
Searching

(and other operations)

in different general purpose data structures

What is this Lecture about?

Search strategies

Algorithms for searching, insertion and deletion in

- arrays
- linked lists
- trees

and their **performance**.

At end, answer to question: **When to use which data structure?**

Simple things first: Arrays

- Most commonly used data storage structure
- Built into most programming languages

Problem: Find a specific data item in an array.

- Array of customer account records identified by account numbers
(Account number is used as a **key** to identify a customer record.)
Find a record given the account number of a customer.

What is the simplest search strategy?

- **Sequential search**
 - How does it work?
 - How does it perform?

Sequential Search in Arrays

Input:

- array of customer records (and its dimensions),
- key account number (to be found in array)

Search routine:

- compare account number of each array element with key
- stop if a match is found or if end of array is reached
- if a match was found remember its position in the array (and locate the customer record at this position)
- else the key (account number) is not in the array

Return: (variety of return values possible)

- position of match (range 0 . . **MAXINDEX**) or “not found” (-1)
- address of customer record or “not found” **NULL**

Write a prototype for a **search** function.

Performance of Sequential Search

Array contains n (distinct) elements

Average case:

- Look through half the array elements
 - ⇒ Takes $n/2$ steps, linear complexity, $O(n)$
- (• What happens when duplicates are allowed?)

Worst case:

- Search for a key that is not in the array
 - ⇒ Examine every element before reporting “not found”
 - ⇒ Takes n steps, linear complexity, $O(n)$


Other Operations on Arrays

Array contains n (distinct) elements

Insertion:

- Insert into first vacant cell in the array
- ⇒ We know how many items the array currently holds (dimensions)
- ⇒ Takes constant time, $O(1)$

Deletion:

- To delete an element you first have to **find** it
- ⇒ Search on average through $n/2$ elements, $O(n)$
-  Holes are not allowed in the array.
- ⇒ Shift contents of each subsequent cell down one space
- ⇒ Move on average the remaining $n/2$ elements
- ⇒ In total deletion takes n steps, linear complexity, $O(n)$

Conclusion: Sequential Search in Arrays

- Sequential search is very straightforward
- On average $n/2$ steps to find an element in an array of size n
- 💡 • Can be time consuming when searching in large arrays

There is a cleverer method:

- **Binary Search** or **Binary Chop** method.

First sort the array, then take advantage of the fact that the array is (now) sorted.

Binary Chop in Ordered Arrays

- Array A of n customer account records identified by account numbers in ascending order

Problem: Find the record with key account number k

Strategy:

- **If search interval is empty** then terminate search with no success (i.e. k was not found in A)
- **Choose** element in middle $A[\text{middle}]$ where $\text{middle} = (n-1)/2$
- **Compare** k with account number at $A[\text{middle}]$
- **If** k and account number of record at $A[\text{middle}]$ are **identical**
 - Terminate search successfully
- **Else if** k is **less than** account number of record at $A[\text{middle}]$
 - Continue searching in array $A[0:\text{middle}-1]$
- **Else** continue searching in array $A[\text{middle}+1:n-1]$

Performance of Binary Search

Analysis: What is maximum number of comparisons/decisions?

- (• Maximum) range n of array to be searched
- (• Maximum) number of decisions s needed to find an element
 - Repeatedly divide range in half until it is too small to divide.
 - Number of **divisions** equals number of **decisions** (comparisons).

n	s
2	1
4	2
8	3
16	4
32	5
64	6
128	7
1024	10

There is an equation for this. $n = 2^s$ or $s = \log_2 n$

Performance of Binary Search

Analysis: What is maximum number of comparisons/decisions?

- (• Maximum) range n of array to be searched
- (• Maximum) number of decisions s needed to find an element
 - Repeatedly divide range in half until it is too small to divide.
 - Number of **divisions** equals number of **decisions** (comparisons).

n	s	n	s
2	1	10	4
4	2	50	6
8	3	100	7
16	4	1.000	10
32	5	10.000	14
64	6	100.000	17
128	7	1.000.000	20
1024	10	10.000.000	24

Number of decisions rises in proportion to **$\log n$** .

What have we gained by using an ordered array?

Assumption: Array contains n (distinct) elements

Search times are much faster (compared to unordered array)

⇒ logarithmic complexity, $O(\log n)$

Insertion takes longer

- Because we have to keep the array in order.
- Elements with higher value must be moved up to make room.

