Chapter 9 Tables and Information Retrieval

信息科学与技术学院　　黄方军

data_structures@163.com

东校区实验中心B502

9.1 Introduction

- **It is possible to complete a search of *n* items fewer than lg*n*.**
- **In this chapter we study to implement and access tables in contiguous storage.**

9.2 Rectangular Tables

1. Row-Major and column-Major Ordering

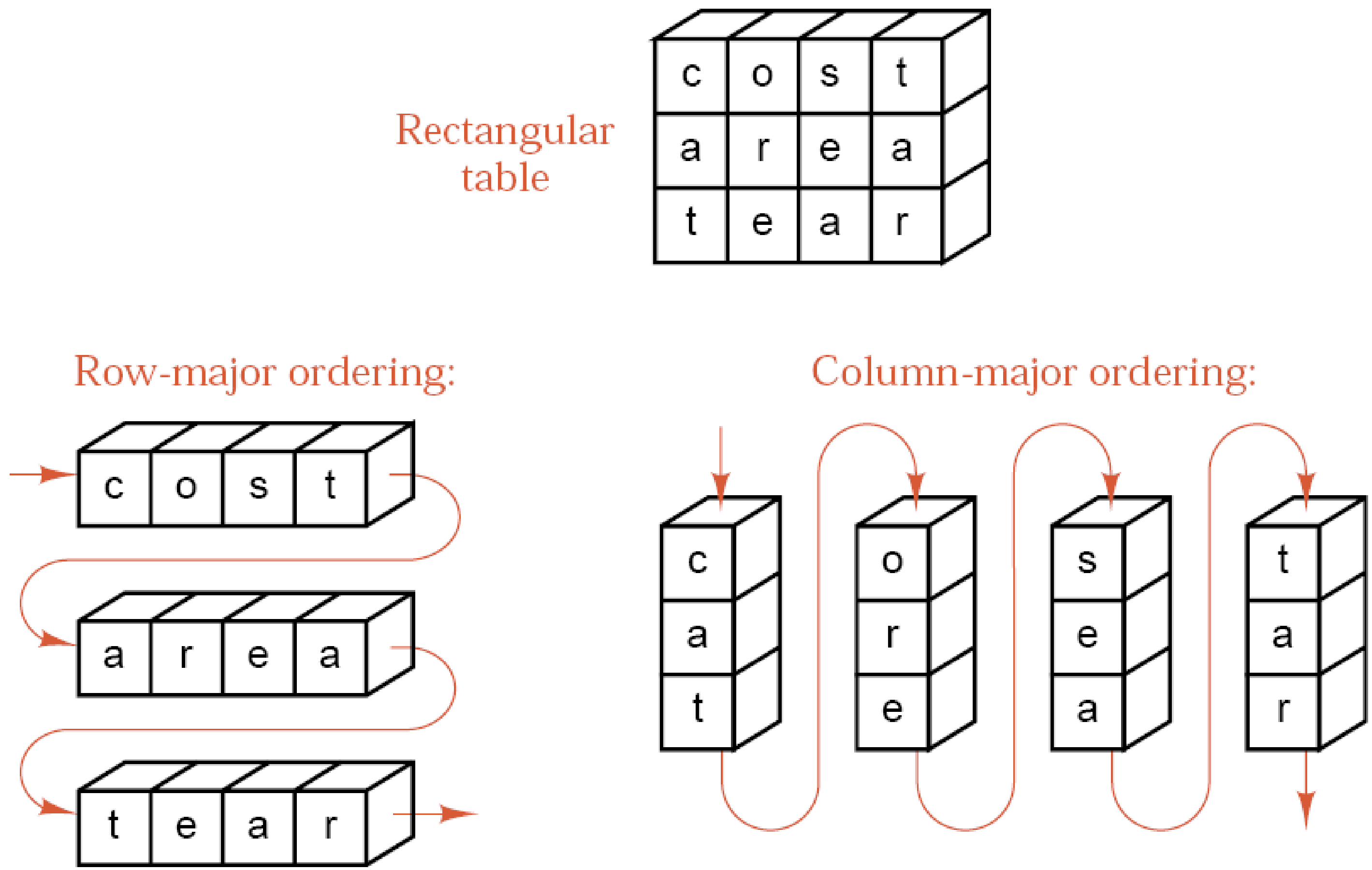


Figure 9.1. Sequential representation of a rectangular array

9.2 Rectangular Tables

2. Indexing Rectangular Tables

For simplicity we shall use only row-major ordering and suppose that rows are numbered from 0 to m-1 and columns from 0 to n-1.

In general, the entries of row i are preceded by ni earlier entries, so the desired formula is

Index Function:

Entry (i, j) in a rectangular table goes to position $ni + j$ in a sequential array.

