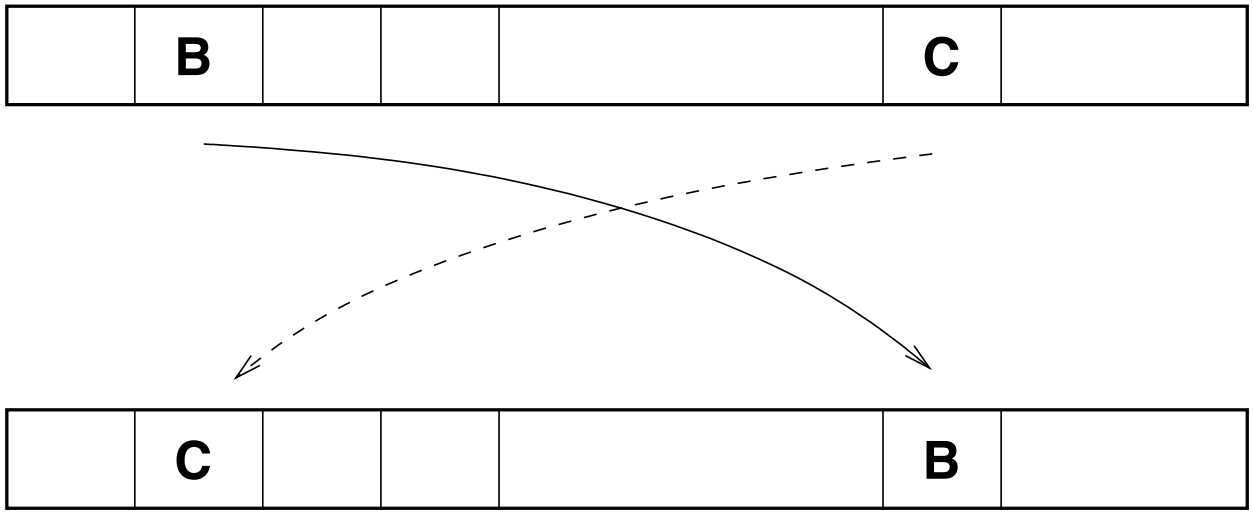
Topics This Week

1. Organization of data — ordered vs. un-ordered
2. Searching, Updating ordered data (summary only)
3. Searching, Updating un-ordered data:
   - Hashing Techniques
   - Approximate Performance Analysis
4. Heap data structure and HeapSort

Data

- Every piece of data D has a *Key* which identifies it uniquely.
- Typical collections of data: *files*, *directories*, *symbol tables*, *mathematical tables*.
- Operations on data: *insert/delete (updates)*, *search/select* — internal or external?
- Manipulate data directly or indirectly?

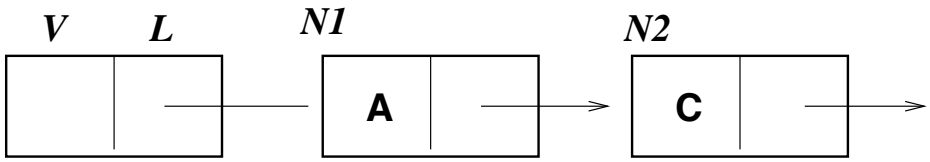
Direct Manipulation

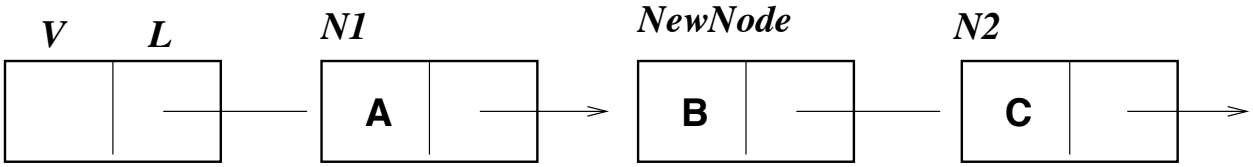


Swapping two elements in an array

This is fine if the array entries are simple types like int or char.