⇒ Takes n steps, linear complexity, $O(n)$

Deletion same time

- Elements must be moved down to fill the hole left by deleted element.

⇒ Takes $\log n + n/2$ steps, linear complexity, $O(n)$

Summary (so far)

Algorithm	Average Running Time in O-Notation
Sequential Search	$O(n)$
Binary Search	$O(\log n)$
Insertion (unordered array)	$O(1)$
Insertion (ordered array)	$O(n)$
Deletion (unordered array)	$O(n)$
Deletion (ordered array)	$O(n)$

Ordered arrays: Useful when searches are more frequent, but insertions or deletions are not.

Searching in other data structures (I)

Assume we now keep our account records in a **linked list**.

Which **search methods** can we use to find a customer account?

Sequential search only!

```
typedef struct L {
    int      account    ...
    struct L *next
} ListElem ;

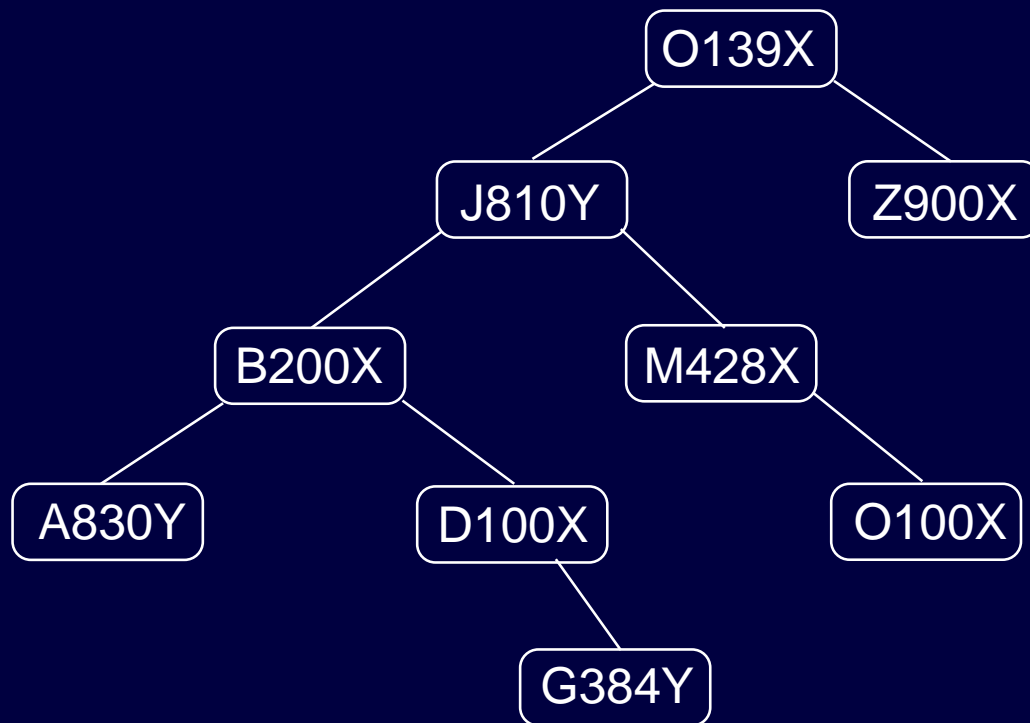
ListElem *search(int account_number, ListElem *record_
    while (record_list != NULL) {
        if (record_list->account == account_number) {
            return record_list }
        record_list = record_list->next
    }
    return 0
}
```

Searching in other data structures (II)

Assume we now keep account records in a **binary search tree**.

A **binary tree** where a node \mathfrak{N} with key \mathfrak{K} is inserted so that

- keys in left subtree of \mathfrak{N} are less than \mathfrak{K} , and
- keys in right subtree of \mathfrak{N} are greater than \mathfrak{K} .



Where to insert **N401X** ?

