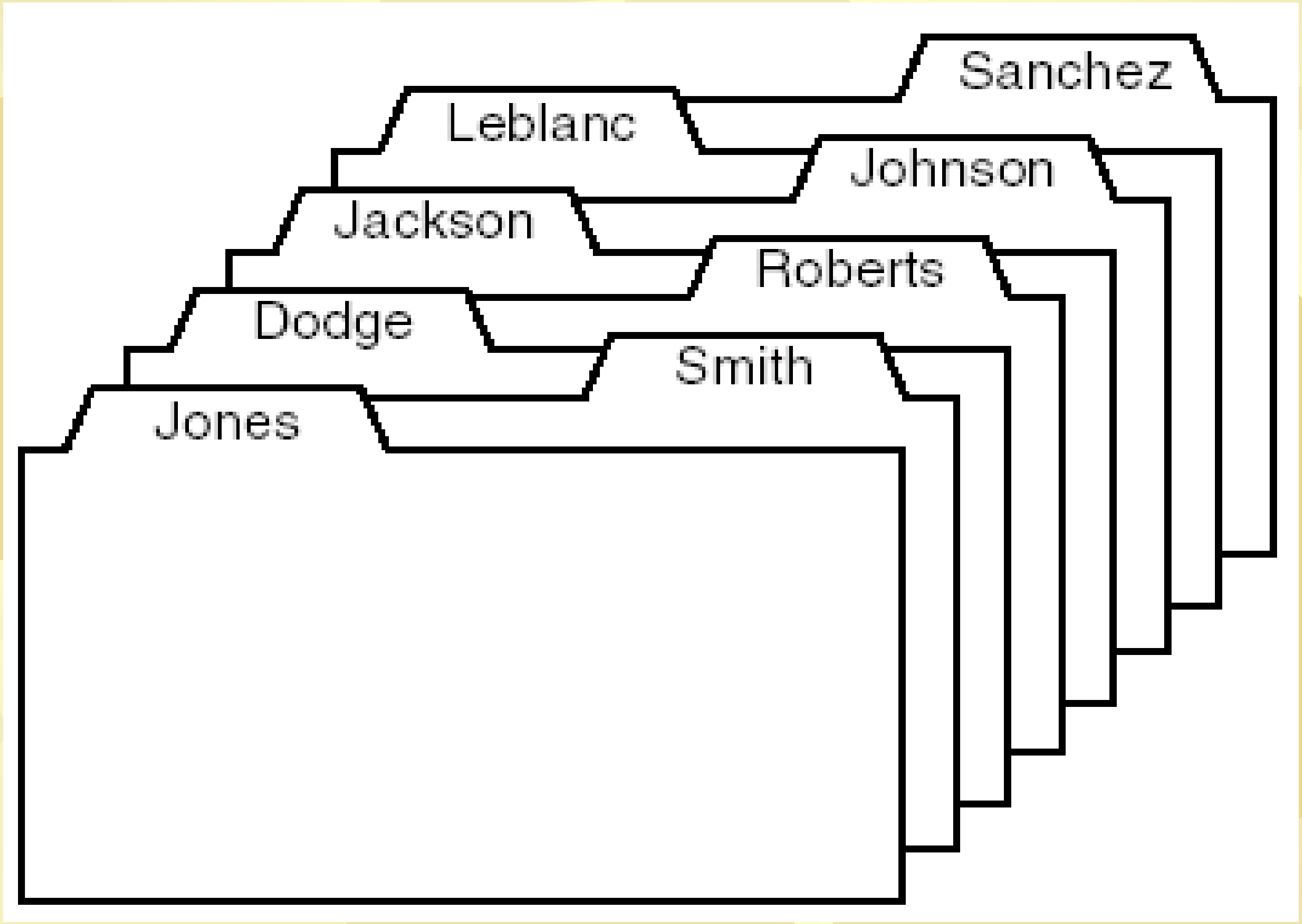
The 8th Course

SEARCH

Records and Keys



Records and Keys

- We are given a ***list*** of ***records***, where each record is associated with one piece of information, which we shall call a ***key***.
- We are given one key, called the ***target***, and are asked to search the list to find the record(s) (if any) whose key is the same as the target.

Records and Keys

- There may be more than one record with the same key, or there may be no record with a given key.
- We often ask *how times* one key is compared with another during a search. This gives us a good measure of the total amount of work that the algorithm will do.

Records and Keys

- ***Internal searching*** means that all the records are kept in high-speed memory. In ***external searching***, most of the records are kept in disk files. We study only internal searching.
- For now, we consider only contiguous lists of records.

Records and Keys in C++

- The records (from a class Record) that are stored in the list being searched (generally called the list) must conform to the following minimal ***standards***:
  - Every Record is associated to a key (of a type or class called Key).
  - Key objects can be compared with the standard operators == , != , <, >, <= , >= .
  - There is a conversion operation to turn any Record into its associated Key.
  - Record objects can be compared to each other or to a Key by first converting the Record objects to their associated keys.

Records and Keys in C++

- Examples of the conversion operation:
  - A method of the class Record, with the declaration operator Key( ) const;
  - A constructor for the class Key, with declaration Key(const Record &);
  - If the classes Key and Record are identical, no conversion needs to be defined, since any Record is automatically a Key.
- We do not assume that a Record has a Key object as a data member, although often it does. We merely assume that the compiler can turn a Record into its corresponding Key.

Parameters for Search Functions

- Each searching function has two input parameters:
  - First is the *list* to be searched;
  - Second is the *target* key for which we are searching.

Parameters for Search Functions

- **Each searching function will also have an output parameter and a returned value:**
  - **The returned value has type *Error_code* and indicates whether or not the search is successful in finding an entry with the target key.**
  - **If the search is successful, then the returned value is *success,*and the output parameter called position will locate the target within the list.**
  - **If the search is unsuccessful, then the value *not_present* is returned, and the output parameter may have an undefined value or a value that will differ from one method to another.**

Key Definition in C++

- To select existing types as records and keys, a client could use type definition statements such as:

```
typedef int Key;
typedef int Record;
```

Sequential Search and Analysis

- Begin at one end of the list and scan down it until the desired key is found or the other end is reached:
- To analyze the behavior of an algorithm that makes comparisons of keys, we shall use the count of these key comparisons as our measure of the work done.

Sequential Search and Analysis

- The number of comparisons of keys done in sequential search of a list of length n is
  - Unsuccessful search: *n* comparisons.
  - Successful search, best case: *1* comparison.
  - Successful search, worst case: *n* comparisons.
  - Successful search, average case: *½(n+1)* comparisons.

Test Data for Searching

- Most later searching methods require the data to be ordered, so use a list with integer keys in increasing order.
- To test both successful and unsuccessful searches, insert only keys containing odd integers into the list.
- If n denotes the number of entries in the list, then the targets for successful searches will be 1,3,5, ... , 2n – 1.
- For unsuccessful searches, the targets will be 0, 2, 4, 6,..., 2n.

Test Data for Searching

- In this way we test all possible failures, including targets less than the smallest key in the list, between each pair, and greater than the largest.
- To make the test more realistic, use pseudo-random numbers to choose the target, by employing the method
  - Random :: random_ integer

from Appendix B.

Ordered Lists

- DEFINITION: ***An ordered list*** is a list in which each entry contains a key, such that the keys are in order. That is, if entry i comes before entry j in the list, then the key of entry i is less than or equal to the key of entry j .

Ordered Lists

- All List operations except insert and replace apply without modification to an ordered list.
- Methods insert and replace must fail when they would otherwise disturb the order of a list.
- We implement an ordered list as a class derived from a contiguous List. In this derived class, we shall override the methods insert and replace with new implementations.

Overloaded Insertion

- We also overload the method insert so that it can be used with a single parameter. The position is automatically determined so as to keep the resulting list ordered:
- The scope resolution is necessary, because we have overridden the original List insert with a new Ordered list method:

Overridden Insertion

- Note: The overridden method replaces a method of the base class by one with a matching name and parameter list. The overloaded method matches in name but has a different parameter list.

Binary Search

- Idea: In searching an ordered list, first compare the target to the key in the center of the list. If it is smaller, restrict the search to the left half; otherwise restrict the search to the right half, and repeat. In this way, at each step we reduce the length of the list to be searched by half.
- Keep two indices, top and bottom, that will bracket the part of the list still to be searched.

Binary Search

- The target key, provided it is present, will be found between the indices bottom and top, inclusive.

- Initialization:
  - Set bottom = 0; top = the_list.size( ) - 1;
- Compare target with the Record at the midpoint,
  - mid = (bottom + top)/2;

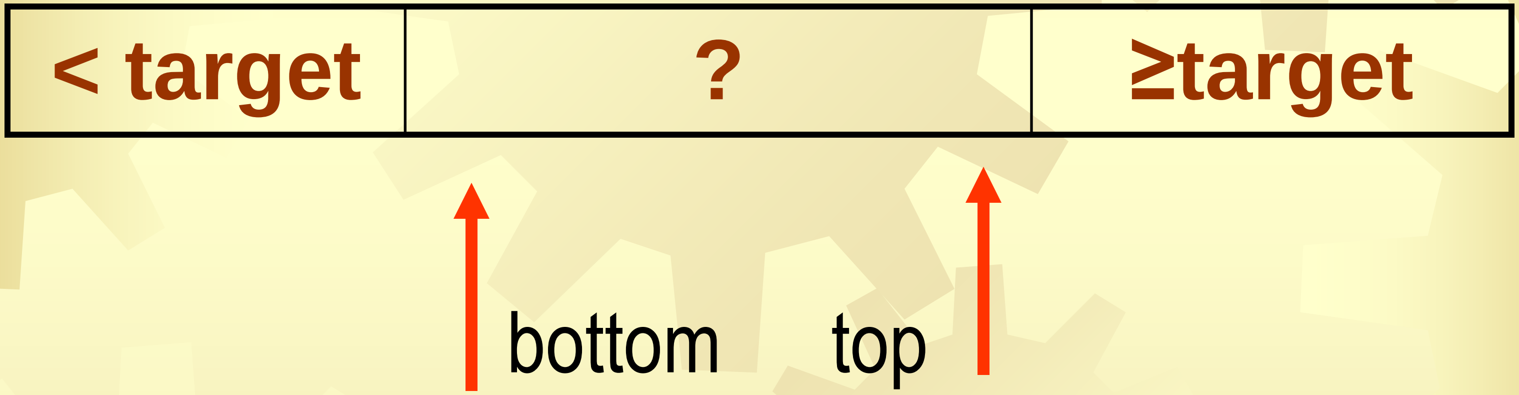
Binary Search

- Change the appropriate index top or bottom to restrict the search to the appropriate half of the list.
- Loop terminates when top≤ bottom, if it has not terminated earlier by finding the target.
- Make progress toward termination by ensuring that the number of items remaining to be searched, top - bottom + 1, strictly decreases at each iteration of the process.

```
{  Record data;
   if (bottom < top) { // List has more than one entry.
        int mid = (bottom + top)/2;
        the_list.retrieve(mid, data);
        if (data < target)          // Reduce to top half of 
list.
              return recursive_binary_1(the_list,
                          target, mid + 1, top, 
position);
        else                     // Reduce to bottom half of list.
              return recursive_binary_1(the_list,
                          target, bottom, mid, 
position);
}
```

Algorithm Verification

- The division of the list into sublists is described in the following diagram:



Algorithm Verification

- Only entries strictly less than target appear in the first part of the list, but the last part contains entries greater than or equal to target.
- When the middle part of the list is reduced to size 1, it will be guaranteed to be the first occurrence of the target if it appears more than once in the list.