Another Direct Manipulation



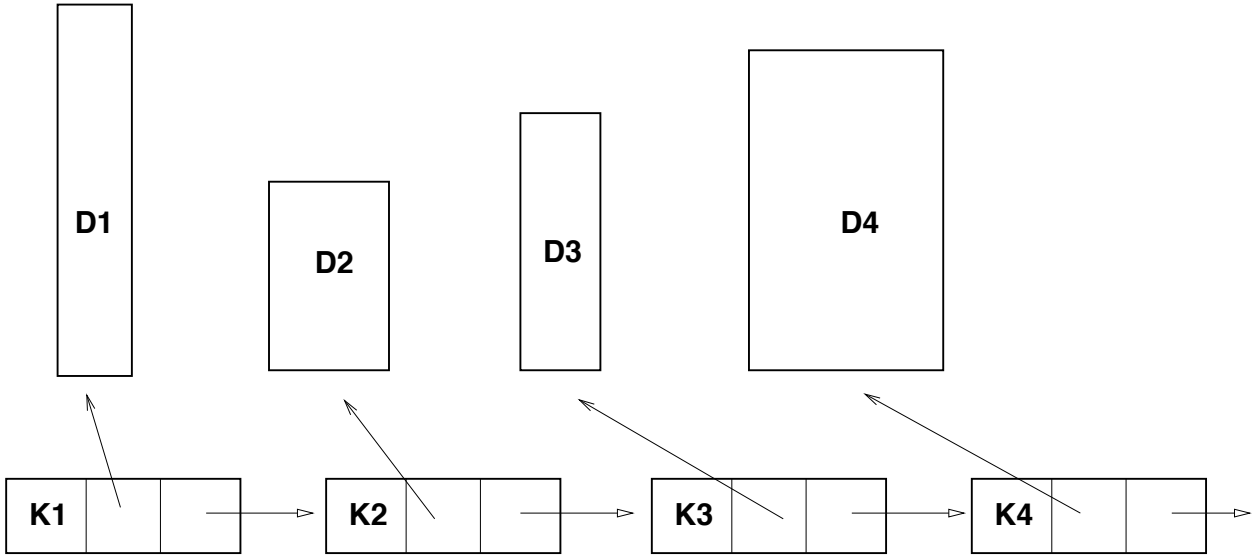


Linked lists are better than arrays for frequent insertions/deletions

```
TempL = N1.L;  /* N2, don't lose it! */
N1.L = NewNode;
NewNode.L = TempL;
```

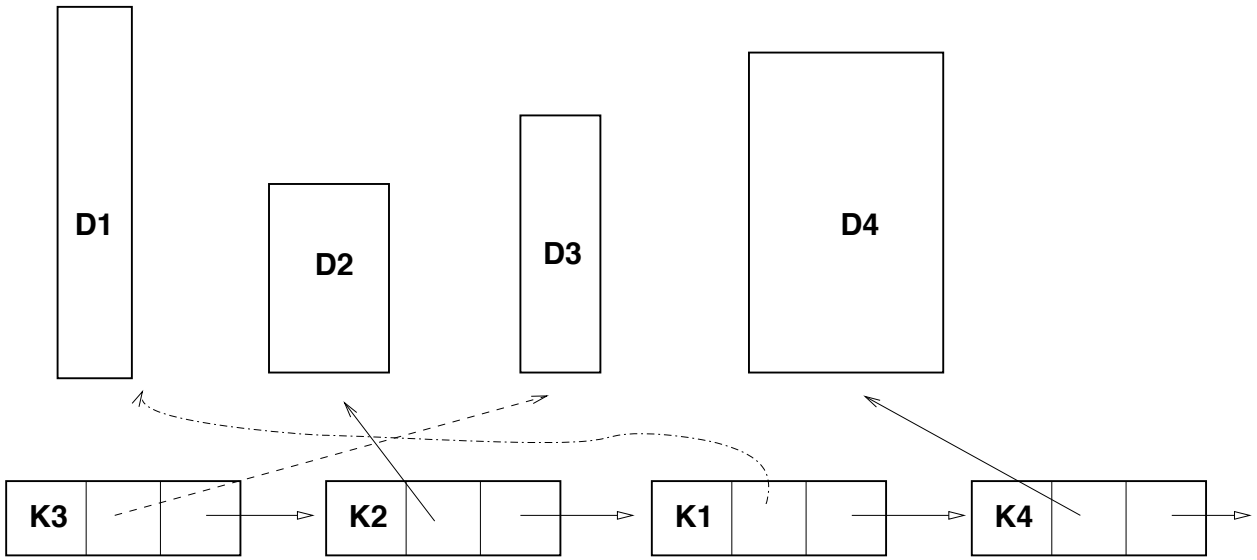
This is also fine if V-fields are not too big.