3. Variation: An Access Array

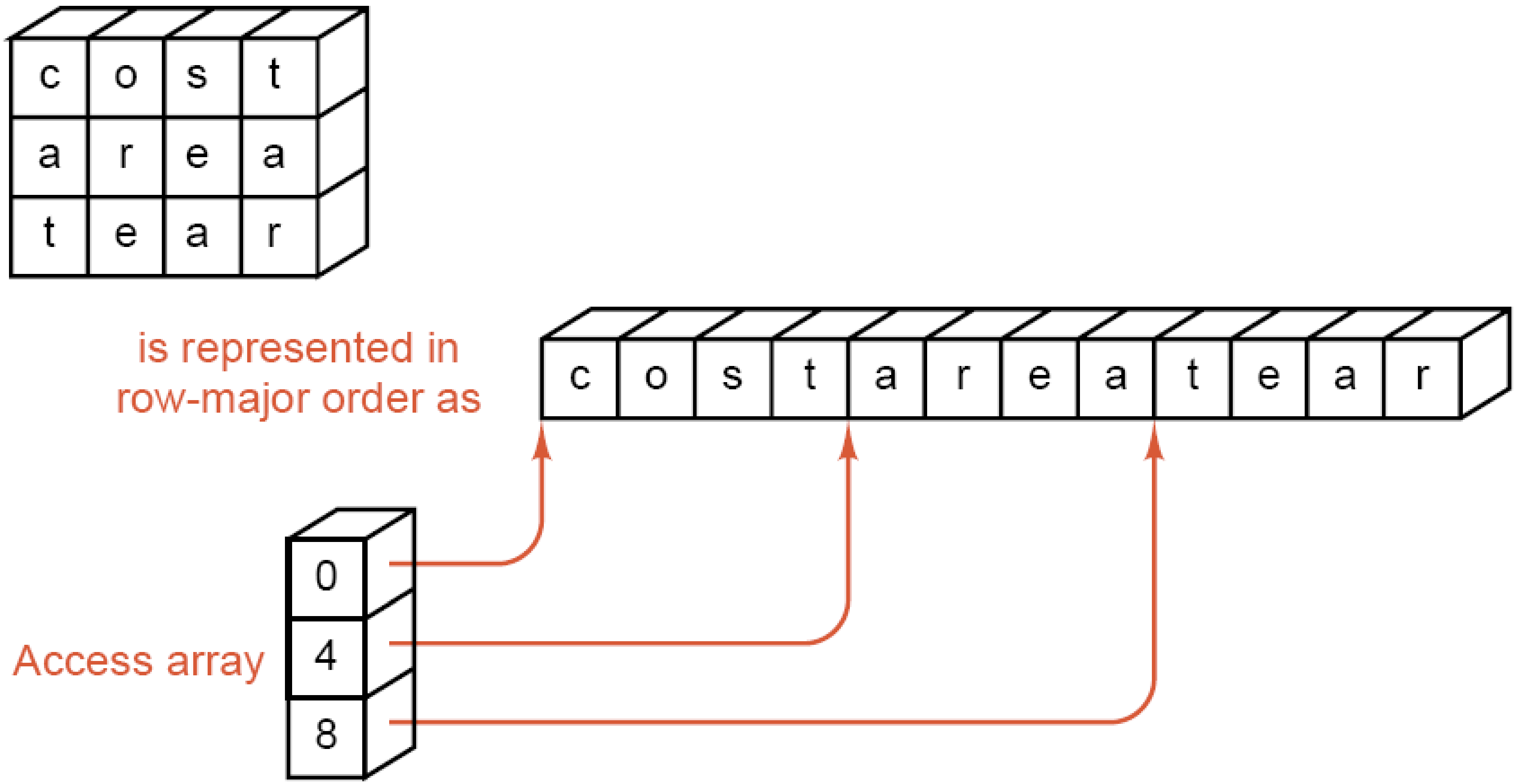


Figure 9.2. Access array for a rectangular table

9.3 Tables of Various Shapes

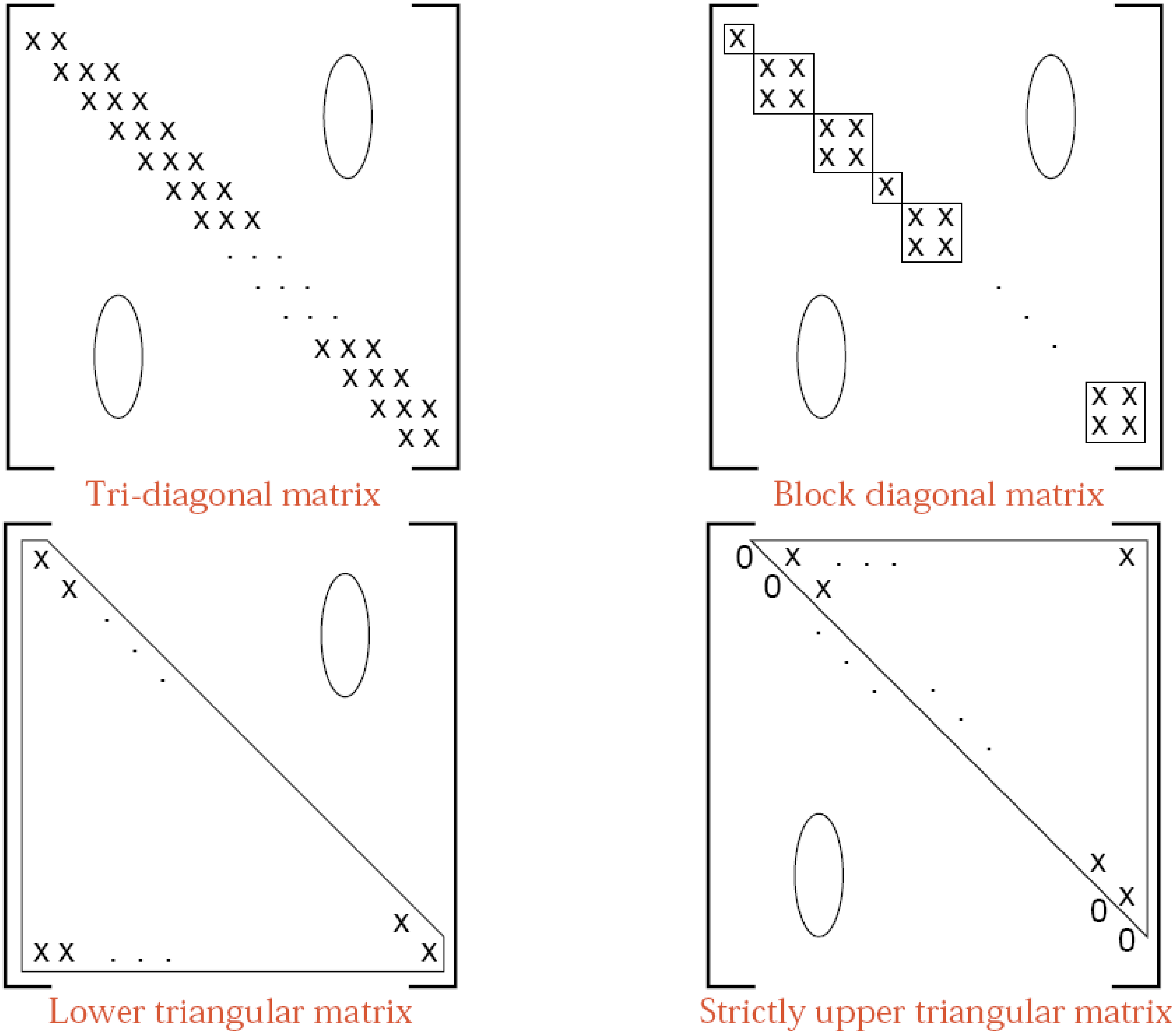


Figure 9.3. Matrices of various shapes

9.3.1 Triangular Tables

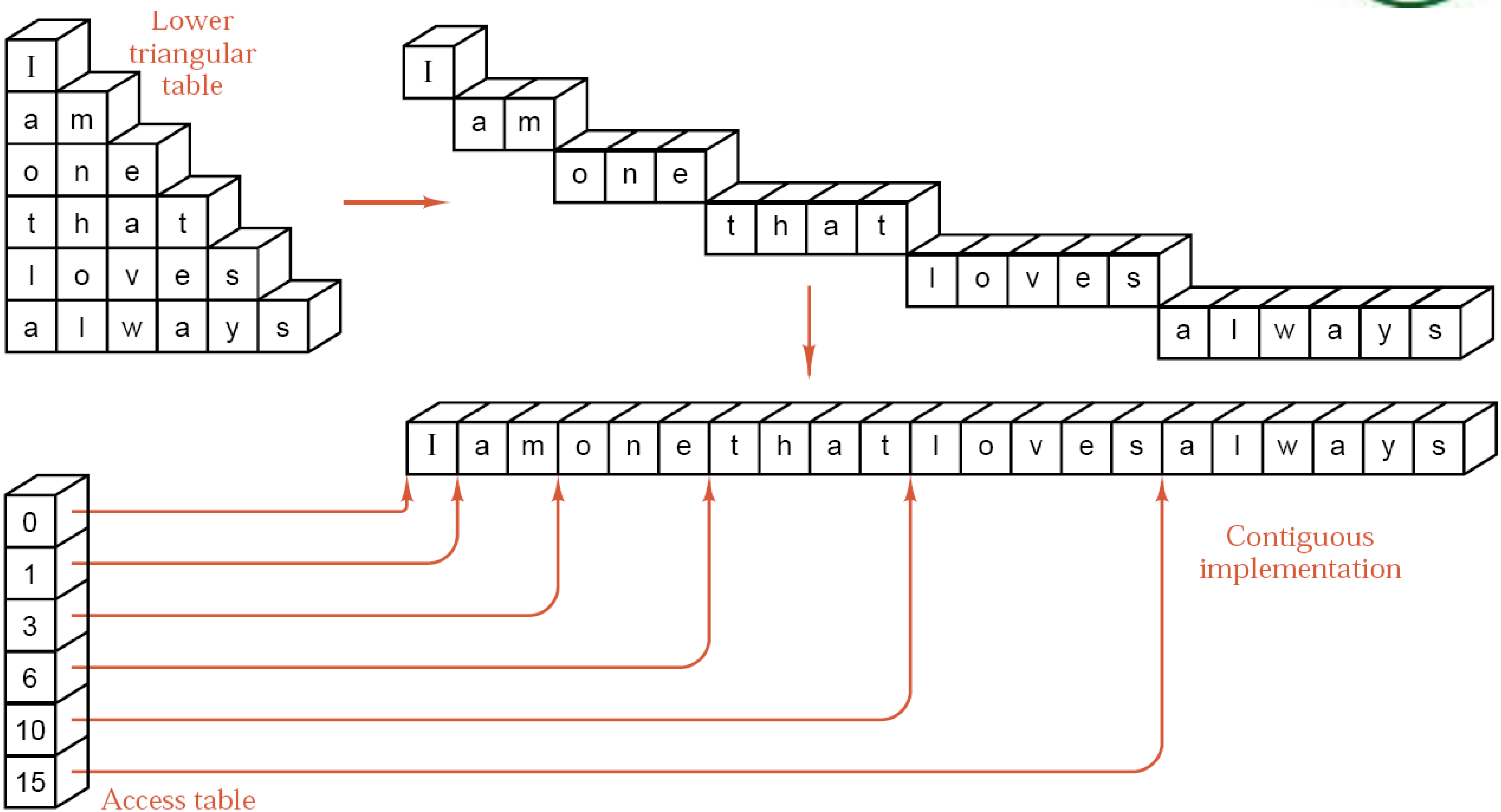


Figure 9.4. Contiguous implementation of a triangular table

9.3.2 Jagged Tables