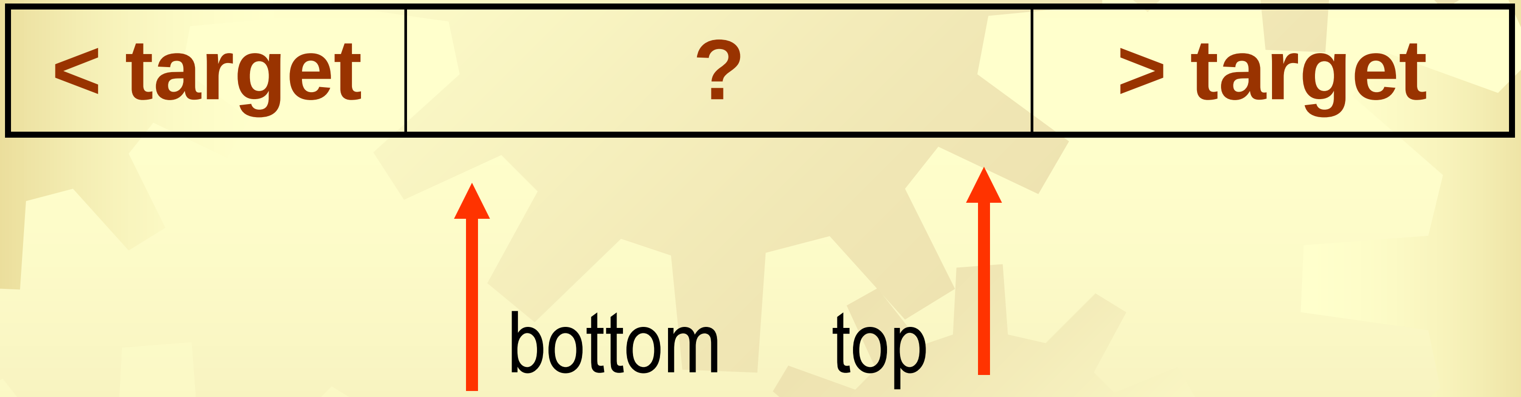
Algorithm Verification

- We must prove carefully that the search makes progress towards termination. This requires checking the calculations with indices to make sure that the size of the remaining sublist strictly decreases at each iteration. It is also necessary to check that the comparison of keys corresponds to the division into sublists in the above diagram.

```
if (bottom <= top) {
   int mid = (bottom + top)/2;
   the_list.retrieve(mid, data);
   if (data == target) {
        position = mid;  return success;}
   else if (data < target)
        return recursive binary 2(the_list, target,
                         mid + 1, top, position);
   else return recursive_binary_2(the_list,
   target,
                         bottom, mid - 1, position);}
```

Algorithm Verification

- **The division of the list into sublists is described in the following diagram:**



Algorithm Verification

- The first part of the list contains only entries strictly less than target, and the last part contains only entries strictly greater than target.
- If target appears more than once in the list, then the search may return any instance of the target.
- Proof of progress toward termination is easier than for the first method.

Comparison Trees: Definitions

- The ***comparison tree*** of an algorithm is obtained by tracing the action of the algorithm, representing each comparison of keys by a ***vertex*** of the tree (which we draw as a ***circle***). Inside the circle we put the index of the key against which we are comparing the target key.

Comparison Trees: Definitions

- ***Branches (lines)*** **drawn down from the circle represent the possible outcomes of the comparison. When the algorithm terminates, we put either *F* (for failure) or the location where the target is found at the end of the appropriate branch, which we call a *leaf*,and draw as a square.**

Comparison Trees: Definitions

- The remaining vertices are called the ***internal vertices*** of the tree. The number of comparisons done by an algorithm in a particular search is the number of internal vertices traversed in going from the top of the tree, called its ***root***, down the appropriate path to a leaf.

Comparison Trees: Definitions

- The number of branches traversed to reach a vertex from the root is called the ***level*** of the vertex. Thus the root itself has level 0, the vertices immediately below it have level 1, and so on. The largest level that occurs is called the ***height*** of the tree.

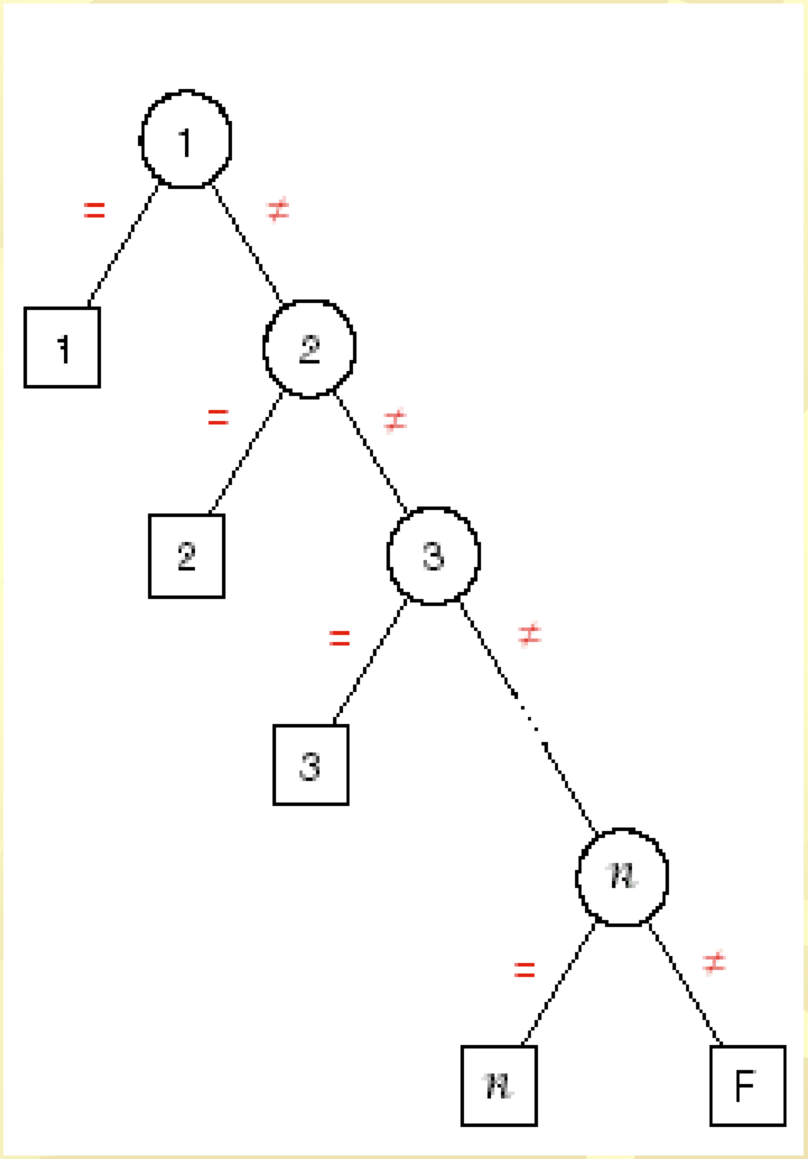
Comparison Trees: Definitions

- We call the vertices immediately below a vertex v the ***children*** of v and the vertex immediately above v the ***parent*** of v.
- The ***external path length*** of a tree is the sum of the number of branches traversed in going from the root once to every leaf in the tree.