Indirect Manipulation I



Uneven or massive data –– use Indirect method; K's are the Keys for Data pieces; in sorting/searching/updates, manipulate Linked list rather than moving Data.

Indirect Manipulation II



After swappping ordering of data as K3 < K2 < K1.

Ordered Keys

1. Must search for correct position to insert/delete. About O(log n) time, using binary (or k-ary) chop.
2. Can speed up by using tricks like:
   - Doubly-linked lists
   - Trees — as balanced as possible, maybe k-ary
   - Partial, exetendable arrays
3. The run time is worst-case, but guaranteed.
4. Output is "user-friendly", e.g. White pages entries.

Un-ordered Keys

Can we organize data so that the keys are not ordered?
Yes, provided that:

Output does not have to be "user-friendly";
Worst-case run time is not guaranteed, although average is OK;
Sophisticated analysis or empirical simulations for run times;

The big advantage is that with "correct" organization, updates can be very fast since no sorting is necessary.

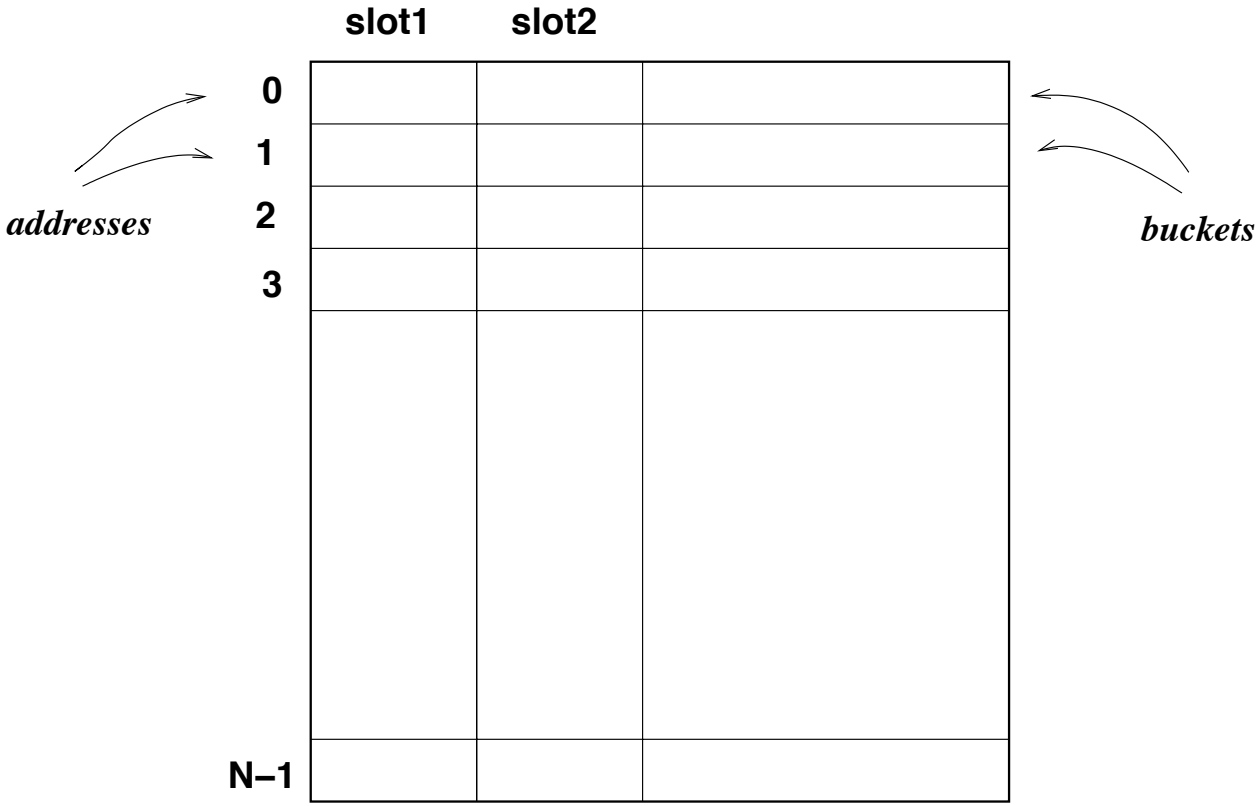
Hashing for Un-ordered Keys

The standard method for organizing data without ordering keys is *Hashing*.