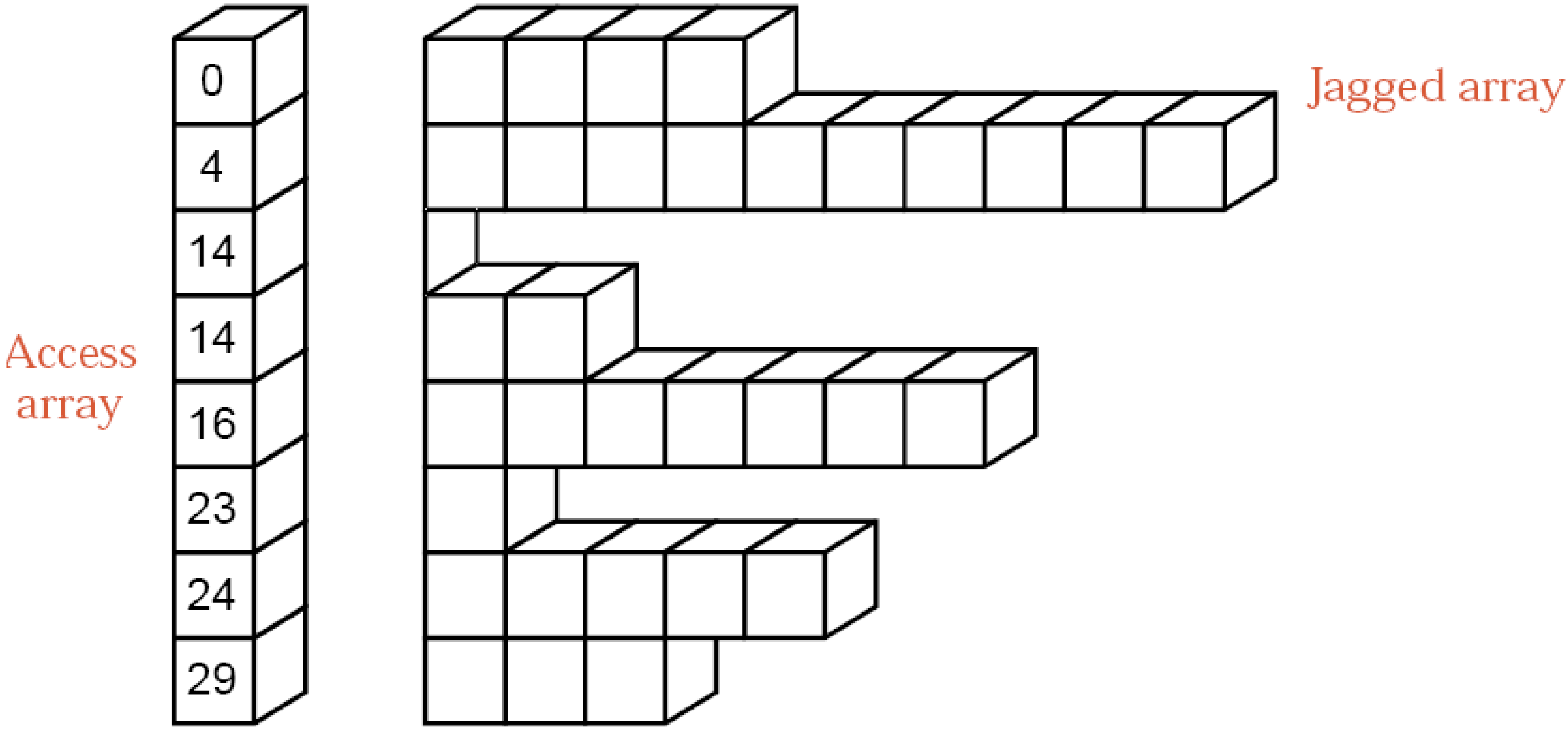


Figure 9.5. Access array for jagged table

9.3.3 Inverted Tables

Index	*Name*	*Address*	*Phone*
1	Hill, Thomas M.	High Towers #317	2829478
2	Baker, John S.	17 King Street	2884285
3	Roberts, L. B.	53 Ash Street	4372296
4	King, Barbara	High Towers #802	2863386
5	Hill, Thomas M.	39 King Street	2495723
6	Byers, Carolyn	118 Maple Street	4394231
7	Moody, C. L.	High Towers #210	2822214

