Data abstraction, revisited

- Design tradeoffs:
  - Speed vs robustness
    modularity
    ease of maintenance

- Table abstract data type: 3 versions

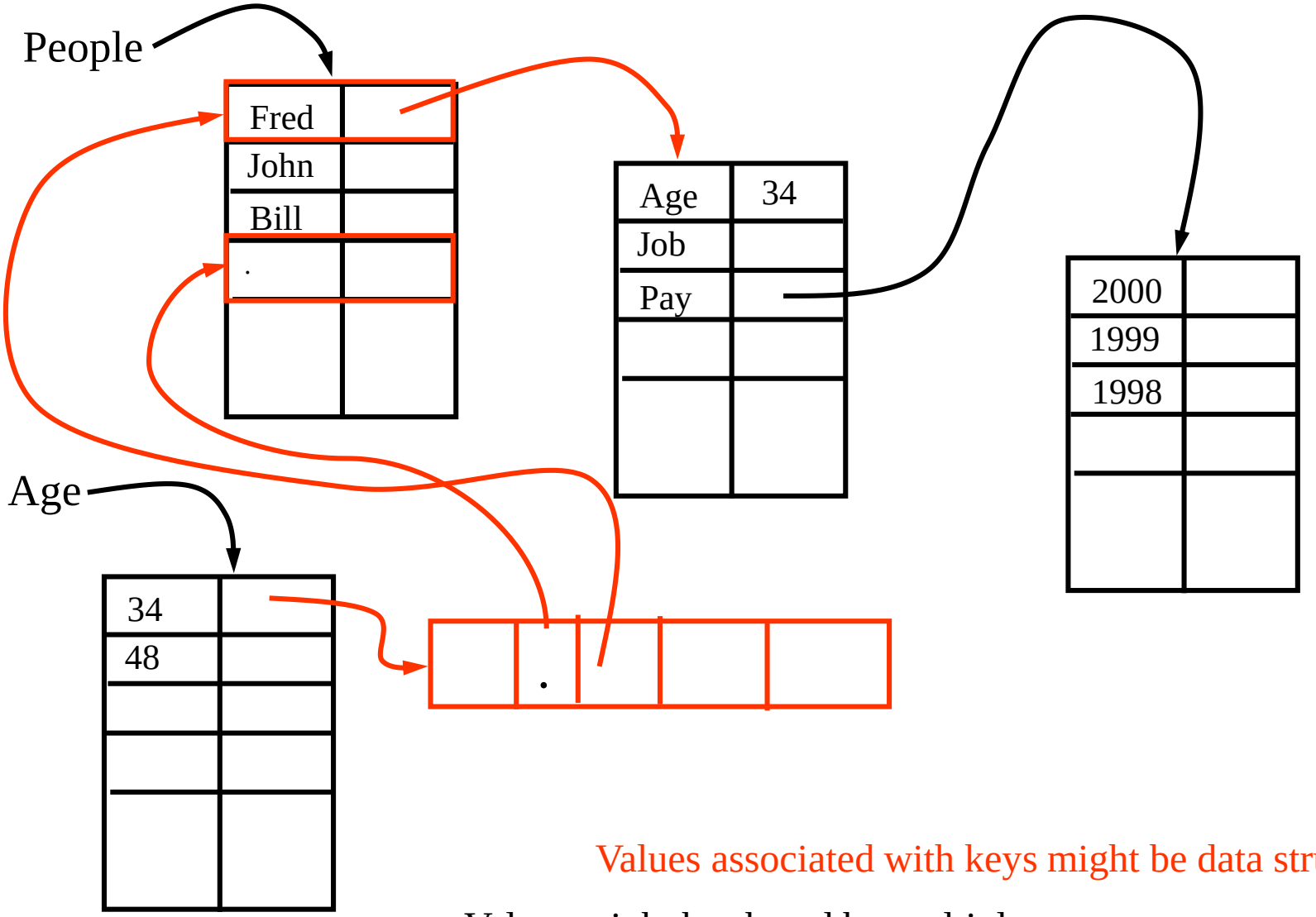
- No implementation of an ADT is necessarily "best"