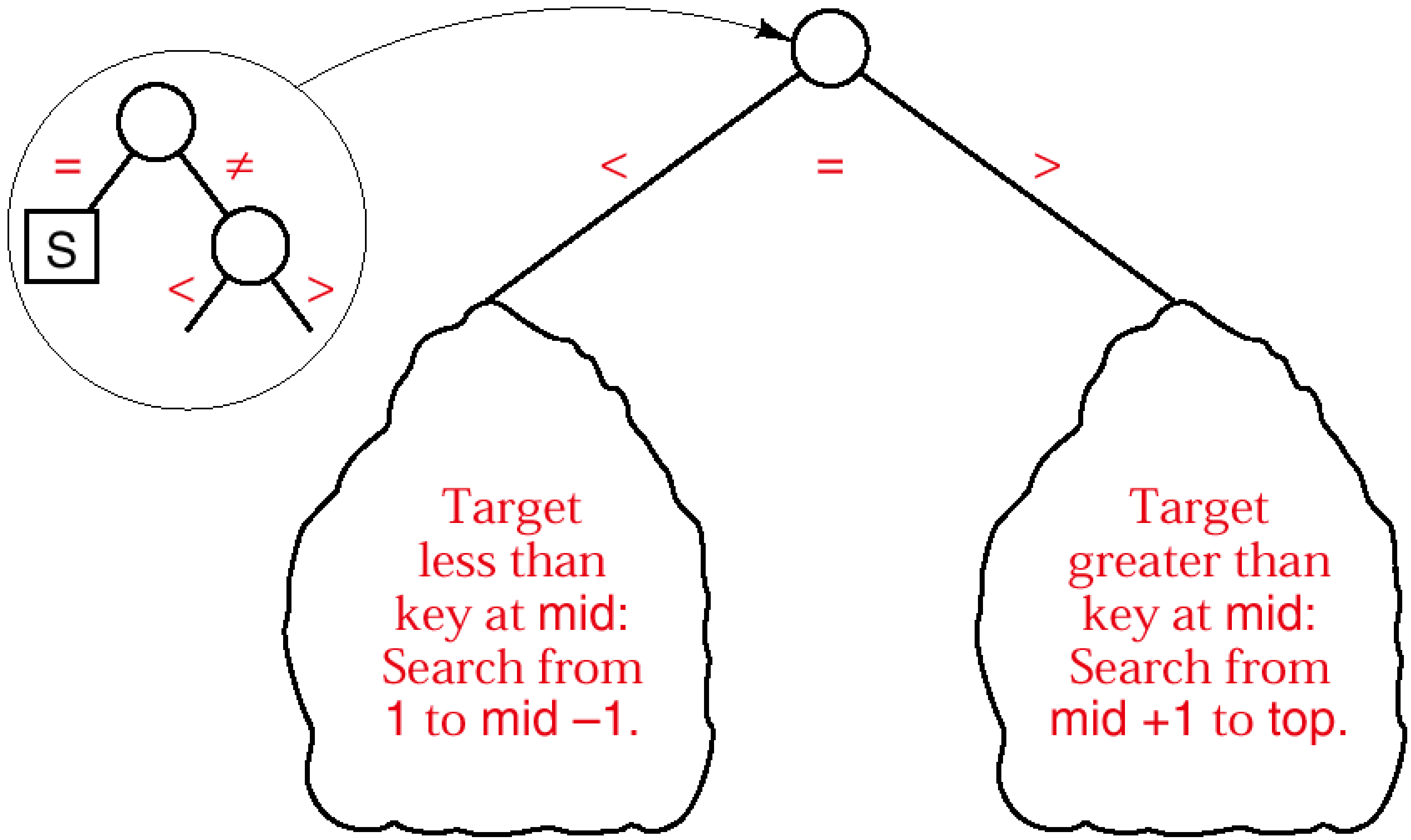
Comparison Trees: Definitions

- The ***internal path length*** is defined to be the sum, over all vertices that are not leaves, of the number of branches from the root to the vertex.

Sample Comparison Trees

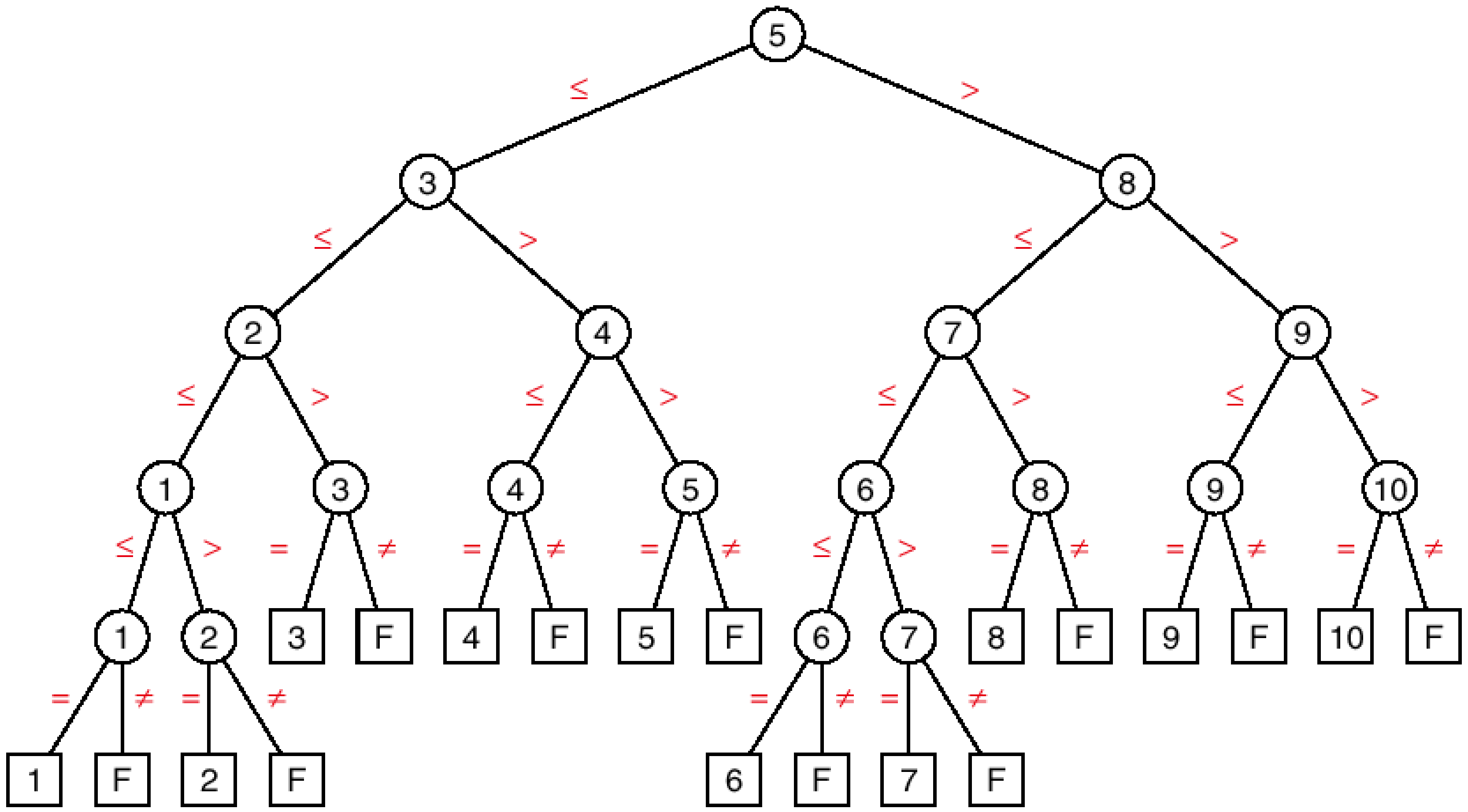


Sample Comparison Trees


=
≠
S
<
>
<
=
>
Target less than key at mid: Search from 1 to mid −1.
Target greater than key at mid: Search from mid +1 to top.

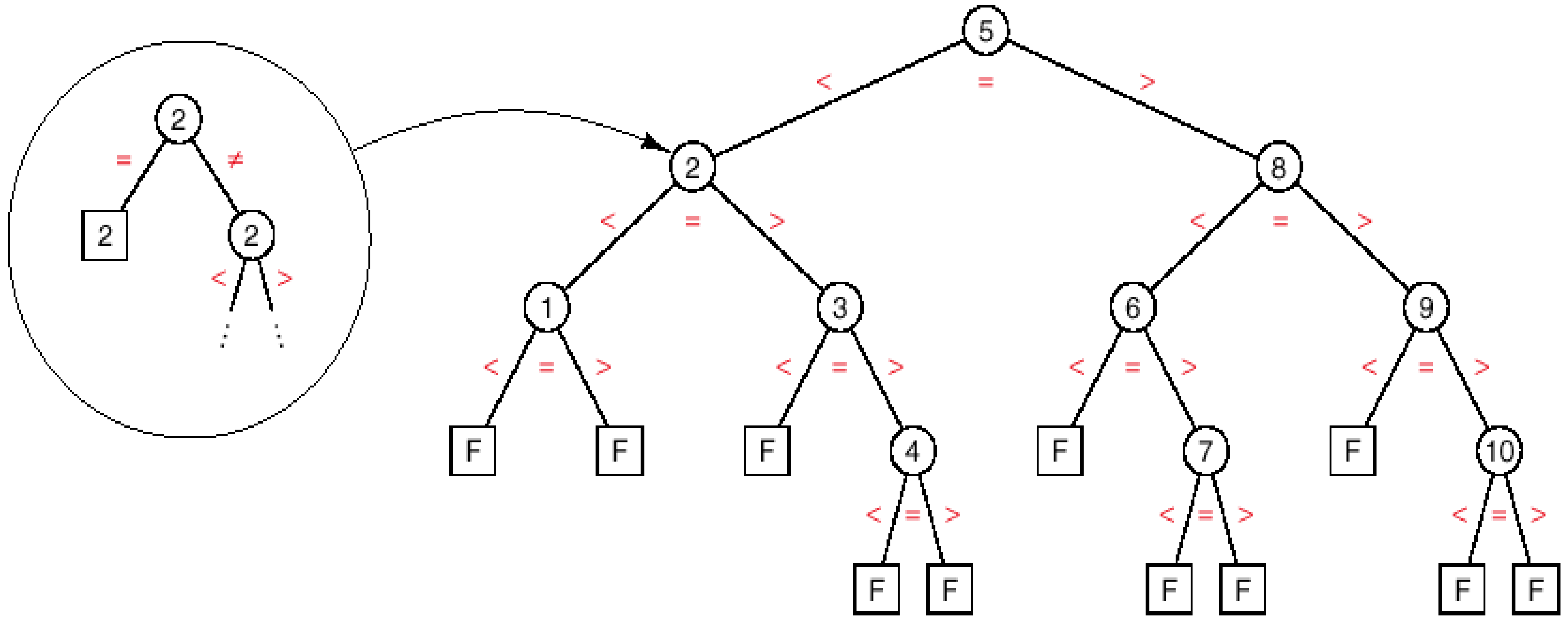
Comparison Trees for Binary Search

First version:



Comparison Trees for Binary Search

Second version:



2-Trees

- As *2-tree* is a tree in which every vertex except the leaves has exactly two children.
- *Lemma 7.1* The number of vertices on each level of a 2-tree is at most twice the number on the level immediately above. Hence, in a 2-tree, the number of vertices on level t is at most 2^t for $t\geq 0$.
- *Lemma 7.2* If a 2-tree has k vertices on level t , then $t\geq \lg k$,where lg denotes a logarithm with base 2.

Binary Search Analysis

- The number of comparisons of keys done by binary_search_1 in searching a list of n items is approximately

  lg n + 1

  in the worst case and

  lg n

  in the average case. The number of comparisons is essentially independent of whether the search is successful or not.