Search in a Binary Search Tree

To search for a key κ in a binary tree \mathcal{T} :

- **If tree is empty** then terminate search with no success
- **Compare** κ to the key κ_R in the root of \mathcal{T}
- **If** $\kappa == \kappa_R$
 - Terminate search successfully
- **Else if** $\kappa < \kappa_R$
 - Continue searching in the left subtree of \mathcal{T}
- **Else** continue searching in the right subtree of \mathcal{T}

We have seen a similar strategy before: Binary search in ordered array

Performance of Binary Search Trees

Search:

- Best case
 - Tree is balanced $\Rightarrow O(\log n)$

- Worst case
 - Tree is as deep and as skinny as possible
 - * left linear, right linear, or zig zag shapes
 - (– When does this happen?) $\Rightarrow O(n)$

- Average case:
 - On random input data tree is almost balanced $\Rightarrow O(\log n)$

Speeds of General Purpose Data Structures

Assume a problem size n

Data Structure	Search	Insertion	Deletion	Traversal
Array	$O(n)$	$O(1)$	$O(n)$	-
Ordered array	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$
Linked list	$O(n)$	$O(1)$	$O(n)$	-
Ordered linked list	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Binary tree (best case)	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Binary tree (worst case)	$O(n)$	$O(n)$	$O(n)$	$O(n)$
(Next lecture: Hash table)	$O(1)$	$O(1)$	$O(1)$	-)

The one that beats it is hashing, next lecture...

Where to get more information?

Thomas A. Standish

“Data Structures, Algorithms & Software Principles in C”

Addison-Wesley, 1995

Various Chapters: Ch1 p. 16 ff, Ch2 p. 43 ff, Ch6 p. 241 ff ...

J. Glenn Brookshear

“Computer Science - An Overview” (sixth edition)

Addison-Wesley, 2000

4.5 Recursive Structures (p. 196 ff)

D.E. Knuth

“The Art of Computer Programming”

Volume 3, third edition, Addison-Wesley Longman, 1998

Hashing: Fast access to data records.

What does *hashing* mean?

When is hashing used?

Different hashing techniques

Problems with hashing and how to overcome them

Assignment involves hashing.

Hashing: Fast Access to Data Records

Data record **(K,I)** consists of a unique **key, K**, and some **information, I**, related to the key.

Requirement: Fast (constant time) access to data records.

- Store employee records for a company that employs 5000 people
 - use NI number as key, **K**

How to organise 5000 data records, so that data retrieval is as efficient as possible?

The Basics of Hashing

Hashing is typically used when:

- **Fast (constant time) access** to data records is required.
- Out of all possible **key values** in the data records, only a **small number** is actually **used**.
 - * NI number has 9 places, e.g. RT 846201 K. In a company with 5000 employees, only 5000 NI numbers will be used at any one time.
 - * Address book, only few names are valid.

Instead of storing the data records in a sequential order, e.g. a linked list (sequential access only), or using the key directly as an index into a table (table would typically be far too big), **the key of a data record is mapped to a table index which is within a smaller range of values.**

The function which defines this mapping is called **hash function**.

Key Idea of Hashing

Given:

- key, $\mathbf{K} \in K$, of a data record (\mathbf{K}, \mathbf{I})
- “hash” table, \mathbf{T} , of data records
 - indexed by values, $\mathbf{A} \in A$, called *(hash) addresses*

Wanted:

- location (or address) of record identified by \mathbf{K} in \mathbf{T}

Compute location of data record (\mathbf{K}, \mathbf{I}) in \mathbf{T} by applying a **hash function, h** , to the key, \mathbf{K} .

- $\mathbf{h} :: K \mapsto A$

The address $\mathbf{h}(\mathbf{K})$ is used as **entry point** into \mathbf{T} .

\Rightarrow This allows quick access to the data record (\mathbf{K}, \mathbf{I}) .

The Hash Table

Abstract storage device that models the storage and retrieval properties of tables.

- Collection of **table entries (K,I)**, where **K** is a key and **I** is some information associated with **K**, so that *no two distinct entries have the same key*.
- **Operations** on a table:
 - **Initialise** to the empty table.
 - Determine whether or not table is **full**.
 - **Retrieving, updating, inserting and deleting** table entries.
 - (Typically no traversal operation!)

⇒ Will show later that this is an Abstract Data Type.

Organising data in a hash table (I)

The **size, m** , of a hash table defines:

- Number of data records that can be stored in table.
- Number of indexes (addresses) available.

How to project the key values onto m table indexes?

Assuming table size $m = 40$, and keys K are numeric.

- **Division method:**

- Divide any key by size, m , of table, using integer division.

- This gives:

- quotient (integer value)

- remainder (integer value) **Always in the range of 0..39!**

- Exactly one table index can be related to each possible remainder.

⇒ **Hash function: $h(K) = K \bmod 40$**

Organising data in a hash table (II)

Task 1: Transfer any **key field value** into a numeric value.