- Abstract data types *hide information*, in types as well as in the code

Table: a set of bindings

- binding: a pairing of a key and a value
- Abstract interface to a table:
  - make
    create a new table
  - put! key value
    insert a new binding
    replaces any previous binding of that key
  - get key
    look up the key, return the corresponding value
- This definition IS the table abstract data type
  - Code shown later is a particular implementation of the ADT

Examples of using tables



Values associated with keys might be data structures

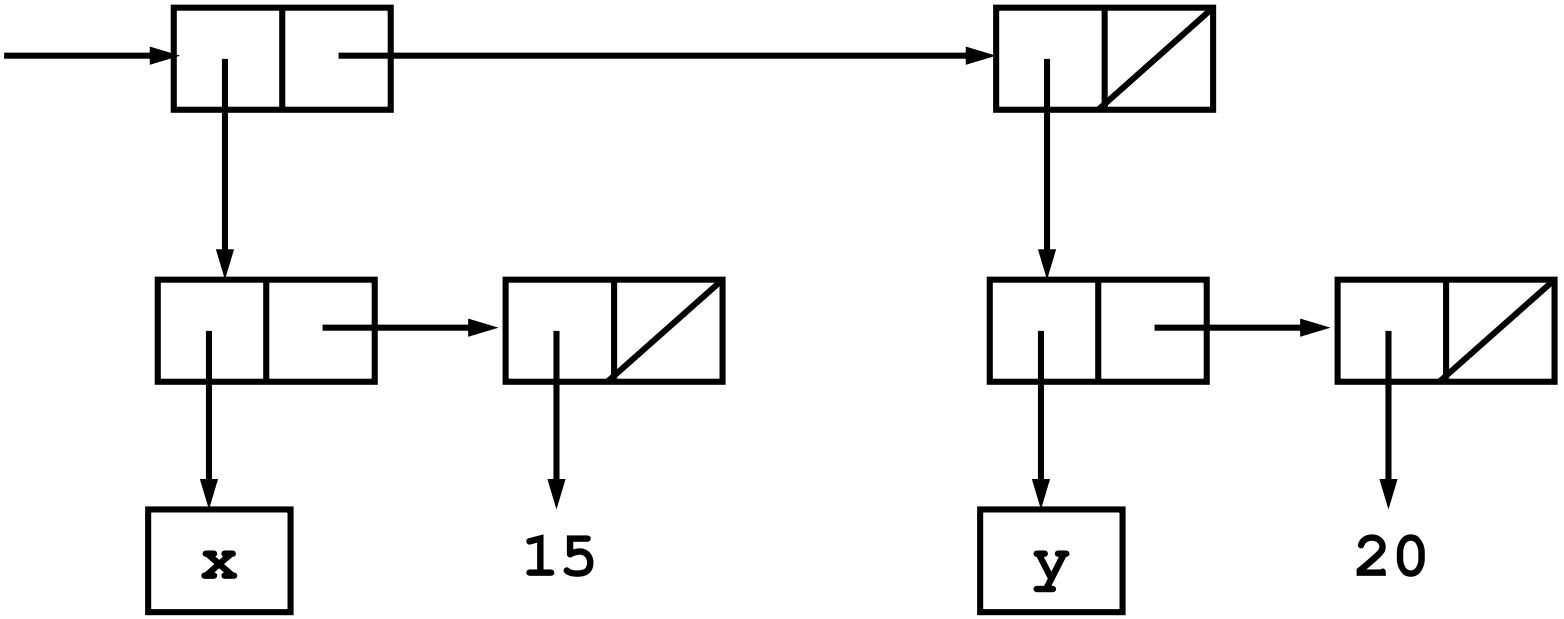
Values might be shared by multiple structures

Traditional LISP structure: association list

- A list where each element is a list of the key and value.
- Represent the table