Access Arrays

Name	*Address*	*Phone*
2	3	5
6	7	7
1	1	1
5	4	4
4	2	2
7	5	3
3	6	6

Figure 9.6. Multikey access arrays: an inverted table

9.4 Tables: A New Abstract Data Type

1. Functions

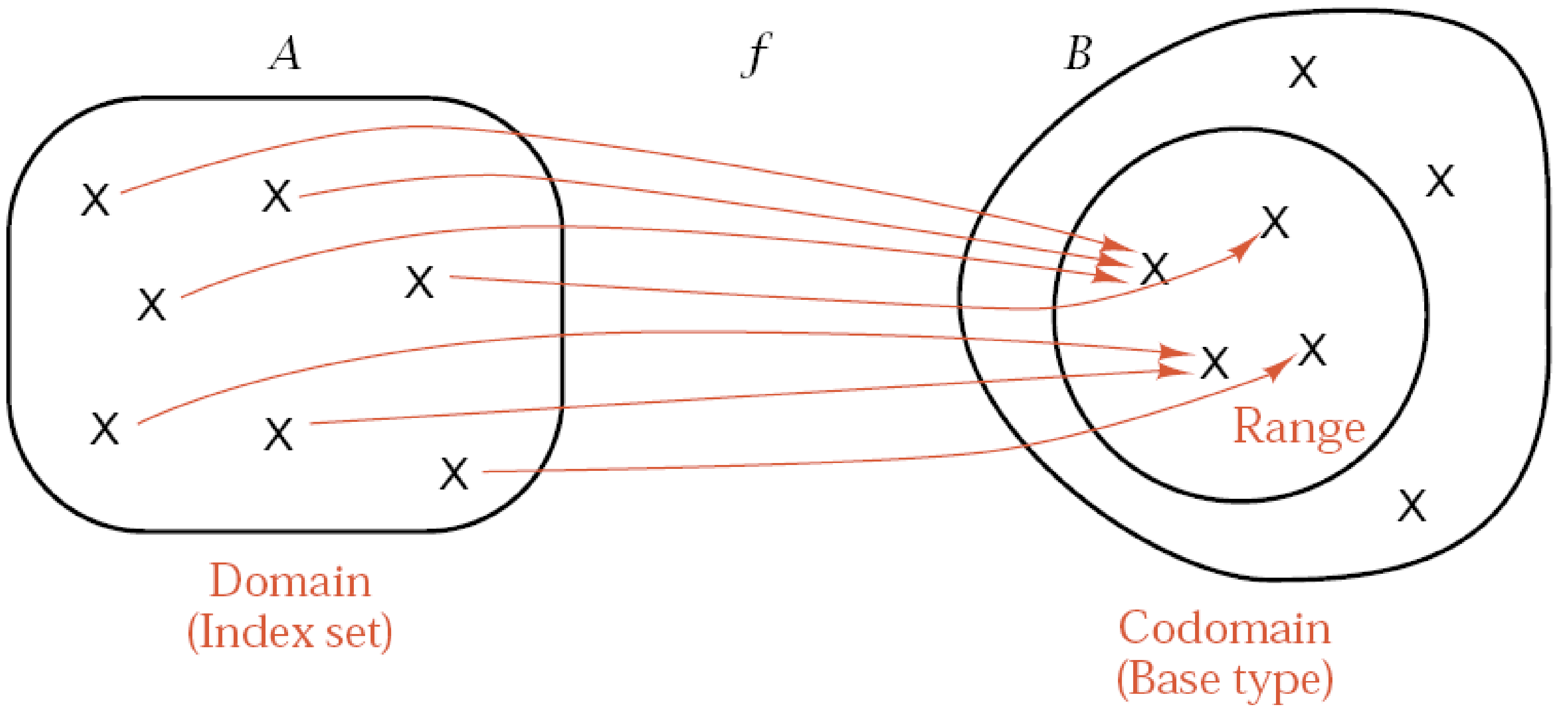


Figure 9.8. The domain, codomain, and range of a function

double array[n]; $\{(i, j) \mid 0 \le j \le i < m\}$.

9.4 Tables: A New Abstract Data Type

2. An Abstract Data Type

A ***table*** with index set I and base type T is a function from I into T together with the following operations.

1. *Table access*: Evaluate the function at any index in I.
2. *Table assignment*: Modify the function by changing its value at a specified index in I to the new value specified in the assignment.
3. *Creation*: Set up a new function from I to T.
4. *Clearing*: Remove all elements from the index set I, so the remaining domain is empty.
5. *Insertion*: Adjoin a new element x to the index set I and define a corresponding value of the function at x.
6. *Deletion*: Delete an element x from the index set I and restrict the function to the resulting smaller domain.

9.5 Application: Radix Sort

216 → 521 → 425 → 116 → 91 → 515 → 124 → 34

521 → 91 → 124 → 34 → 425 → 515 → 216 → 116

Chain after sorting on least significant digit

515 → 216 → 116 → 521 → 124 → 425 → 34 → 91

Chain after sorting on 2nd least significant digit

34 → 91 → 116 → 124 → 216 → 425 → 515 → 521

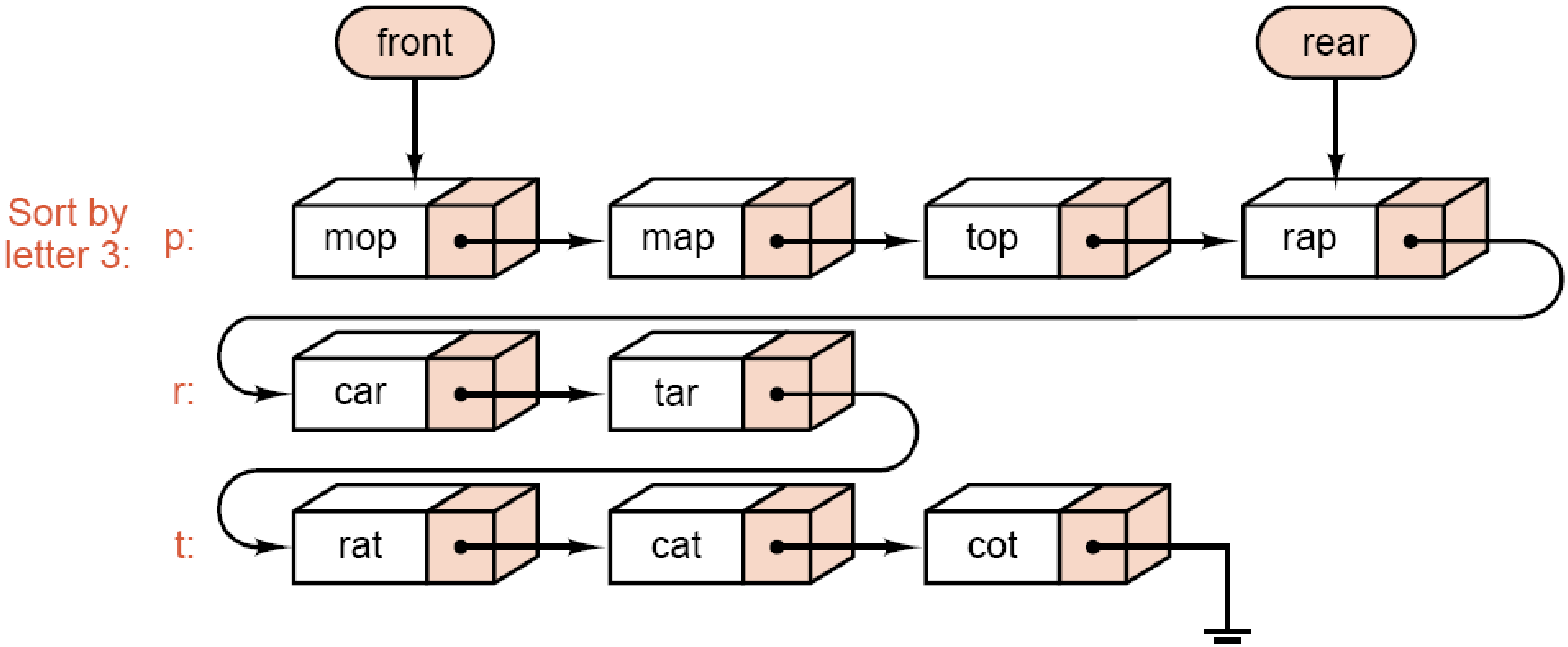
Chain after sorting on most significant digit