- Key **K: 25X3Z**

ASCII: **0110010 0110101 1011000 0110011 1011010**

Equivalent base 10 value: **13,534,370,266**

Result: Numerical key.

Alternatively, sum the ASCII values of the characters in the key.

- Key **K: 25X3Z**

Sum of ASCII values: **50 + 53 + 88 + 51 + 90 = 332**

Organising data in a hash table (III)

Task 2: Convert any **numeric key into** an integer that identifies where the corresponding data record is stored in the table, i.e. **the table index of the data record**.

⇒ Application of the hash function to the numeric key value.

- Key: **13,534,370,266**

Hash function: **$h(K) = K \bmod 40$**

Hash address: **$26 = h(13,534,370,266)$**

Result: Storage location number (address) in hash table.

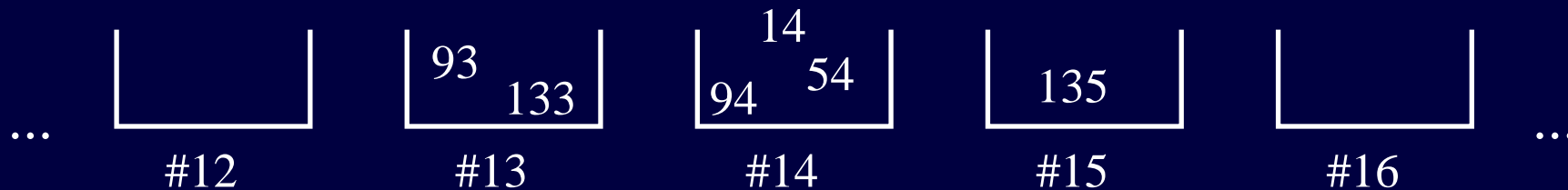
Organising data in a hash table (IV)

Inserting a data record (K,I) into a hash table T :

- Convert key K into an integer, giving K' .
- Apply the hash function, h , to K' .
- Insert (K,I) into T at address $h(K')$.

Retrieving a data record (K,I) from a hash table T :

- Compute the address of the record as above.
- Access T at the computed address.
- **Search the data records** at that address **for the record with key K .**
 - Example for $h(K) = K \bmod 40$



Problems with Hashing (I)

The **explanation** so far was **simplified**. It overlooked complications.

Once a **hash function** is chosen, there is no more control over which address will hold which record(s).

- **$h(K) = K \bmod 40$**

- Used on keys that tend to be multiples of 40

- ⇒ Index #0 will **overflow**.

- Used on keys that tend to be multiples of 5

- ⇒ Data records will **cluster**, and **overflow** the indexes

- #0, #5, #10, . . . , #35.

- Similar for key values that are factors of 40, i.e. 2, 4, 8,

- ⇒ **Minimise overflow and clustering** by selecting the table size to have as few factors as possible. **Use a prime number!**

- Instead of 40, use 37 or 41.

- ⇒ **Hash functions need to be chosen very carefully.**

Choosing a Hash Function (I)

Division method:

- Choose table size, m a prime number gives the best spread.
- Interpret keys, K , as integers.
- Hash function: $h(K) = K \bmod m$
 - $K = 031782$, $m = 199$, $h(031782) = 141$
 - * Tends to **spread keys uniformly and randomly** if m is a prime.
 - ⇒ Otherwise **clustering** can occur.

Folding method:

- Divide key into sections and add these together.
 - $K = 031\ 782$, $h(031\ 782) = 48$

Choosing a Hash Function (II)

Middle squaring method:

- Take middle digits of key and square them.
 - $K = 031782$, $h(031782) = 289$

Truncation method:

- Delete part of key and use remaining symbols.
 - $K = 031782$, $h(031782) = 31$
 - + Takes hardly any time to compute.
 - | Tends not to spread keys uniformly and randomly.
 - ⇒ Often **used in combination** with other methods.

Choosing a Hash Function (III)

In general

- Aim is to select a hash function that distributes keys **uniformly and randomly**.
- Choice is often based on:
 - statistical analysis,
 - ability and experience of programmer,
 - rules of thumb,
 - test results.
- **Different hashing methods** can be **combined** to reduce the disadvantages of a single method.
- **Test performance** of a hash function on sample data records **BEFORE** making final choice!

Problems with Hashing (II)

However, ultimately **collisions** will cause overflow regardless of the hash function.