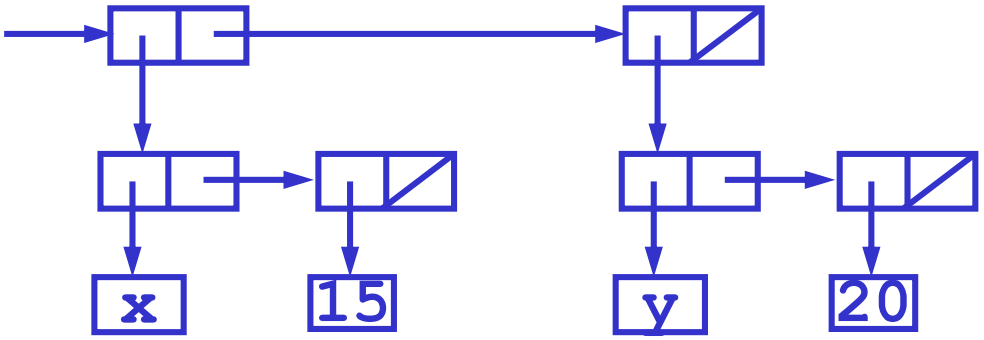
```
x: 15
y: 20
```

as the alist: `((x 15) (y 20))`



Alist operation: find-assoc

```
(define (find-assoc key alist)
  (cond
    ((null? alist) #f)
    ((equal? key (caar alist)) (cadar alist))
    (else (find-assoc key (cdr alist)))))

(define a1 '((x 15) (y 20)))
(find-assoc 'y a1)  ==> 20
```



An aside on testing equality

- = tests equality of numbers
- Eq? Tests equality of symbols
- Equal? Tests equality of symbols, numbers or lists of symbols and/or numbers that print the same

Alist operation: add-assoc

```
(define (add-assoc key val alist)
   (cons (list key val) alist))

(define a2 (add-assoc 'y 10 a1))

a2                    ==> ((y 10) (x 15) (y 20))

(find-assoc 'y a2) ==> 10
```

We say that the new binding for y
"shadows" the previous one

Alists are not an abstract data type

- Missing a constructor:
  - Used **`quote`** or **`list`** to construct

```
(define a1 '((x 15) (y 20)))
```

- There is no abstraction barrier: the implementation is exposed.

- User may operate on alists using standard list operations.

```
(filter (lambda (a) (< (cadr a) 16)) a1))
            ==> ((x 15))
```

Why do we care that Alists are not an ADT?

- Modularity is essential for software engineering
  - Build a program by sticking modules together
  - Can change one module without affecting the rest

- Alists have poor modularity
  - Programs may use list ops like **`filter`** and **`map`** on alists
  - These ops will fail if the implementation of alists change
  - Must change whole program if you want a different table

- To achieve modularity, hide information
  - Hide the fact that the table is implemented as a list
  - Do not allow rest of program to use list operations
  - ADT techniques exist in order to do this

