Data Structure for Symbol Table

Compiler Design

Contents

- Symbol table
  - Contests of a symbol table
- Data structure for symbol table
  - Lists
  - Self-organizing lists
  - Search trees
  - Hash tables

Symbol Table

- Compiler needs to collect and use information about the names appearing in the source program
- Each entry in symbol table is a pair containing
  - Name (Variable, procedures, functions, defined constants, labels, structures, etc.)
  - Information (type, form, location, scope, other attributes like array limit, parameters, return values etc.)

- Each time a name encountered, symbol table is searched and/or the name is entered, if it is new.
- There may be separate table for names.

Symbol Table and Other Parts of Compiler

- Lexical analysis and syntax analysis: Fills symbol table.
- Semantics analysis: Checking that uses of names are consistent with their implicit and explicit declaration.
- Code optimization: Flagging temporaries that are used more than once.
- Code generation: Knowing how much and what kind of run time storage must be allocated to a name.
- Error Detection and Recovery: Printing error message like “undefined variable”

Symbol Table Design

- Need to find principle ways to organize or access the symbol table.
- Primary issues in symbol table design
  - Format of entries
  - Method of access
  - Place where stored

- Block-structured language use same identifier to represent distinct names with nested scopes
- Ensures that innermost occurrence of identifier always found first
- Removal of such names from the action portion of the ST when they are no longer active.

Contents of the Symbol Table

- Symbol table: Two fields (name and information)

- Requires capabilities:
  - Search
  - Add new name
  - Access information with a given name
  - Add new information for a given name
  - Delete name or group of names

Name	Type	Size	Dimension	Line of Declaration	Line of Usage	Address

Names in the Symbol Table

- Names in ST denotes objects of various sorts
- May have separate tables for variables names, labels, procedure names, constants, field names (for structure) and other types of names, depending upon the language).
- Useful to have more than one table with varying size and format of information.

Data Structures for Symbol Table

- Symbol table is searched every time an identifier is encountered.
- Data are added if a new name or information is discovered.
- Required to add new entries and finding existing entries in ST efficiently.
- Evaluation of scheme: time required to add n entries and make m inquiries.

