Printing Requisition / Graphic Services

11511

1. **Please complete unshaded areas** on form as applicable.
2. **Distribute copies** as follows: White and Yellow to Graphic Services. **Retain Pink Copies for your records.**
3. On completion of order the **Yellow copy will be returned** with the printed material.
4. **Please direct enquiries, quoting requisition number and account number**, to extension 3451.

TITLE OR DESCRIPTION
CS-80-23 A Handbook of Algorithms and Data Structures

DATE REQUISITIONED: October 24, 1984
DATE REQUIRED:
ACCOUNT NO.: 1 2 6 4 4 3 1 0 2

REQUISITIONER – PRINT:
PHONE:
SIGNING AUTHORITY: /A. Harris

MAILING INFO – NAME: D. McCracken | DEPT.: Computer Science | BLDG. & ROOM NO.: 6081E | [ ] DELIVER [ ] PICK-UP

Copyright: I hereby agree to assume all responsibility and liability for any infringement of copyrights and/or patent rights which may arise from the processing of, and reproduction of, any of the materials herein requested. I further agree to indemnify and hold blameless the University of Waterloo from any liability which may arise from said processing or reproducing. I also acknowledge that materials processed as a result of this requisition are for educational use only.

NUMBER OF PAGES: 68
NUMBER OF COPIES: 2

TYPE OF PAPER STOCK
[X] BOND [ ] NCR ____ PT. [ ] COVER [ ] BRISTOL [ ] SUPPLIED [ ] ____

PAPER SIZE
[X] 8½ x 11 [ ] 8½ x 14 [ ] 11 x 17 [ ] ____

PAPER COLOUR: [X] WHITE [ ] ____
INK: [ ] BLACK [ ] ____

PRINTING: [X] 1 SIDE ____ PGS. [ ] 2 SIDES ____ PGS.
NUMBERING: FROM ____ TO ____

BINDING/FINISHING
[X] COLLATING [X] STAPLING [ ] ____ HOLE PUNCHED [ ] PLASTIC RING

FOLDING/PADDING:
CUTTING SIZE:

Special Instructions

Kocher K

COPY CENTRE — OPER. NO.: 34 BLDG.: MC MACH. NO.:

DESIGN & PASTE-UP — OPER. NO. | TIME | LABOUR CODE

TYPESETTING — QUANTITY | OPER. NO. | TIME | LABOUR CODE

PROOF
P R F
P R F
P R F

NEGATIVES — QUANTITY | OPER. NO. | TIME | LABOUR CODE
F L M
F L M
F L M
F L M
F L M

PMT
P M T
P M T
P M T

PLATES
P L T
P L T
P L T

STOCK

BINDERY

OUTSIDE SERVICES

$ ____ COST

UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT

UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT

UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT

A Handbook of Algorithms and Data Structures

Gaston H. Gonnet

CS-80-23

May, 1980

University of Waterloo
Waterloo, Ontario, Canada

Table of Contents

1. Introduction

This is a preliminary version of the 0^{th} edition of the Handbook of Algorithms and Data Structures.

This handbook is intended to contain most of the tabular information available on algorithms and their data structures; thus it is designed to a wide spectrum of users, from the programmer who wants to code efficiently, to the student, or researcher who needs quick information.

In one way or another, computer science is not so much a science as an art. Although tremendous effort is being made in the theory and in the practice of the profession to improve this condition, we still cannot compare our software technology with other technologies. For example the construction of the CN tower in Toronto is a task similar in magnitude to the construction of a major operating system; however, it is unthinkable letting the tower fall a few times until its design is "debugged". We hope that this handbook will contribute to an improvement in the quality and reliability of computer science technology.

This early version is about 35% of its target size. It is intended for internal use in the University of Waterloo. The table of contents gives the flavour of the algorithms missing at present. Moreover algorithms will be presented in more than one language. Depending on the practicallity of the algorithm they will be coded in Pascal and/or Cobol and/or Fortran.

2. Basic Concepts and Notation

An algorithm is a function which operates on data structures. More formally, an algorithm is a map from $S \times Q \rightarrow R$, where S,Q, and R are all structures; S is called the Data Structure, Q is called the Query, and R is the Result.

The two following examples will make this formal definition more understandable.

(1) B-tree insertion: This is an algorithm which inserts a new record Q into a B-tree S, giving a new B-tree as a result. In function notation,

B-tree insertion: B-tree $\times$ new record $\rightarrow$ new B-tree.

(2) Quicksort: This algorithm takes an array and sorts it. Since there is no data structure acting as a Query in this case, we write

Quicksort: Array $\times$ nil $\rightarrow$ Sorted Array.

Algorithms can be described in several ways: verbally, by flowcharts, or with the aid of an algorithmic language. We will use this last approach, since such languages tend to make algorithms easy to describe, easy to understand, and easy to transmit. Furthermore, it is usually a straightforward matter to translate an algorithmic language into genuine computer code; indeed, most algorithmic languages are themselves programming languages.

To understand the process of designing an algorithm, we must consider three factors:

(a) The objects (data structures) on which the algorithm operates;

(b) The basic operations an algorithm may perform;

(c) The building operations, i.e. the way basic operations may be combined.

We will find that describing algorithms in terms of these three components not only makes the algorithms easier to understand, but sometimes leads to useful variations and occasionally to new algorithms altogether.

2.1. The Objects Definition

When we study analytic functions, we must first be familiar with the complex number system. In the same way, when we study algorithms, we must first look at data structures, since they are the objects which an algorithm takes as input and yields as output.

Data structures generally have rules of creation, syntactic rules, which allow us to build complicated structures from simpler ones, and semantic rules, which indicate which structures will be considered valid and which will not.

2.1.1. The Data Structures

To describe the data structures dealt with in this handbook and their rules of creation, we will use what is known as a W-grammar (also called a two level grammar or a van Wijngaarden grammar). Actually, we will not need the full capabilities of a W-grammar; all we require here are the standard BNF productions and the uniform replacement rule which we shall describe shortly.

A W-grammar generates a language in two steps (levels). In the first step, a collection of generalized rules are used to create more specific production rules. In the second step, the production rules generated in the first step are used to define the actual data structures.

This two-step process can be understood more easily using the following illustration. A sequence of real numbers can be defined by the BNF production

S :: [{ real, S }] | nil .

Thus a sequence of reals can have the form nil, [{ real, nil }], [{real, [{real, nil}] }] , and so on. Now in a similar way we could define sequences of integers, characters, strings, boolean constants, etc. However, this would make for a bulky collection of production rules which are all very much alike. One might first try to get around this repetitiveness by defining

S :: [{D,S}] | nil

where D is given as the list of data-types,

D :: real | int | bool | string | char .

However, this pair of productions generates unwanted sequences such as

[{real, [{int, nil}] }]

as well as the kind of sequences we are looking for. Thus we need a different approach.

In a W-grammar, we solve the problem of listing repetitive production rules by starting out with generalized rule-forms rather than the rules themselves. We let D now be given as

D :: real; int; bool; string; char; ...

To distinguish this from a simple production, we will call it a metaproduction. The generalized form of a sequence S is given by the hyperrule

s-D :: [{D,s-D}] | nil .

Under this new system, we define a sequence of real numbers in two steps. The first step consists of choosing a value to substitute for **D** from the list of possibilities given by the metaproduction above. In this instance we select

D → real.

Next we make use of a W-grammar's uniform replacement rule which allows us to substitute the string **real** for **D** everywhere it appears in the hyperrule which defines **s−D**. This substitution gives us

s-real :: [{real, s-real}] | nil .

Thus we have used the metaproduction and the hyperrule to generate an ordinary BNF production defining real sequences. The same two statements can generate a

production rule for sequences of any other valid data-type (integer, character, etc.).

We are ready to define the W-grammar which will generate the data structures we wish to work with. Our metaproductions are

D ::	**real**;**int**;**bool**;**string**;**char**;...	# atomic data types #
	$\{$ **D** $\}_{\mathbf{N}}$;	# array #
	{ **DS** } ;	# record #
	[**D**] ;	# reference #
	s−D;	# sequence #
	bt−D−LEAF;	# binary tree #
	mt−D−LEAF;	# multiway tree #
	gt−D−LEAF;	# general tree #
	tr−D−LEAF;	# data structure for Tries #
	... .	

Thus all of the above are valid data-type substitutions for **D**. Other metaproductions are

DS :: **D**; **D, DS**.
LEAF :: **nil**; **D**.
N :: **DIGIT**; **DIGIT, N**.
DIGIT :: 0;1;2;3;4;5;6;7;8;9.

Our hyperrule definitions are

HR[1] **data structure** : **D**
HR[2] **s−D** : [{ **D,s−D** }] ; **nil**.
HR[3] **bt−D−LEAF** : [{ **D,bt−D−LEAF,bt−D−LEAF** }] ; **LEAF**.
HR[4] **mt−D−LEAF** : [{ **N,**$\{$**D**$\}_{\mathbf{N}}$**,**$\{$**mt−D−LEAF**$\}_{\mathbf{0}}^{\mathbf{N}}$ }] ; **LEAF**.
HR[5] **gt−D−LEAF** : [{ **D,s−gt−D−LEAF** }] ; **LEAF**.

As an example, consider what happens when we let

D→real and **LEAF→nil**.

With these substitutions, HR[3] generates the production rule

bt−real−nil : [{ **real,bt−real−nil,bt−real−nil** }] | **nil**

This production rule defines a binary tree which has a real entry in each node. Since **bt−real−nil** is one of the legitimate values for **D** according to the metaproduction for **D** we let

D→bt−real−nil

and learn from HR[1] that such a binary tree is a legitimate data structure.

As another example we can generate a production rule for B-trees of strings using HR[4] and the proper substitutions to yield

mt−string−nil : [{ 10,$\{$**string**$\}_{1}^{10}$,$\{$**mt−string−nil**$\}_{0}^{10}$ }] | **nil**.

This is a multi-way tree in which each node contains ten keys and has eleven descendants.

A data structure that is derived from a hyperrule that contains its lefthand side symbol in the righthand side is called a *recursive* data structure. Hyperrules 2 to 5 all generate recursive data structures. The object defined by the lefthand side symbol is called the *parent* of the objects defined by the righthand symbols (*descendants*).

2.1.2. Semantic Rules or Constraints

A semantic rule or constraint may be regarded as a boolean function on data structures ($S \rightarrow$ **bool**) that tells us which structures are valid and which are not. The objects defined are those in the intersection of the objects produced by the W-grammars and those that satisfy the constraints. Whenever we perform an operation which modifies a data structure (additions, deletions, etc.) we may violate some of these semantic rules. The restructuring activity done by an algorithm to restore validity is called organization. Below we list some examples of semantic rules which may be imposed on data structures. Note that these constraints are additional ones placed on data structures which have been legitimately produced by the rules given in the previous section. Thus when we refer here to an invalid data structure we are not speaking of one that has been illegally produced, but rather a legal data structure which merely does not satisfy all the additional requirements we name in a given application.

Order

Many data structures are kept in some fixed order (e.g. the records in a file are often arranged alphabetically or numerically according to some key). Whatever work is done on such a file should not disrupt this order.

Height Balance

Let s be any node of a tree (binary or multiway). Define $h(s)$ as the height of the subtree rooted in s, i.e. the maximum number of nodes one must pass through to reach the end of branch starting at s. One structural quality that is sometimes required is that the height of a tree along any pair of adjacent branches should be approximately the same. More formally, we require that

$$|h(s_1)-h(s_2)| \leqslant \delta$$

where s_1 and s_2 are any two subtrees of the same node, and δ is a constant giving the maximum allowable height difference. In B-trees, for example, we have $\delta=0$, while in AVL-trees, $\delta=1$.

Weight Balance

For any tree, we define the weight function $w(s)$ as the number of nodes in the subtree rooted at s. A weight balance condition requires that for all nodes s_1 in the subtree rooted at s

$$\frac{w(s_1)}{w(s)} \leqslant r$$

where r is a positive constant less than 1.

Lexicographical Trees

A lexicographical tree is a tree which satisfies the following condition for every node s: If s has n keys $(key_1, key_2, \ldots, key_n)$ stored in it, s must have $n+1$ descendant subtrees $t_0, t_1, \ldots, t_n$. Furthermore, if d_0 is any key in any node of t_0, d_1 any key in any node of t_1, and so on, we have the inequality

$$d_0 \leqslant key_1 \leqslant d_1 \leqslant \ldots \leqslant key_n \leqslant d_n$$

Priority Queues

A priority queue can be any kind of recursive structure in which an order relation has been established between each node and its descendants (most priority queues are implemented using recursive data structures). One example of such an order relation would be to require that $key_p \leqslant key_d$, where key_p is any key in a parent node, and key_d is any key in any descendant of that node.

Brent's Reorganization and Binary Tree Hashing

Brent's reorganization applies to the insertion of keys in a hashing table. A key is inserted in the location which minimizes the total cost of accessing every element in the table when we are allowed to move only one other table element ahead in its hashing path.