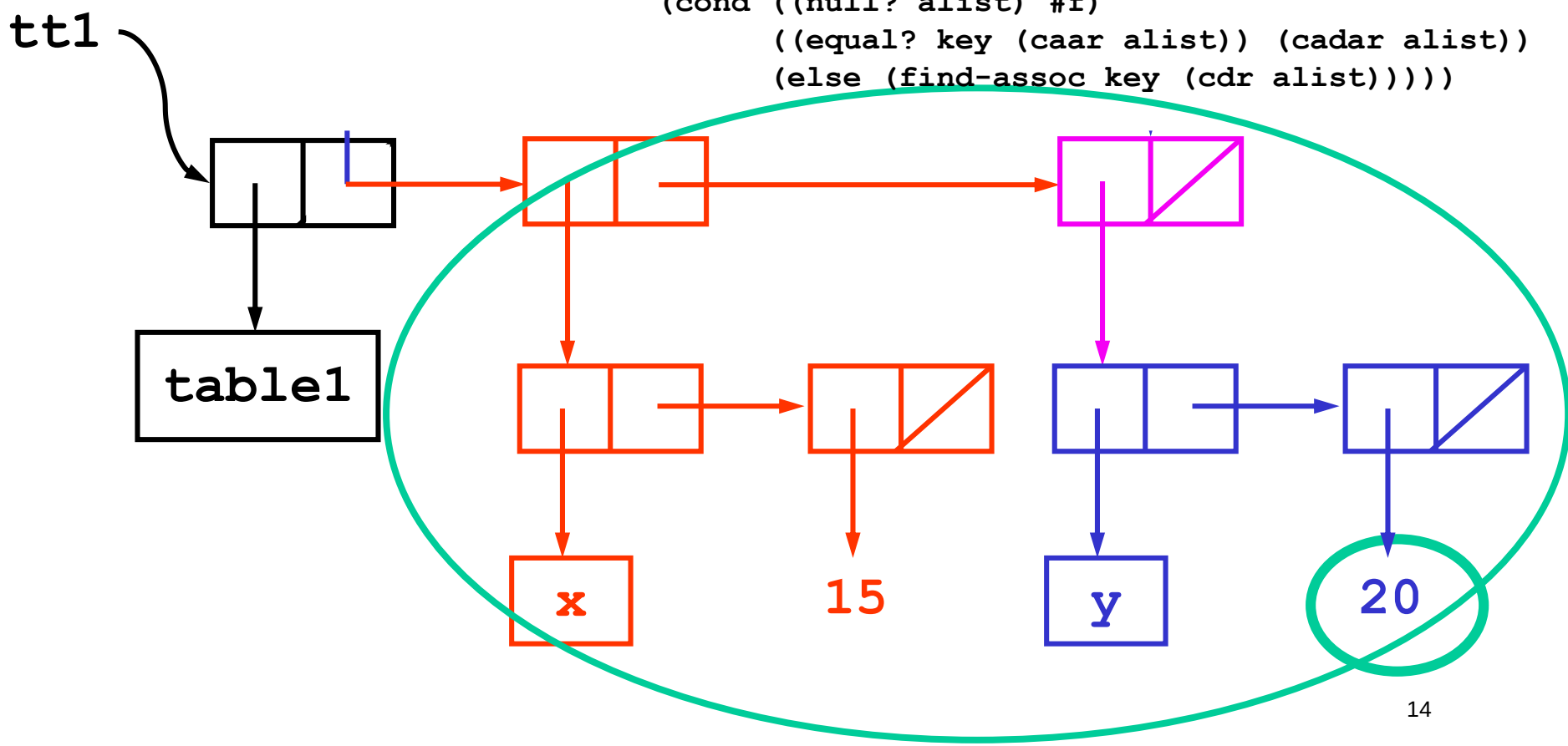
Table1: Table ADT (implemented as an Alist)

```
(define table1-tag 'table1)

(define (make-table1) (cons table1-tag nil))

(define (table1-get tbl key)
  (find-assoc key (cdr tbl)))

(define (table1-put! tbl key val)
  (set-cdr! tbl (add-assoc key val (cdr tbl))))
```

Table1 example

```
(define tt1 (make-table1))

(table1-put! tt1 'y 20)

(table1-put! tt1 'x 15)
(table1-get tt1 'y)
```

```
(define (table1-get tbl key)
  (find-assoc key (cdr tbl)))

(define (table1-put! tbl key val)
  (set-cdr! tbl
        (add-assoc key val (cdr tbl))))

(define (add-assoc key val alist)
  (cons (list key val) alist))

(define (find-assoc key alist)
  (cond ((null? alist) #f)
        ((equal? key (caar alist)) (cadar alist))
        (else (find-assoc key (cdr alist)))))
```



How do we know Table1 is an ADT implementation

- Potential reasons:
  - Because it has a type tag — No
  - Because it has a constructor — No
  - Because it has mutators and accessors — No

- Actual reason:
  - *Because the rest of the program does not apply any functions to Table1 objects other than the functions specified in the Table ADT*
  - For example, no **`car`**, **`cdr`**, **`map`**, **`filter`** done to tables

- The implementation (as an Alist) is hidden from the rest of the program, so it can be changed easily

Information hiding in types: opaque names

- Opaque: type name that is defined but unspecified
- Given functions **`m1`** and **`m2`** and unspecified type MyType:

```
(define (m1 number) ...)    ; number → MyType
(define (m2 myt) ...)       ; MyType → undef
```

- Which of the following is OK? Which is a type mismatch?

```
(m2 (m1 10))     ; return type of m1 matches
                 ; argument type of m2
(car (m1 10))    ; return type of m1 fails to match
                 ; argument type of car
                 ; car: pair<A,B> → A
```

- Effect of an opaque name:
  no functions have the correct types except the functions of the ADT