9.5 Application: Radix Sort

rat	mop	map	car
mop	map	rap	cat
cat	top	car	cot
map	rap	tar	map
car	car	rat	mop
top	tar	cat	rap
cot	rat	mop	rat
tar	cat	top	tar
rap	cot	cot	top
Initial order	Sorted by letter 3	Sorted by letter 2	Sorted by letter 1

Figure 9.10. Trace of a radix sort

9.5 Application: Radix Sort



9.5 Application: Radix Sort

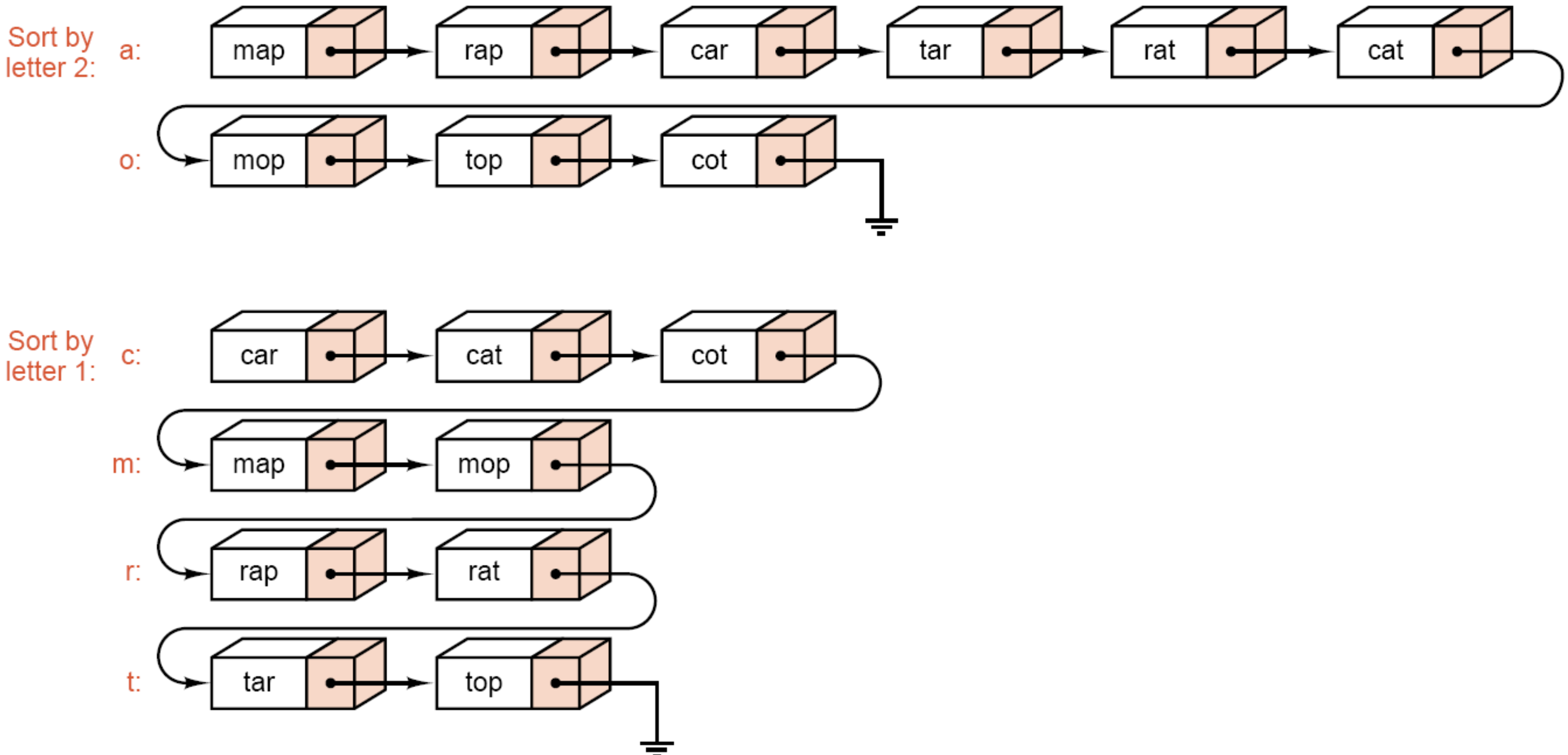


Figure 9.11. Linked radix sort

9.5 Application: Radix Sort

```
template <class Record>
class Sortable_list:  public List<Record> {
public:                                    //    sorting methods
   void radix_sort( );
   //    Specify any other sorting methods here.
private:                                   //    auxiliary functions
   void rethread(Queue queues[ ]);
};
```

```
class Record {
public:
   char key_letter(int position) const;
   Record( );                            //    default constructor
   operator Key( ) const;                //    cast to Key
//    Add other methods and data members for the class.
};
```

9.5 Application: Radix Sort

```
const int max_chars = 28;
template <class Record>
void Sortable_list<Record>::radix_sort()
/* Post: The entries of the Sortable_list have been sorted so all their keys are in
         alphabetical order.
   Uses: Methods from classes List, Queue, and Record;
         functions position and rethread. */
{
   Record data;
   Queue queues[max_chars];
   for (int position = key_size − 1; position >= 0; position−−) {
      //   Loop from the least to the most significant position.
      while (remove(0, data) == success) {
         int queue_number = alphabetic_order(data.key_letter(position));
         queues[queue_number].append(data);   //   Queue operation.
      }
      rethread(queues);                       //   Reassemble the list.
   }
}
```

9.5 Application: Radix Sort

```
int alphabetic_order(char c)
/* Post: The function returns the alphabetic position of character c, or it returns 0
         if the character is blank. */
{
   if (c == ' ') return 0;
   if ('a' <= c && c <= 'z') return c − 'a' + 1;
   if ('A' <= c && c <= 'Z') return c − 'A' + 1;
   return 27;
}
```

9.5 Application: Radix Sort

```
template <class Record>
void Sortable_list<Record>::rethread(Queue queues[ ])
/* Post: All the queues are combined back to the Sortable_list, leaving all the
         queues empty.
   Uses: Methods of classes List and Queue. */
{
   Record data;
   for (int i = 0; i < max_chars; i++)
      while (!queues[i].empty()) {
         queues[i].retrieve(data);
         insert(size(), data);
         queues[i].serve();
      }
}
```

9.5 Application: Radix Sort

- Note that the time used by radix sort is $\Theta(nk)$
- The best time was that of mergesort, which was $n \lg n + O(n)$.
- If the keys are long but there are relatively few of them, then k is large and $\lg n$ relatively small, and other methods (such as mergesort) will outperform radix sort;
- but if k is small (the keys are short) and there are a large number of keys, then radix sort will be faster than any other method we have studied.

9.6 Hashing

- Worst-case time for search, insert, and delete is O(size).
- Expected time is O(1).