The binary tree hashing scheme is similar to Brent's, but here we are allowed to move ahead more than one element already in the table if it decreases the total access cost. The optimal reorganization algorithm would require us to move any number of keys either forward or backwards in their hashing paths.

Optimality

Any condition on a data structure which minimizes a complexity measure (such as the expected number of accesses; maximum number of comparisons) is an optimality condition. If this minimized measure of complexity is a worst case value, we call the value the *minimax*; when the minimized complexity measure is an average value, it is the *minave*.

2.2. Basic or Atomic Algorithms

One cannot define a set of basic operations for algorithms without paying some attention to the building operations as well. After all, the richer the set of building operations we have, the simpler (and possible fewer) atomic operations we need to construct usable algorithms. On the other hand, with a large collection of basic operations, we may not need many building operations to be able to construct the algorithms we want. It is possible that there are an infinite number of ways to define basic operations and building operations which will produce equivalent algorithms. Thus we do not claim that the division of operations we will make is in any way unique or optimal; our motivation for choosing the following system is simply that it seems natural. Moreover, we do not intend to present a formal proof that our atomic operations are indivisible, since we will not be treating these operations formally enough to justify such a proof. We prefer to

list a few of our basic operations below and describe them in a way that conveys their flavour rather than taking a completely rigorous approach.

Direct Addressing

Many data structures consist of a collection of records which are each distinguished by an identification key. Direct addressing uses the actual record key (or its binary configuration) as an integer. This integer will normally be used as an index into an array.

Multiway (Binary) Decision

This operation is defined as ranking a scalar X in a set of different scalars $X_1, X_2, \ldots, X_n$. By ranking, we mean finding out how many of the X_j values are less than or equal to X, thus determining what rank X would have if all the values were ordered. More precisely, ranking is finding an integer i and a subset $A \subseteq \{X_1, X_2, \ldots, X_n\}$ such that

$$|A| = i$$
$$X_j \in A \Rightarrow X \geqslant X_j$$
$$X_j \notin A \Rightarrow X < X_j.$$

A binary decision is the simplest case of a multiway decision. In a binary decision $n = 1$, and i is zero if $X < X_1$, one otherwise.

Hashing

Like direct addressing, hashing is an operation which normally makes use of a record key. Rather than using the actual key value however, hashing transforms the key into an integer in a prescribed range by means of a hashing function and then uses the generated integer value. We would like this hashing transformation $h: K \rightarrow h(K)$ to display some randomness; thus we require that the distribution of the values $h(K)$ be discrete rectangular for a random key K.

Interpolation

This operation computes a tentative location for a record in a file, basing its guess on the value of the record's key, the number of records in the file, the values of the smallest and largest keys, and the distribution of the values of the keys throughout the file. Interpolation normally gives the statistical mode of the location of the desired record in an ordered file, i.e. the most probable location of the record.

Collision Resolution

This operation assumes that a file has been partitioned into a number of sets of records. The algorithm then stores these sets in a linear array according to some prescribed rule. Collision Resolution is closely linked to hashing algorithms.

2.3. Building Operations

Building operations allow us to combine basic algorithms to produce more complicated ones. The definition of these building procedures thus provides a criterion for deciding whether or not an algorithm is basic: the algorithm is basic

(or atomic) if it has not been built from simpler operations. In this section, we will define four building operations.

2.3.1. Composition

Composition is our main operation for producing algorithms from atomic operations. Typically, but not exclusively, the composition of F_1 and F_2 can be expressed in a functional notation as $F_2(F_1(S,Q_1),Q_2)$. A more general and hierarchical description of composition is that F_2 uses F_1 instead of a basic operation.

Although the formal definition is enough to include all types of composition, we may indentify several common structures of composition.

Divide and Conquer

is a composition of two algorithms. The first is used to split a problem into (usually two) smaller problems. The composed algorithm is then recursively applied to each non-empty component. Finally the second algorithm is used to assemble the components' results into one result.

Iterative Application

takes an algorithm and a sequence of data structures as parameters. The algorithm is iteratively applied using successive elements of the sequence in the place of the single element for which it was written (e.g. insertion sort, relational join).

Tail Recursion

takes one algorithm as its parameter. This algorithm specifies the criterion for splitting a problem into (sometimes two) components and selecting one of them to be solved recursively (e.g. binary search, priority queue merge).

Inversion

is a composition of two search algorithms used to search for secondary keys (i.e. we expect repetitions of the key values). The first algorithm is used to search for the set of keys that have the searched value and the second search algorithm is used to search within the set. The data structures required by the two composed algorithms are usually called *inverted files*.

Digital Decomposition

is applied to a problem of size n by attacking preferred-size pieces. An algorithm is applied to all these pieces to produce the desired result. It is assumed that we have an algorithm that works very well for these preferred-size problems (e.g. binary (Bentley & Saxe) decomposition).

Merge

applies an algorithm, and a discarding rule to two sequences of data structures. The algorithm is iteratively applied using successive elements of the sequences in place of the single elements for which it was written. The discarding rule controls the iteration process (e.g. set union, intersection, merge sort, polynomial addition).

2.3.2. Superimposition

This building operation combines two or more algorithms, allowing them to operate on the same data structure more or less simultaneously. Two algorithms F_1 and F_2 may be superimposed over a structure S if $F_1(S,Q_1)$ and $F_2(S,Q_2)$ can both operate together. A typical example of this situation is a file that can be searched by one attribute using F_1 and by another attribute using F_2. Interleaving is a special case of superimposition. This happens when one algorithm does not need to wait for other algorithms to terminate before starting its execution. For example one algorithm might add records to a file while a second algorithm makes deletions; interleaving the two would give an algorithm which performs additions and deletions in a single pass through the file.

2.3.3. Organization

If an algorithm creates or changes a data structures, it is sometimes necessary to perform more work to ensure that semantic rules and constraints on the data structure are not violated. For example, when we insert or delete nodes in a tree, its height balance may be altered. Then we will have to perform some action to restore the balance in the new tree. When we perform changes to the records in an ordered file, we may find ourselves forced to re-sort the file because our record modifications have disrupted the prescribed ordering. In effect we have combined two algorithms: the original modification algorithm and the sorting algorithm. This process of combining an algorithm with a "clean-up" operation on the data structure involved is called *organization* (sometimes *reorganization*).

2.3.4. Self-Organization

This is a supplementary heuristic activity which an algorithm may often perform in the course of querying a structure. Not only does the algorithm do its primary work, it also reaccomodates the data structure involved in a way designed to improve the performance of future queries. For example, a search algorithm may promote the searched element once it is found so that future searches through the file will locate this record more quickly.

2.4. Interchangeability

It is easy to see that there is some interchangeability among data structures, basic algorithms and semantic rules. For example, in place of a list of real numbers, we could use an array of real numbers and simply treat it in sequential fashion. We may use either height or weight balance criteria to eliminate sizable irregularities in tree structures.

Without going into detail, we can establish what amounts to equivalence classes of objects in each of these areas. By this we mean that whenever we have an algorithm which uses one element in an equivalence class, we can create a new algorithm by replacing that element with one of its equivalents. Some of these equivalence classes are listed below.

Data Structures	{array (used linearly); sequence} {binary trees; multiway trees}
Basic Algorithms	{hashing; interpolation; direct addressing} {collision resolution methods} {binary partition; Fibonaccian partition; median partition; mode partition}
Semantic Rules	{height balance; weight balance} {lexicographical order; priority queues} {Ordered hashing; Brent's hashing; Binary tree hashing} {minimax; miniave}

2.5. Examples

In this section we give examples of the classification of algorithms according to the above criteria. We also suggest some possible variations of known algorithms.

2.5.1. Direct chaining hashing is the sequential composition of a hashing step with a sequential search through a linked list. We could improve its performance by incorporating a step which self-organizes the lists, thus obtaining a new algorithm.

2.5.2. The Interpolation Search is a tail recursion composition where we interpolate, compare the probe position and select part of the file to continue the search recursively. Sequentially composing the interpolation operation with a Sequential Search in the proper direction gives the Interpolation Sequential search algorithm.

2.5.3. If we sequentially compose one interpolation step with sequential lists search we obtain an algorithm that behaves exactly as direct chaining hashing. This algorithm has the advantage that range searches are possible and efficient and that the file is very close to total order compared to hashing.

2.5.4. If we compose hashing with interpolation, in other words we interpolate on the result of a hashing function applied on a key, instead on the keys themselves, we obtain a new algorithm. This algorithm is as efficient as interpolation search and does not suffer the same difficulties as pure interpolation search does when the keys are not uniformly distributed.

2.5.5. Composing interpolation with the linear collision resolution scheme produces an interesting algorithm which is similar to linear probing but which constructs an almost ordered table. From this we can derive a fairly efficient sorting method which we might call the Linear Probing Sort.

2.6. Notation

Many of the complexity measures in this handbook are for situations where the size of the problem increases asymptotically. The asymptotic notation we will use is given below.

$f(n) = O(g(n)) \Rightarrow$ there exists k and n_0 such that $|f(n)| < kg(n)$ for $n > n_0$.

$$f(n) = o(g(n)) \Rightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$f(n) = \Theta(g(n)) \Rightarrow$ there exists k_1, k_2, $(k_1 \times k_2 > 0)$ and n_0 such that $k_1 g(n) < f(n) < k_2 g(n)$ for $n > n_0$.

$f(n) = \Omega(g(n)) \Rightarrow g(n) = O(f(n))$.

$f(n) = \omega(g(n)) \Rightarrow g(n) = o(f(n))$.

$f(n) \approx g(n) \Rightarrow f(n) - g(n) = o(g(n))$.

Whenever we write $f(n) = O(g(n))$ it is with the understanding that we know of no better bound, i.e. we know of no $h(n) = o(g(n))$ such that $f(n) = O(h(n))$.

The probability of a given event is denoted by $Pr\{event\}$. Random variables will always be capitalized. The expected value of a random variable X is written $E[X]$ and its variance is $\sigma^2(X)$.

At the end of each major section in the following chapters, we will list some references related to the material in the section.

Our algorithm descriptions will all have roughly the same format, though we may make slight deviations or omissions when information is unavailable or trivial. The general format is as follows.

(1) Definition of the algorithm in words and classification according to the basic operations described in this section.

(2) Theoretical results on the algorithm's complexity. We are mainly interested in measurements which indicate an algorithm's running time and its space requirements. Useful quantities to measure for this information include the number of comparisons, data accesses, assignments, or exchanges an algorithm might make. When looking at space requirements we might consider the number of words, records, or pointers involved in an operation. Time complexity covers a much broader range of measurements. For example, in our examination of Search algorithms, we might be able to attach meaningful interpretations to most of the combinations of

the { average / variance / minimum / worst case / average w.c. } of { comparisons / accesses / assignations / exchanges } when we { query / add a record into / delete a record of / modify a record of / reorganize / create / read sequentially } the structure.

Other theoretical results may also be presented, such as enumerations, generating functions, or behaviour of the algorithm when the data elements are distributed according to special distributions.

(3) The Algorithm. We present each algorithm in a real programming language. We have selected Algol 68 for this purpose because it allows us to express algorithms in a very compact and precise way. Furthermore, it is relatively simple to translate this language into other languages such as Pascal, Fortran, Cobol, Assembler, or PL/I. Actually, the selection of a language is not an easy decision, and we are not completely happy with any current language. For this reason we have enriched the language we use here with a small number of simple constructs as in Algol 68C. For algorithms which are only of theoretical interest, we do not provide this formal description.

(4) Hints and Tips. Following the algorithm description we give several tips and recommendations on how to use the algorithm. We may point out pitfalls to avoid in coding, suggest when to use the algorithm and when not to, tell when to expect best and worst performances, and provide a variety of other comments.

(5) Tables. Whenever possible, we present tables which show exact values of complexity measures in selected cases. These are intended to give a feeling for how the algorithm behaves. When precise theoretical results are not available we give simulation results, generally in the form $xxx \pm yy$ where the value yy is chosen so that the resulting interval has a confidence level of 95%. In other words, the actual value of the complexity measure falls out of the given interval in at most one out of every twenty simulations.

(6) Differences between internal and external storage. Some algorithms may perform better for internal storage than external, or vice versa. When this is true, we will give recommendations for applications in the two different cases. Since most of our analysis up to this point implicitly assumes the internal case, in this section we will look more closely at the external case (if appropriate). We analyze the algorithm's behaviour when working with external storage, and discuss any significant practical considerations in using the algorithm externally.