Binary Search Analysis

- The number of comparisons done in an unsuccessful search by binary_search_2 is approximately 2lg(n + 1).
- In a successful search of a list of n entries, binary search 2 does approximately

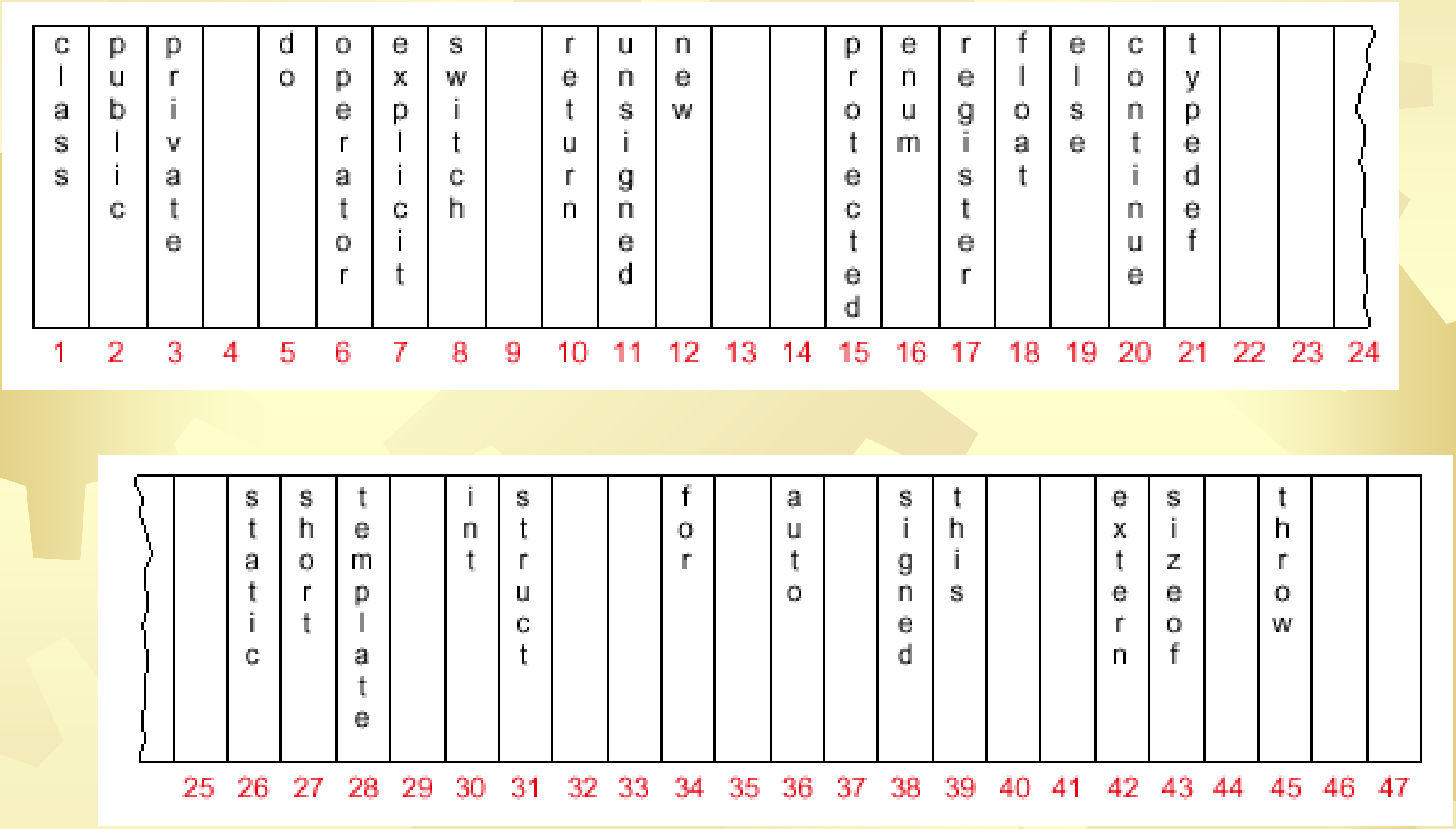
$$\frac{2(n+1)}{n}\lg(n+1)-3$$

comparisons of keys.

Binary Search Analysis

	Successful search	*Unsuccessful search*
binary_search _1	lg n +1	lg$n$ + 1
binary_search _2	2 lg n -3	2lg$n$

Hash Tables



1	class
2	public
3	private
4	
5	do
6	operator
7	explicit
8	switch
9	
10	return
11	unsigned
12	new
13	
14	
15	protected
16	enum
17	register
18	float
19	else
20	continue
21	typedef
22	
23	
24	
25	
26	static
27	short
28	template
29	
30	int
31	struct
32	
33	
34	for
35	
36	auto
37	
38	signed
39	this
40	
41	
42	extern
43	sizeof
44	
45	throw
46	
47	

Hash Tables

- Start with an ***array*** that holds the hash table.
- Use a ***hash function*** to take a key and map it to some index in the array. This function will generally map several different keys to the same index.

Hash Tables

- If the desired record is in the location given by the index, then we are finished; otherwise we must use some method to resolve the *collision* that may have occurred between two records wanting to go to the same location.
- To use hashing we must
  - find good hash functions and
  - determine how to resolve collisions.

Choosing a Hash Function

- A hash function should be easy and quick to compute.
- A hash function should achieve an even distribution of the keys that actually occur across the range of indices.
- The usual way to make a hash function is to take the key, chop it up, mix the pieces together in various ways, and thereby obtain an index that will be uniformly distributed over the range of indices.

Choosing a Hash Function

- Note that there is nothing random about a hash function. If the function is evaluated more than once on the same key, then it must give the same result every time, so the key can be retrieved without fail.

Choosing a Hash Function

- ***Truncation:*** Sometimes we ignore part of the key, and use the remaining part as the index.
- ***Folding:*** We may partition the key into several parts and combine the parts in a convenient way.
- ***Modular arithmetic:*** We may convert the key to an integer, divide by the size of the index range, and take the remainder as the result.
- A better spread of keys is often obtained by taking the size of the table (the index range) to be a *prime number.*

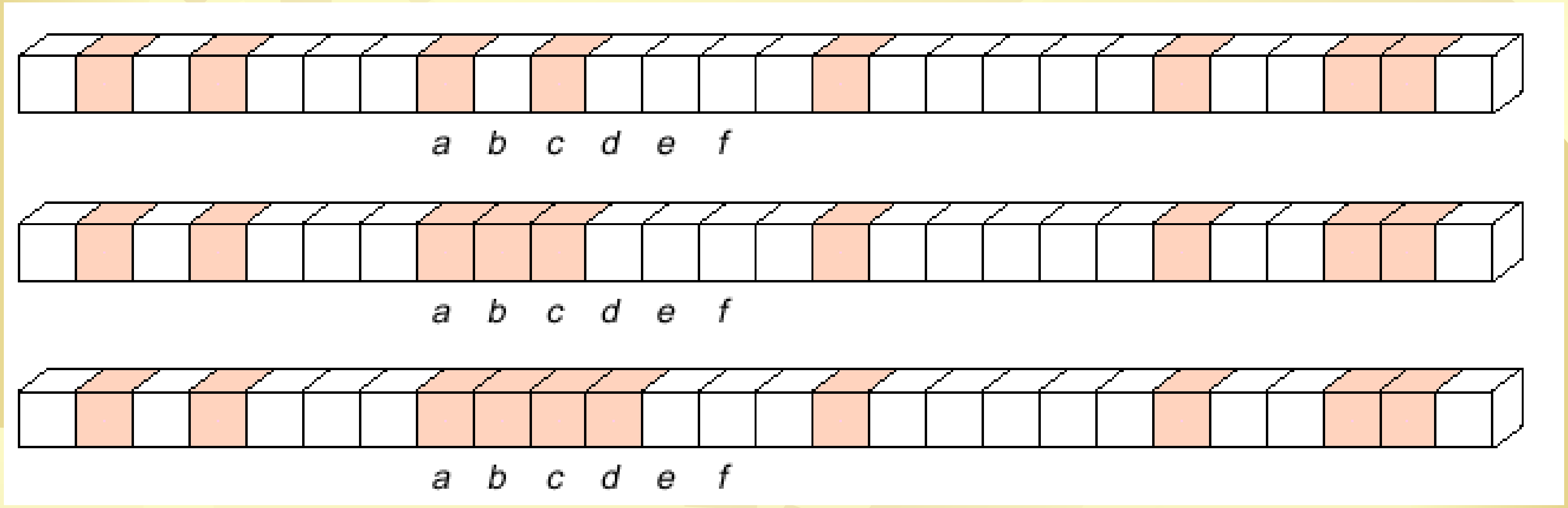
Collision Resolution with Open Addressing

- Linear Probing:

  Linear probing starts with the hash address and searches sequentially for the target key or an empty position. The array should be considered circular, so that when the last location is reached, the search proceeds to the first location of the array.

Collision Resolution with Open Addressing

- **Clustering:**



Collision Resolution with Open Addressing

- Quadratic Probing:

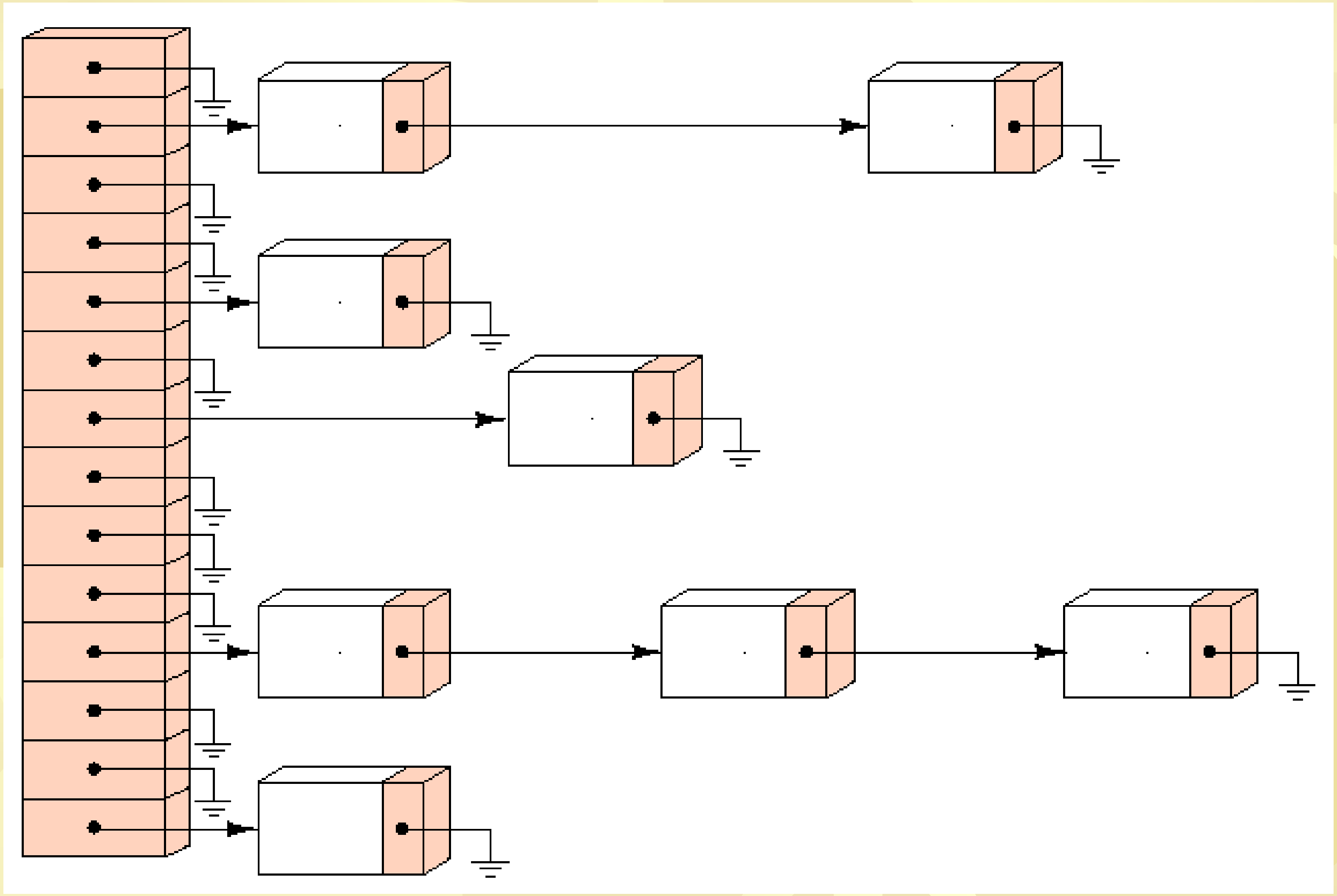
  If there is a collision at hash address h, quadratic probing goes to locations h+1, h + 4, h + 9, ... , that is, at locations h + i^2 (mod hashsize) for i = 1, 2,.... .

- Other methods:
  - Key-dependent increments;
  - Random probing.

Hash Table Insertion

- **Error_code Hash_table :: insert(const Record &new_entry)**
- **/* Post: If the Hash_table is full, a code of overflow is returned. If the table already contains an item with the key of new_ entry a code of duplicate_error is returned. Otherwise: The Record new_ entry is inserted into the Hash_table and success is returned.**
- **Uses: Methods for classes Key , and Record . The function hash . */**

Chained Hash Tables



Chained Hash Tables

- The linked lists from the hash table are called ***chains***.
- If the records are large, a chained hash table can save space.
- Collision resolution with chaining is simple; clustering is no problem.
- The hash table itself can be smaller than the number of records; overflow is no problem.
- Deletion is quick and easy in a chained hash table.
- If the records are very small and the table nearly full, chaining may take more space.

The Birthday Surprise

- How many randomly chosen people need to be in a room before it becomes likely that two people will have the same birthday (month and day)?

The Birthday Surprise

- The probability that m people all have different birthdays is

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

- This expression becomes less than 0.5 whenever m≥ 23.
- For hashing, the birthday surprise says that for any problem of reasonable size, collisions will almost certainly occur.

Analysis of Hashing

- A probe is one comparison of a key with the target.
- The *load factor* of the table is $\lambda = n/t$, where n positions are occupied out of a total of t positions in the table.
- Retrieval from a chained hash table with load factor λ requires approximately $1 + \frac{1}{2}\lambda$ probes in the successful case and λ probes in the unsuccessful case.

Analysis of Hashing

- **Retrieval from a hash table with open addressing, random probing, and load factor requires approximately**

$$\frac{1}{\lambda}\ln\frac{1}{1-\lambda}$$

probes in the successful case and

1/(1 – λ)

probes in the unsuccessful case.

Analysis of Hashing

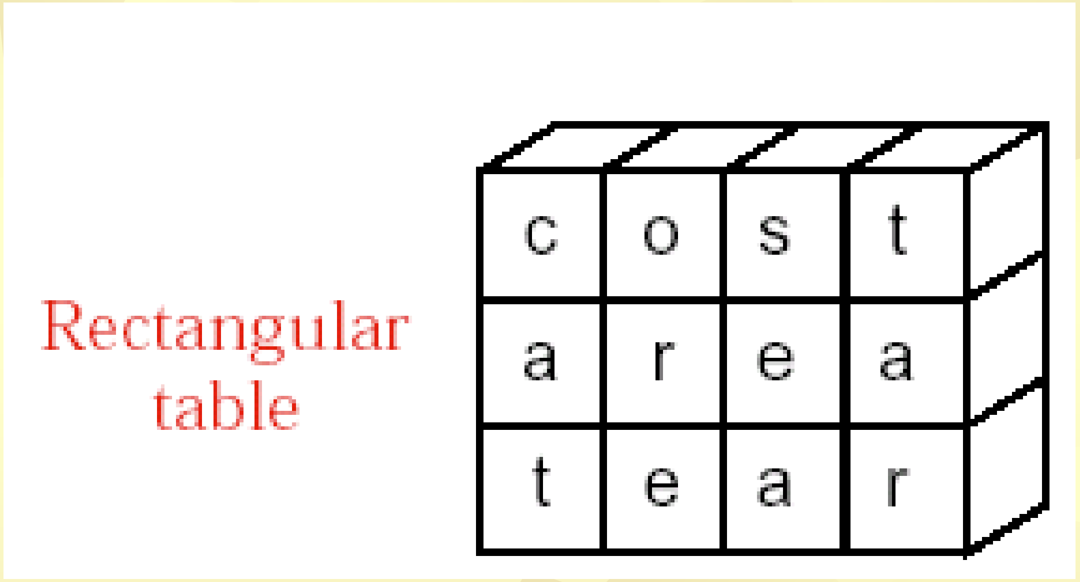
- Retrieval from a hash table with open addressing, linear probing, and load factor λrequires approximately

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

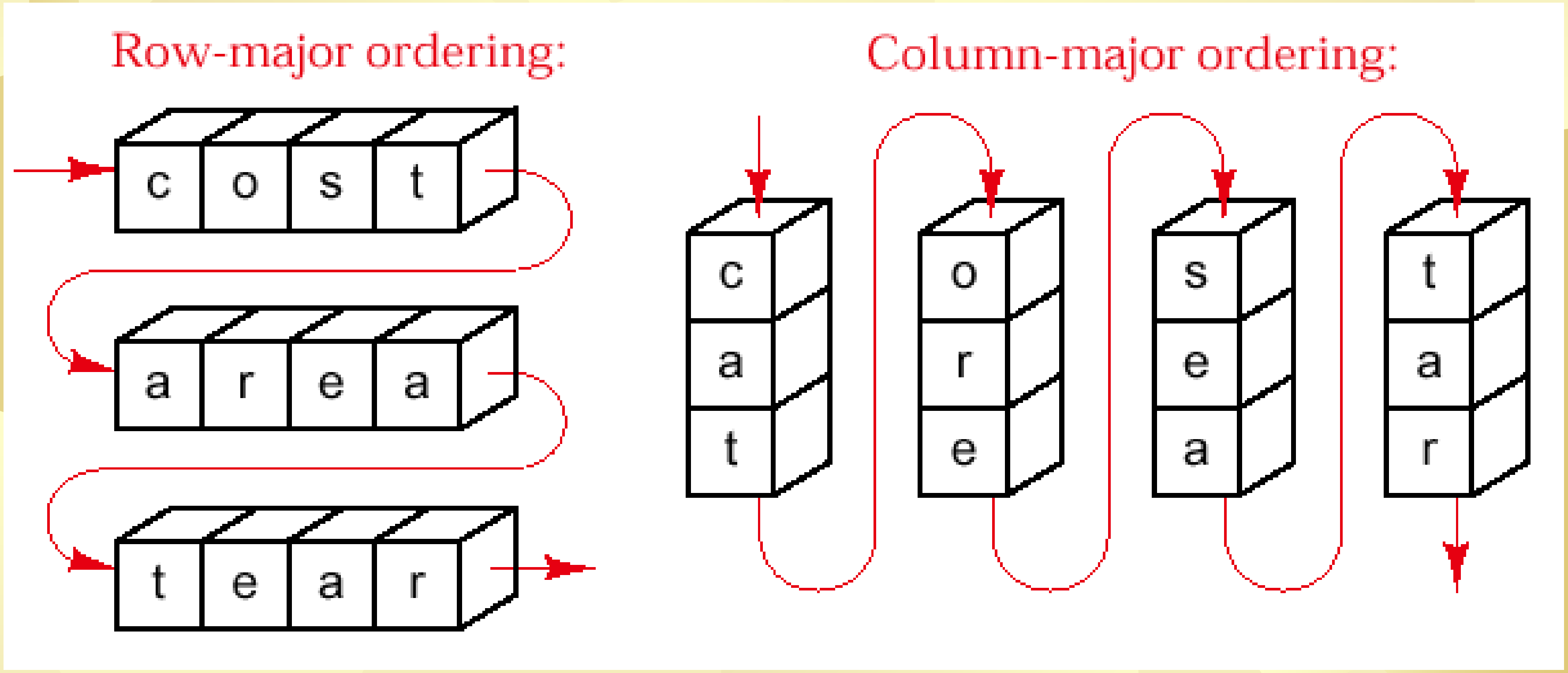
- probes in the successful case and in the unsuccessful case, respectively.