Types for table1

- Here is everything the rest of the program knows

Table1<k,v>	opaque type
`make-table1`	void → Table1<anytype,anytype>
`table1-put!`	Table1<k,v>, k, v → undef
`table1-get`	Table1<k,v>, k → (v | nil)

- Here is the hidden part, only the implementation knows it:

Table1<k,v> = symbol × Alist<k,v>

Alist<k,v> = list< k × v >

Lessons so far

- Association list structure can represent the table ADT
- The data abstraction technique (constructors, accessors, etc) exists to support information hiding
- Information hiding is necessary for modularity
- Modularity is essential for software engineering
- Opaque type names denote information hiding

Now let's talk about efficiency

- Speed of operations
  - put Fast
  - get Slow

- What if it's the Boston Yellow Pages?

  Really need to use other information to get to right place to search

Hash tables

- Suppose a program is written using Table1
- Suppose we measure that a lot of time is spent in **`table1-get`**

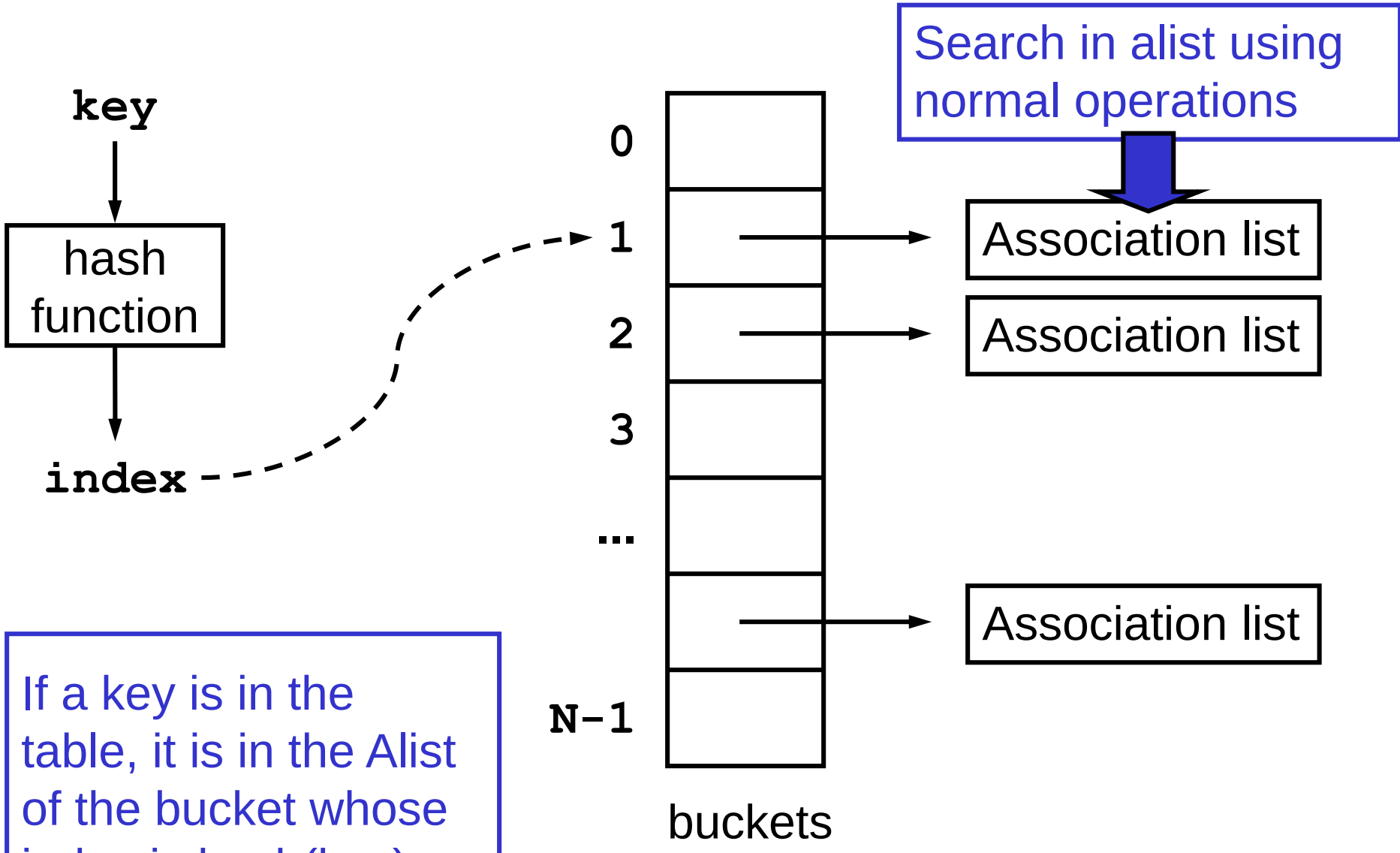
- Want to replace the implementation with a faster one

