

Tables

CIS*2520 Summer 2006

Overview

- Can we break the " $\log_2 n$ " barrier for searching?
- Of course - we do it all the time
 - store records, indexed $0..(N-1)$ in an array of size N
 - use the index n as the key to locate the record of item n by ordinary table look-up
 - $O(1)$
- The trick is making efficient use of tables; consider:
 - records in files
 - very large number of very large records (similar issue)
 - non-integer keys
 - multiple keys

CIS*2520 (S06)

Rectangular Arrays

- Generally provided by all high level languages
- Note that compiler storage is inherently linear
 - accessing rectangular arrays involves some work to map the 2-component index to a linear space
- Typically **row-major** ordering
 - i.e. the i in an (i,j) pair refers to the row in the array
 - as opposed to column-major ordering
- If we represent a rectangular array in a linear space, how do we map (i,j) pairs \rightarrow linear space?

CIS*2520 (S06)

Rectangular Arrays (cont.)

- let m and n be the number of rows and columns respectively
 - let us define a function to compute the index given i,j

Index Function:

in C: $(i,j) \rightarrow (n * i) + j$

1st element of the row
offset into the row

Note: if arrays are indexed 1 to n rather than 0 to $(n-1)$ then,

$$(i,j) \rightarrow ((n * (i - 1)) + 1) + (j - 1) = n(i-1) + j$$

CIS*2520 (S06)

Rectangular Arrays Example

```
typedef struct 2DArray_t
{
    int *array; /* linear array */
    int dim_m, dim_n; /* dimensions of virtual 2-D array */
} 2DArray;

2DArray *2DArrayCreate(int m, int n)
{
    2DArray *new;
    new->array = calloc(m*n, sizeof(int));
    new->dim_m = m; new->dim_n = n;
    return(new);
}

#define 2DAGet(A,i,j) ((A)->array[(i)*(A)->dim_n + (j)])
```

A = 2DArrayCreate(4,4);

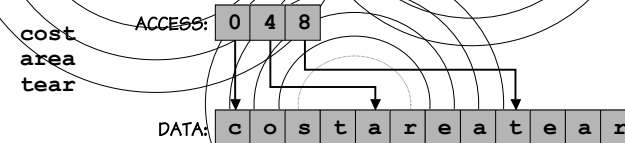
... 2DAGet(A,1,3) => (1*4) + 3 = 7



CIS*2520 (906)

Access Tables

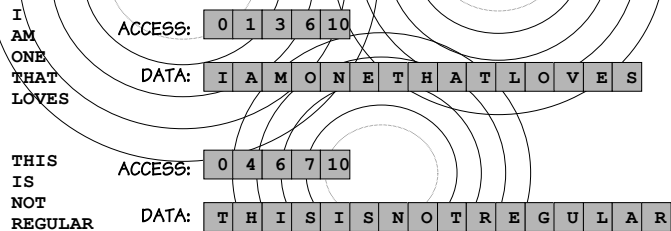
- Use of an auxiliary array to reference the start points of rows in linear storage (i.e. indirection)
- e.g.



CIS*2520 (906)

Access Tables (cont.)

- Note that this permits tables of varying shapes
 - some forms may have easy index functions (triangular)
 - others will have no index function in general



CIS*2520 (906)

Index Functions

- Note that the idea is to permit [i,j] style indexing
 - this does not imply that we can necessarily enforce "legal" references (e.g. [0,2] in a lower triangular array)
 - given appropriate data abstraction you can even handle this
 - e.g. traversal of a lower triangular array:

```
for (i = 0; i < n; i++)
    for (j = 0; j < i; j++)
        ... (i,j) = 1;
...
```

CIS*2520 (906)

Inverted Tables

- Extend the idea of an “access table”
 - user more than one access table to permit a single table or records to be referred to by several different keys
- e.g. Phone Company
 - to publish phone directory, want alphabetic
 - to handle billing, want telephone number
 - to schedule maintenance, want sort by address
 - we could three sets of records and keys
 - wasteful
 - just use 3 access tables (sorted status of records is not important)

CIS*2520 (S06)

Inverted Table (cont.)

INDEX	Name	Address	Phone
0	Hill, Thomas M.	High Towers #315	282-9478
1	Baker, John S.	17 King Street	288-3285
2	Roberts, L.B.	53 Ash Street	437-2296
3	King, Barbara	High Towers #802	286-3386
4	Moody, C.L.	39 King Street	249-5723
5	Byers, Carolyn	718 Maple Street	439-4231

ACCESS TABLES:

Name	Address	Phone
1	2	4
5	0	0
0	3	3
3	1	1
4	4	2
2	5	5

CIS*2520 (S06)

Inverted Tables (cont.)

