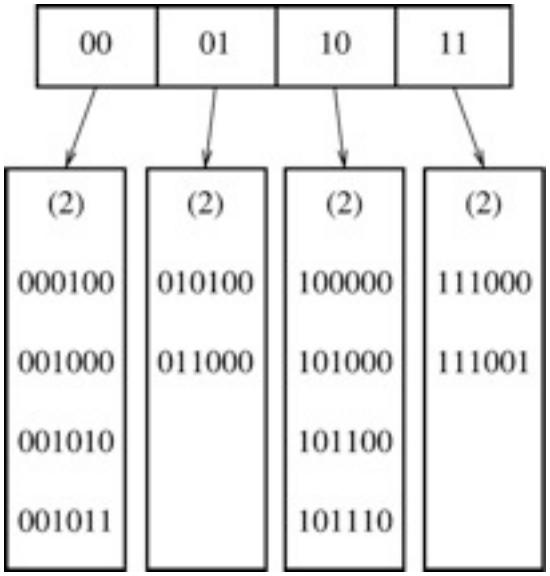
Extensible Hashing

Fagin, R.; Nievergelt, J. & Pippenger, N. et al. (September, 1979), "Extendible Hashing - A Fast Access Method for Dynamic Files", *ACM Trans. on Database Systems* 4(3)

- Choose a hash function with a large range.
- Use *i* "senior" bits as an offset into a table of bucket addresses.
- Store entries in the buckets
  - Entries in the same bucket share common prefix in hash values
- Search for key K
  - Compute h(K)
  - Take i senior bits
  - Look up the directory entry using i-bit offset
  - Follow the bucket pointer
- Insert key K with h(K) = 100100

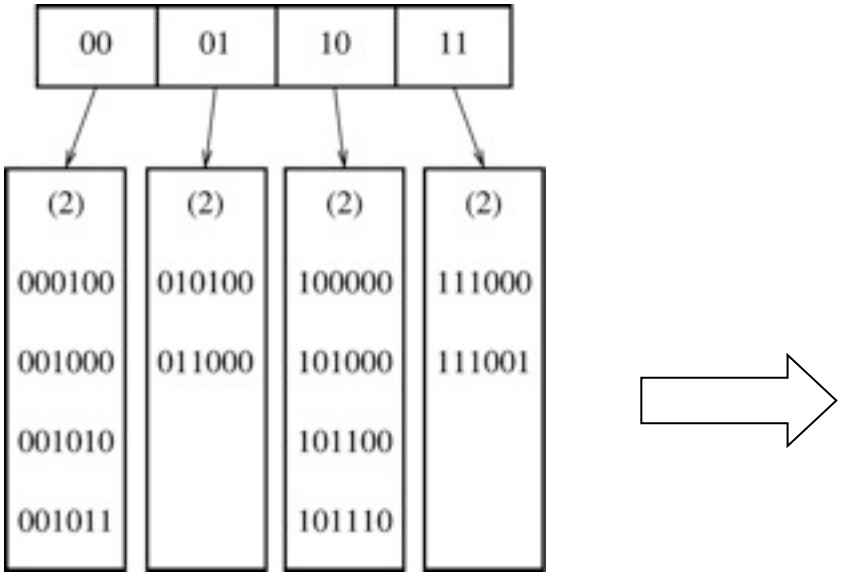


Insertions into Extendible Hash (Case 1)