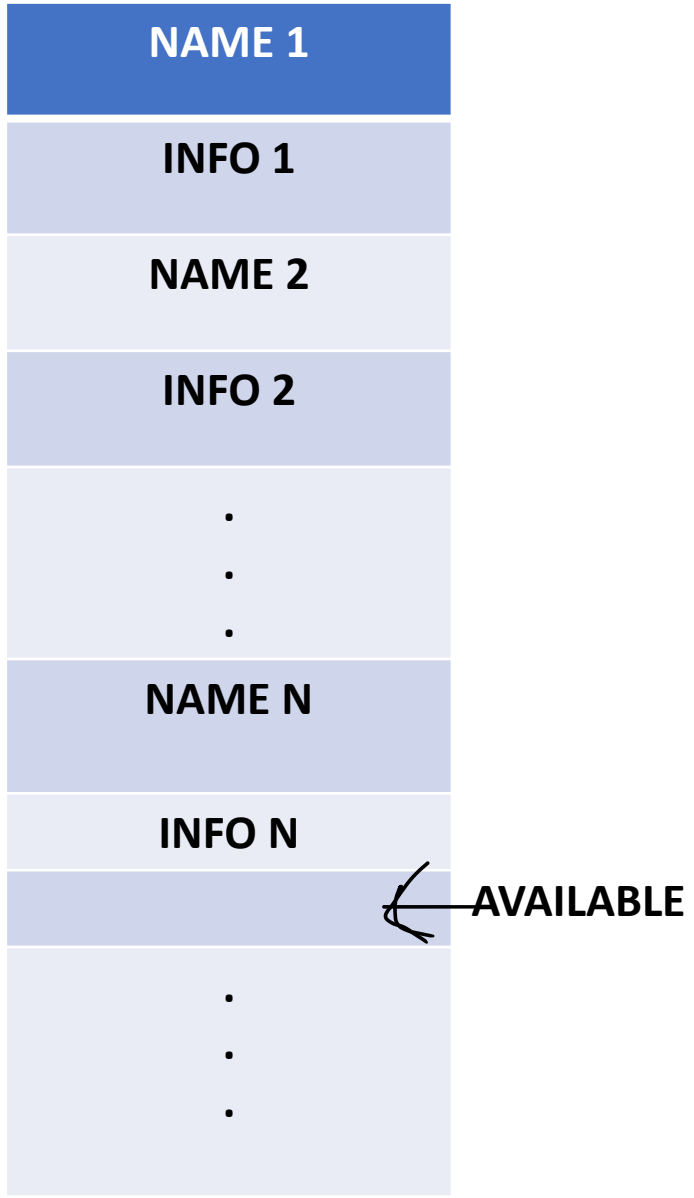
Data Structures for Symbol Table

Commonly use data structure for symbol tables:

- Linear list:
  - Simplest
  - Inefficient with large number of entries and inquiries
- Trees (BST):
  - Better performance
  - Some increases in implementation difficulty
- Hash Tables:
  - Best performance
  - Greater programming efforts and some extra space

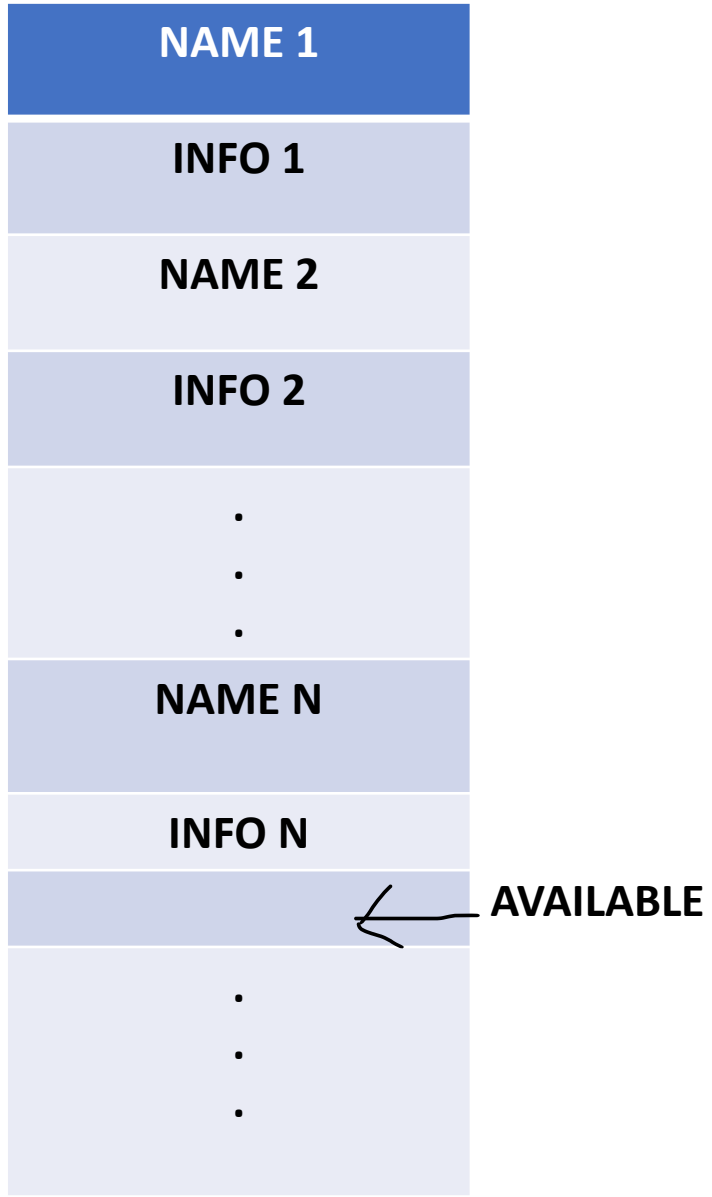
Lists

- Simplest and easiest data structures
- Use array or equivalently several array to store name and their associated information.
- New names are added to the list in the order in which they are encountered.
- AVAILABLE: indicates position of first empty portion of the array.