9.6 Hashing

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
class	public	private		do	operator	explicit	switch		return	unsigned	new			protected	enum	register	float	else	continue	typedef			

continued below

25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
	static	short	template		int	struct			for		auto		signed	this			extern	sizeof		throw		

Figure 9.12. A hash table

Ideal Hashing

- Uses a 1D array (or table) table[0:b-1].
  - Each position of this array is a bucket.
  - A bucket can normally hold only one record.
- Uses a hash function f that converts each key k into an index in the range [0, b-1].
  - f(k) is the home bucket for key k.
- Every record (key, element) is stored in its home bucket table[f[key]].

Ideal Hashing Example

- Pairs are: (22,a), (33,c), (3,d), (73,e), (85,f).
- Hash table is table[0:7], b = 8.
- Hash function is key/11.
- Pairs are stored in table as below:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
(3,d)		(22,a)	(33,c)			(73,e)	(85,f)

- search, insert, and delete take O(1) time.

What Can Go Wrong?

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

- Where does (26,g) go?
- Keys that have the same home bucket are synonyms.
  - 22 and 26 are synonyms with respect to the hash function that is in use.
- The home bucket for (26,g) is already occupied.

What Can Go Wrong?

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)

- A collision occurs when the home bucket for a new record is occupied by a record with a different key.
- An overflow occurs when there is no space in the home bucket for the new pair.
- When a bucket can hold only one record, collisions and overflows occur together.
- Need methods to handle collisions (and overflows).

Hash Table Issues

- Size (number of buckets) of hash table.
- Choice of hash function.
- collisions handling method.

9.6.2 Choosing a Hash Function

Two principal criteria:

- It should be easy and quick to compute.
  - Convert key into an integer in case the key is not an integer.
    - Done by the method hashCode().
- It should achieve an even distribution of the keys that actually occur across the range of indices.
  - f(k) is an integer in the range [0, b-1], where b is the number of buckets in the table.

9.6.2 Choosing a Hash Function

1. Truncation
   - The first, second, and fifth digits from the right might make the hash function, so that 21296876 maps to 976.
2. Folding
   - 21296876 maps to 212+968+76 = 1256.
3. Modular Arithmetic
   - The best choice for modulus is often, but not always, a prime number.

9.6.2 Choosing a Hash Function

4. C++ Example

- A simple example for transforming a key consisting of eight alphanumeric characters into an integer in the range 0…has_size-1.

```
class Key:  public String{
public:
    char key_letter(int position) const;
    void make_blank();
    //     Add constructors and other methods.
};
```

9.6.2 Choosing a Hash Function

```
int hash(const Key &target)
/* Post: target has been hashed, returning a value between 0 and hash_size −1.
   Uses: Methods for the class Key. */
{
   int value = 0;
   for (int position = 0;  position < 8;  position++)
      value = 4 * value + target.key_letter(position);
   return value % hash_size;
}
```

Map Into A Home Bucket

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
(3,d)		(22,a)	(33,c)			(73,e)	(85,f)

- Most common method is by

homeBucket =

Math.abs(theKey.hashCode()) % divisor;

- divisor equals number of buckets b.
- 0 <= homeBucket < divisor = b

Uniform Hash Function

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
(3,d)		(22,a)	(33,c)			(73,e)	(85,f)

- Let keySpace be the set of all possible keys.
- A uniform hash function maps the keys in keySpace into buckets such that approximately the same number of keys get mapped into each bucket.

Uniform Hash Function

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

- Equivalently, the probability that a randomly selected key has bucket i as its home bucket is $1/b$, $0 <= i < b$.

- A uniform hash function minimizes the likelihood of an overflow when keys are selected at random.

Selecting The Divisor

- Because of this correlation, applications tend to have a bias towards keys that map into odd integers (or into even ones).
- When the divisor is an even number, odd integers hash into odd home buckets and even integers into even home buckets.
  - 20%14 = 6, 30%14 = 2, 8%14 = 8
  - 15%14 = 1, 3%14 = 3, 23%14 = 9
- The bias in the keys results in a bias toward either the odd or even home buckets.

Selecting The Divisor

- When the divisor is an odd number, odd (even) integers may hash into any home.
  - 20%15 = 5, 30%15 = 0, 8%15 = 8
  - 15%15 = 0, 3%15 = 3, 23%15 = 8
- The bias in the keys does not result in a bias toward either the odd or even home buckets.
- Better chance of uniformly distributed home buckets.
- So do not use an even divisor.

Selecting The Divisor

- Similar biased distribution of home buckets is seen, in practice, when the divisor is a multiple of prime numbers such as 3, 5, 7, …
- Ideally, choose b so that it is a prime number.
- Alternatively, choose b so that it has no prime factor smaller than 20.

9.6.3 Collision Resolution with Open Address

- Clustering
- Linear Probing
- Increment Functions
- Quadratic Probing
- Key-Dependent Increments
- Random Probing

9.6.3 Collision Resolution with Open Address

- Clustering

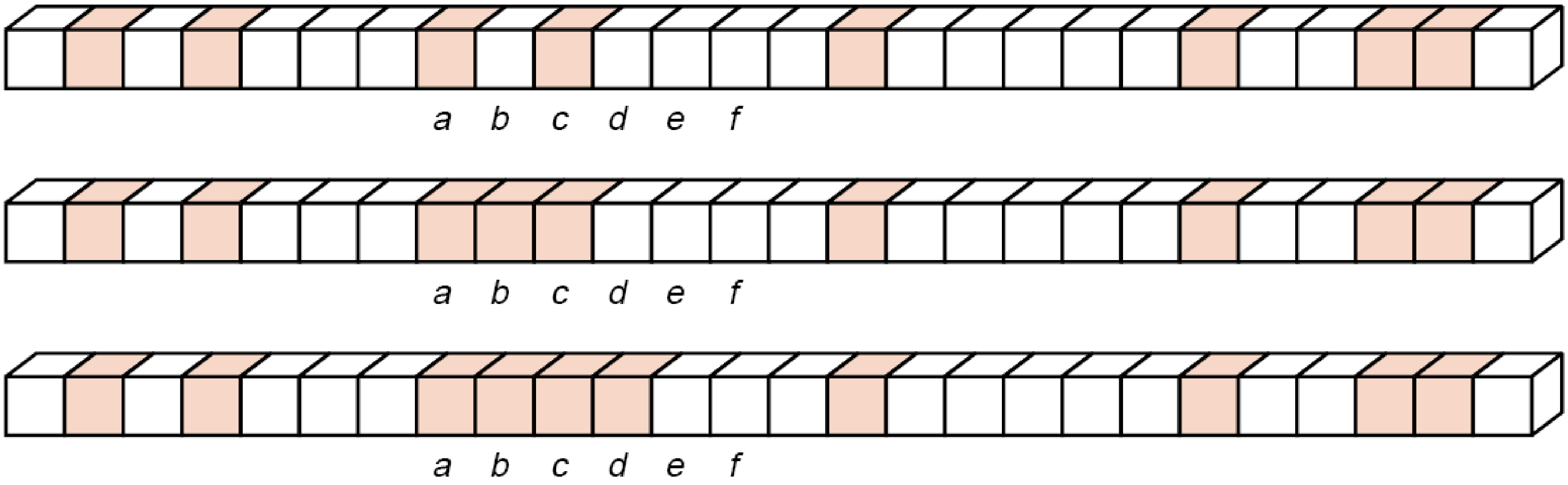


Figure 9.13. Clustering in a hash table

Linear Probing – Get And Put

- divisor = b (number of buckets) = 17.
- Home bucket = key % 17.