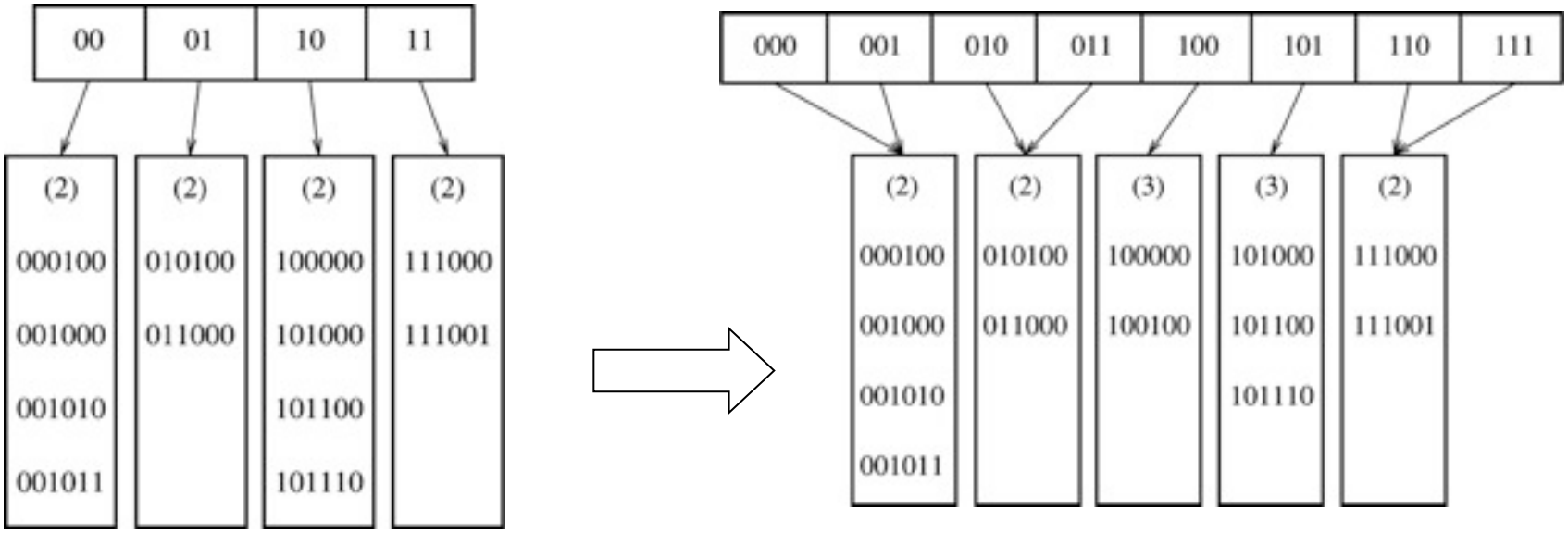
- Insertion of 100100: leaf split and directory split
  - Only involves reorganization of directory (main memory) and one bucket (localized damage)
- Bucket's common prefix can be shorter than *i*

Insertions into Extendible Hash (Case 1)



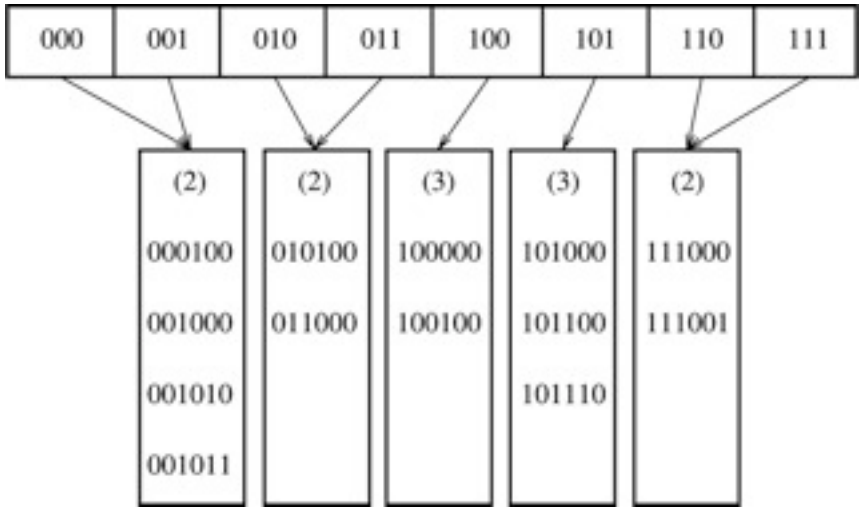
- Insertion of 100100: leaf split and directory split
  - Only involves reorganization of directory (main memory) and one bucket (localized damage)
- Bucket's common prefix can be shorter than *i*