Lists

- Retrieve: search from beginning up to the position marked by pointer AVAILABLE
  - Found: Associated information can be found in the word following next
  - Not found: announce fault – the use of an undefined name
- Insert: scan down the list to be sure that it is not already there
  - If it is, we have another fault – a multiple-defined variable
  - If not, store new word and increase the pointer by the width of a symbol – table record



Lists

Number of names in symbol table = n

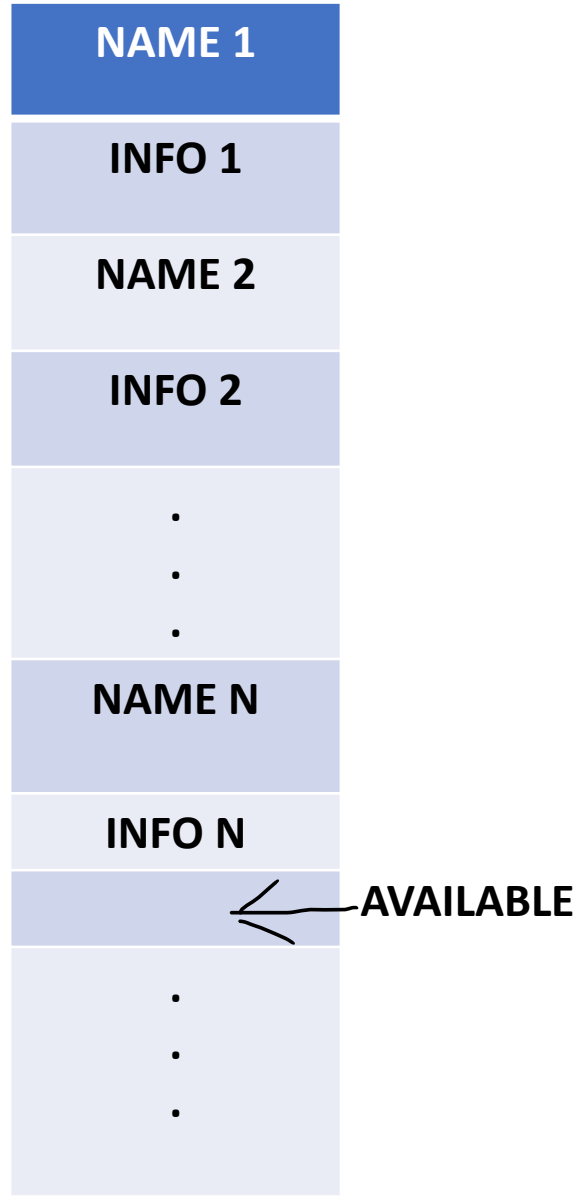
Work necessary to insert a name is proportional is = n

Average searches to find data = n/2

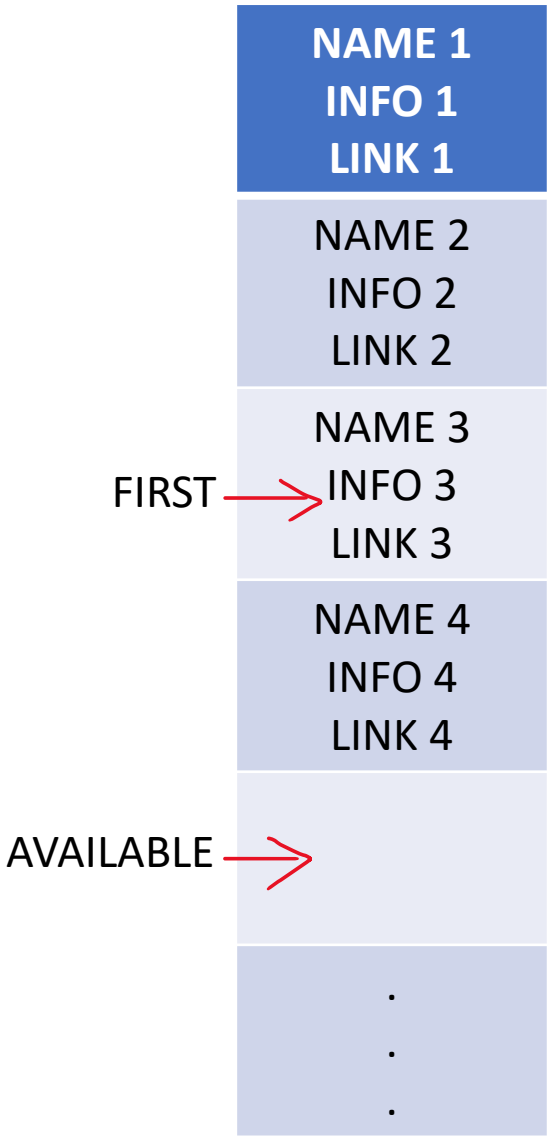
Cost of inquiries is proportional to = n