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

- Put in pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45

Linear Probing -- Remove

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

- remove(0)

0				4				8				12				16
34		45				6	23	7			28	12	29	11	30	33

- Search cluster for pair (if any) to fill vacated bucket.

0				4				8				12				16
34	45					6	23	7			28	12	29	11	30	33

Linear Probing – remove(34)

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

0				4				8				12				16
	0	45				6	23	7			28	12	29	11	30	33

- Search cluster for pair (if any) to fill vacated bucket.

0				4				8				12				16
0		45				6	23	7			28	12	29	11	30	33

0				4				8				12				16
0	45					6	23	7			28	12	29	11	30	33

Linear Probing – remove(29)

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

0				4				8				12				16
34	0	45				6	23	7			28	12		11	30	33

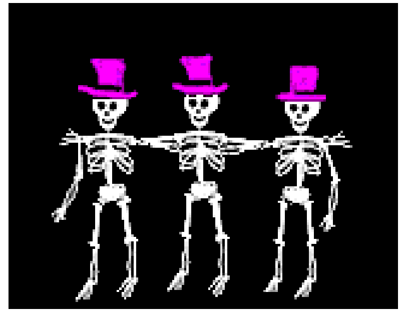
- Search cluster for pair (if any) to fill vacated bucket.

0				4				8				12				16
34	0	45				6	23	7			28	12	11		30	33

0				4				8				12				16
34	0	45				6	23	7			28	12	11	30		33

0				4				8				12				16
34	0					6	23	7			28	12	11	30	45	33

Performance Of Linear Probing



0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

- Worst-case get/put/remove time is $\Theta(n)$, where n is the number of pairs in the table.
- This happens when all pairs are in the same cluster.

9.6.3 Collision Resolution with Open Address

- Linear Probing
- Increment Functions (*rehashing*)
- Quadratic Probing
- Key-Dependent Increments
- Random Probing

9.6.3 Collision Resolution with Open Address

```
const int hash_size = 997;        //   a prime number of appropriate size
class Hash_table {
public:
   Hash_table();
   void clear();
   Error_code insert(const Record &new_entry);
   Error_code retrieve(const Key &target, Record &found) const;
private:
   Record table[hash_size];
};
```

9.6.3 Collision Resolution with Open Address

```
Error_code Hash_table :: insert(const Record &new_entry)
    Error_code result = success;
    int probe_count,          //   Counter to be sure that table is not full.
        increment,            //   Increment used for quadratic probing.
        probe;                //   Position currently probed in the hash table.
    Key null;                 //   Null key for comparison purposes.
    null.make_blank( );
    probe = hash(new_entry);
    probe_count = 0;
    increment = 1;
    while (table[probe] != null   //   Is the location empty?
            && table[probe] != new_entry   //   Duplicate key?
            && probe_count < (hash_size + 1)/2) {   //   Has overflow occurred?
      probe_count++;
      probe = (probe + increment) % hash_size;
      increment += 2;         //   Prepare increment for next iteration.
    }
    if (table[probe] == null) table[probe] = new_entry;   //   Insert new entry.
    else if (table[probe] == new_entry) result = duplicate_error;
    else result = overflow;   //   The table is full.
    return result;
}
```

9.6.4 Collision Resolution by Chaining

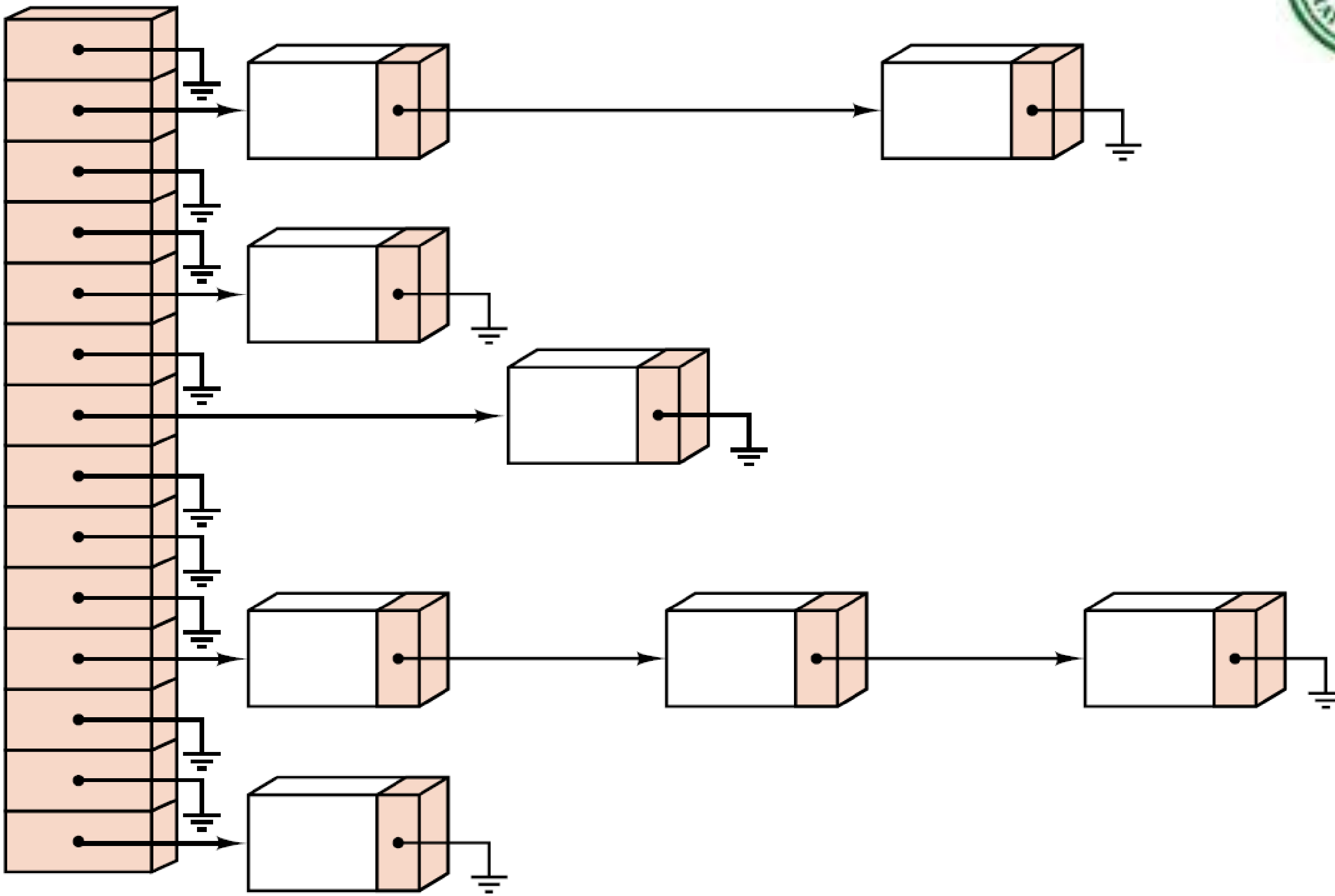
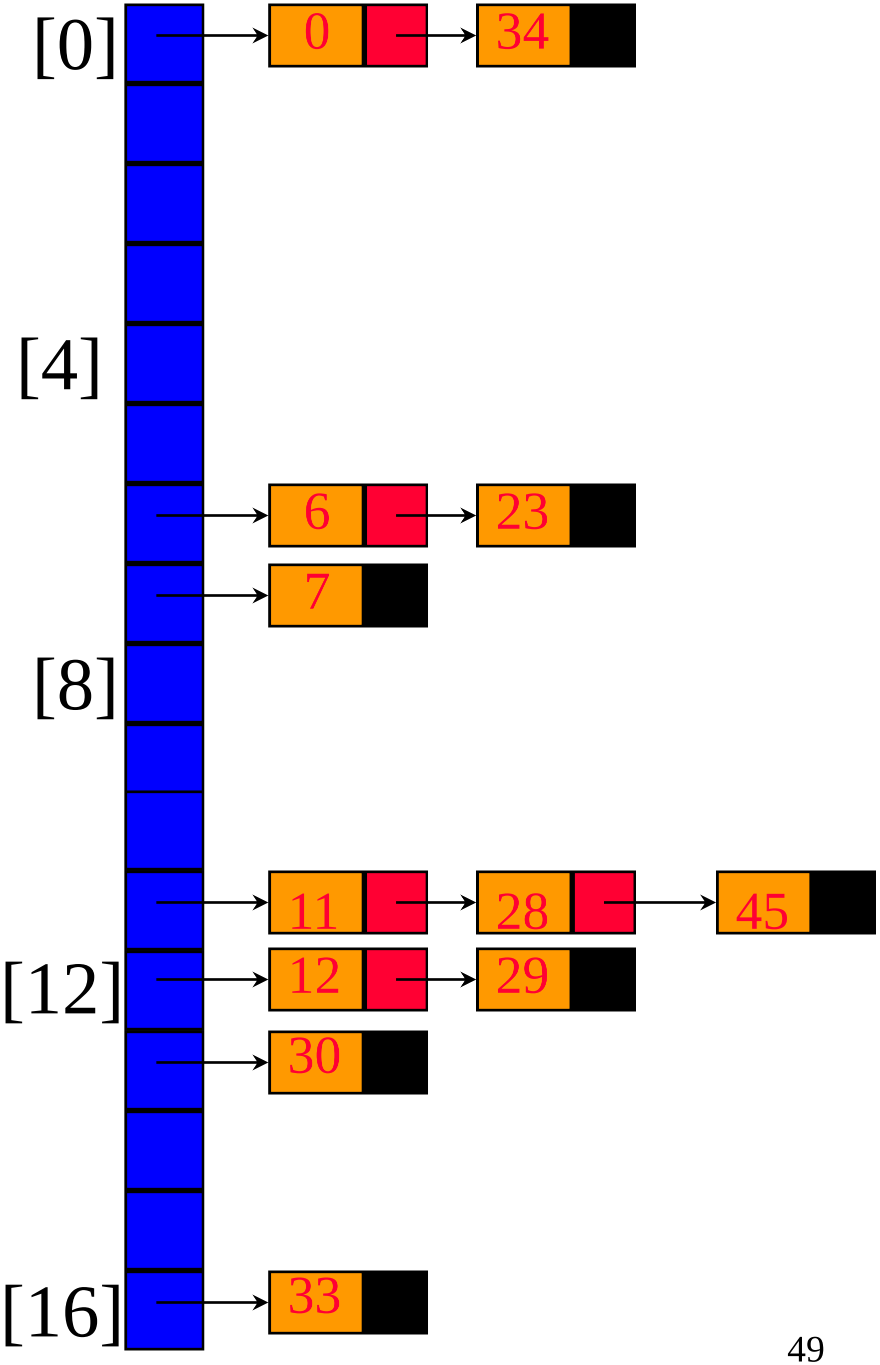