3. Search Algorithms

3.1. Sequential Search

3.1.1. Sequential Search

This very basic algorithm is also known as the Linear search or search by Brute Force. It searches for a given element in an array or list by looking through the records sequentially until it finds the element or it reaches the end of the structure. Let A_n be a random variable representing the number of comparisons made between keys during a successful search and let A'_n be a random variable for the number of comparisons in an unsuccessful search. We have

$$Pr\{A_n = i\} = \frac{1}{n},$$

$$E[A_n] = \frac{n+1}{2},$$

$$\sigma^2(A_n) = \frac{n^2-1}{12},$$

$$A'_n = n.$$

Below we give pseudo-code descriptions of the sequential search algorithm in several different situations. The first algorithm searches an array $r[i]$ for the first occurrence of a record with the required key; this is known as primary key search. The second algorithm also searches through an array, but does not stop until it has found every occurrence of the desired key; this is known as secondary key search. The last two algorithms deal with the search for primary and secondary keys in linked lists.

Algorithm for arrays (primary keys)

```
k of r[n+1] := key;
i := 1;
while key≠k of r[i] do i +:= 1 od;
if i⩽n then # found # else # not found # fi;
```

Algorithm for arrays (secondary keys)

```
for i to n do if key = k of r[i] then # found # fi od;
```

Algorithm for lists (primary keys)

```
p := list;
while p≠nil do if key = k of p then # found #; p := nil
        else p := next of p fi od;
```

Algorithm for lists (secondary keys)

```
p := list;
while p≠nil do if key = k of p then # found # fi;
        p := next of p od;
```

The Sequential search is the simplest search algorithm. Although it is not very efficient in terms of the average number of comparisons needed to find a record, we can justify its use in the following cases:

a) when our files only contain a few records (say, $n \leqslant 20$);

b) when the search will be performed only infrequently;

c) when we are looking for secondary keys and a large number of hits ($O(n)$) is expected.

The Sequential search can also look for a given range of keys instead of one unique key, at no significant extra cost. Another advantage of this search algorithm is that it imposes no restrictions on the order in which records are stored in the list or array.

The efficiency of the Sequential search improves somewhat when we use it to examine external storage. Suppose each physical I/O operation retrieves b records; we say that b is the *blocking factor* of the file, and we refer to each block of b records as a *bucket*. Assume that there are a total of n records in the external file we wish to search and let $k = \lfloor n/b \rfloor$. If we use E_n as a random variable representing the number of external accesses needed to find a given record, we have

$$E[E_n] = k+1-\frac{kb(k+1)}{2n} \approx \frac{k+1}{2}.$$

$$\sigma^2(E_n) = \frac{bk(k+1)}{n}\left[\frac{2k+1}{6}-\frac{kb(k+1)}{4n}\right] \approx \frac{k^2}{12}.$$

3.1.2. Self-Organizing Sequential Search: Move to Front Method

This algorithm is basically the Sequential search, enhanced with a simple heuristic method of improving the order of the list or array. Whenever a record is found, that record is moved to the front of the table and the other records are slid back to make room for it (note that we only need to move the elements which were ahead of the given record in the table; those elements further on in the table need not be touched). The rationale behind this procedure is that if some records are accessed more often than others, moving those records to the front of the table will decrease the time for future searches. It is, in fact, very common for records in a table to have unequal probabilities of being accessed; thus, the Move to Front technique may often reduce the average access time needed for a successful search.

We will assume that there exists a probability distribution in which $Pr\{accessing\ key\ K_i\} = p_i$. Further we will assume that the keys appear in the table in order of decreasing access probability, i.e. $p_1 \geqslant p_2 \geqslant \cdots \geqslant p_n > 0$. With this model we have

$$E[A_n] = C_n = \frac{1}{2}+\sum_{i,j}\frac{p_i p_j}{p_i+p_j}.$$

$$\sigma^2(A_n) = (2-C_n)(C_n-1) + 4\sum_{i<j<k} \frac{p_i p_j p_k}{p_i+p_j+p_k}\left(\frac{1}{p_i+p_j}+\frac{1}{p_i+p_k}+\frac{1}{p_j+p_k}\right).$$

$$A_n' = n.$$

$$C_n \leqslant \frac{2n}{n+1} C_n^{Optimal\ arrangement} = \frac{2n}{n+1}\sum ip_i < 2\mu_1'.$$

If we let $T(z) = \sum_{i=1}^{n} z^{p_i}$ then

$$C_n = \int_0^1 z\,[T'(z)]^2\,dz.$$

It is conjectured that

$$C_n \leqslant \frac{\pi}{2}\mu_1' \approx 1.5708\mu_1'.$$

Let $C_n(t)$ be the average number of additional accesses required to find a record, given that t accesses have already been made. Working with a randomly ordered table (i.e. not necessarily in decreasing access probability order as was required above) we have

$$|C_n(t)-C_n| = O(n^2/t).$$

Below we give a pseudo-code description of the Move to Front algorithm as it can be implemented to search linked lists. The algorithm is less well suited to working with arrays -- it is really only efficient when we can guarantee a search will be successful.

Search and Move algorithm

```
p := list;
while p≠nil do if key = k of p then # found #; p := nil
        else p := next of p fi od;
list := p :=:= next of p :=:= list;
```

There are more sophisticated heuristic methods of improving the order of a list than the Move to Front technique; however, this algorithm can be recommended as particularly appropriate when we have reason to suspect that the accessing probabilities for individual records will change with time.

Moreover, the Move to Front approach will quickly improve the organization of a list when the accessing probability distribution is very skewed.

Below we give some efficiency measures for this algorithm when accessing probabilities follow a variety of distributions.

Zipf's law (harmonic): $p_i = (iH_n)^{-1}$

$$C_n = \frac{1}{2}+\frac{(2n+1)H_{2n}-2(n+1)H_n}{H_n} = \frac{2\ln(2)n}{H_n}-\frac{1}{2}+o(1).$$

Lotka's law: $p_i = (i^2 H_n^{(2)})^{-1}$

$$C_n = \frac{3}{\pi^2} \ln n - 0.00206339... + O\left(\frac{\ln n}{n}\right);$$

Exponential distribution: $p_i = (1-a)a^{i-1}$

$$C_n = -\frac{2 \ln 2}{\ln a} - \frac{1}{2} - \frac{\ln a}{24} - \frac{\ln^3 a}{2880} + O(\ln^5 a).$$

Wedge distribution: $p_i = \dfrac{2(n+1-i)}{n(n+1)}$

$$C_n = H_n \left(\frac{4n+2}{3} - \frac{1}{8n(n+1)} \right) - H_{2n} \left(\frac{4n^2+5n-3}{3n} + \frac{3}{4n(n+1)} \right) + \frac{4n+4}{3} - \frac{13}{12(n+1)}$$

$$= \frac{4(1-\ln 2)}{3} n - H_n + \frac{5(1-\ln 2)}{3} + \frac{H_n}{n} + O(n^{-1}).$$

Generalized Zipf's: $p_i \propto i^{-\lambda}$

$$C_n \leqslant \frac{1}{\lambda} \left[\psi\left(\frac{\lambda+1}{2\lambda}\right) - \psi\left(\frac{1}{2\lambda}\right) \right].$$

The table below gives the expected number of accesses required to find an element in lists of various lengths, when the list elements have accessing probabilities which follow several different distributions.

	C_n			
n	Zipf's law	80%-20% rule	Bradford's law ($b=3$)	Lotka's law
5	2.6104	1.6471	2.8264	1.8916
10	4.2949	2.4513	5.1391	2.4124
50	14.9484	8.9700	23.6736	3.7949
100	26.2614	17.1580	46.8458	4.4301
500	101.569	82.7880	232.227	5.9412
1000	184.724	164.865	463.953	6.5991
10000	1415.90			8.7932

3.1.3. Self-Organizing Sequential Search: Transpose Method

This is another algorithm based on the basic Sequential search and enhanced by a simple heuristic method of improving the order of the list or array. In this model, whenever a search succeeds in finding a record, that record is transposed with the record that immediately precedes it in the table (provided of course that the record being sought

was not already in the first position). As with the Move to Front technique, the object of this rearrangement process is to improve the average access time for future searches by moving the most frequently accessed records closer to the beginning of the table. We have

$$E[A_n] = C_n = Prob(I_n)\sum_{\pi}\left\{\left\{\prod_{i=1}^{n} p_i^{\,i-\pi(i)}\right\}\sum_{j=1}^{n} p_j\pi(j)\right\}$$

where π denotes any permutation of the integers 1,2,....,n, $\pi(j)$ is the location of the number j in the permutation π, and $Prob(I_n)$ is given by

$$Prob(I_n) = \left\{\sum_{\pi}\prod_{i=1}^{n} p_i^{\,i-\pi(i)}\right\}^{-1}$$

This expected value of the number of the accesses to find an element can be written in terms of permanents by

$$C_n = \frac{\sum_{k=1}^{n} perm(P_k)}{perm(P)}$$

where P is a matrix with elements $p_{i,j} = p_i^{\,-j}$ and P_k is a matrix which is the derivative of P with respect to p_k^{-1}. We can put a bound on this expected value by

$$C_n \leqslant \frac{2n}{n+1} Opt < 2\mu_1'.$$

In general the Transpose method gives better results than the Move to Front technique. In fact, for all record accessing probability distributions, we have

$$C_n^{Transpose} \leqslant C_n^{MTF}.$$

When we look at the case of the unsuccessful search, however, both methods have the identical result

$$A_n' = n.$$

Below we give a pseudo-code description of the Transpose algorithm as it can be applied to linked lists. The Transpose method can also be implemented efficiently for arrays, using an obvious adaptation of the list algorithm.

Algorithm

```
q := p := list;
while p≠nil do if key = k of p then # found #; break
        else q := p :=:= next of p fi od;
if p≠q then p := next of p :=:= q :=:= p fi;
```

It is possible to develop a better self-organizing scheme by allocating extra storage for counters which record how often individual elements are accessed; however, it is conjectured that the Transpose algorithm is the optimal heuristic organization scheme when allocating such extra storage is undesirable.

It should be noted that the Transpose algorithm may take quite some time to rearrange a randomly ordered table into close to optimal order. In fact, it may take $\Omega(n^2)$ accesses to come within a factor of $1+\epsilon$ of the final steady state.

Because of this slow adaptation ability, the Transpose algorithm is not recommended for applications where accessing probabilities may change with time.

3.1.4. Optimal Sequential Search

When we know the accessing probabilities for a set of records in advance, and we also know that these probabilities will not change with time, we can minimize the average number of accesses in a Sequential search by arranging the records in order of decreasing accessing probability (so that the most often required record is first in the table, and so on). With this preferred ordering of the records, the efficiency measures for the Sequential search are

$$E[A_n] = \mu_1' = \sum_{i=1}^{n} i p_i,$$

$$\sigma^2(A_n) = \sum_{i=1}^{n} i^2 p_i - (\mu_1')^2,$$

$$A_n' = n.$$

Naturally, these improved efficiencies can only be achieved when the accessing probabilities are known in advance and do not change with time. In practice, this is often not the case.

Further, this ordering requires the overhead of sorting all the keys initially according to access probability.

Once the sorting is done, however, the records do not need reorganization during the actual search procedure.

References

[Allen,1978], [Bitner,1979], [Gonnet,1979], [Hendricks,1976], [Knuth,1973], [Knuth,1974], [McCabe,1965], [McKellar,1978], [Rivest,1976], [Shneiderman,1978], [Tanenbaum,1978].

3.2. Sorted Array Search

The following algorithms are designed to search for a record in an array whose keys are arranged in increasing (or decreasing) order.

3.2.1. Binary Search

This algorithm is also known as the Bipartition search, the Bisection search, or the Dichotomic search. It searches a sorted array by the "divide and conquer" technique. At each step of the search, a comparison is made with the middle element of the array. If there is no match, the algorithm decides which half of the array will contain the required key, and discards the other half. The process is repeated recursively, halving the number of records to be searched at each step until the key is found. If the array contains n elements and $k = \lfloor \log_2 n \rfloor$ then we have

$$1 \leqslant A_n \leqslant k+1$$

$$E[A_n] = k+1-\frac{2^{k+1}-k-2}{n} \approx \log_2(n)-1+\frac{k+2}{n},$$

$$\sigma^2(A_n) = \frac{3\times 2^{k+1}-(k+2)^2-2}{n}-\left(\frac{2^{k+1}-k-2}{n}\right)^2 \approx 2.125 \pm .125+o(1).$$

$$1 \leqslant A_n' \leqslant k+1$$

$$E[A_n'] = C_n' = k+2-\frac{2^{k+1}}{n+1} \approx \log_2 n$$

$$\sigma^2(A_n') \leqslant \frac{1}{12}.$$

$$C_n = (1+\frac{1}{n})C'_n-1.$$

(The random variables A_n and A_n' are as defined in section 3.1; C_n and C_n' are the expected values of A_n and A_n' respectively.)

Binary Search Algorithm.

```
low := 0;          high := n+1;
while high-low > 1 do
        j := (high+low) / 2;
        if key > k of r[j] then low := j
        elif key < k of r[j] then high := j
        else high := low fi od;
if key = k of r[j] then # found #
        else # not found # fi;
```

There are more efficient search algorithms than the Binary search but such methods must perform a number of special calculations: for example, the Interpolation search (Section 3.2.2) calculates a special interpolation function while hashing algorithms (Section

3.3) must compute one or more hashing functions. The Binary search is an optimal search algorithm when we restrict our operations to only comparisons between keys.