- First name entry is the position where the records of the subscriber whose name is first in alpha-order
 - 2nd == second alphabetic
 - etc.
- Multi-Key access tables in general.
 - an “inverted table”

CIS*2520 (S06)

Tables and Functions

- All of these concepts are analogous to **functions**
- Function:
 - start with an argument and compute a corresponding value
 - $f: \text{domain} \rightarrow \text{range}$
 - table access:
 - index (via table look-up) \rightarrow corresponding value
 - domain == INDEX SET
 - range = BASE TYPE (or VALUE TYPE)
- e.g. `double darray [MAX];`
 - index set = {1, 2, 3, ..., MAX} (integers)
 - value type = set of real numbers

CIS*2520 (S06)

Table ADT Specification

- **DEFIN:**
 - A table with index set I and value type T is a function from I to T
- **OPERATIONS:**
 - **create**
 - create a new finite-sized table initialized to be empty
 - PRE: $size > 0$
 - POST: table is initialized and empty
 - **destroy**
 - destroy a table
 - PRE: n/a
 - POST: table is destroyed

CIS*2520 (S06)

Table ADT Specification (cont.)

- **OPERATIONS (cont.):**
 - **getEntry**
 - evaluate the function at any index in I
 - PRE: index value is from index set
 - POST: provides value of the function at index
 - **addEntry**
 - adjoin a new item x to the index set I and define a corresponding value of the function at x
 - PRE: table is not full (if applicable)
 - POST: function (i.e. table) now evaluates to specified value given index value x
 - **removeEntry**
 - delete an element x from the index set I and restrict the function to the resulting smaller domain
 - PRE: index value is from the index set
 - POST: function (i.e. table) is now undefined if evaluated at index x

CIS*2520 (S06)

Table ADT Specification (cont.)

- **OPERATIONS (cont.):**
 - **size**
 - obtain the physical size of the table
 - PRE: n/a
 - POST: provides the physical size of the table
 - **entries**
 - obtain the number of entries in the table
 - PRE: n/a
 - POST: provides the current number of entries in the table
 - **load**
 - obtain load factor for table
 - PRE: n/a
 - POST: evaluates and provides value $entries / size$

CIS*2520 (S06)

Table ADT Specification (cont.)

- **OPERATIONS (cont.):**
 - **isEmpty**
 - determine if table is empty
 - PRE: n/a
 - POST: evaluates to "true" if table is empty, "false" otherwise
 - **write**
 - **read**
 - **others?**

CIS*2520 (S06)

Table ADT Implementation

- **Note:** the notion of an “index set” is dynamic
 - depends on what is currently a legal mapping within the table
- **Issues:**
 - perform mapping with index function? access table? ...?
 - mathematical distinction
 - list == **sequence** --- has implicit order
 - table == **function** --- only *sometimes* has a natural order
 - efficiency of retrieval:
 - O(1) generally
 - faster than list searching
 - traversal not always clear

CIS*2520 (S06)

Hashing

- **Problem: sparse storage**
 - indexed by very large set with relatively few positions actually occupied
 - e.g. alphabetic words of 8 letters
 - 26⁸ possible keys
 - only a small fraction of these will occur
- **Problem: non-integer keys**
 - or keys otherwise not suitable for direct indexing
- We wish to have a 1-1 correspondence between keys and indices for array access
 - solution: *hashing*

CIS*2520 (S06)

Hashing (cont.)

- **Hash Table**
 - allow many different keys to be mapped to the same index
- **HASH FUNCTION**
 - takes a key and maps it to some array index
 - generally maps several different keys to the same index
- **COLLISION**
 - two or more records wanting to go to the same index location

CIS*2520 (S06)

Hashing Implementation Issues

- **Array to hold the hash table**
 - must store the key --- cannot just use index
- **Must know if entry is “empty”**
 - a pointer/reference/ADT commonly used for this (especially for chaining)
- **To insert a record: evaluate hash function for a given key**
 - if location empty: **store record**
 - if keys are equal: **duplicate key** (overwrite? disallow?)
 - if record with different key in same location: **collision**
- Retrieval is a similar process

CIS*2520 (S06)