- Standard data structure for fast table lookup: hash table
- Idea:
  - keep N association lists instead of 1
  - choose which list to search using a hash function
    - given the key, hash function computes a number x where 0 <= x <= (N-1)
- Speed of hash table?

What's a hash function?

- Maps an input to a fixed length output (e.g. integer between 0 and N)
- Ideally the set of inputs is uniformly distributed over the output range
- Ideally the function is very rapid to compute
- Example:
  - First letter of last name:
    - 26 buckets
    - Non-uniform
  - Convert last name by position in alphabet, add, take modular arithmetic
    - GRIMSON: 7+18+9+13+19+15+14 = 95 (mod 26 = 17)
    - GREEN: 7+18+5+5+14=49 (mod 26 = 23)
- Uses:
  - Fast storage and retrieval of data
  - Hash functions that are hard to invert are very valuable in cryptography

Hash function output chooses a bucket



If a key is in the table, it is in the Alist of the bucket whose index is hash(key)

Store buckets using the vector ADT

- Vector: fixed size collection with indexed access

vector<A>	opaque type	Vector has constant speed access
`make-vector`	number, A → vector<A>	
`vector-ref`	vector<A>, number → A	
`vector-set!`	vector<A>,number, A → undef	

(make-vector size value) ==> a vector with size locations;
each initially contains value

(vector-ref v index) ==> whatever is stored at that index of v
(error if index >= size of v)

(vector-set! v index val) stores val at that index of v
(error if index >= size of v)

The Bucket Abstraction

```
(define (make-buckets N v) (make-vector N v))
(define make-buckets make-vector)
(define bucket-ref vector-ref)
(define bucket-set! vector-set!)
```

Table2: Table ADT implemented as hash table

```
(define t2-tag 'table2)
(define (make-table2 size hashfunc)
   (let ((buckets (make-buckets size nil)))
     (list t2-tag size hashfunc buckets)))
(define (size-of tbl) (cadr tbl))
(define (hashfunc-of tbl) (caddr tbl))
(define (buckets-of tbl) (cadddr tbl))
```

- For each function defined on this slide, is it
  - a constructor of the data abstraction?
  - an accessor of the data abstraction?
  - an operation of the data abstraction?
  - none of the above?

get in table2

```
(define (table2-get tbl key)
  (let ((index
         ((hashfunc-of tbl) key (size-of tbl))))
    (find-assoc key
          (bucket-ref (buckets-of tbl) index))))
```