Binary search is a very stable algorithm: the range of search times stays very close to the average search time, and the variance of the search times is $O(1)$.

Another advantage of the Binary search is that it is well suited to searching for keys in a given range as well as searching for one unique key.

One drawback of the Binary search is that it requires a sorted array. Thus additions, deletions, and modifications to the the records in the table can be expensive, requiring work on the scale of $O(n)$.

Below we give figures showing the performance of the Binary search for various array sizes.

n	C_n	$\sigma^2(A_n)$	C'_n
5	2.2000	0.5600	2.6667
10	2.9000	0.8900	3.5455
50	4.8600	1.5204	5.7451
100	5.8000	1.7400	6.7327
500	7.9960	1.8600	8.9780
1000	8.9870	1.9228	9.9770
5000	11.3644	2.2004	12.3619
10000	12.3631	2.2131	13.3618

3.2.2. Interpolation Search

This is also known as the Estimated Entry search. It is one of the most natural ways to search an ordered table which contains numerical keys. Like the Binary search, it uses the "divide and conquer" approach, but in a more sophisticated way. At each step of the search, the algorithm makes a guess (or interpolation) of where the desired record is apt to be in the array, basing its guess on the value of the key being sought and the values of the first and last keys in the table. As with the Binary search, we compare the desired key with the key in the calculated probe position; if there is no match, we discard the part of the file we know does not contain the desired key and probe the rest of the file using the same procedure recursively.

Let us suppose we have normalized the keys in our table to be real numbers in the closed interval [0,1] and let $\alpha \in [0, 1]$ be the key we are looking for. For any integer $k \leqslant n$, the probability of needing more than k probes to find α is given by

$$Pr\{A_n > k\} \approx \prod_{i=1}^{k} (1 - \frac{1}{2}\xi^{2-i}),$$

where $\xi = \frac{2}{\pi n \alpha(1-\alpha)}$. For this model we have normalized ([0,1]) searched key.

$$E[A_n] = \log_2 \log_2 n + O(1) \approx \log_2 \log_2(n+3),$$

$$\sigma^2(A_n) = O(\log_2 \log_2 n).$$

$$E[A_n'] = \log_2 \log_2 n + O(1) \approx \log_2 \log_2 n + 0.58$$

When implementing the Interpolation search, we must make use of an interpolating formula. This is a function $\phi(\alpha,n)$ which takes as input the desired key $\alpha(\alpha \in [0,1])$ and the array of length n, and which yields an array index between 1 and n, essentially a guess at where the desired array element is. Two of the simplest linear interpolation formulae are $\phi(\alpha,n) = \lceil n\alpha \rceil$ and $\phi(\alpha,n) = \lfloor n\alpha+1 \rfloor$. Below we give a pseudo-code description of the Interpolation search, showing how the function ϕ is used.

Search Algorithm.

```
low := 0;          high := n+1;
while high−low > 1 do
        j := φ( (key − k of r[low]) / (k of r[high] − k of r[low]), high−low−1) + low;
        if key > k of r[j] then low := j
        elif key < k of r[j] then high := j
        else high := low fi od;
if key = k of r[j] then # found #
        else # not found # fi;
```

The Interpolation search is asymptotically optimal among all algorithms which search arrays of numerical keys. However, it is very sensitive to a non-uniform [0,1] distribution of the keys. Simulations show that the Interpolation search can lose its $O(\text{loglog}n)$ behavior under some non-uniform key distributions.

While it is relatively simple in theory to adjust the algorithm to work suitably even when keys are not distributed uniformly [6] difficulties can arise in practice. First of all, it is necessary to know how the keys are distributed and this information may not be available. Furthermore, unless the keys follow a very simple probability distribution, the calculations required to adjust the algorithm for non-uniformities can become quite complex and hence impractical.

The table below gives figures for the efficiency measures of the Interpolation search for various array sizes. The results represented by an interval are simulation results with a 95% confidence interval.

n	$E[A_n]$	$\sigma^2(A_n)$	$E[A_n']$
5	1.5935		1.9611
10	1.8765	0.680	2.3470

50	2.4878	1.056	3.0848
100	2.688±0.021	1.127	3.331±0.051
500	3.161±0.031	1.237	3.800±0.063
1000	3.327±0.030	1.282	4.037±0.058
5000	3.626±0.036	1.295	4.335±0.125
10000	3.769±0.043	1.327	4.489±0.060

3.2.3. Interpolation Sequential Search

This algorithm is a combination of the Interpolation and Sequential search methods. An initial interpolation probe is made into the table, just as in the Interpolation algorithm; if the given element is not found in the probed position, the algorithm them proceeds to search through the table sequentially, forwards or backwards depending on which direction is appropriate. Let A_n and A_n' be random variables representing the number of array accesses for successful and unsuccessful searches respectively. We have

$$\begin{aligned} E[A_n] &= 1+\frac{2}{n}\sum_{k=1}^{n-1}\frac{\Gamma(n)}{\Gamma(k)\Gamma(n-k)}(k/n)^k(1-k/n)^{n-k} \\ &= 1+\left(\frac{n\pi}{32}\right)^{1/2}(1-\frac{7}{12n})+O(n^{-1}) \\ E[A_n'] &= \frac{2}{n+1}\left(1+\sum_{k=1}^{n-1}\frac{(n+1)k}{n}I_{k/n}(k+1,n-k)-(k+1)I_{k/n}(k+2,n-k)\right) \\ &= \left(\frac{n\pi}{32}\right)^{1/2}+O(1) \end{aligned}$$

As with the standard Interpolation search (Section 3.2.2), this method requires an interpolation formula ϕ such as $\phi(\alpha,n) = \lceil n\alpha \rceil$ or $\phi(\alpha,n) = \lfloor n\alpha + 1 \rfloor$. Below we give a pseudo-code description of the algorithm.

Search Algorithm.

```
j := φ((key-lowestkey)/(highestkey-lowestkey)); # initial probe position #
if key < k of r[j] then
                for i from j-1 by -1 to 1 while key ⩽ k of r[i] do j := i od
        elif key > k of r[j] then
                for i from j+1 to n while key ⩾ k of r[i] do j := i od fi;
if key = k of r[j] then # found #
        else # not found # fi;
```

Asymptotically, this algorithm behaves significantly worse than the pure Interpolation search; its performance is generally not acceptable.

When we use this search technique with external storage, we have a significant improvement over the internal case. Suppose we have storage buckets of size b (that is, each physical I/O operation reads in a block of b records); then the number of external accesses the algorithm must make to find a record is given by

$$E[E_n] = 1+\frac{1}{b}\left(\frac{n\pi}{32}\right)^{1/2}$$

In addition to this reduction the accessed buckets are contiguous and hence the seek time may be reduced.

Below we list the expected number of accesses required for both successful and unsuccessful searches for various table sizes.

n	$E[A_n]$	$E[A_n']$
5	1.5939	1.9613
10	1.9207	2.3776
50	3.1873	3.7084
100	4.1138	
500	7.9978	
1000	10.9024	
5000	23.1531	
10000	32.3310	

References

[Gonnet,1977], [Gonnet,1977], [Gonnet,1980], [Knuth,1973], [Kruijer,1974], [Nat,1979], [Perl,1977], [Perl,1978], [Peterson,1957], [Price,1971], [Yao,1976]

3.3. Hashing

Hashing or scatter storage algorithms are distinguished by the use of a *hashing function*. This is a function which takes a key as input and yields an integer in a prescribed range [1,m] as a result. The function is designed so that the integer values it produces are uniformly distributed throughout the range. These integer values are then used as indices for an array of size m called the *hashing table*. Records are both inserted into the table and retrieved from the table by using the hashing function to calculate the required indices from the record keys.

When the hashing function yields the same index value for two different keys, we have a *collision*. A complete hashing algorithm consists of a hashing function and a method for handling the problem of collisions. Such a method is called a *collision resolution scheme*.

There are two distinct classes of collision resolution schemes. The first class is called *open addressing*. Schemes in this class resolve collisions by computing new indices based on the value of the key; in other words, they "rehash" into the table. In the second class of resolution schemes, all elements which "hash" to the same table location are linked together in a chain.

To insert a key using open-addressing we follow a sequence of probes in the table. This sequence of probe positions is called a *path*. In open addressing a key will be inserted in the first empty location of its path. There are at most $m!$ different paths through a hashing table and most open-addressing methods use far less paths than $m!$ Several keys may share a common path or portions of a path. The portion of a path which is fully occupied with keys will be called a *chain*.

The undesirable effect of having chains longer than expected is called *clustering*. More precisely there are two possible definitions for clustering.

a) Let $p = \Theta(m^k)$ be the maximum number of different paths. We say that a collision resolution scheme has $k+1$ clustering if it allows p different circular paths. A *circular path* is the set of all paths that are obtained from circular permutations of a given path. I.e. all the paths in a circular path share the same order of table probing except for their starting position.

b) If the path depends exclusively on the first k initial probes we say that we have k-clustering. It is generally agreed that linear probing suffers from primary clustering; quadratic and double hashing from secondary clustering and uniform and random probing from no clustering.

Assume our hashing table of size m has n records stored in it. The quantity $\alpha = n/m$ is called the *load factor* of the table. We will let A_n be a random variable which represents the number of times a given algorithm must access the hashing table to locate any of the n elements stored there. It is expected that some records will be found on the first try, while for others we may have to rehash several times or follow a chain of other records before we locate the record we want. We will use L_n to denote the length of the longest probe sequence needed to find any of the n records stored in the table. Thus our random variable A_n will have the range

$$1 \leqslant A_n \leqslant L_n ;$$

its actual value will depend on which of the n records we are looking for.

In the same way, we will let A_n' be a random variable which represents the number of accesses required to insert an $n+1^{st}$ element into a table already containing n records. We have

$$1 \leqslant A_n' \leqslant n+1.$$

The search for a record in the hashing table starts at an initial probe location calculated by the hashing function and from there follows some prescribed sequence of accesses determined by the algorithm. If we find an empty location in the table while following this path, we may conclude that the desired record is not in the file. Thus it is important that an open addressing scheme be able to tell the difference between an empty table position (one that has not yet been allocated) and a table position which has had its record deleted. The probe sequence may very well continue past a deleted position, but an empty position marks the end of any search. When we are inserting a record into the hashing table rather than searching for one, we use the first empty location we find, unless we encounter a deletion position earlier in the probe sequence.

Let

$$C_n = E[A_n],$$

and

$$C_n' = E[A_n'].$$

C_n denotes the expected number of accesses needed to locate any individual record in the hashing table while C_n' denotes the expected number of accesses needed to insert a record. Thus

$$C_n = \frac{1}{n}\sum_{i=0}^{n-1} E[A_i'] = \frac{1}{n}\sum_{i=0}^{n-1} C_i'.$$

3.3.1. Uniform Probing Hashing

Uniform Probing Hashing is an open addressing scheme which resolves collisions by probing the table according to a permutation of the integers $[1,m]$. The permutation used depends only on the key of the record in question. Thus for each key, the order in which the table is probed is a random permutation of all table locations. This method will equally likely use any of the $m!$ possible paths.

Uniform Probing is a theoretical hashing model which has the advantage of being relatively simple to analyze. The following list summarizes some of the pertinent facts about this scheme:

$$Pr\{A_n' > k\} = \frac{n^k}{m^k},$$

$$E[A_n] = C_n = \frac{m+1}{n}[H_{m+1} - H_{m-n+1}] \approx -\alpha^{-1}\ln(1-\alpha)$$

$$\sigma^2(A_n) = \frac{2(m+1)}{m-n+2} - C_n(C_n+1) \approx \frac{2}{1-\alpha} + \alpha^{-1}\ln(1-\alpha) - \alpha^{-2}\ln^2(1-\alpha).$$

$$C_m = \frac{m+1}{m}[H_{m+1}-1] = \ln m + \gamma - 1 + o(1).$$

$$C_n^{(worst\ file)} = \frac{n+1}{2}$$

$$E[A_n'] = C_n' = \frac{m+1}{m-n+1} \approx \frac{1}{1-\alpha}$$

$$\sigma^2(A_n') = \frac{(m+1)n(m-n)}{(m-n+1)^2(m-n+2)} \approx \frac{\alpha}{(1-\alpha)^2}.$$

$$C_m' = m.$$

$$C_n'^{(worst\ file)} = C_n'$$

$$L_n = \max_{0 \leqslant i < n} A_i'$$

$$1 \leqslant L_n \leqslant n$$

$$E[L_n] = -\log_\alpha m - \log_\alpha(-\log_\alpha m) + O(1)$$

$$E[L_m] = 0.631587\ldots \times m.$$

The following table gives figures for some of the quantities we have been discussing in the cases $m = 100$ and $m = \infty$.