Insertions into Extendible Hash (Case 1)

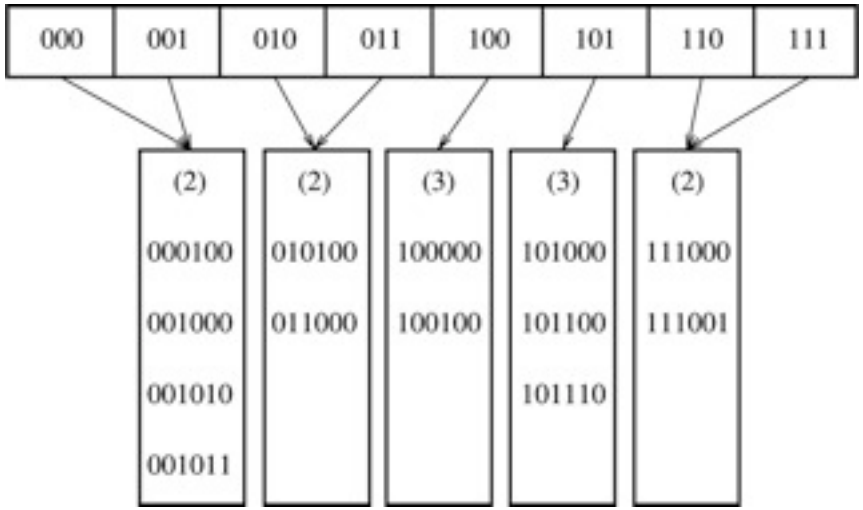


- Insertion of 100100: leaf split and directory split
  - Only involves reorganization of directory (main memory) and one bucket (localized damage)
- Bucket's common prefix can be shorter than *i*

Insertions into Extendible Hash (Case 2)

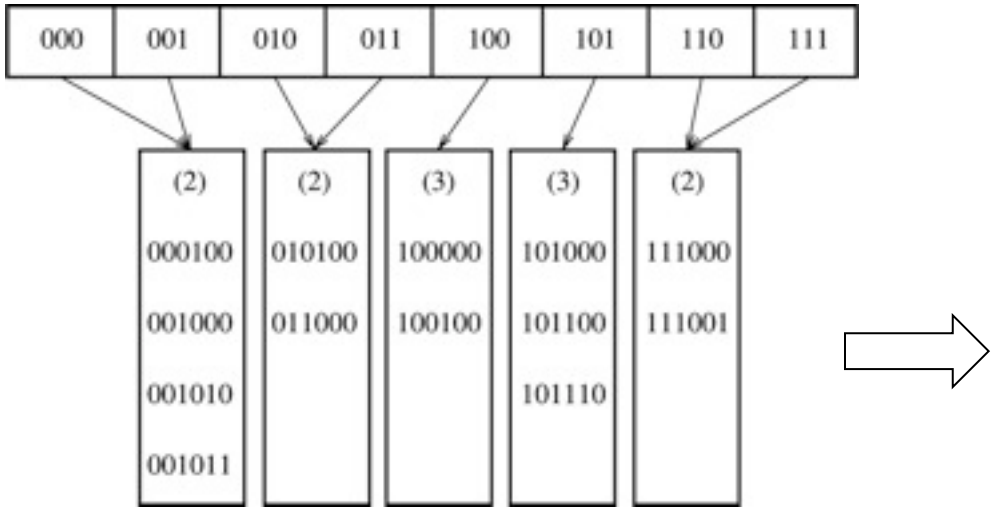


Insertions into Extendible Hash (Case 2)