A *Hash Function* F is a numeric function that maps Keys into (memory) addessses, either absolutely or base-relatively, i.e. $F : Keys \rightarrow Addresses$. The addresses usually refer to entries in a *Table*. Each such entry is a *bucket*, which may be divided in turn into *slots* for data.

Keys might as well be numeric, as they have ASCII, Unicode, etc encoding interpretable as binary numbers. So, F maps numbers to numbers.

Picture of Hash Table



F(key) = some number between 0 and N–1

Loading Density

T = theoretically possible number of keys (e.g. identifiers)
K = practically useful number of keys ("normal indentifiers")
S = number of slots per bucket
N = number of buckets

Each slot can contain one piece of data.

K/T = *Key Density*
K/(S*N) = *Loading Density*

Usually, K << T and N << T. Also, S is usually 1.

Overflows and Collisions

A *collision* occurs when there are two distinct keys K1 and K2 such that F(K1) = F(K2). K1 and K2 are then *synonyms* or *aliases*.

An *overflow* occurs when a new key K is mapped by F to a bucket that is already full (no vacant slots). In some texts, this is also (confusingly) called a collision.

Collisions are not a problem as long as a bucket still has empty slots.

Overflows are always a problem.

Example

	s1	s2
0	Allan	
1	Betty	Brad
2	Charles	
3		
4	Effi	
25		

Key of a Person's Name is its first letter.

F(Letter) = position in alphabet, 0=A, 1=B, etc.

Operation	*Result*
Add "Doyle"	**Put in s1 of bucket 3**
Add "Emma"	**Alias with "Effi"; Put in s2 of bucket 4**
Add "Bill"	**Overflow bucket 1**

This is not a realistic hash function, however.

Hash Functions

Desirable Features:

1. Efficient to compute
2. Minimizes collisions

To achieve (2), we should make F *uniform*, i.e., if there are N buckets, for all I in the range 0 to N-1, Prob(F(K) = I) should be about 1/N, independent of K.

Quasi-uniform F's

Some standard techniques to get such an F:

1. Modular arithmetic
2. Mid-square
3. Folding

Modular Arithmetic

The idea here is to achieve uniformity by spreading the keys evenly across the table by associating them with the numbers 0, 1, 2, ... N-1.

$$F(K) = K \bmod N$$

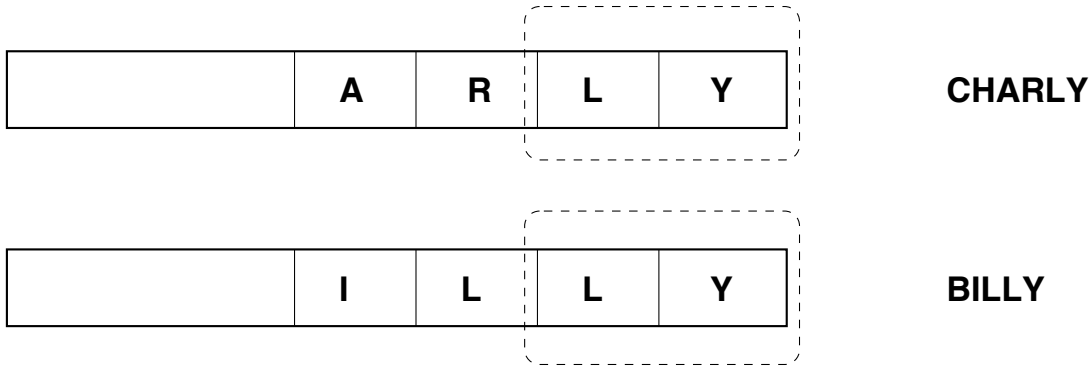
where N is a number (divisor), and K mod N is the *remainder* on dividing K by N. Recall that Keys "are" numbers.

Hence $0 \leq F(K) \leq N\text{-}1$, and the table is of size at most N.

N must be carefully chosen if F is to be uniform.

Common Naming Habits

Many names, variables, etc, have similar endings. For these, choosing N to be 2**k for some k is a bad idea.



Left justification is worse

C	H	A	R	L	Y	0	0		0

CHARLY

B	I	L	L	Y	0	0	0		0

BILLY

This is even worse, for most names will hash to 0

Pragmatics for Modular Method

Number-theoretic considerations provide reasons for certain guidelines for good choices of N.