Total cost to insert n names and m inquiries = cn (n+m)

where, c is a constant representing the time necessary for a few machine operations

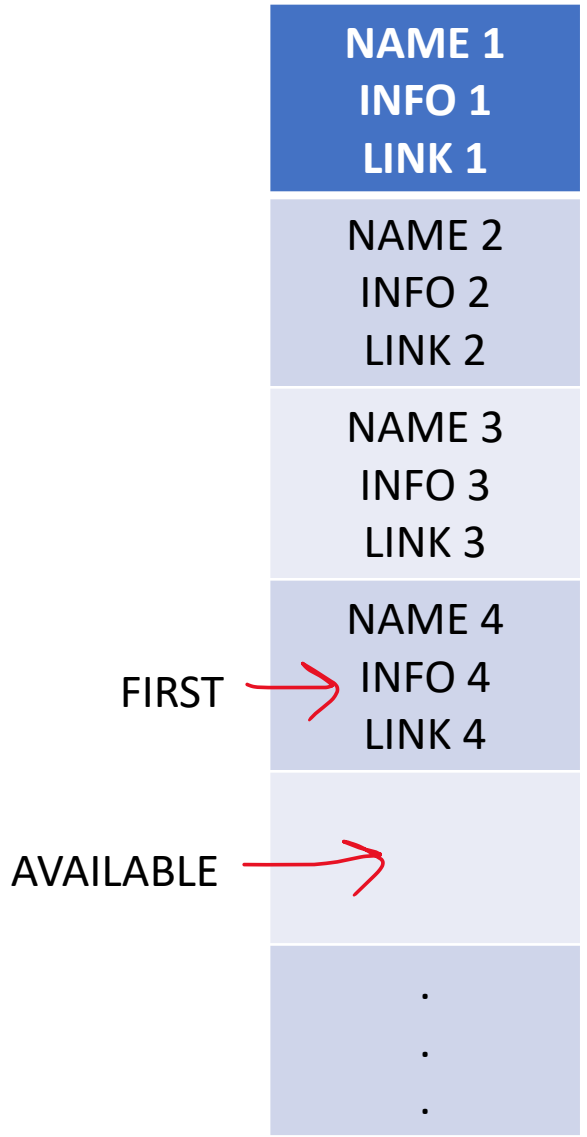


Self-organizing Lists

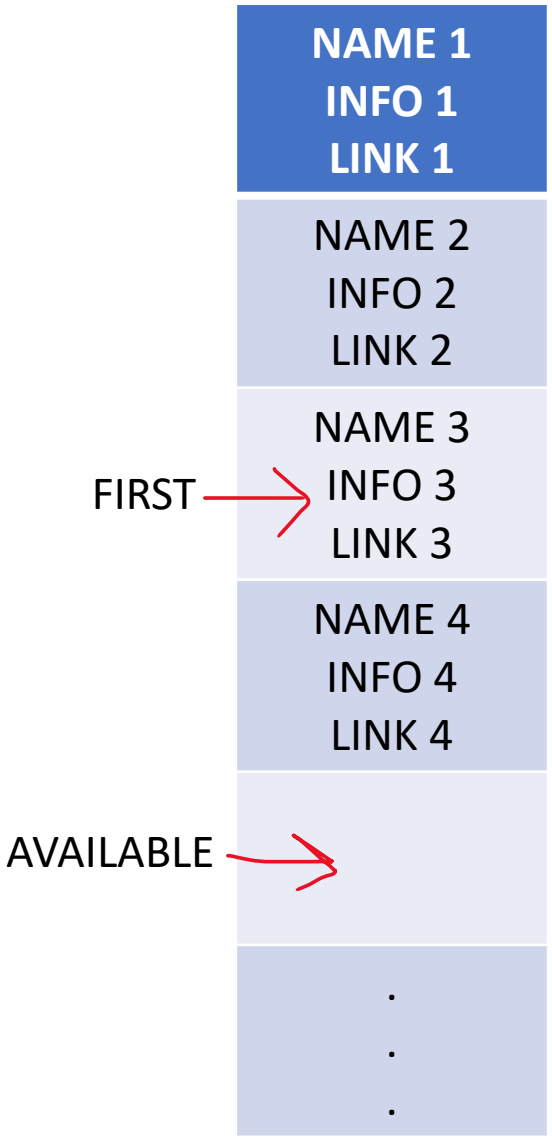


- Substantial fraction of searching time is saved at the cost of little extra cost.
- Names that are referenced frequently will tend to be at the front of the list to found it quickly.