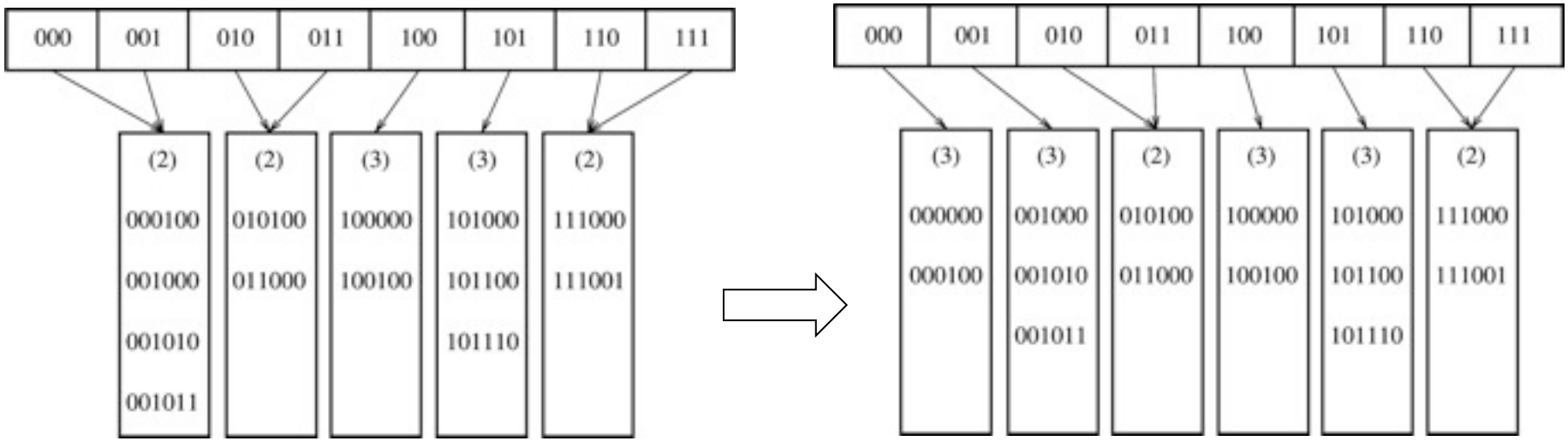
- Adding 000000

Insertions into Extendible Hash (Case 2)



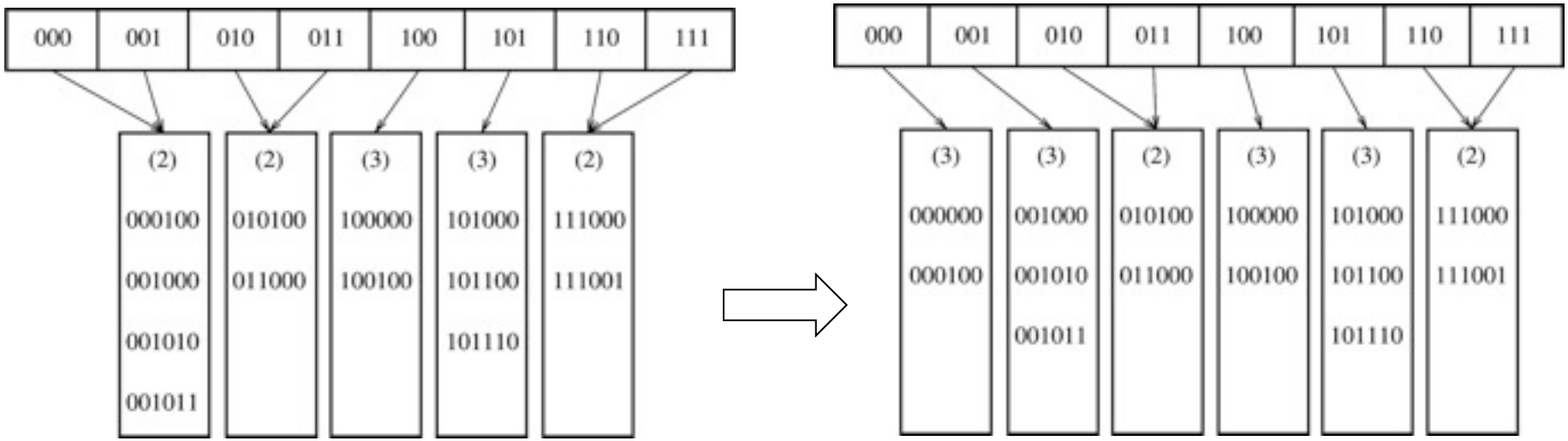
- Adding 000000

Insertions into Extendible Hash (Case 2)