Hash Function

- Ideally, a hash function should be
 - easy/quick to compute
 - produce even distribution of keys across indices
- If we know in advance *exactly* the keys to be used this is easy
 - generally this is not the case (e.g. string-based keys)
- Common to have hash function take the key, “chop it up” and mix the pieces in various ways in hopes of getting a good distribution
 - contrast this with the generation of pseudo-random numbers

CIS*2520 (S06)

Hashing by Truncation

- Ignore part of key and use the rest
- e.g. 8-digit integer keys, 1000 table entries
 - choose 1st, 2nd and 5th digits from the right
 - 62538194 ----> 394
- Benefits:
 - fast
- Drawbacks:
 - often fails to distribute keys well

CIS*2520 (S06)

Hashing by Folding

- Partition and combine
- e.g. 8-digit integer keys, 1000 table entries
 - break into groups of 3, 3, 2 numbers and add together, truncate if necessary
 - 62538194 ----> 625 + 381 + 94 = 1100 -> truncate to 100
- Benefits:
 - still fairly fast (obviously slower than plain truncation)
 - tends to obtain better distributions than truncation alone
- Drawbacks:
 - can still produce weak distributions in many cases

CIS*2520 (S06)

Hashing by Modular Arithmetic

- Convert key to integer (use truncation/folding if necessary) and MODULUS with array size
- e.g. 8-digit integer keys, 1000 table entries
 - do modulus with key directly
 - 62538194 % 1000 = 194
- Distribution achieved is highly correlated with particular modulus
 - poor choice: powers of small integers (2 or 10)
 - good choice: prime number (tends to be quite uniform)
- Can achieve good distribution of keys while ensuring correct range
 - characteristics of keys should be taken into account when designing the conversion to integer process

CIS*2520 (S06)

Collision Resolution by Open Addressing

- Apply function to compute a new index from the original
 - **incremental functions** (e.g. 2nd hash)
 - little to gain
 - **linear probing** (increment index by 1, 2, ...)
 - clustering quickly becomes a problem
 - **quadratic probing** (probe at $h_1 + j^2$, $j = 1, 2, \dots$)
 - reduces clustering but may not consider all positions
 - **key-dependent increments**
 - apply some other function to the key: $f_2(\text{key})$
 - **random probing**
 - relies on having same seed value -> same stream of values
- *In all cases, deletions may be non-trivial*

CIS*2520 (S06)

Collision Resolution by Chaining

- We wish to use contiguous storage for the table
 - for efficiency
 - we can still use references/pointers to the data itself (save space if records are large/many empty records)
- Simple and efficient collision handling
 - organize all the records with the same hash address as a list (i.e. a List ADT, or add a "link" field to the record)
 - good hashing function means few keys will give the same address so lists should be short (ideally)
 - clustering is not an issue
 - size of table need not exceed number of records
 - deletion is trivial
 - references/pointers/ADTs do require space however (i.e. ratio with small records may be significant)

CIS*2520 (S06)

Chaining Implementation

- General idea:
 - **getEntry**
 1. hash key
 2. do sequential search of chain at hashed index
 - **addEntry**
 1. hash key
 2. insert new entry at head of chain at hashed index (may require a search if replacement is desired)
- Note: collision resolution is substantially simpler here than would be the case with open addressing

CIS*2520 (S06)

Load Factor

- Let n be the number of entries in the table, let t be the number of positions in the array
- **LOAD FACTOR** = n / t
- NOTE:
 - load factor = 0 (empty)
 - load factor = 0.5 (half as many entries as positions)
 - maximum load factor
 - 1.0 for open addressing
 - no limit for chaining

CIS*2520 (S06)

Theoretical Comparison of Hashing Methods

Adapted from *Data Structures and Program Design in C*, Kruse & Leung

LOAD:	0.10	0.50	0.80	0.90	0.99	2.00
Successful search, expected # of comparisons (probes)						
chaining	1.05	1.25	1.40	1.45	1.50	2.00
open, random	1.05	1.40	2.00	2.60	4.60	-
open, linear	1.06	1.50	3.00	5.50	50.50	-
Unsuccessful search, expected # of comparisons (probes)						
chaining	0.10	0.50	0.80	0.90	0.99	2.00
open, random	1.10	2.00	5.00	10.00	100.00	-
open, linear	1.12	2.50	13.00	50.00	5000.00	-

Computed as infinite sum of probabilities

CIS*2520 (906)