- N should be prime.
- Let each byte be of length L bits (one byte per character). N should *not* divide L**k + a nor L**k - a for "small" k and a.
- In practice, it seems to suffice to choose N to have no prime divisors less than 20.
- Prime testing is quite efficient (Rabin test)

Randomizing Keys

Another idea is to invent an F that has the effect of *randomizing* keys.

It is an observed fact that keys arising in practice are often *names* of one kind or another, e.g, "Beethoven", "Velocity", "Xvar12".

The occurrence of alphamerics in them are *biased* and *correlated*, i.e., vowel clusters tend to follow consonants, numbers tend to be at the end, etc.

Any hashing function that does not reduce these biases will cause a lot of collisions.

Shuffling the Keys

The next two methods, *Mid-Square* and *Folding*, are two heuristic ways to randomize keys by spreading the dependency of F(K) across the entire key and by shuffling them.

You can experiment with some other ways yourself, e.g., take the ASCII of a variable name, do a cyclic shift of it by j bits, and add it to the original. Then see what name you get on decoding. Does it appear more "random"?

Mid-Square

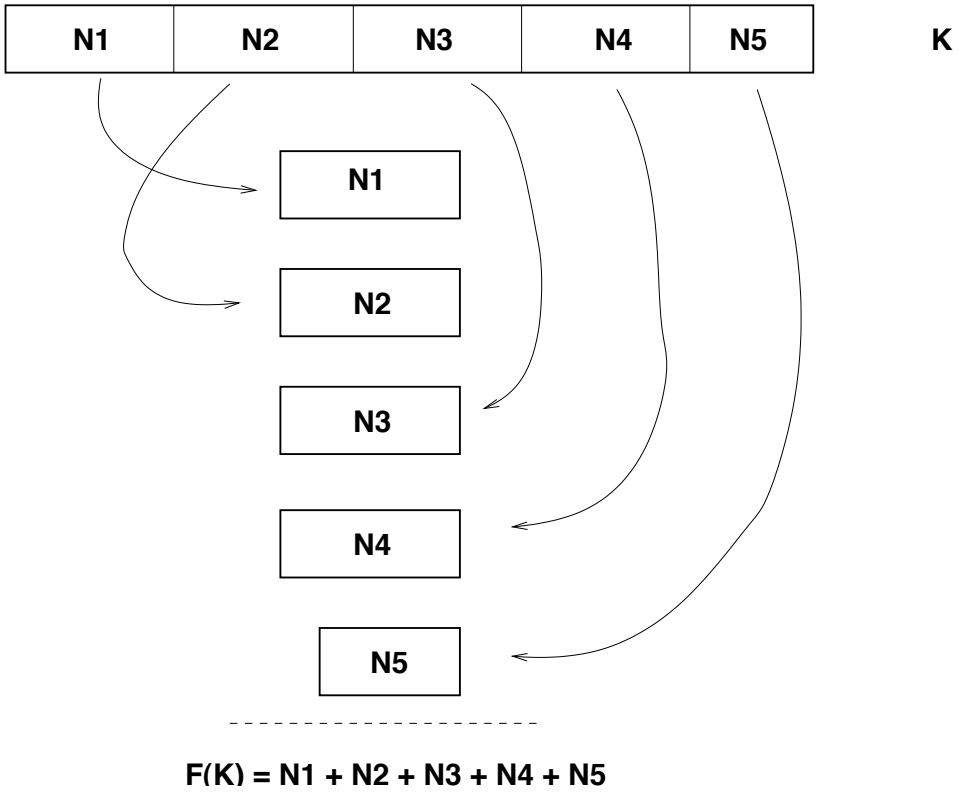
This is a popular method as it has a lot of empirical backing.

Take the ACSII, EBCDIC, Unicode, etc binary of the key as a number, square it, then extract the middle k bits as the bucket address. This intuitively both spreads the dependencies and shuffles the key.

It implies that the table is of size 2**k. It seems to work quite well in practice.

Folding

This also intuitively spreads dependencies and shuffles the key.