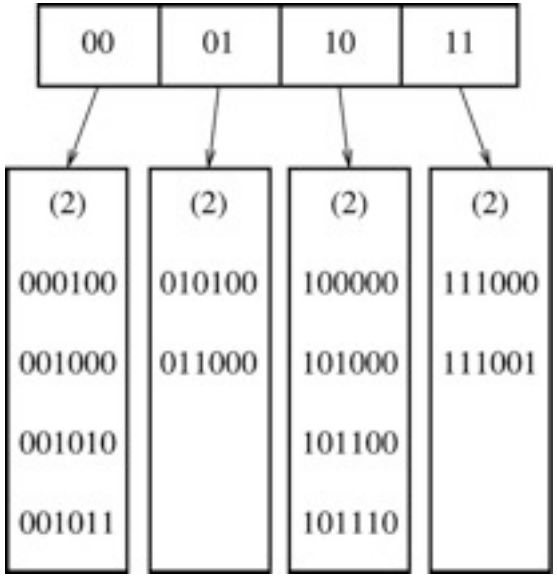
- Adding 000000

Insertions into Extendible Hash (Case 2)

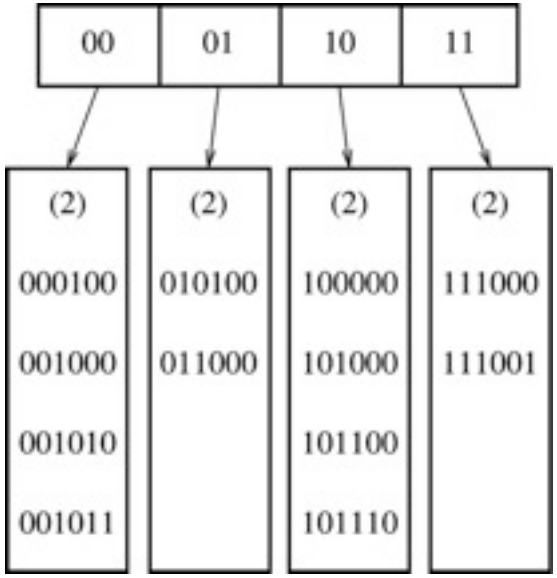


- Adding 000000
  - leaf split only

Insertions into Extendible Hash (Case 3)

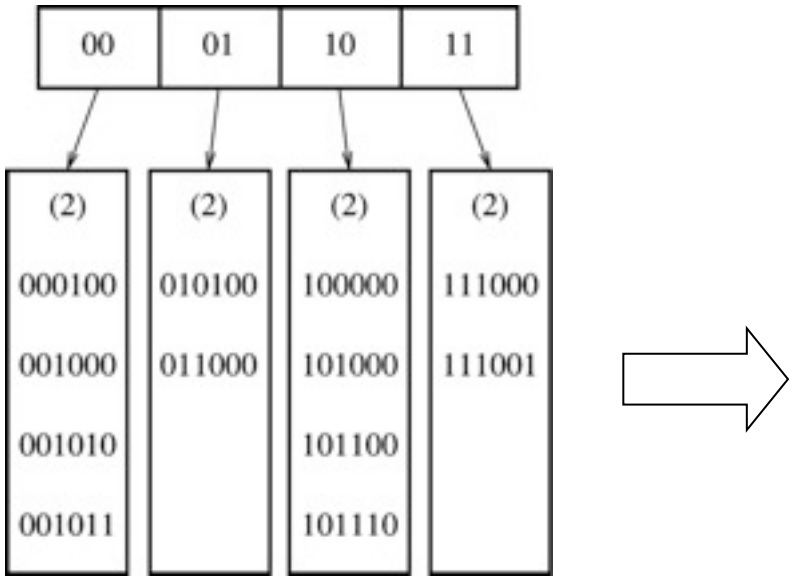


Insertions into Extendible Hash (Case 3)