中国南开大学 Nankai University 1919–

Rectangular Arrays

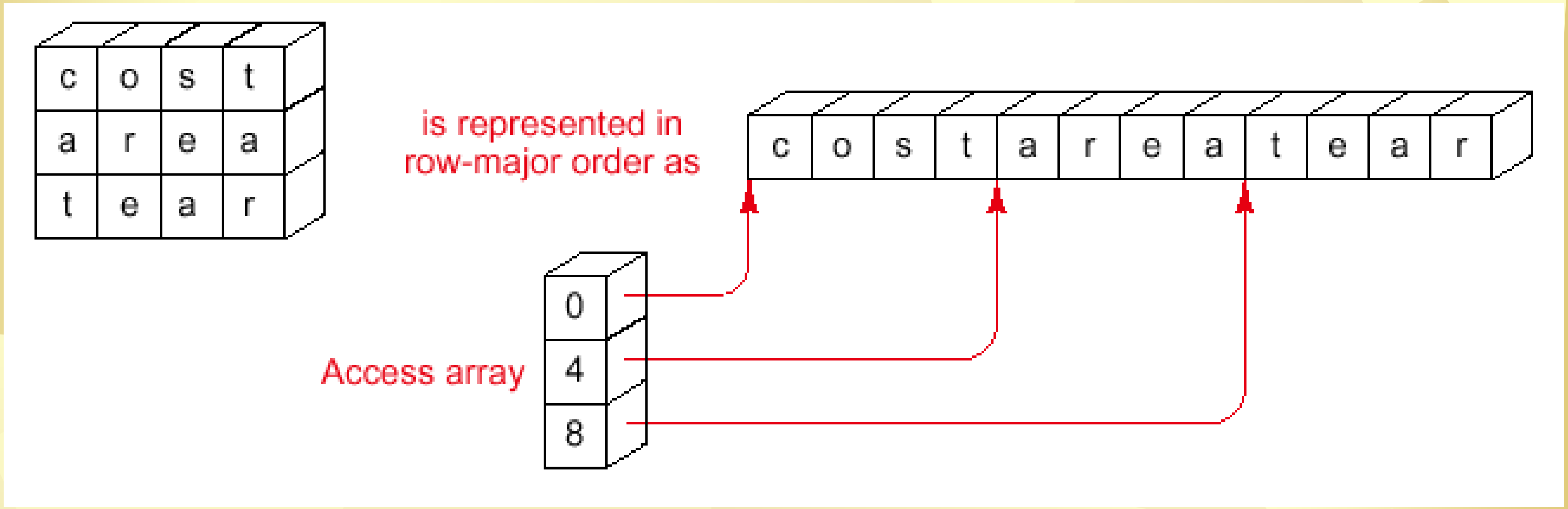


Rectangular Arrays

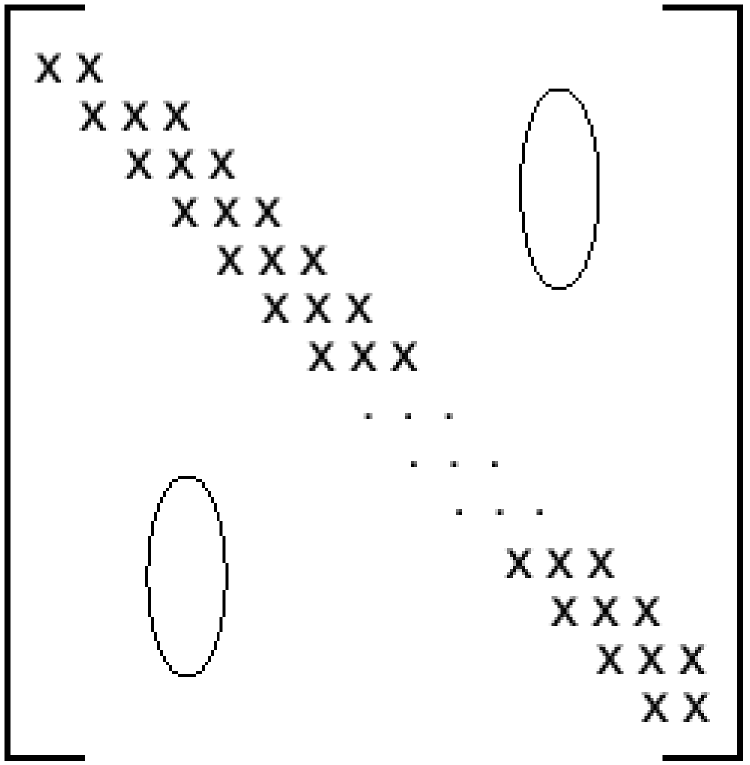


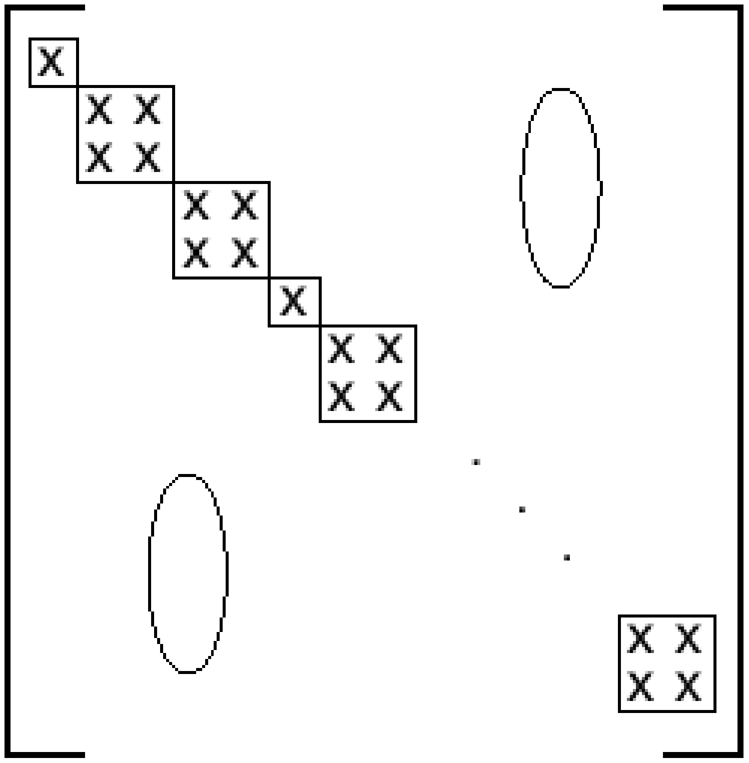
Rectangular Arrays

- In row-major ordering, entry [i, j] goes to position ni+j .