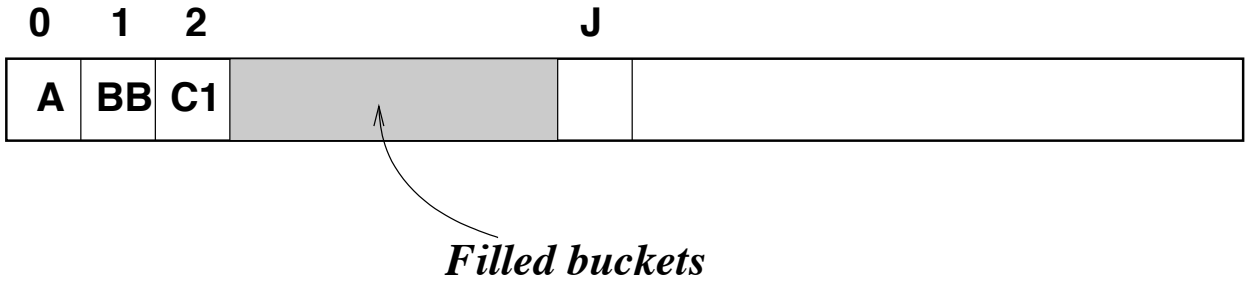
Overflows

Two standard ways of handling overflows are (i) linear probing, and (ii) chaining.

These are easier to understand if we assume there is only one slot per bucket, in which case *collision* = *overflow*.

Linear probing (and its relatives, e.g quadratic probing) handles a collision by searching forward in the table for the *next vacant bucket* and places the key there. The table is considered to be "circular".

Linear



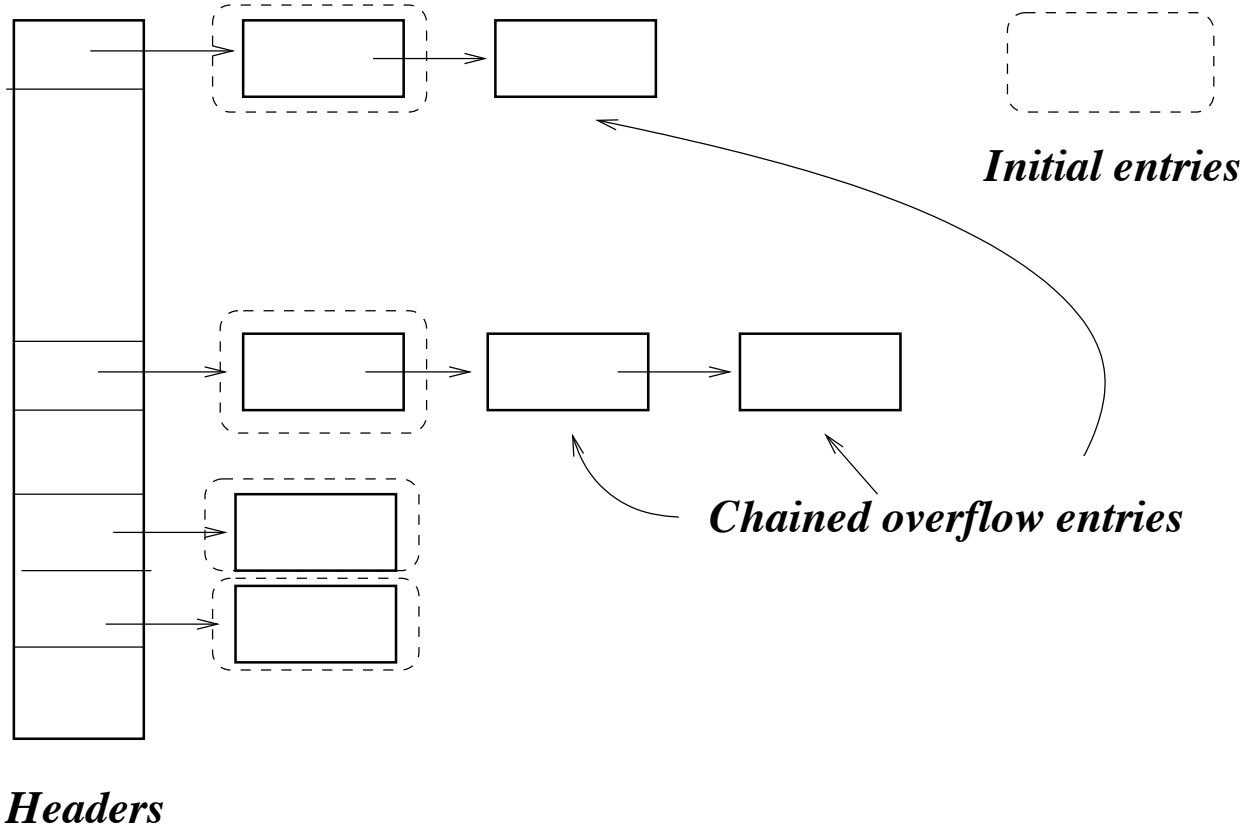
Where new key CC is put if it collides with C1

The table "wraps around" in looking for the next vacant bucket.