💣 Collisions are relatively frequent.

According to *von Mises Birthday Paradox*, if there are **23 or more people in a room**, the chance is **greater than 50%** that **2 or more** of them will have the **same birthday**.

- See a year as a hash table with 365 slots. If 23 entries are made randomly into the table.
- The chance is greater than 50% that two or more of them fall into the same slot.

The (birthday) table is **only 6.3% full!** (Try it out!)

Problems with Hashing (III)

A **collision between** two keys, **K1** and **K2**, occurs if,

- when trying to store both keys in a hash table,
- **both keys** have the **same hash address**, **$h(K1) = h(K2)$** .

How to find an empty table entry *in which to store the data record with key K2, if the table location given by the hash address $h(K2)$ is already occupied by another data record with key K1?*

\Rightarrow Collision resolution strategies are needed.

Resolving Collisions: A simple Example

- Address book

Indexed A-Z, one index letter per page.

- **Direct chaining:**

- Allow more than one entry per page.
- All names starting with the same letter go into one page.

- **Open addressing:**

- Allow only one entry per page (or a fixed number).
- If a page is already occupied (full), use the next free one.

Resolving Collisions (I)

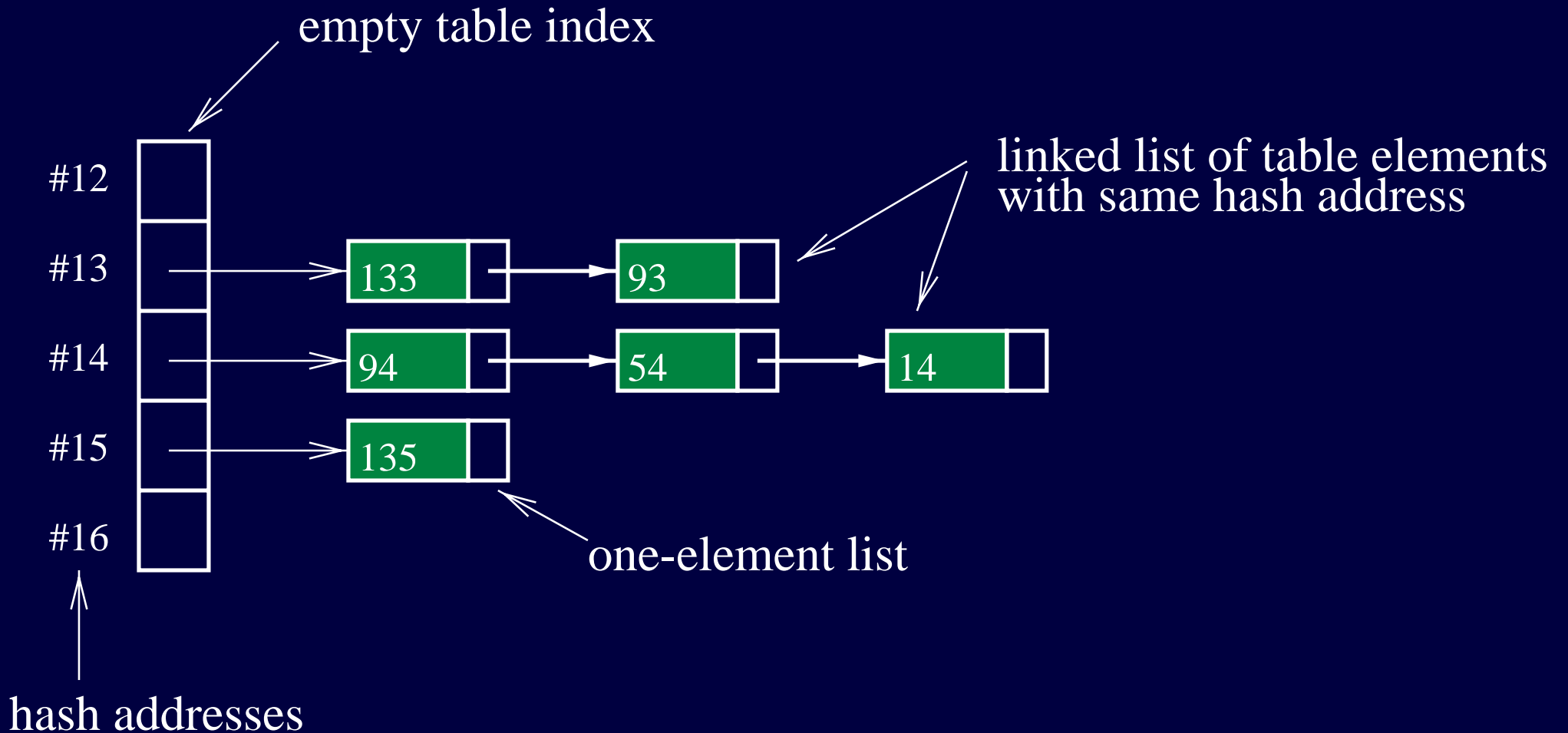
Direct chaining:

- **Extend each data item** to include a pointer to another item.
 - Use composite data structures `struct` in C.
- **Initialise** this pointer to a `NULL` pointer.
- When a **collision occurs**, space for the collision item can be allocated dynamically. The pointer is used to record the location of the collision item.

The table contains pointers, which are:

- `NULL` pointers for empty indexes, or
- pointers to the first item of a linked list of items with the same hash address.


Direct chaining on an example



Resolving Collisions (II)

Direct chaining (continued):

For **data retrieval**

- **Enter** hash table at hash address $h(K)$ computed for given key, K .
- If key of first record in linked list is not equal to K , **search through list sequentially**.
-  If collisions occur very often,
 - the overflow lists grow and
 - ⇒ the search efficiency drops!

Resolving Collisions (III)

Open addressing:

- Data records which cause a collision are stored in the **next available location** in the hash table.
- If table slot for particular hash address is already full:
 - **search remaining table** sequentially until a **vacant slot** is found,
 - **insert** the collision record there.
- 💡 For data insertion and retrieval the table needs to be **searched circularly!**

Animation at:

<http://www.cs.pitt.edu/~kirk/cs1501/animations/Hashing.html>

Resolving Collisions (IV)

Hash bucket methods:

- **Divide a big hash table** into a number of smaller subtables, called *buckets*.
- **Hash function** maps a key into one of the buckets, where data records are stored sequentially.
- Bucket methods work quite well for large collections of data stored on external memory devices.

- 💣 Important to find right **compromise** between table size and bucket size.

- 💣 Even buckets will eventually overflow.
⇒ A strategy for dealing with overflowing buckets is still needed.

Summary (I)

Hashing: Method for storing and retrieving information in tables with constant time complexity.

Items of information, I , are stored in **table entries** of the form **(K,I)** , identified by a **unique key K** .

Given the key K , to find (K,I) :

- A **hash address** is computed by applying a **hash function, h** to **K** .
- The address **$h(K)$** is the **table index** where the search for **(K,I)** can start.
- If the entry cannot immediately be found at **$h(K)$** , a **collision resolution strategy** is invoked.

Summary (II)

The **hash function $h(\mathbf{K})$** maps keys \mathbf{K} to hash table addresses (indexes).

Ideally, it maps keys

- **uniformly** and
- **randomly**

into the **entire range** of the table indexes.

A well-designed hash function **spreads clusters of keys**.

Each table index is equally likely to be the target of $h(\mathbf{K})$ for a randomly chosen \mathbf{K} .

⇒ Use first step of random number generation, i.e. key is seed.

Conclusion

The **design of a hash table** requires a careful analysis of:

- the **size** of the table,
- the **hash function**, and
- the **collision resolution strategy**.

Aim

Efficient (constant time) access to data records over a long time of insertions and deletions of data records.

Where to get more information?

Thomas A. Standish

“Data Structures, Algorithms & Software Principles in C”

Addison-Wesley, 1995

Chapter 11: Hashing and the Table Abstract Data Type (p. 450 ff)

J. Glenn Brookshear

“Computer Science - An Overview” (sixth edition)

Addison-Wesley, 2000

8.5 Hashing (p. 385 ff)

A. D. Dewdney **“The New Turing Omnibus”**

Computer Science Press, 1997

Chapter 43: Storage by Hashing (p. 288 ff)

Applications of hash tables

There are a variety of applications, here are two of them:

- **Address generation for files held on mass storage.**
 - Hash function determines **disk block address** for a record.
 - Allows records to be accessed quickly.
- **Production of symbol tables by compilers.**
 - **Identifiers in the source code** need to be recorded, together with details about their type, value, etc.
 - Entries are accessed very frequently.
 - Access needs to be made as efficient as possible.