Figure 9.14. A chained hash table

Sorted Chains

- Put in pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45
- Home bucket = key % 17.



1. The Birthday Surprise

The probability that *m* people all have different birthdays is

$$\frac{364}{365} \times \frac{363}{365} \times \frac{362}{365} \times \cdots \times \frac{365-m+1}{365}.$$

This expression becomes less than 0.5 whenever $m \geq 23$.

2. Counting Probes

- Let n be the number of entries in the table.
- Let t (which is the same as hash_size) be the number of positions in the array holding the hash table.

Load factor

$$\lambda = n/t.$$

Thus $\lambda = 0$ signifies an empty table;

$\lambda = 0.5$ a table that is half full.

3. Analysis of Chaining

Suppose that the chain that will contain the target has *k* entries. Note that *k* might be 0.

For a unsuccessful search:

$$\lambda = n/t.$$

For a successful search:

$$\frac{1}{2}(k+1) \approx \frac{1}{2}(1+\lambda+1) = 1+\frac{1}{2}\lambda.$$

$1+(n-1)/t$

4. Analysis of Open Addressing (随机探测)

The expected number of probes in an unsuccessful search is therefore

$$U(\lambda)=\sum_{k=1}^{\infty} k\lambda^{k-1}(1-\lambda)=\frac{1}{(1-\lambda)^2}(1-\lambda)=\frac{1}{1-\lambda}.$$

The average number of probes in successful search is approximately

$$S(\lambda)=\frac{1}{\lambda}\int_0^{\lambda} U(\mu)d\mu=\frac{1}{\lambda}\ln\frac{1}{1-\lambda}.$$

9.7 Analysis of Hashing

- 线性探测

Retrieval from a hash table with open addressing, linear probing, and load factor λ requires, on average, approximately

$$\frac{1}{2}\left(1+\frac{1}{1-\lambda}\right)$$

probes in the successful case and

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

probes in the unsuccessful case.

9.7 Analysis of Hashing

5. Theoretical Comparisons

Load factor	0.10	0.50	0.80	0.90	0.99	2.00
Successful search, expected number of probes:						
Chaining	1.05	1.25	1.40	1.45	1.50	2.00
Open, random probes	1.05	1.4	2.0	2.6	4.6	—
Open, linear probes	1.06	1.5	3.0	5.5	50.5	—
Unsuccessful search, expected number of probes:						
Chaining	0.10	0.50	0.80	0.90	0.99	2.00
Open, random probes	1.1	2.0	5.0	10.0	100.	—
Open, linear probes	1.12	2.5	13.	50.	5000.	—

Figure 9.15. Theoretical comparison of hashing methods

6. Empirical Comparisons

Load factor	0.1	0.5	0.8	0.9	0.99	2.0
Successful search, average number of probes:						
Chaining	1.04	1.2	1.4	1.4	1.5	2.0
Open, quadratic probes	1.04	1.5	2.1	2.7	5.2	—
Open, linear probes	1.05	1.6	3.4	6.2	21.3	—
Unsuccessful search, average number of probes:						
Chaining	0.10	0.50	0.80	0.90	0.99	2.00
Open, quadratic probes	1.13	2.2	5.2	11.9	126.	—
Open, linear probes	1.13	2.7	15.4	59.8	430.	—

Figure 9.16. Empirical comparison of hashing methods

9.7 Analysis of Hashing

We can summarize these observations for retrieval from n entries as follows:

- Sequential search is $\Theta(n)$.
- Binary search is $\Theta(\log n)$.
- Hash-table retrieval is $\Theta(1)$.