Costs of Linear Probing

- Aliases tend to *cluster*. (Quadratic probing can reduce this.)
- Clustering causes sequential search after collision detection.
- Collision can occur even with keys K1 and K2 are such that F(K1) $\neq$ F(K2), since a bucket could have been filled by an overflow from somewhere else.

Chaining



Costs for Chaining

- Space for chains.
- List maintenance.
- But all aliases are "real", and sit in the same chain.

Example of Approximate Analysis

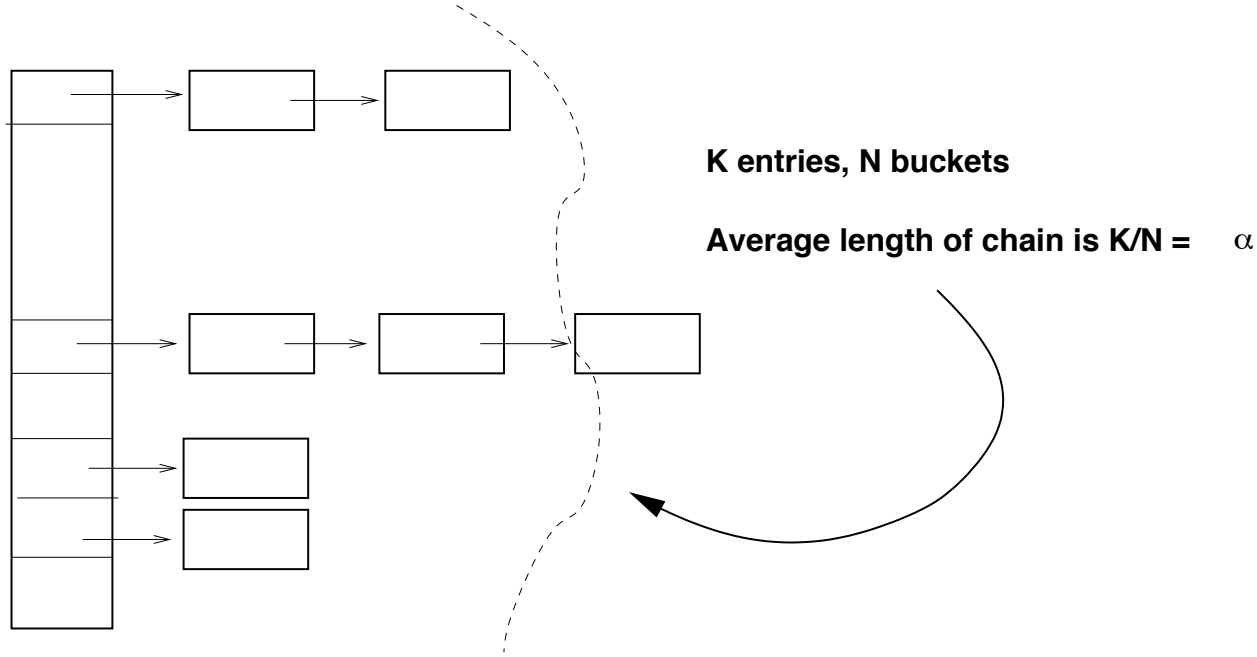
We will do this for Chaining, and for average time performance.

Assume a uniform hash function F, slot = bucket, K items, N buckets. Then the loading density α = K/N.

Let S_K be the number of comparisons to find a key that is already in the table; and T_K be that to find a vacant spot.