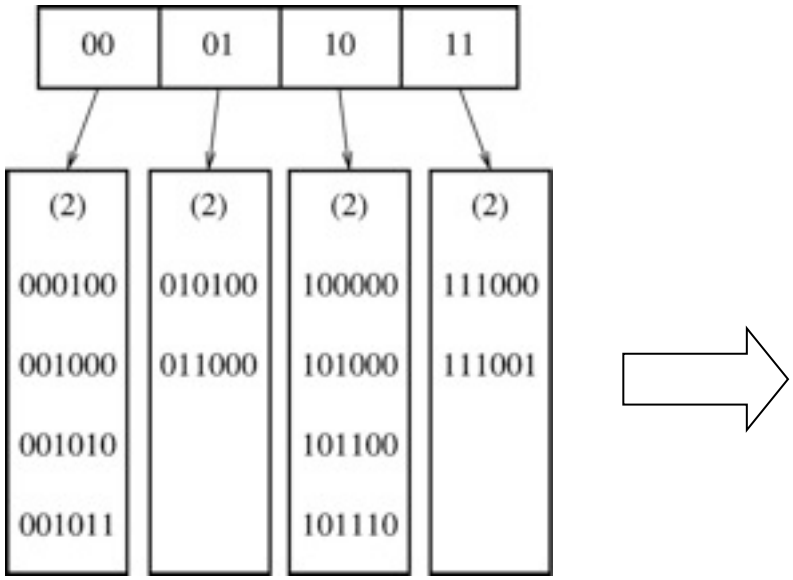
- Insert 111010, 111011, 111100

Insertions into Extendible Hash (Case 3)