	$m = 100$			$m = \infty$		
α	C_n	$\sigma^2(A_n)$	C_n'	C_n	$\sigma^2(A_n)$	C_n'
50%	1.3705	0.6358	1.9804	1.3863	0.6919	2.0
80%	1.9593	3.3837	4.8095	2.0118	3.9409	5.0
90%	2.4435	8.4190	9.1818	2.5584	10.8960	10.0
95%	2.9208	17.4053	16.8333	3.1534	26.9027	20.0
99%	3.7720	44.7151	50.0	4.6517	173.7101	100.0

It does not seem practical to implement a clustering-free hashing function.

Double hashing behaves very similarly to Uniform Probing. For all practical purposes they are indistinguishable.

3.3.2. Random Probing Hashing

This is an open addressing hashing scheme in which collisions are resolved by additional probes into the table. The sequence of these probes is considered to be random and dependent only on the value of the key. The difference between this scheme and Uniform Probing is that here some positions may be repeated in the probe sequence, whereas in Uniform Probing no position is examined more than once. Random probing is another theoretical model which is relatively simple to analyze.

The pertinent formulae for this scheme are given by:

$$Pr\{A_n' > k\} = \alpha^k.$$

$$E[A_n] = C_n = \frac{m}{n}[H_m - H_{m-n}] = -\alpha^{-1}\ln(1-\alpha) + O(\frac{1}{m-n})$$

$$\sigma^2(A_n) = \frac{2m^2}{n}[H_m^{(2)} - H_{m-n}^{(2)}] - C_n(C_n+1)$$

$$= \frac{2}{1-\alpha} + \alpha^{-1}\ln(1-\alpha) - \alpha^{-2}\ln^2(1-\alpha) + O(\frac{1}{m-n})$$

$$C_n^{(worst\ file)} = \infty.$$

$$1 \leqslant A_n' \leqslant \infty$$

$$E[A_n'] = C_n' = \frac{1}{1-\alpha}$$

$$\sigma^2(A_n') = \frac{\alpha}{(1-\alpha)^2}$$

$$C_m = H_m = \ln m + \gamma + O(m^{-1}).$$

In some hashing schemes the order in which elements are inserted has no effect on the total number of accesses required to insert a set of keys into the table. In Random Probing, however, order does make a difference. If we have some kind of reorganization scheme which optimizes the order in which keys are inserted, we have the bounds

$$\ln(m) + \gamma + \frac{1}{2} + o(1) \leqslant E[L_m].$$

$$[-\alpha^{-1}\ln(1-\alpha)] \leqslant E[L_n].$$

The following table gives figures for some of the basic complexity measures in the case of $m = 100$ and $m = \infty$.

	$m = 100$			$m = \infty$		
α	C_n	$\sigma^2(A_n)$	C_n'	C_n	$\sigma^2(A_n)$	C_n'
50%	1.3763	0.6698	2.0	1.3863	0.6919	2.0
80%	1.9870	3.7698	5.0	2.0118	3.9409	5.0

90%	2.5093	10.1308	10.0	2.5584	10.8960	10.0
95%	3.0569	23.6770	20.0	3.1534	26.9027	20.0
99%	4.2297	106.1598	100.0	4.6517	173.7101	100.0

Notice that the asymptotic results ($m \to \infty$; α fixed) coincide with Uniform Probing, while for finite values of m, Uniform Probing gives better results.

Random Probing could be implemented using pseudo-random probe locations, however, it does not seem to be a good alternative to the Double Hashing algorithm described in Section 3.3.4.

3.3.3. Linear Probing

Linear Probing is an open addressing hashing algorithm that resolves collisions by probing to the next table location modulo m. In other words, it probes sequentially through the table starting at the initial hash index, possibly running until it reaches the end of the table, rolling to the beginning of the table if necessary, and continuing the probe sequence from there. This method resolves collisions using using only one circular path. For this model:

$$E[A_n] = C_n = \frac{1}{2}\left(1+\sum_{k\geqslant 0}\frac{(n-1)^{\underline{k}}}{m^k}\right) \approx \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$$

$$\sigma^2(A_n) = \frac{1}{6}+\frac{m}{n}\left(\sum_{k\geqslant 0}\frac{k^2+k+3}{6}\frac{n^{\underline{k}}}{m^k}-\frac{1}{2}\right)-(C_n)^2 \approx \frac{(1-\alpha)^{-3}}{3}-\frac{(1-\alpha)^{-2}}{4}-\frac{1}{12}$$

$$C_n^{(worst\ file)} = \frac{n+1}{2}.$$

$$E[A_n'] = C_n' = \frac{1}{2}\left(1+\sum_{k\geqslant 0}\frac{(k+1)n^{\underline{k}}}{m^k}\right) \approx \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$$

$$\sigma^2(A_n') = \frac{1}{6}+\sum_{k\geqslant 0}\frac{(k+1)(2k^2+6k+7)}{12}\frac{n^{\underline{k}}}{m^k}-(C_n')^2 \approx \frac{3(1-\alpha)^{-4}}{4}-\frac{2(1-\alpha)^{-3}}{3}-\frac{1}{12}$$

$$C_n'^{(worst\ file)} = 1+m'.$$

$$C_m = \sqrt{\pi m/8}+\frac{1}{3}+O(m^{-1/2}).$$

Because this algorithm is of practical value, we include a pseudo-code description of the scheme's search and insertion algorithms. We denote the hashing table as an array r, with each element $r[i]$ having a key k.

Search Algorithm.

```
i := hash function(key);
to n while (k of r[i] ≠ key) and not empty(r[i]) do
```

```
        i := (i mod m) + 1 od;
if k of r[i] ≠ key then # not found #
        else # found # fi;
```

Insertion Algorithm

```
i := hash function(key);
to m while not empty(r[i]) and not deleted(r[i]) do
        i := (i mod m) + 1 od;
if empty(r[i]) or deleted(r[i]) then # insert here #; n := n + 1
        else # table full # fi;
```

Linear Probing hashing uses one of the simplest collision resolution techniques available, requiring a single evaluation of the hashing function. It suffers, however, from a piling-up phenomenon called primary clustering. The longer a contiguous sequence of keys grows, the more likely it is that collisions with this sequence will occur when new keys are added to the table. Thus the longer sequences grow faster than the shorter ones. Furthermore, there is a greater probability that longer chains will coalesce with other chains, causing even more clustering. This problem makes the Linear Probing scheme undesirable in certain instances; for example, the scheme is not recommended for probing tables with a high load factor α.

It should be noted that the number of accesses in a successful or unsuccessful search has a very large variance. Thus it is possible that there will be a sizable difference in the number of accesses needed to find different elements.

It should also be noted that given any set of keys, the order in which the keys are inserted has no effect on the total number of accesses needed to install the set. Thus the average number of accesses to insert a single element is not affected by the insertion order either.

An obvious variation on the Linear Probing scheme would be to move backward through the table instead of forward, when resolving collisions. More generally, we could move through a unique permutation of the table entries, which would be the same for every key; only the starting point of the permutation would depend on the key in question. Clearly, both these variations would exhibit exactly the same behaviour as the standard Linear Probing model.

As noted previously, deletions from the table must be marked as such in order for the algorithm to work correctly. The presence of deleted records in the table is called contamination, a condition which clearly interferes with the efficiency of an unsuccessful search. When new keys are inserted after deletions, the successful search is also deteriorated.

Up until now, we have been considering the shortcomings of Linear Probing when it is used to access internal storage. With external storage, the performance of the scheme improves significantly, even for fairly small storage buckets. Let b be the blocking factor, i.e. the number of records per storage bucket. We find that the number of external accesses (E_n) is

$$E_n = 1 + \frac{A_n - 1}{b}.$$

while the number of accesses required to insert an $n+1^{st}$ record is

$$E_n' = 1 + \frac{A_n' - 1}{b}.$$

Furthermore, for external storage, we may change the form of the algorithm so that we scan each bucket completely before examining the next bucket This improves the efficiency somewhat over the simplest form of the Linear Probing algorithm.

The following table gives figures for the efficiency of the Linear Probing scheme with $m = 100$, and $m = \infty$.

	$m = 100$			$m = \infty$		
α	C_n	$\sigma^2(A_n)$	C_n'	C_n	$\sigma^2(A_n)$	C_n'
50%	1.4635	1.2638	2.3952	1.5	1.5833	2.5
80%	2.5984	14.5877	9.1046	3.0	35.3333	13.0
90%	3.7471	45.0215	19.6987	5.5	308.25	50.5
95%	4.8140	87.1993	32.1068	10.5	2566.58	200.5
99%	6.1616	156.583	50.5	50.5	330833.	5000.5

3.3.4. Double Hashing

Double hashing is an open addressing hashing algorithm which resolves collisions by means of a second hashing function. This second function is used to calculate an increment less than m which is added on to the index to make successive probes into the table. Each different increment gives a different path, hence this method uses $m-1$ circular paths. We have

$$E[A_n] = C_n = -\alpha^{-1}\ln(1-\alpha)+o(1) \quad (\alpha<0.319...)$$

$$1 \leqslant A_n' \leqslant n+1$$

$$E[A_n'] = C_n' = (1-\alpha)^{-1}+o(1) \quad (\alpha<0.319...)$$

If $m = 13$ then

$$C_{13}^{Doub.\ Hash.} - C_{13}^{Unif.\ Prob.} = 0.0009763...$$

$$E[L_{13}^{Doub.\ Hash.}] - E[L_{13}^{Unif.\ Prob.}] = 0.001371...$$

Below we give pseudo-code descriptions of search and insertion algorithms which implement the Double Hashing scheme. Both algorithms require the table size m to be a prime number; otherwise there is the possibility that the probe sequence for some keys will not cover the entire table.

Search Algorithm.

```
i := hash function(key);
to n while (k of r[i] ≠ key) and not empty(r[i]) do
        i := (i + hash increment(key) - 1) mod m + 1 od;
if k of r[i] = key then # found #
        else # not found # fi
```

Insertion Algorithm

```
i := hash function(key);
to m while not empty(r[i]) and not deleted(r[i]) do
        i := (i + hash increment(key) - 1) mod m + 1 od;
if empty(r[i]) or deleted(r[i]) then # insert here #; n := n + 1
        else # table full # fi;
```

Double Hashing is a practical and efficient hashing algorithm. Since the increment we use to step through the table depends on the key we are searching for. Double Hashing does not suffer from primary clustering. This also implies that changing the order of insertion of a set of keys may change the average number of accesses required to do the inserting. Thus several reorganization schemes have been developed to reorder insertion of keys in ways which make Double Hashing more efficient.

As with Linear Probing, deletion of records leads to contamination and decreases the efficiency of the unsuccessful search. When new keys are inserted after deletions, the successful search is also deteriorated. The unsuccessful search can be drastically improved by keeping in a counter the length of the longest probe sequence in the file. Thus the search algorithm becomes

to llps **while** (· · ·).

Whenever we insert a new key we may need to update this counter.

Extensive simulations show that it is practically impossible to establish statistically whether Double Hashing behaves differently from Uniform Probing. For example we would need a sample of 3.4×10^7 files of size 13 to statistically show with 95% confidence that double hashing is different from uniform probing. The following tables list some sample results.

$m = 101$

n	C_n	L_n	$\sigma^2(A_n)$	C'_n
51	1.3759±0.0021	4.564±0.027	0.6385±0.0069	2.0020±0.0038
81	1.9686±0.0035	10.996±0.071	3.436±0.033	4.8763±0.0091
91	2.4560±0.0051	18.22±0.13	8.623±0.088	9.306±0.017
96	2.9353±0.0066	27.26±0.20	17.74±0.19	17.004±0.028
100	3.7920±0.0097	48.81±0.34	49.99±0.52	51.0

$m = 4999$

n	C_n	L_n	$\sigma^2(A_n)$	C'_n
2500	1.3872±0.0023	9.42±0.23	0.6956±0.0078	1.9991±0.0040
3999	2.0089±0.0038	25.76±0.59	3.906±0.041	4.9960±0.0100
4499	2.5555±0.0060	48.91±1.43	10.91±0.15	9.989±0.021
4749	3.1544±0.0086	90.3±2.6	27.05±0.42	19.914±0.040
4949	4.629±0.020	314.2±12.4	167.0±4.5	97.99±0.19

3.3.5. Quadratic Hashing

3.3.6. Ordered Hashing

This method is a composition of a hashing step, followed by double hashing collision resolution. The chains for collision resolution are kept ordered. This is equivalent to inserting all keys in increasing order. By doing this there is no gain in the successful search, but we have a significant gain in the unsuccessful search.