3→1 →4 →2



Self-organizing Lists



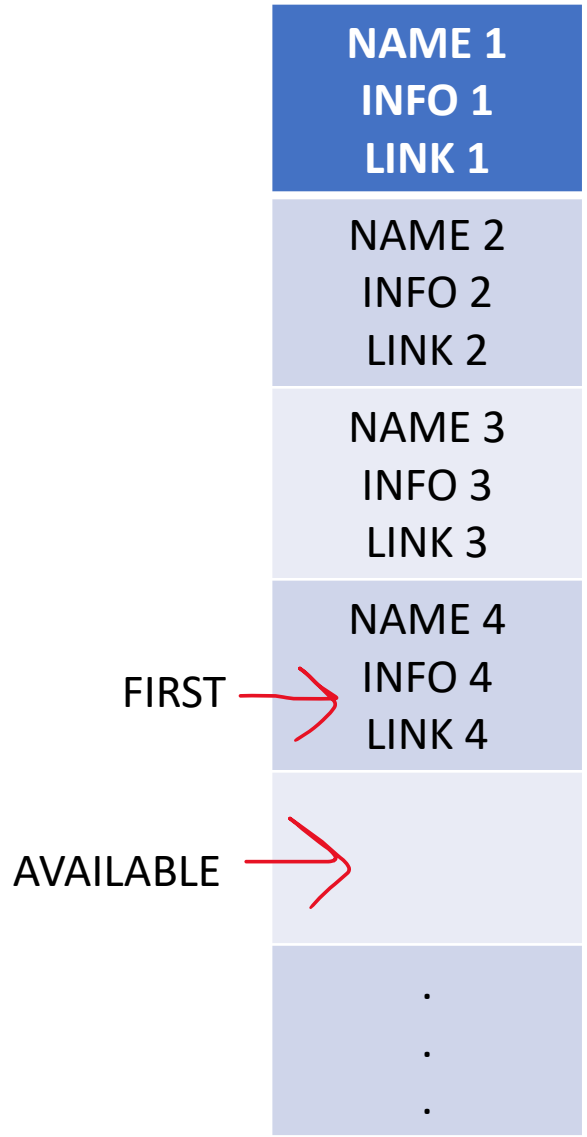
3→1 →4 →2

LINK p← LINK I

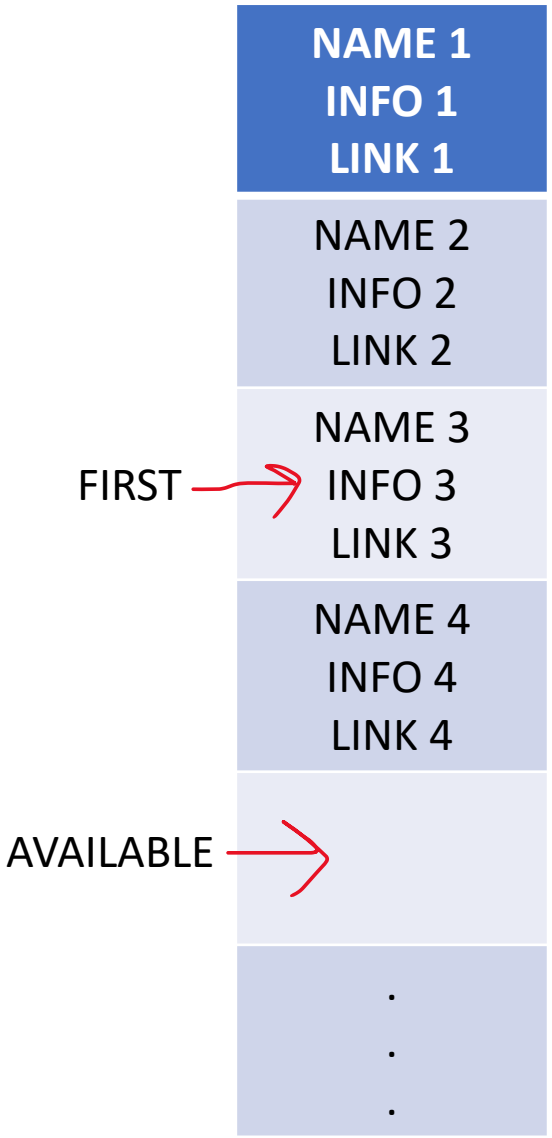
LINK i ← FIRST

FIRST ← NAME i