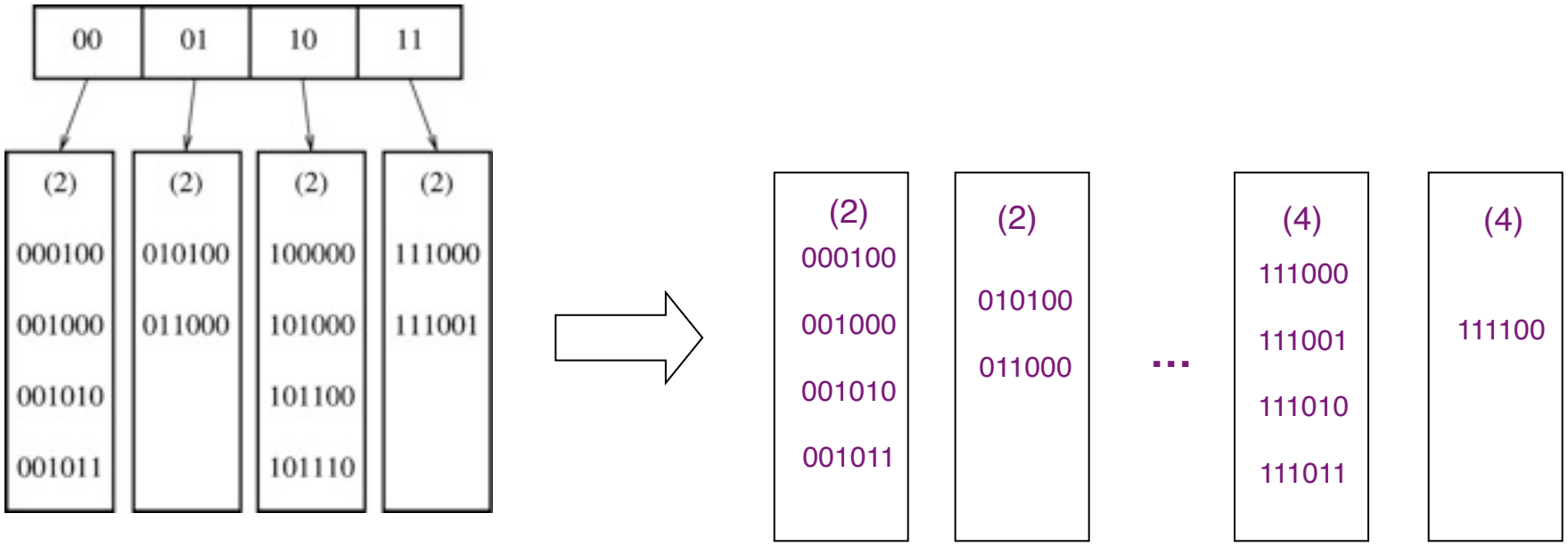
- Insert 111010, 111011, 111100

Insertions into Extendible Hash (Case 3)



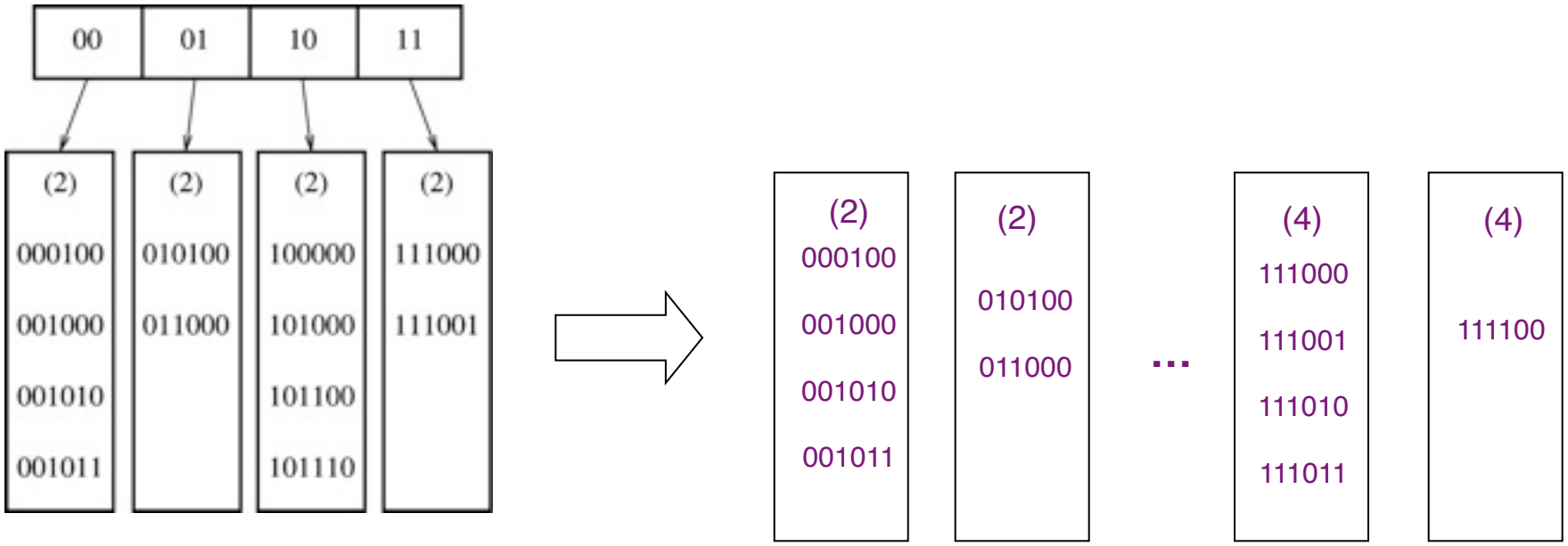
- Insert 111010, 111011, 111100
- Last bucket to split

Insertions into Extendible Hash (Case 3)