For the analysis we will assume that we have no clustering, i.e. similar to uniform probing. Let x be the probability that a randomly selected key in the file is less than the searched key. Then

$$Pr\{A'_n(x) > k\} = \frac{n^{\underline{k}}}{m^{\underline{k}}} x^k$$

$$E[A'_n(x)] = \sum_{k \geq 0} \frac{n^{\underline{k}}}{m^{\underline{k}}} x^k = \frac{m}{m - nx} + O\left(\frac{1}{m-n}\right)$$

$$C'_n = E[A'_n] = \int_0^1 E[A'_n(x)]dx$$

$$= \sum_{k=0}^{n} \frac{(m-k)^{\underline{n-k}}}{m^{\underline{n-k}}(k+1)}$$

$$= \frac{m}{n}\ln\frac{m}{m-n} + O\left(\frac{1}{m-n}\right) \approx -\alpha^{-1}\ln(1-\alpha).$$

$$C'_m = E[A'_m] = H_{m+1}$$

The values for A_n and C_n are the same as those for Double Hashing.

Search Algorithm.

```
i := hash function(key);
to n while (k of r[i] ≠ key) and not empty(r[i]) do
        i := (i + hash increment(key) - 1) mod m + 1 od;
if k of r[i] = key then # found #
        else # not found # fi
```

Insertion Algorithm

```
i := hash function (key);
to m while not empty (k[i]) and not deleted(k[i]) do
        if key = k[i] # error #
        elif key > k[i] then key := k[i] :=:= key fi;
        i := (i + hash increment (key) - 1) mod (m + 1)
        od;
if empty(k[i]) or deleted(k[i]) then # insert here #; n := n + 1
        else # table full # fi;
```

This variation of Double Hashing makes equivalent the unsuccessful search to the successful search at a small cost during insertion.

3.3.10. Direct Chaining Hashing (or Separate Chaining, or Separate Overflow Chaining).

This method is also known as Separate Chaining or Separate Overflow Chaining. It makes use of both hashing functions and sequential lists in the following way. The hashing function first computes an index into the hashing table using the record key. This table location does not hold an actual record, but a pointer to the head of a linked list of all records which hash to that location. This is a composition of hashing with linked lists. Let P_n and P'_n be random variables which represent the number of pointers (chain links) inspected for the successful and unsuccessful searches respectively. Thus

$$P_n = A_n. \qquad P'_n = A'_n + 1.$$

The pertinent facts about this algorithm are listed below:

$$Pr\{list\ with\ length\ i\} = \binom{n}{i}\frac{(m-1)^{n-i}}{m^n}.$$

$$E[A_n] = C_n = 1 + \frac{n-1}{2m} \approx 1 + \frac{\alpha}{2}.$$

$$\sigma^2(A_n) = \frac{(n-1)(n-5)}{12m^2} + \frac{n-1}{2m} \approx \frac{\alpha^2}{12} + \frac{\alpha}{2}$$

$$E[A_n'] = C_n' = \frac{n}{m},$$

$$\sigma^2(A_n') = \frac{n(m-1)}{m^2} \approx \alpha.$$

$$E[L_n] = \Gamma^{-1}(m)\left(1 + \frac{\ln\alpha}{\ln\Gamma^{-1}(m)} + O\left(\frac{1}{\ln^2\Gamma^{-1}(m)}\right)\right),$$

$$E[L_m] = \Gamma^{-1}(m) - \frac{3}{2} + \frac{\gamma-1}{\ln\Gamma^{-1}(m)} + O\left(\frac{1}{\ln m}\right) + P(\ln\ln m).$$

A pseudo-code description of the search algorithm is given below. The insertion algorithm is similar in some respect, but has the added complexity of allocating storage for the new record and either adding the new record to the proper linked list, or else setting a pointer to the record in the table if no other keys have previously hashed to that location.

Search Algorithm.

```
p := hash table[ hash function(key) ];
while p≠nil do if k of p = key then break;
        else p := next of p
        od;
if p=nil then # not found #
        else # found # fi;
```

The Direct Chaining method has several advantages over open addressing schemes. It is very efficient in terms of the average number of accesses for both successful and unsuccessful searches, and in both cases the variance of the number of accesses is small. L_n grows very slowly with respect to n.

Unlike in open addressing schemes, contamination of the table due to deletions does not occur; to delete a record all that is required is an adjustment in the pointers of the linked list involved.

Another important advantage of Direct Chaining is that the load factor α can be greater than 1; that is, we can have $n > m$. This makes the algorithm a good choice for dealing with files which may grow beyond expectations.

There are two slight drawbacks to the Direct Chaining method. The first is that it requires additional storage for the $(m+n)$ pointers used in linking the lists of records. The second is that the method requires some kind of memory management capabilities to handle allocation and deallocation of list elements.

References

[Amble,1974], [Anderson,1979], [Bell,1970], [Bell,1970], [Blake,1977], [Bolour,1979], [Brent,1973], [Cichelli,1979], [Cichelli,1980], [Clapson,1977], [Deutscher,1975], [Gonnet,1979], [Gonnet,1977], [Guibas,1978], [Guibas,1978], [Halatsis,1978], [Knott,1975],

[Knuth,1973], [Konheim,1966], [Lipton,1980], [Lum,1971], [Lum,1973], [Lyon,1978], [Lyon,1978], [Mallach,1977], [Maurer,1975], [Maurer,1968], [Mendelson,1979], [Morris,1968], [Peterson,1957], [Pippenger,1979], [Rivest,1978], [Trabb,1978], [Ullman,1972].

3.4. Recursive Structures Search

3.4.1. Binary Tree Search

The Binary tree search is an algorithm for searching a lexicographically ordered binary tree. Without loss of generality we may assume that the left descendants nodes of any node contain keys whose values are less than or equal to the root, and that the right descendants nodes contain keys whose values are greater than the root.

Let A_n be the number of accesses (or key comparisons) made in the course of a successful search for a given key in a binary tree of size n, and let A_n' be the number of accesses made in the course of an unsuccessful search of size n.

The symbol $h(n)$ denotes the *height* of a tree of size n, i.e. the number of nodes in the longest path through the tree. With this definition, a null tree has height 0, a single node tree has height 1. The *depth* of a node in a tree is that node to the distance from the root; thus the depth of the root is 0.

Several variations on the basic binary tree structure arise with the introduction of semantic rules or constraints such as height balance, weight balance, or heuristic organization schemes. The search algorithm for all binary trees is given below.

Search Algorithm:

```
mode tree = struct ( typekey k, ref tree left, right );

proc search = (ref tree t, typekey key) void:
    if t = nil then #not found#
                        #can insert new element here#
    elif k of t = key then #found#
    elif k of t < key then search(right of t, key)
    else search ( left of t, key) fi;
```

The number of different trees with n nodes is

$$t_n = \frac{1}{n+1}\binom{2n}{n}.$$

and the associated generating function is:

$$T(z) = \sum_{n\geqslant 0} t_n z^n = 1+zT^2(z) = \frac{1-\sqrt{1-4z}}{2z}.$$

The internal path length, I_n, of a tree with n nodes is defined as the sum of the depths of all its nodes. The external path length E_n, of a tree is the sum of the depths of all its leaves. For any binary tree

$$E_n = I_n+2n.$$

Let a_k be the expected number of nodes at depth k and let b_k be the expected number of leaves at depth k. We have the associated generating function

$$A(z) = \sum_k a_k z^k,$$

$$B(z) = \sum_k b_k z^k = (2z-1)A(z)+1.$$

For a successful search we have

$$C_n = E[A_n] = \frac{E[I_n]}{n}+1 = \frac{A(1)+A'(1)}{n} = (1+1/n)C_n'-1,$$

$$\sigma^2(A_n) = 3C_n-2+\frac{A''(1)}{n}-C_n^2,$$

$$1 \leqslant A_n \leqslant h(n),$$

and for an unsuccessful search

$$C_n' = E[A_n'] = \frac{E[E_n]}{n+1} = \frac{B'(1)}{n+1},$$

$$\sigma^2(C_n') = \frac{B''(1)}{n+1}+C_n'(1-C_n'),$$

$$1 \leqslant A_n' \leqslant h(n).$$

The ordered binary tree is a structure which allows us to perform many operations efficiently: inserting in a time of $O(h(n))$; deleting a record also takes $O(h(n))$; finding the maximum or minimum key requires $O(h(n))$ comparisons; and retrieving all the elements in ascending or descending order can be done in a time of $O(n)$. With small changes, it allows to retrieve the k^{th} ordered record in the tree in $O(h(n))$.

References

[Choy,1978], [Driscoll,1978], [Flajolet,1978], [Kemp,1979], [Lee,1980], [Nievergelt,1973], [Nievergelt,1974], [Proskurowski,1980], [Rotem,1978], [Solomon,1980].

3.4.1.1. Randomly Generated Binary Trees

These structures are also known as Random Search trees. Such trees are generated by taking elements in a random order and inserting them into an empty tree using the same algorithm as the one in the previous section. Ordered binary search trees are normally considered to be created in this way. The efficiency measures for searching such trees are

$$C_n = 1 + n^{-1}\sum_{i=0}^{n-1} C_i',$$

$$E[A_n] = C_n = 2(1+1/n)H_n-3 \approx 1.3863 \log_2 n - 1.8456,$$

$$\sigma^2(A_n) = (2+10/n)H_n-4(1+1/n)(H_n^2/n+H_n^{(2)})+4 \approx 1.3863\log_2 n - 1.4253,$$

$$E[A_n'] = C_n' = 2H_{n+1}-2 \approx 1.3863 \log_2 n - 0.8456,$$

$$\sigma^2(A_n') = 2H_{n+1}-4H_{n+1}^{(2)}+2 \approx 1.3863 \log_2 n - 3.4253,$$

$$3.634...\ln(n)+o(\ln n) \leqslant E[h(n)] \leqslant 4.3110...\ln(n)+O(\ln^{1/2} n).$$

$$E[h(n)] \approx 4.011...\ln n - 6.568 \quad (exper.)$$

At the cost of two extra pointers per element, randomly generated binary trees display an excellent behaviour in searches. Unfortunately, the worst case can be generated when the elements are sorted before they are put into the tree. In particular, if any portion of the input records is sorted, it will cause the tree to degenerate badly. Compared to the random binary trees of the next section though, ordered binary trees generated from random input are exceptionally well behaved.

Below we give numerical values for several efficiency measures in trees of various sizes.

n	C_n	$\sigma^2(A_n)$	C_n'	$\sigma^2(A_n')$	$E[h(n)]$
5	2.4800	1.1029	2.900	0.9344	3.8000
10	3.4437	2.1932	4.0398	1.8076	5.6411
50	6.1784	5.6159	7.0376	4.5356	10.8103
100	7.4785	7.2010	8.3946	5.8542	13.2858
500	10.6128	10.7667	11.5896	9.0179	
1000	11.9859	12.2391	12.9729	10.3972	
5000	15.1927	15.5608	16.1894	13.6105	
10000	16.5772	16.9667	17.5754	14.9961	

References

[Knuth,1973], [Knuth,1974], [Robson,1979].

3.4.1.2. Random Binary Trees

When speaking of Random Binary Trees, we consider the situation where all possible trees with the same number of nodes are equally likely to occur. In this case,

$$E[A_n] = \frac{4^n - \frac{3n+1}{n+1}\binom{2n}{n}}{\frac{n}{n+1}\binom{2n}{n}} = \sqrt{\pi n}\,(1+\frac{9}{8n}+\frac{17}{128n^2}+O(n^{-3}))-3-\frac{1}{n},$$

$$E[A_n'] = \frac{4^n - \frac{n-1}{n+1}\binom{2n}{n}}{\binom{2n}{n}} = \sqrt{\pi n}\,(1+\frac{1}{8n}+\frac{1}{128n^2}+O(n^{-3}))-\frac{n-1}{n+1},$$

$$E[h(n)] = 2\sqrt{\pi n}+O(n^{1/4+\delta}) \quad (for\ any\ \delta>0).$$

$$\sigma^2(I_n) = (\frac{10}{3}-\pi)n^3 - \frac{n^2\sqrt{\pi n}}{2}+9(1-\pi/4)n^2-\frac{25n\sqrt{\pi n}}{16}+O(n).$$

If $t_{n,h}$ is the number of trees of height h and size n, then the associated generating function is

$$B_h(z) = \sum_{n=0}^{\infty} t_{n,h} z^n = zB_{h-1}^2(z)+1.$$

When all trees of height h are considered equally likely to occur, then

$$E[nodes] = (0.62896...)2^h-1+O(\delta^{-2^h}) \quad (\delta>1).$$

This situation is primarily a theoretical model. In practice, very few situations give rise to random trees.

References

[Flajolet,Unp].