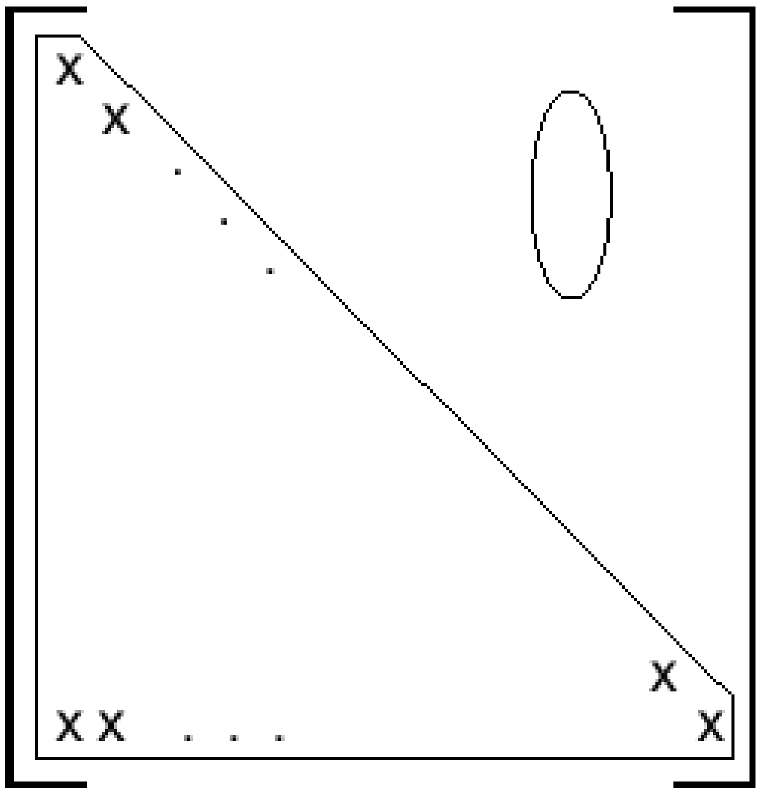


Matrices of Various Shapes

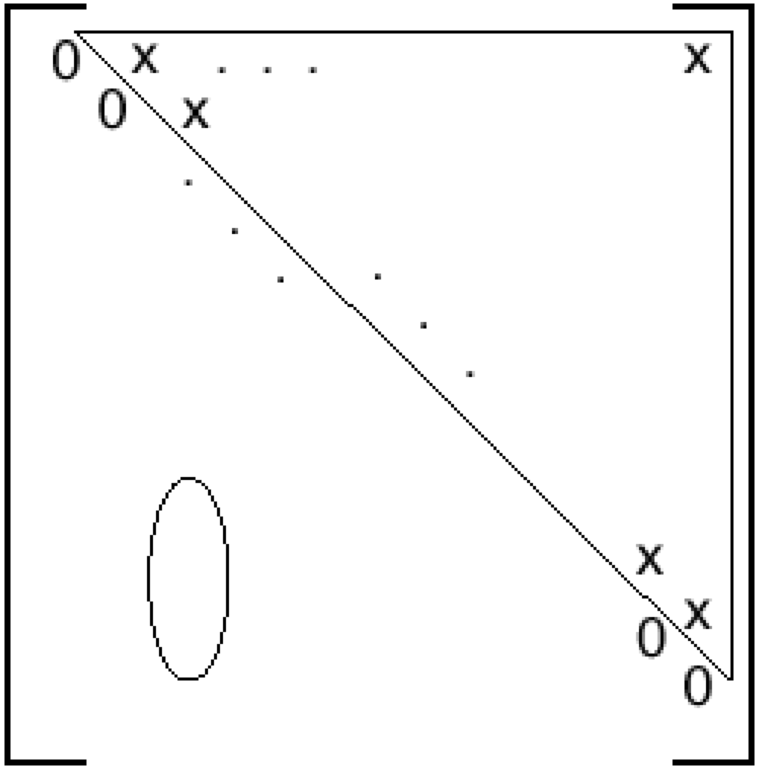

Tri-diagonal matrix


Block diagonal matrix

Matrices of Various Shapes

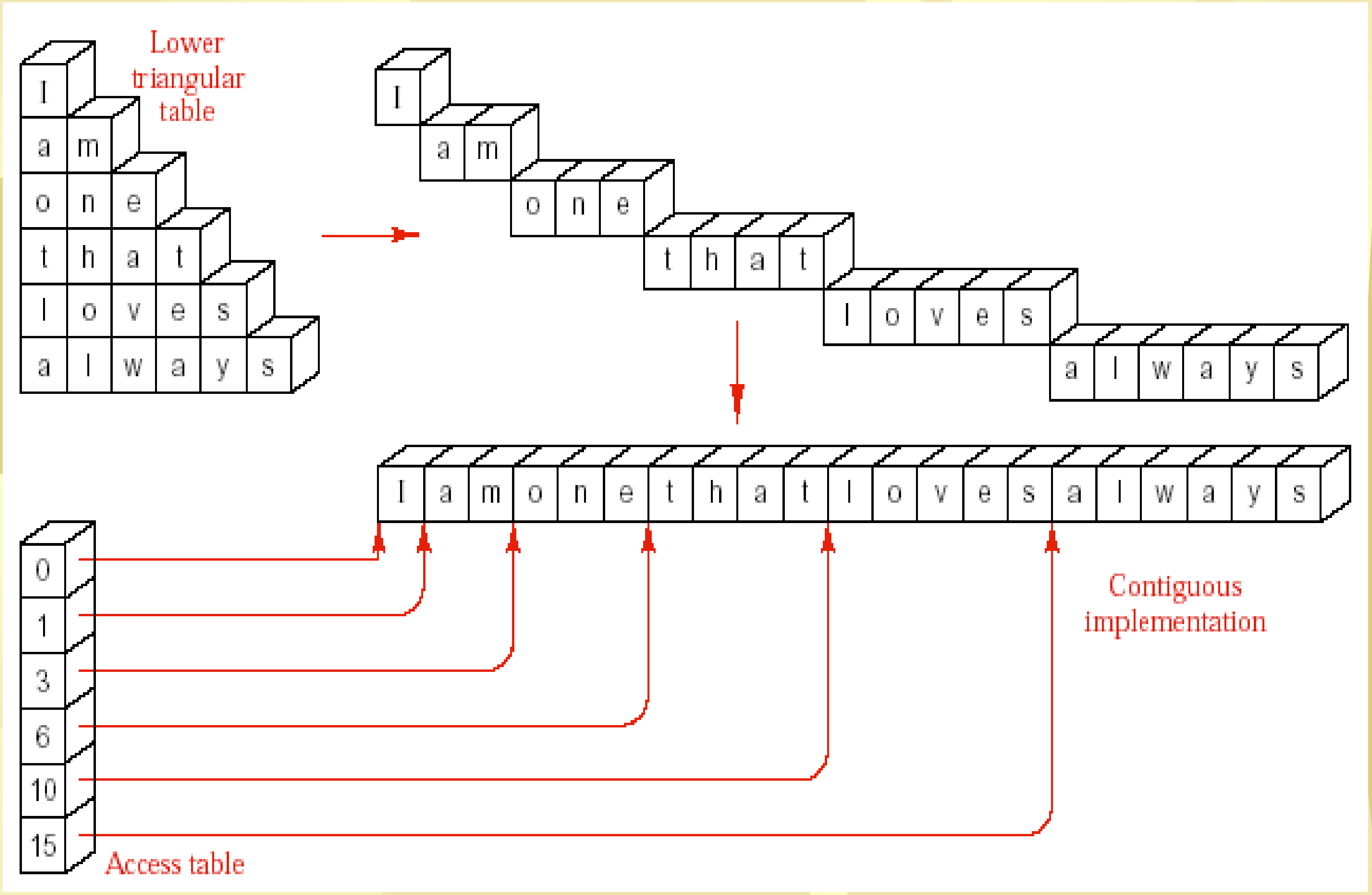


Lower triangular matrix

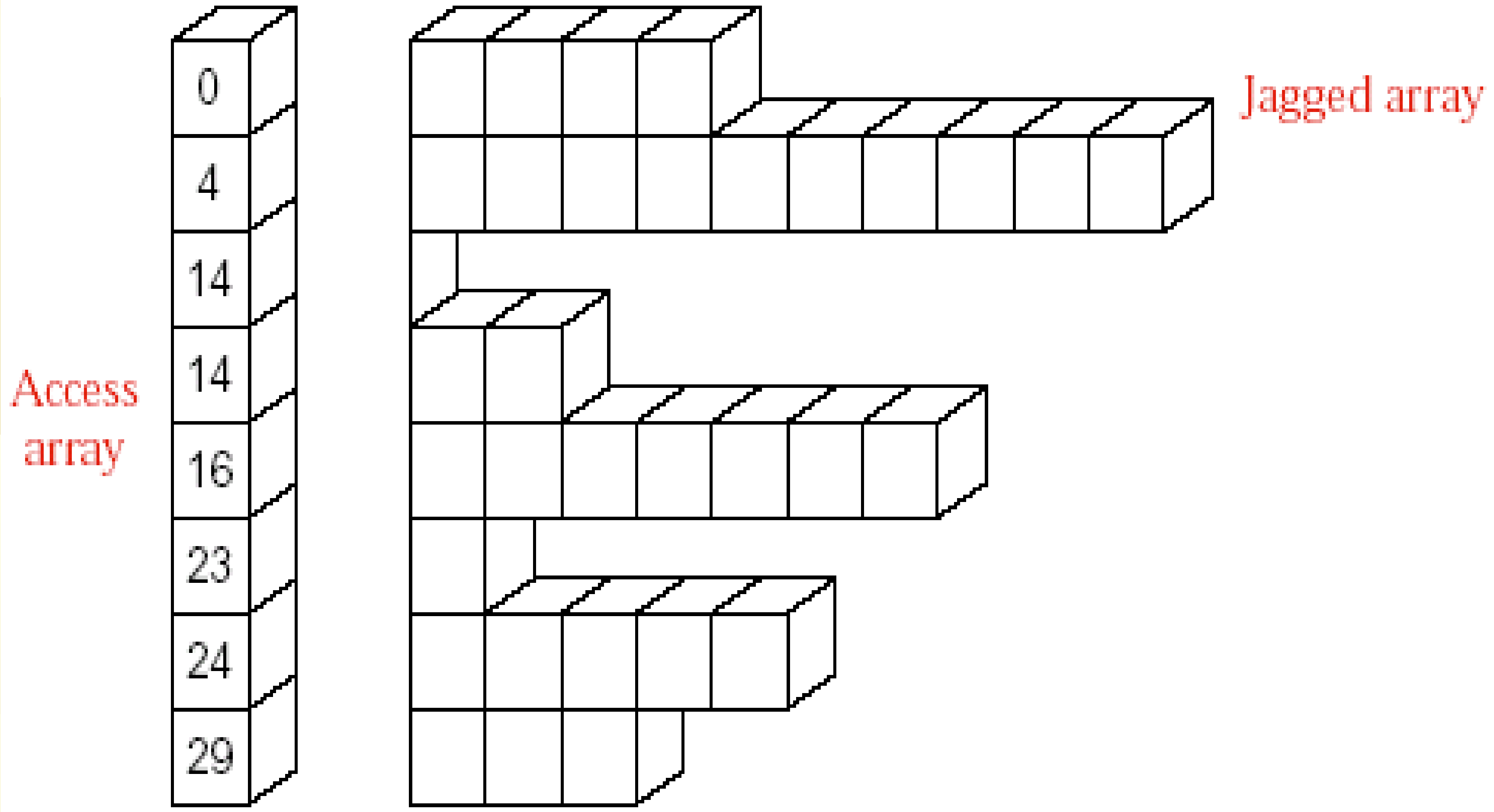


Strictly upper triangular matrix

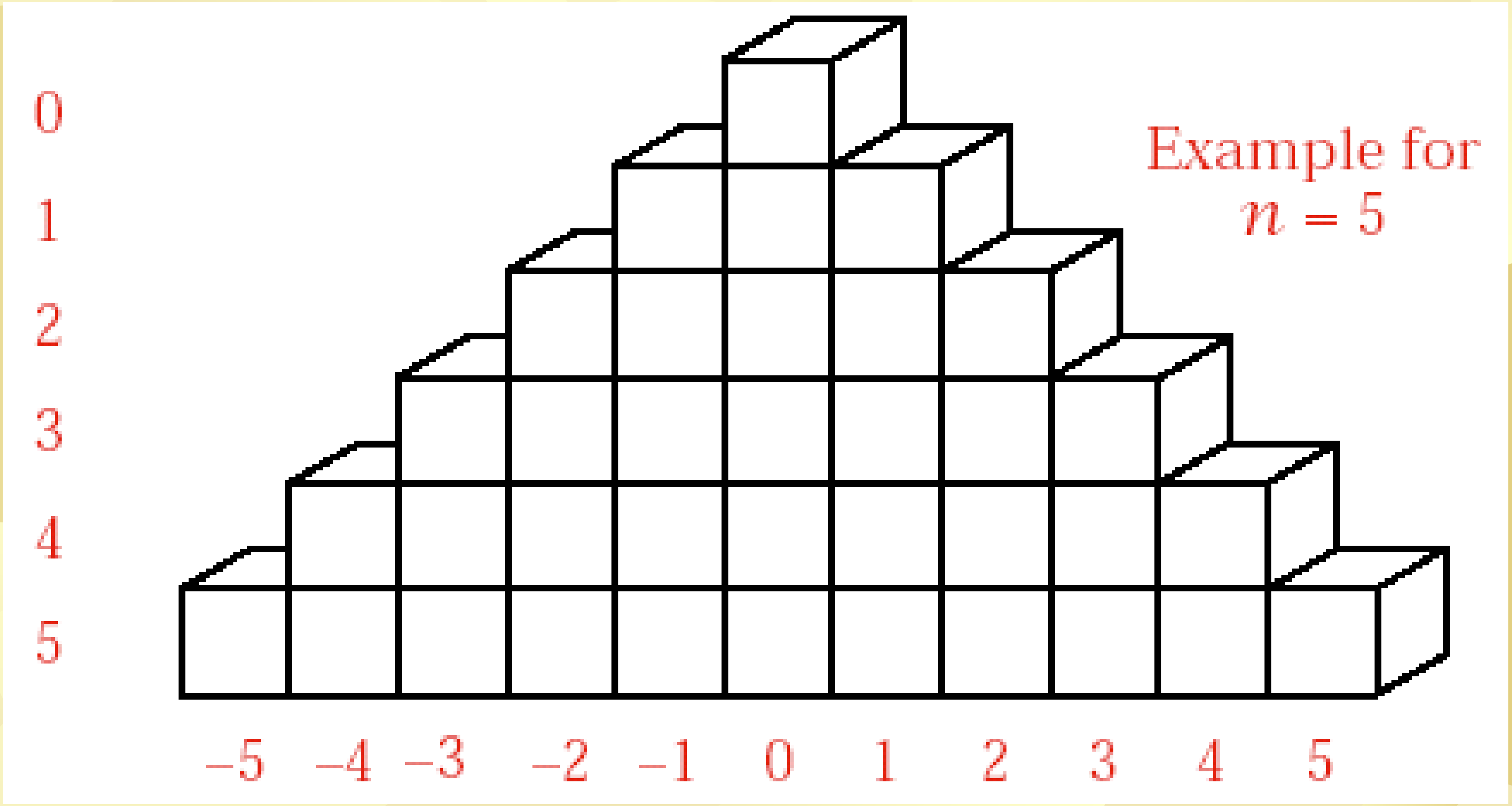
Matrices of Various Shapes



Jagged Table with Access Table



Symmetrically Triangular Table



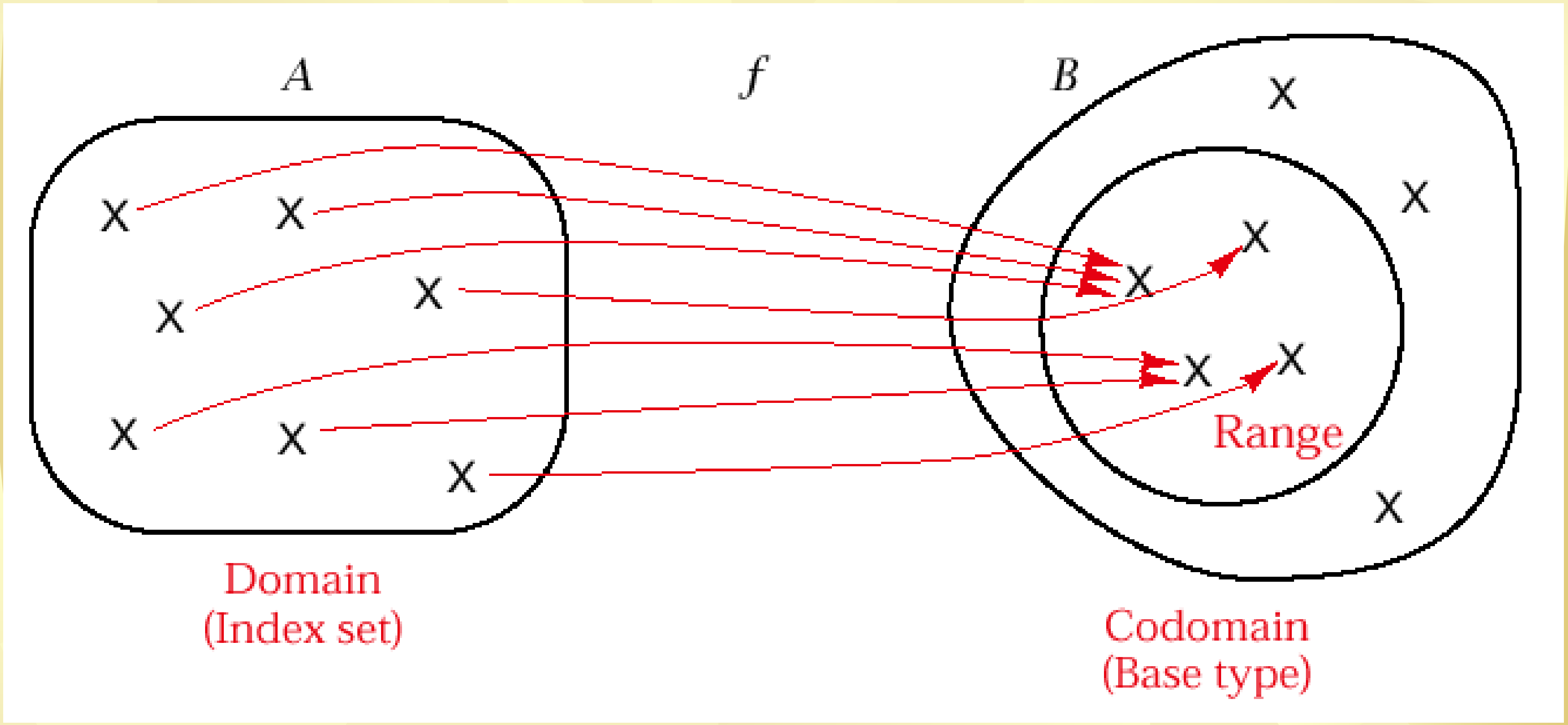
Inverted Table

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 Tables

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

Functions



Functions

- In mathematics a ***function*** is defined in terms of two sets and a correspondence from elements of the first set to elements of the the second. If ***f*** is a function from a set A to a set B, then ***f*** assigns to each element of A a unique element of B.
- The set A is called the ***domain*** of f , and the set B is called the ***codomain*** of f .
- The subset of B containing just those elements that occur as values of f is called the ***range*** of f .

Functions

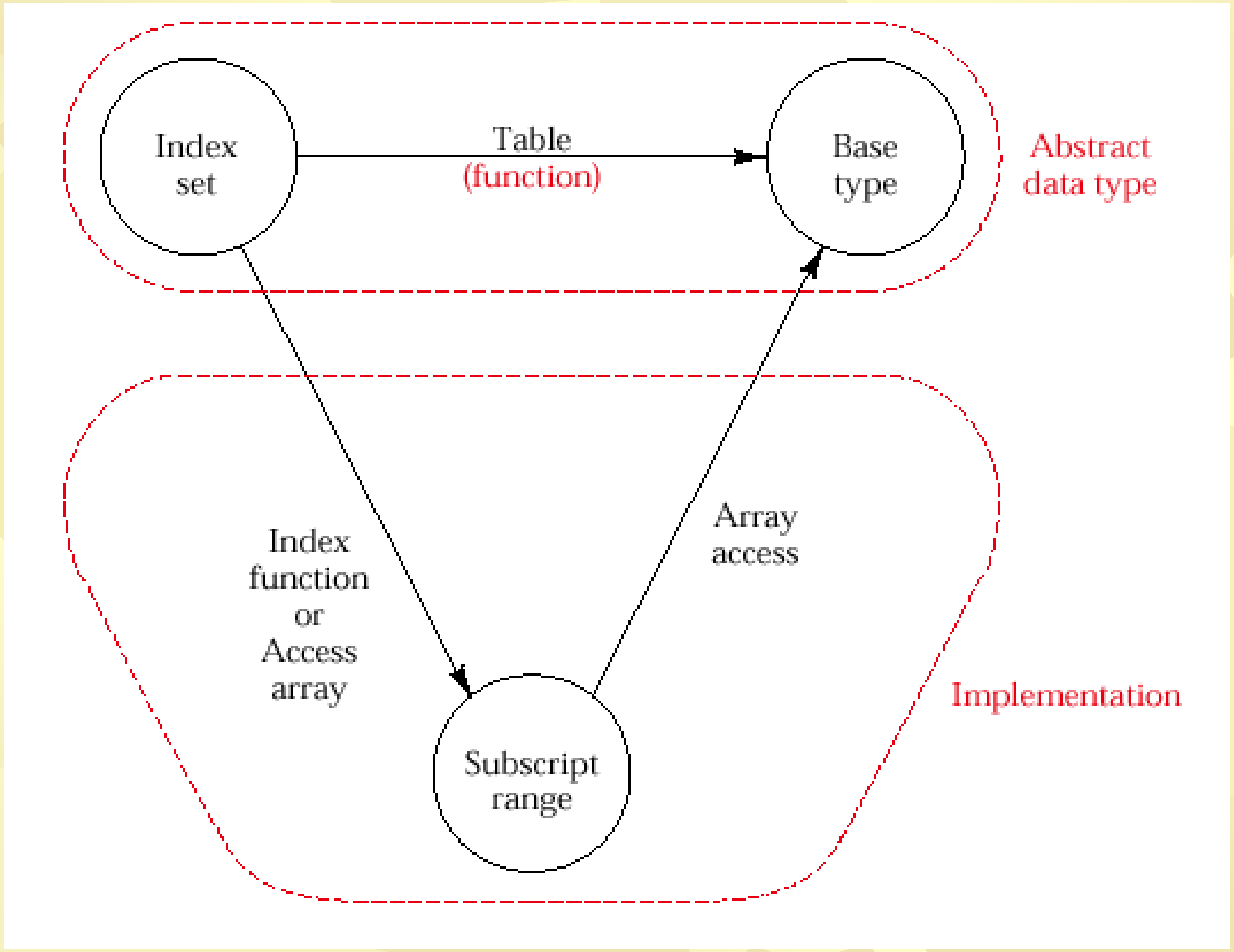
- DEFINITION: A ***table*** with index set I and base type T is a function from I to T together with the following operations.

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.

Functions

4. **Clearing:** **Remove all elements from the index set I , so there is no remaining domain.**
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.**

Functions


Index set
Table (function)
Base type
Abstract data type
Index function or Access array
Array access
Implementation
Subscript range

Conclusions: Comparison of Methods

- We have studied four principal methods of information retrieval, the first two for lists and the second two for tables. Often we can choose either lists or tables for our data structures.
- Sequential search is O(n).
  - Sequential search is the most flexible method. The data maybe stored in any order, with either contiguous or linked representation.
- Binary search is O(log n).
  - Binary search demands more, but is faster: The keys must be in order, and the data must be in random-access representation (contiguous storage).

Conclusions: Comparison of Methods

- Table lookup is O(1).
  - Ordinary lookup in contiguous tables is best, both in speed and convenience, unless a list is preferred, or the set of keys is sparse, or insertions or deletions are frequent.
- Hash-table retrieval is O(1).
  - Hashing requires the most structure, a peculiar ordering of the keys well suited to retrieval from the hash table, but generally useless for any other purpose. If the data are to be available for human inspection, then some kind of order is needed, and a hash table is inappropriate.