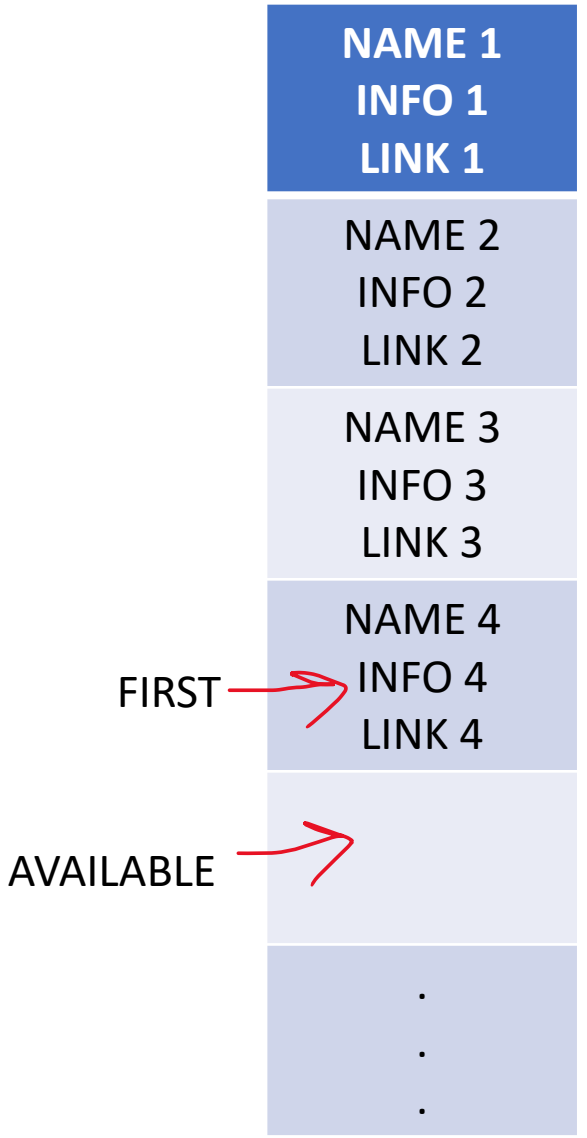
4→3 →1 →2



Self-organizing Lists



- Preferred: when small set of names is heavily used
- Costs time and space: when references are random
- Average: time saving