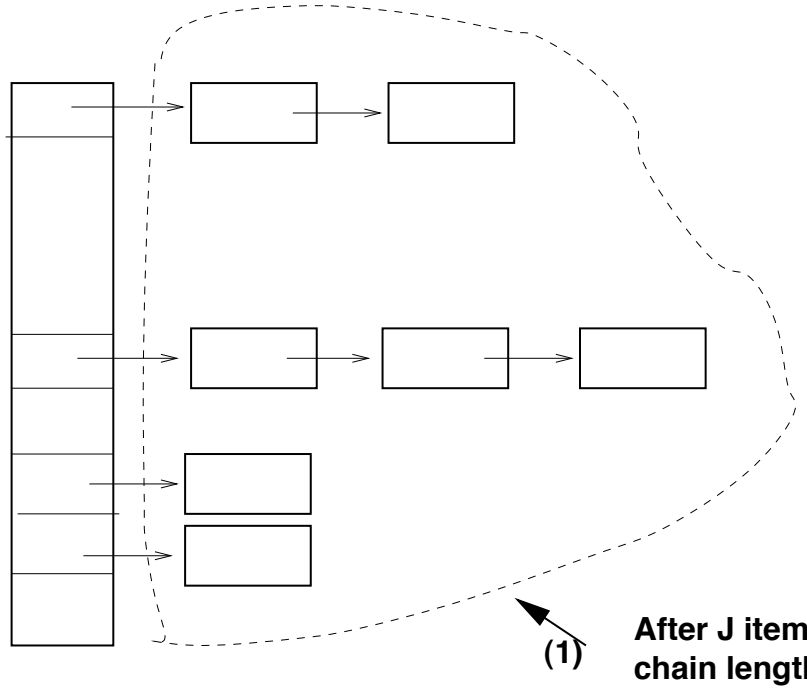
Result: T_K is approx α, and S_K is approx $1 + \frac{\alpha}{2}$.

Average Chain Length



This is also the average number of comaparisons T_K to find a vacant spot

Locating the J-th Item Entered

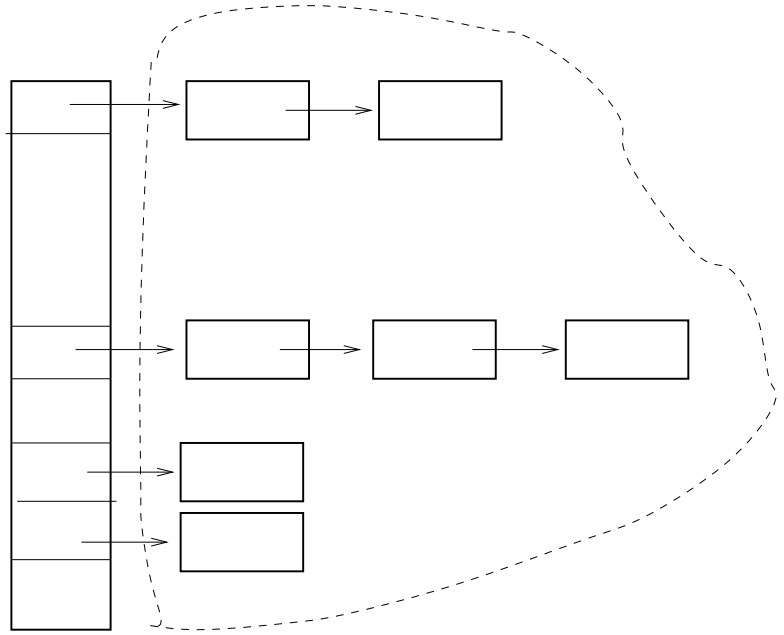


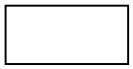
item J+1

(2) We now add this to the table

(1) After J items are in the table, average chain length is J/N.

Locate Time for J-th Item





item J+1

Where is this item added?

To the END of a chain!

Hence, after this addition, if we are looking for this (J+1)th item in the future, we can expect to search to the end of the average chain created **at this point in time** ***when the chain length is average 1+J/N***

Average Locate Time for Any of K Items

Suppose K items have been entered, what is the average time S_K to locate any item?

Since the locate time for the J-th item is approx $(1 + \frac{J}{N})$, and all J items between 1 and K are equally likely, we average over these times:

Thus $S_K = \frac{1}{K} \sum_{J=1}^{K} (1 + \frac{J}{N})$

This sums to approximately $1 + \frac{K}{2N} = 1 + \frac{\alpha}{2}$.

Miscellaneous Remarks on Hashing

- Standard reference is D. Knuth, *The Art of Computer Programming, vol 3: Sorting and Seaching.*
- Most methods have approximate analysis with reasonable assumptions — concrete computational complexity.
- Some omitted techniques: re-hashing for overflow, random probing, digit analysis, simulation studies.
- Best practice recommendation? Use modular arithmetic method for hashing, and chaining for overflow.