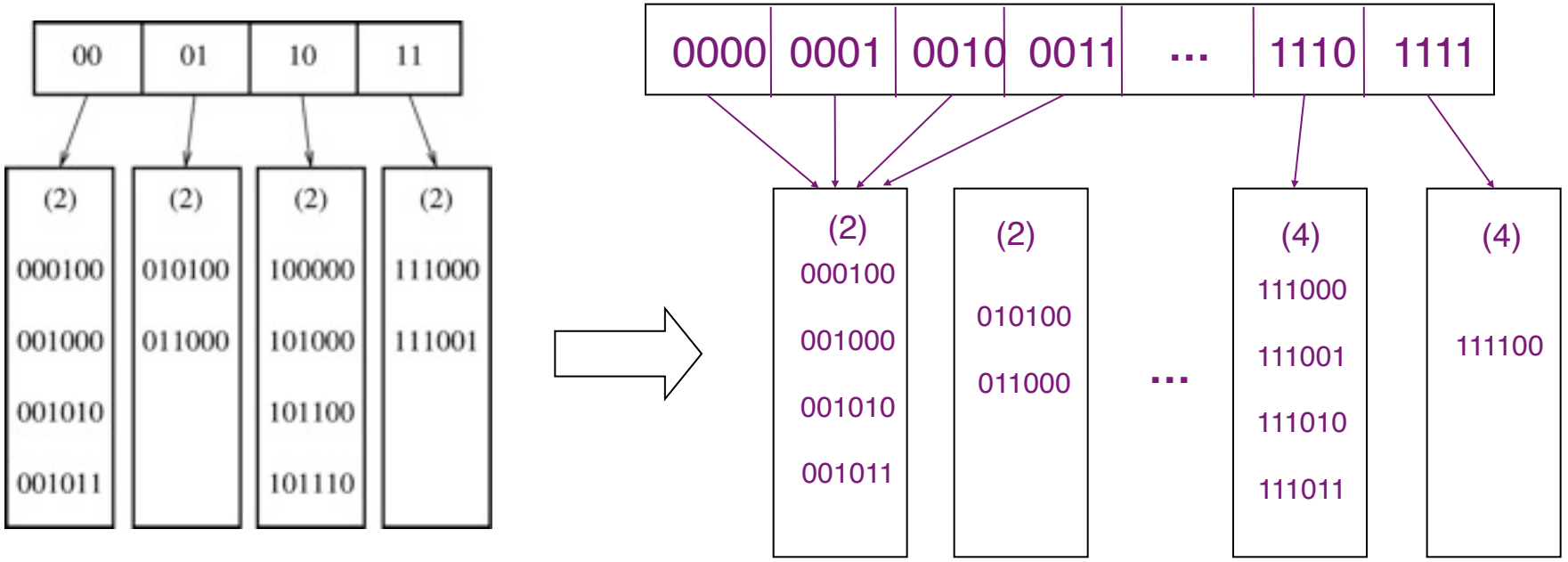
- Insert 111010, 111011, 111100
- Last bucket to split

Insertions into Extendible Hash (Case 3)



- Insert 111010, 111011, 111100
- Last bucket to split
- Can't distinguish two new buckets with i+1 bits
  - Increase directory offset bits by 2

Insertions into Extendible Hash (Case 3)



- Insert 111010, 111011, 111100
- Last bucket to split
- Can't distinguish two new buckets with i+1 bits
  - Increase directory offset bits by 2

Deletions from Extendible Hash

- Reverse to insertions
- Buckets may merge
- Directory may be reorganized

Extendible Hash Discussion

Extendible Hash Discussion

- B-trees
  - B-tree nodes are less space efficient (store key ranges)
  - Processing at a node (including root) requires matching a key against key ranges (logarithmic at best)
- Extendible hash
  - Constant-time directory access
  - Directory is space-efficient (just bucket pointers)
  - But: directory must fit into memory

Extendible Hash Discussion

- B-trees
  - B-tree nodes are less space efficient (store key ranges)
  - Processing at a node (including root) requires matching a key against key ranges (logarithmic at best)
- Extendible hash
  - Constant-time directory access
  - Directory is space-efficient (just bucket pointers)
  - But: directory must fit into memory
- Researcher's questions:
  - What if directory does not fit into memory?
  - Is this likely to happen?
    - ✓ If yes, we have a problem to solve!
    - ✓ If not, when do you ever want to use a B-tree?