3.4.1.3. Height Balanced Trees

These are also known as AVL trees. Height balanced trees have the property that any two subtrees at a common node differ in height by at most 1. This balance property can be efficiently maintained by means of a counter in each node, indicating the difference in height between the left and right subtrees. Because of the height balancing

$$\log_2 n+1 \leqslant h(n) \leqslant 1.4404 \log_2(n+2) - 0.3277,$$

$$C_n' \approx \log_2 n + 0.25 \qquad (exper.).$$

Let $t_{n,h}$ be the number of height balanced trees of height h and size n. The associated generating function is

$$T_h(z) = \sum_{n\geqslant 0} t_{n,h} z^n = zT_{h-1}(z)(2T_{h-2}(z)+T_{h-1}(z)).$$

If we assume that all trees of height h are equally likely to occur, the average number of nodes in a balanced tree of height h is

$$E[nodes] = (0.70118...)2^h.$$

References

[Adel'son-Vel'skii,1962], [Foster,1965], [Karlton,1976], [Kosaraju,1978], [Luccio,1978], [Ottmann,1978], [Raiha,1979], [Zaki,1979], [Zweben,1978].

3.4.1.5. Heuristic Organization Schemes on Binary Trees

When the keys in a binary tree have different accessing probabilities, a randomly generated tree may not be fully satisfactory. The following heuristic organization schemes offer ways to create better trees when the accessing probabilities are known.

(a) Insert in decreasing probability order. In this way, the keys most likely to be sought are closer to the root and have shorter search paths. This method requires a reordering of the keys before they are put into the tree.

(b) Median Split. In this scheme we are closer to the root so that the accessing probabilities of both the left and right subtrees are close to ½. This is repeated recursively on both sides of the root; thus if a key is a descendant of any node, it should be equally likely to be in the left subtree as the right. This arrangement is the information theoretic optimum.

(c) It is possible to mix approaches (a) and (b). We allow a tolerance δ, and examine the elements for which the accessing probabilities of the left and right subtrees fall into the range $½\pm\delta$. From these elements, we choose the one with the highest accessing probabilitiy to be the root. This selection procedure is repeated recursively for the nodes of each subtree.

When we do not know the accessing probabilities we may try heuristic organization schemes similar to the Transpose and Move to Front techniques in linear searching.

(d) The Transpose method can be adapted for trees by exchanging a node with its parent each time the node is accessed. If the probability of accessing any key i is $p_i = 1/n$, then this Exchange with Parent technique has the result

$$C_n^{EP} \approx \sqrt{\pi n}.$$

(e) Corresponding to the Move to Front scheme in linear searching, we have the technique of moving an accessed element to the root. With this Move to Root approach we have

$$E[C_n^{MR}] = 1+\sum_{1\leqslant i<j\leqslant n}\frac{p_ip_j}{p_i+\ldots+p_j} \leqslant 2ln(2)H(\vec{p})+1.$$

$$\sigma^2(C_n^{MR}) \leqslant \sqrt{2\ln n}.$$

where $H(\vec{p})=\sum_i -p_i\log_2 p_i$.

References

[Allen,1978], [Horibe,1979], [Sheil,1978].

3.4.2. B-Trees

A B-tree is a balanced multiway tree with the following properties: (a) Every node has at most $2m+1$ sons. (b) Every node except the root and the leaves has at least $m+1$ sons; the root either is a leaf or has at least two sons. (c) The leaves are null nodes which all appear at the same depth. B-trees are used mainly as a primary key access method for large databases where secondary storage is mandatory.

Let A_n and A_n' represent the number of key comparisons in successful and unsuccessful searches respectively. Let E_n and E_n' be the number of nodes of the B-tree accessed in the two cases. Finally, let α be the occupation factor of the tree, i.e. the ratio between the number of keys currently in the tree and the maximum number of keys that a B-tree of the same shape can hold. Then

$$\alpha = \frac{number\ of\ keys}{2m(number\ of\ nodes)}.$$

We also have

$$1 \leqslant E_n \leqslant h(n),$$

$$E_n \approx h - \frac{1}{m} - \frac{m(h+1)-1}{m(2(m+1)^{h-1}-1}.$$

$$\lceil \log_{2m+1}(n+1) \rceil \leqslant h \leqslant 1 + \lfloor \log_{m+1}((n+1)/2) \rfloor.$$

Let t_n be the number of different B-trees with n nodes. We have

$$B(z) = \sum_{n=0}^{\infty} t_n z^n = B\left(\frac{z^{m+1}(z^{m+1}-1)}{z-1}\right)+z,$$

$$t_n \approx \frac{\phi^{-n}}{n} u(\log n),$$

where $0<\phi<1$ and ϕ is a root of $z-1=z^m(z^{m+1}-1)$. In general, for randomly generated B-trees, we have

$$1/2 \leqslant \alpha \leqslant 1,$$

$$\lim_{m\to\infty} \alpha = \ln(2) \approx 69\%.$$

In our following description of the algorithm for searching B-trees, in-search(t,x) is a function that searches for the value x in the node pointed to by t; in-search returns the largest i such that $k[i] \leqslant x$.

Search Algorithm

```
proc search = (btree t; key x, int i) btree :
    if t = nil then nil
        else i := in-search (t,x);
            if x = k[i] then t
                else search (p[i],x,i)
            fi
    fi
```

B-trees are well suited to searches which look for a range of keys rather than one unique key. Furthermore, since the B-tree structure is kept balanced during insertions and deletions, there is no need for periodic reorganizations.

References

[Bayer,1971], [Bayer,1977], [Bayer,1977], [Comer,1979], [Held,1978], [Kwong,1978], [Lomet,7461], [Maly,1978], [Miller,1978], [Odlyzko,1980], [Ottmann,1979], [Rosenberg,1979], [Samadi,1976], [Snyder,1978].

3.4.2.1. 2-3 Trees

2-3 trees are the special case of B-trees when $m=1$. Each node has two or three sons, and all the leaves are at the same depth.

Let t_n be the number of different 2-3 trees with n nodes. Then

$$B(z) = \sum_{n=0}^{\infty} t_n z^n = B(z^2+z^3)+z$$

$$t_n \approx \frac{\phi^n}{n} u(\ln n)$$

where $\phi=(1+\sqrt{5})/2$ is the "golden ratio", and $u(x)$ is a periodic function with period $\ln(4-\phi)$ and mean value $(\phi \ln(4-\phi))^{-1}$. Randomly created 2-3 trees have occupation factors in the range

$$0.70+O(n^{-1}) \leqslant \alpha \leqslant 0.79+O(n^{-1}).$$

If we assume all trees of height h are equally likely, then

$$E[nodes] = (0.48061...)3^h,$$

$$E[keys] = (0.72161...)3^h,$$

$$\alpha = 0.75073...$$

References

[Ellis,1978], [Miller,1977], [Reingold,1979], [Rosenberg,1978], [Vaishnavi,1979], [Yao,1978], [Zaki,1979].

3.4.2.2. Symmetric Binary B-Trees

Symmetric Binary B-Trees (or SBB trees) are an implementation of 2-3 trees. They have been suggested as an alternative for AVL trees. Symmetric Binary B-Trees are binary search trees in which the right and left pointers may be either *vertical* (normal) pointers or *horizontal* pointers. In an SBB tree all paths have the same number of vertical pointers (as in a true B-tree). All nodes except the leaves have two sons and in no path has two consecutive horizontal pointers.

Random retrievals, insertions, and deletions of keys in an SBB tree can be done in a time of $O(\ln n)$. If we let k be the maximum number of keys in any path and $h(n)$ be the height of the SBB tree (counting vertical pointers only), we have

$$h(n) \leqslant k \leqslant 2h(n),$$

$$\log_2(n+1) \leqslant k \leqslant 2\log_2(n+2)-2.$$

While every AVL tree can be transformed into an SBB tree, the converse is not true. Thus the class of AVL trees is a proper subclass of the SBB trees.

SBB trees are well suited for primary memory.

References

[Bayer,1972].

4. Sorting Algorithms.

4.1. Techniques for Sorting Arrays.

4.1.1. Bubble Sort

The Bubble Sort algorithm sorts an array by interchanging adjacent records that are in the wrong order. The algorithm makes repeated passes through the array probing all adjacent pairs until the file is completely in order.

Let C_n be the number of comparisons needed to sort a file of size n using the Bubble Sort, and let I_n be the number of interchanges performed in the process. Then

$$n-1 \leqslant C_n \leqslant \frac{n(n-1)}{2}.$$

$$E[C_n] = \frac{n^2-n\ln n-(\gamma+\ln 2-1)n}{2}+O(n^{1/2}).$$

$$0 \leqslant I_n \leqslant \frac{n(n-1)}{2}.$$

$$E[I_n] = \frac{n(n-1)}{4}.$$

$$E[passes] = n-\sqrt{\pi n/2}+1/3+o(1).$$

The simplest form of the Bubble Sort always makes its passes from the top of the array to the bottom. A slightly more complicated form passes from the top to the bottom, then makes a return pass from bottom to top. Descriptions of both these algorithms are given below.

Bubble Sort Algorithm

```
up := n;
while 1<up do
        1>up
        for i to up-1 do
                j := 1;
                if k[i]>k[i+1] then
                        k[i] := k[i+1] :=:= k[i];
                        j := i
                        fi
                od
        up := j
        od;
```

Bubble Sort Algorithm (Double direction)

```
a := 1;  b := n-1;incr := 1;
while (b-a)*incr > 0 do
        j := a;
        for i from a by incr to b do
                if k[i]>k[i+1] then
                        k[i] := k[i+1] :=:= k[i];
                        j := i
                        fi
                od
        a := j-incr;        b := a;  incr := -incr;
        od;
```

The Bubble Sort is a simple sorting algorithm, but inefficient. Its running time is $O(n^2)$, unacceptable even for medium-sized files. Perhaps for very small files its simplicity may justify its use, but the Linear Insertion Sort is just as simple to code and more efficient to run.

For files with very few elements out of place, the double direction Bubble Sort (or cocktail shaker sort) can be very efficient. If k of the n elements are out of order, the running time of the double direction sort is $O(kn)$. One advantage of the Bubble Sort is that it is stable: records with equal keys remain in the same order after the sort as before.

4.1.2. Linear Insertion Sort

The Linear Insertion Sort is one of the simplest sorting algorithms. With a portion of the array already sorted, the remaining records are moved into their proper places one by one. Let C_n be the number of comparisons needed to sort an array of size n using the Insertion Sort. Then

$$E[C_n] = \frac{(n+4)(n-1)}{4}$$

$$\sigma^2(C_n) = \frac{(2n+5)n(n-1)}{72}$$

Algorithm:

```
for i from n-1 by -1 to 1 do
        for j from i to n-1 while k[j]>k[j+1] do
                k[j] := k[j+1] :=:= k[j] od
        od;
```

Algorithm (more efficient)

```
k[n+1] := +∞;
for i from n-1 by -1 to 1 do
        temp := k[i];
        j := i+1;
        while temp>k[j] do k[j-1] := k[j];
                j +:= 1 od;
        k[j-1] := temp
        od;
```

The running time for sorting a file of size n with the Linear Insertion Sort is $O(n^2)$. For this reason, the use of the algorithm is only justifiable for sorting very small files. For files of this size however (say $n<10$), the Linear Insertion Sort may be more efficient than algorithms which perform better asymptotically. The main advantage of the algorithm though is the simplicity of its code.

Like the Bubble Sort, the Linear Insertion Sort is stable: records with equal keys remain in the same order after the sort as before.

4.1.3. Quicksort

Quicksort is a sorting algorithm which uses the divide and conquer technique. To begin each iteration an element is selected from the file. The file is then split into two subfiles, those elements smaller than the selected one and those elements whose keys are larger. In this way, the selected element is placed in its proper final location between the two subfiles. This procedure is repeated recursively on the two subfiles and so on.

Let C_n be the number of comparisons needed to sort an array of size n, let I_n be the number of interchanges performed in the process, and let $k = \lfloor \log_2 n \rfloor$. Then

$$(n+1)k - 2^{k+1} + 2 \leqslant C_n \leqslant \frac{n(n-1)}{2}$$

$$E[C_n] = (n+1)(2H_{n+1} - 2) = 2n(\ln n + \gamma - 1) + 2\ln n + O(1).$$

$$\sigma^2(C_n) = n\sqrt{7 - 2\pi^2/3} + o(n)$$

$$E[I_n] = (n+1)\left(\frac{H_{n+1}}{3} - \frac{5}{6}\right) + \frac{1}{2} = n\left(\frac{\ln n + \gamma}{3} - \frac{5}{6}\right) + O(\ln n)$$

In the description of Quicksort below, note that files and subfiles under a certain size m are sorted by a routine called simplesort, not by Quicksort itself. The choice of m and the algorithm for simplesort are up to the programmer. For example, if $m = 0$ simplesort would be null, while if $m = 1$ simplesort is just a conditional interchange. One could also choose $m = 9$ and use the Linear Insertion Sort. In any case, Quicksort usually behaves more efficiently when $m \neq 0$ and very small files are sorted by a separate algorithm.