- Same type as table1-get

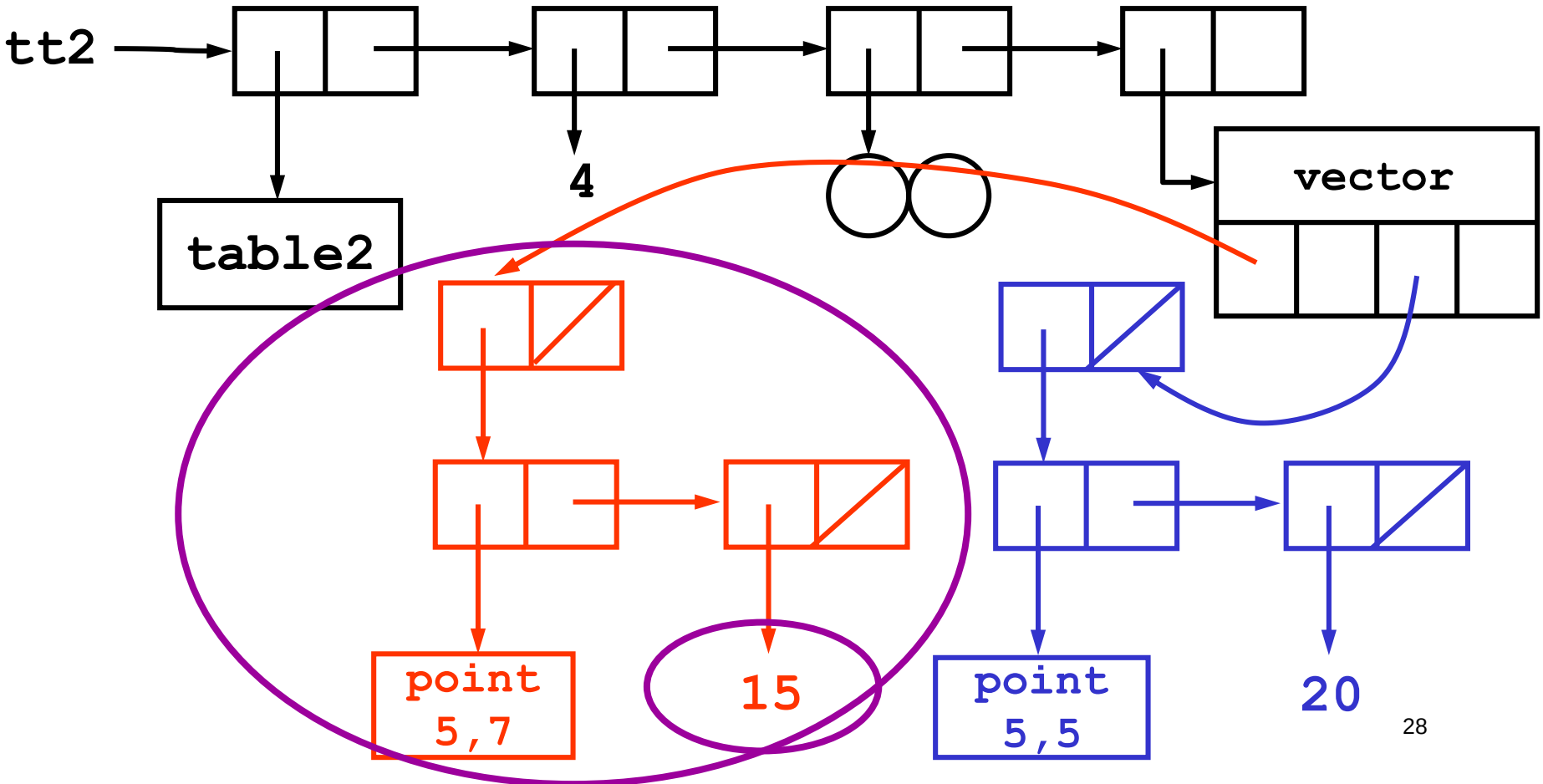
put! in table2

```
(define (table2-put! tbl key val)
  (let ((index
          ((hashfunc-of tbl) key (size-of tbl)))
        (buckets (buckets-of tbl)))
    (bucket-set! buckets index
                 (add-assoc key val
                   (bucket-ref buckets index)))))
```

- Same type as table1-put!

Table2 example

```
(define tt2 (make-table2 4 hash-a-point))
(table2-put! tt2 (make-point 5 5) 20)
(table2-put! tt2 (make-point 5 7) 15)
(table2-get tt2 (make-point 5 5))
```



Is Table1 or Table2 better?

- Answer: it depends!
  - Table1: make — extremely fast
    put! — extremely fast
    get — O(n) where n=# calls to put!
  - Table2: make — space N where N=specified size
    put! — must compute hash function
    get — compute hash function plus O(n) where n=average length of a bucket
- Table1 better if almost no gets or if table is small
- Table2 challenges: predicting size, choosing a hash function that spreads keys evenly to the buckets

Summary

- Introduced three useful data structures
  - association lists
  - vectors
  - hash tables
- Operations not listed in the ADT specification are internal
- The goal of the ADT methodology is to hide information
- Information hiding is denoted by opaque type names