Search Trees

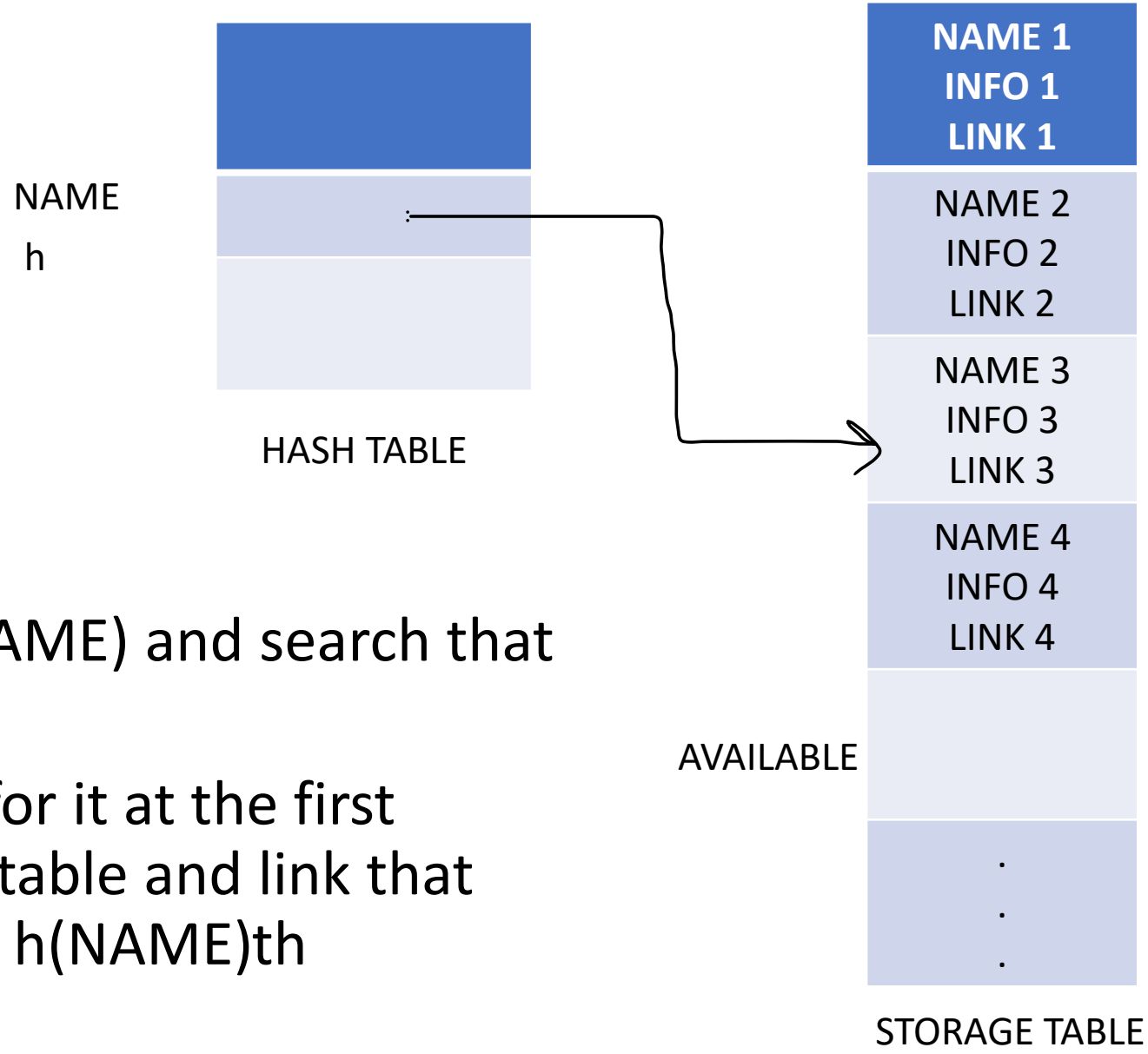
1) while P is not = null do
2) if NAME = NAME(P) then ……..
    1) /*NAME found, take action */
3) else if NAME is less than NAME(P) then P = LEFT (P)
    1) /*visit left child*/
4) else P := RIGHT(P) /*NAME greater than NAME(P)*/
    1) /*visit right child*/
    2) /*if we fall through the loop, we have failed to find NAME */

Time needed to enter n names and make m inquiries is proportional to (n+m) log n

Hash Tables

Hash Table: consists of k words (0,1,….,k-1)

Storage Table: words points to storage table to the heads of k separated linked lists

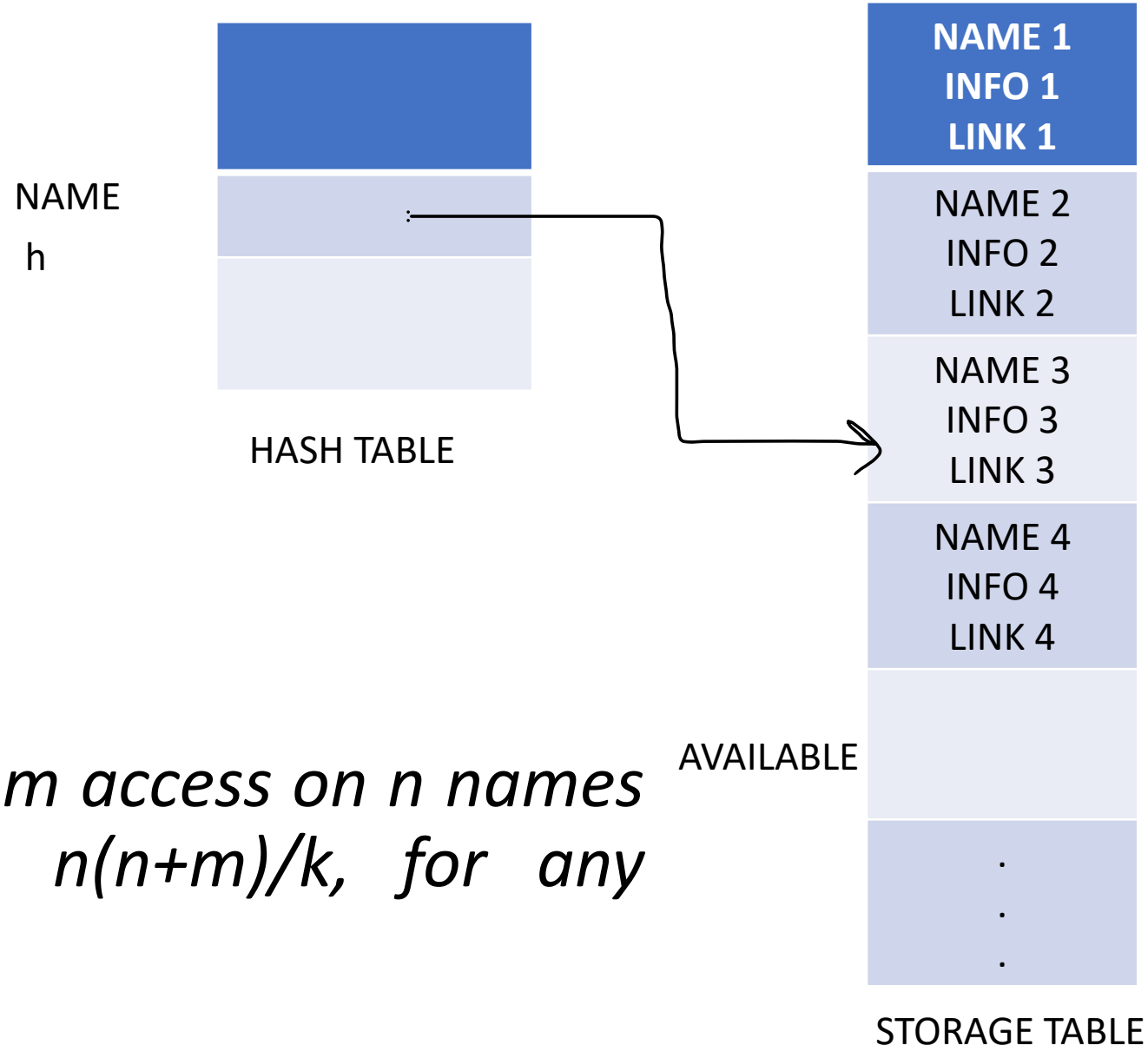


Inquire NAME: Computer h(NAME) and search that list only

Enter NAME: Create a record for it at the first available space in the storage table and link that record to the beginning of the h(NAME)th

Hash Tables

1) h will distribute names uniformly among the k lists, and
2) h is easy to compute for names consisting of strings of characters



The capability of performing m access on n names in time is proportional to n(n+m)/k, for any constant k.

Thank you