Quicksort Algorithm

```
proc quicksort = ( [] typekey k; int lo,up ) void :
    if up-lo≤m then simplesort
        else
        j := up;
        temp := k[lo];
        for i from lo+1 while i≤j do
            if k[i]>temp then
                while k[j]≥temp do j := j-1 od
                if j>i then k[i] := k[j] :=:= k[i] fi
                fi
            od;
        k[lo] := k[j] :=:= k[lo];
        if j-lo<up-j then quicksort(k,lo,j-1);
                quicksort(k,j+1,up)
            else quicksort(k,j+1,up);
                quicksort(k,lo,j-1)
            fi
        fi
```

Quicksort is a very popular sorting algorithm, though its worst case is $O(n^2)$, its average behaviour is excellent.

Unfortunately, this worst case occurs when the given file is nearly in order already, a situation which may well happen in practice. To compensate for this, small tricks in the code of the algorithm can be used to ensure that these worst cases only occur with exponentially small probability.

4.1.4. Shell Sort

4.1.5. Heapsort

4.1.6. Interpolation Sort.

This sorting algorithm is similar in concept to the Bucket Sort. An interpolation function is used to estimate where records should appear in the file. Records with the same interpolation address are grouped together and later bubble-sorted. The main difference between this algorithm and the Bucket Sort is that the Interpolation Sort is implemented in an array, using only one auxiliary index array and with no pointers.

Let C_n be the number of comparisons needed to sort an array of size n using the Interpolation Sort, and let F_n be the total number of interpolation function evaluations made in the process. Then

$$F = 2n,$$

$$n-1 \leqslant C_n \leqslant \frac{n(n-1)}{2}$$

$$E[C_n] = \frac{5(n-1)}{4}$$

$$\sigma^2(C_n) = \frac{(20n-13)(n-1)}{72n}$$

The algorithm below uses the interpolation function ϕ to sort the records $[1{:}n]$ of the array "in" into ascending order in the array "out". The array "iwk" is an auxiliary array of length n.

Algorithm

```
for i to n do iwk[i] := 0 od;
for i to n do iwk[ φ(in[i],n) ] +:= 1 od;
for i from 2 to n do iwk[i] +:= iwk[i-1] od;
for i to n do
        j := φ(in[i],n);
        out[iwk[j]] := in[i];
        iwk[j] -:= 1 od;
for i from 2 to n do
        if out[i-1] > out[i] then
                for k from i by -1 to 2 while out[k-1] > out[k] do
                        out[k-1] := out[k] :=:= out[k-1] od
                fi
        od;
```

Because the standard deviation of C_n is $\approx 0.53n^{1/2}$, the total number of comparisons used by the Interpolation Sort is very stable around its average.

One of the restrictions of the Interpolation Sort is that it can only be used where records have numerical keys which can be handled by the interpolation function. Even in this case, if the distribution of the record key values departs significantly from the uniform distribution, it may mean a dramatic difference in running time. If, however, the key distribution is suitable and we can afford the extra storage required, the Interpolation Sort is remarkably fast, with a running time of $O(n)$.

4.1.7. Linear Probing Sort.

This is an interpolation sort based on a collision resolution technique similar to that of Linear Probing Hashing. It produces ordered chains using an algorithm which is the composition of interpolation and linear collision resolution. The algorithm can be implemented for full tables, but its performance improves dramatically in tables with load factors of less than 100%.

Let the size of our table be $m+w$; we will use the first m locations to interpolate the keys and the last w locations as an overflow area. We will let n be the total number of keys to be sorted and $\alpha = n/m$ be the load factor. Let C_n be the number of comparisons needed to sort the n keys using the Linear Probing Sort, and let F_n be the total number of interpolation function evaluations performed in the process. Then

$$F_n = n.$$

$$E[C_n] \leqslant \frac{n(2m-n)}{2(m-n)}.$$

Let W_n be the number of keys in the overflow section beyond the location m in the table. We have

$$E[W_n] \leqslant \frac{\alpha^2}{2(1-\alpha)}$$

$$\sigma^2(W_n) = \frac{6\alpha^2-2\alpha^3-\alpha^4}{12(1-\alpha)^2}$$

Below we describe the Linear Probing Sort using the interpolation function ϕ.

Algorithm

```
for j to upb k do k[j] := 0 od;
to n do key := get a key;
        i := φ(key,m);
        to n while k[i]≠0 do
                if key<k[i] then key := k[i] :=:= key fi
                i +:= 1;
                if i>upb k then # overflow: k not large enough #
                od;
        k[i] := key
        od;
i := 0;
for j to upb k do if k[j]≠0 then
        k[ (i+:=1) ] := k[j] fi od
while i<upb k do k[ (i+:=1) ] := 0 od;
```

With a good interpolation formula, this algorithm can rank among the most efficient interpolation sort algorithms.

The application of this algorithm to external storage appears to be promising; however, its performance cannot be improved by using larger buckets. Letting E_n be the number of external accesses required to sort n records, we have

$$E[E_n] = n\left(1+\frac{\alpha}{2b(1-\alpha)}\right)+\frac{m}{b}.$$

Below we give efficiency measures for the two table sizes with various load factors.

	$m = 100$			$m = 5000$		
α	C	W	I	C	W	I
50%	72.94±0.18	0.233±0.011	13.83±0.13	3749.7±8.0	0.305±0.073	766.4±5.9
80%	201.40±0.91	1.269±0.033	85.57±0.74	11960.±76.	1.52±0.22	5941.±67.
90%	309.3±1.6	2.476±0.048	164.1±1.4	24095.±293.	4.14±0.48	16130.±273.
95%	398.2±2.2	3.610±0.058	233.8±1.9	45890.±884.	8.73±0.85	36121.±846.
99%	499.2±2.7	5.069±0.063	315.8±2.4	119703.±3564.	25.7±1.9	106957.±3475.

References

[Dobosiewicz,1978], [Floyd,1964], [Hirschberg,1978], [Hoare,1961], [Hoare,1962], [Johnson,1978], [Knuth,1974], [Knuth,1973], [MacLaren,1966], [Manacher,1979], [Manacher,1979], [Munro,1976], [Peltola,1978], [Power,1980], [Sedgewick,1978], [Sedgewick,1977], [Todd,1978], [Williams,1964], [Woodall,1971]

8. Distributions Derived from Empirical Observation

In this chapter we will describe some probability distributions arising from empirical situations. The distributions described here will later be used with other well known distributions to test algorithms under various conditions. Some of these distributions are related directly or indirectly to data processing.

8.1. Zipf's Law

Zipf [1] observed that the frequency of word usage (in written English) follows a simple pattern. When word frequencies are listed in decreasing order, we have the relation

$$f_1 \approx i f_i$$

where f_i denotes the frequency of the i^{th} most frequent word. Zipf observed that the population of cities in the U.S. also follows this relation closely. From this observation we can easily define a Zipfian probability distribution as

$$p_i = \frac{1}{iH_n} \qquad 1 \leqslant i \leqslant n.$$

The first moments and variance of this distribution are

$$\mu_1' = \frac{n}{H_n}$$

$$\mu_2' = \frac{n(n+1)}{2H_n}$$

$$\sigma^2 = \frac{n}{H_n}\left(\frac{n+1}{2} - \frac{n}{H_n}\right)$$

This distribution can be generalized in the two following ways.

(a) First generalization of a Zipfian distribution

In this case the probabilities are defined by

$$p_i = \frac{1}{a(i+b)} \qquad (1 \leqslant i \leqslant n,\ b > -1)$$

where $a = \psi(n+b+1) - \psi(b+1)$. The first moments and variance are

$$\mu_1' = \frac{n}{a} - b$$

$$\mu_2' = \frac{n(n+1) - 2nb + 2ab^2}{2a}$$

$$\sigma^2 = \frac{n}{2a}(n+1+2b-2n/a)$$

Choosing b to be an integer allows us to represent truncated Zipfian distributions. Giving b a small non-integer value may provide a better fit for the first few frequencies.

(b) Second generalization of a Zipfian distribution

This generalization introduces a parameter θ so that we may define

$$p_i = \frac{1}{i^\theta H_n^{(\theta)}} \qquad 0 \leqslant \theta \leqslant 1.$$

Zipf found that some word frequencies matched this distribution closely for values of θ other than 1. In this case the first moments and variance are

$$\mu_1' = \frac{H_n^{(\theta-1)}}{H_n^{(\theta)}} \approx \frac{n(1-\theta)}{2-\theta}$$

$$\mu_2' = \frac{H_n^{(\theta-2)}}{H_n^{(\theta)}} \approx \frac{n^2(1-\theta)}{3-\theta}$$

$$\sigma^2 = \frac{H_n^{(\theta-2)}H_n^{(\theta)} - (H_n^{(\theta-1)})^2}{(H_n^{(\theta)})^2} \approx \frac{n^2(1-\theta)}{(3-\theta)(2-\theta)^2}$$

8.2. Bradford's Law

Bradford's law was first observed in experiments dealing with the number of references made to a selection of books in search of information. This principle can be described in the following way. Assume that we have a collection of n books which treat a given topic, and that these books are placed on a shelf in decreasing order according to the number of times each book is referenced. Thus the most referenced book is first and so on. Lastly we divide these books into k contiguous groups such that each group receives the same number of references. Bradfords Law now states that the number of books in each successive division follows the ratio $1:m:m^2:\cdots:m^{k-1}$.

To translate this description into mathematical terms, we let r_i be the expected value of the number of references to the i^{th} most referenced book on our shelf. Thus we have $r_1 \geqslant r_2 \geqslant \cdots \geqslant r_n$. Let the partial sum of the expected values of these references be

$$\sum_{i=1}^{j} r_i = R(j)$$

and so

$$R(n) = T$$

where T is the total expected number of references. To divide the n books into k divisions satisfying the given ratio, the number of books in each division must be $\frac{n(m-1)}{m^k-1}$, $\frac{nm(m-1)}{m^k-1}$, $\cdots$, $\frac{nm^{k-1}(m-1)}{m^k-1}$. Since each division receives the same number of references, this number must be T/k. Consequently the total expected number of references to the first division will be

$$\sum_{i=1}^{\frac{n(m-1)}{m^k-1}} r_i = R\left(\frac{n(m-1)}{m^k-1}\right) = \frac{T}{k}.$$

For the first and second divisions together we have

$$R\left(\frac{(1+m)n(m-1)}{m^k-1}\right) = \frac{2T}{k}.$$

In general, for the first j divisions we have the equation

$$R\left(\frac{(m^j-1)n}{m^k-1}\right) = \frac{jT}{k}. \tag{1}$$

Now the quantities k and m are related to one another, since for any valid k, Bradford's Law predicts the existence of a unique m. Examination of $R(x)$ for different values of k and m shows that in order for the law to be consistent, the quantity $m^k-1 = b$ must be constant. This constant b defines the shape of the distribution. From equation (1) we can solve for $R(x)$ and obtain

$$R(x) = \frac{T}{k}\log_m\left(\frac{bx}{n}+1\right).$$

Let p_i be the probability that a random reference refers to the i^{th} book. From the above discussion we have

$$p_i = \frac{R(i)-R(i-1)}{T} = \frac{1}{k}\log_m\left(\frac{bi+n}{b(i-1)+n}\right)$$

Since $m^k-1 = b$, we have $k \ln m = \ln(b+1)$; this allows us to simplify the given probability to

$$p_i = \log_{b+1}\left(\frac{bi+n}{b(i-1)+n}\right)$$

The first moment of the above distribution is

$$\mu_1' = \sum_{i=1}^{n} ip_i = n\log_{b+1}\left(\frac{n(b+1)}{b}\right) - \log_{b+1}\left(\frac{\Gamma(n(b+1)/b)}{\Gamma(n/b)}\right), \tag{2}$$

$$= n\left(\frac{1}{\ln(b+1)}-\frac{1}{b}\right)+\frac{1}{2}+\frac{b^2}{12n(b+1)\ln(b+1)}+O(n^{-3}).$$

The second moment is given by

$$\mu_2' = \frac{n^2}{b^2}-\frac{n}{b}+\frac{1}{3}+\frac{1}{\ln(b+1)}\left(\frac{n^2(b-2)}{2b}+n-\frac{b}{6(b+1)}+\frac{b^2}{12(b+1)n}\right)+O(n^{-2}). \tag{3}$$

The variance is

$$\sigma^2 = \frac{n^2}{\ln(b+1)}\left(\frac{b+2}{2b}-\frac{1}{\ln(b+1)}\right)+O(1). \tag{4}$$

This distribution behaves very much like the generalized harmonic (or the first generalization of the Zipf). When the parameter $b \to 0$ Bradford's distribution coincides

with the discrete rectangular distribution.

Although the process of accessing information from books is rarely automated, there is a significant number of automatic processes in which the accessing of information is similar to the situation of referencing books. In these cases Bradford's Law may provide a good model of the access probabilities.

8.3. Lotka's Law

Lotka [3] observed that the number of papers in a given journal written by the same author closely followed an inverse square distribution. In other words, if we were to choose an author at random from the list of contributors to the journal, the probability that he or she had contributed exactly i papers would be proportional to i^{-2}. Later it was observed that for some journals an inverse cube law fit the data more precisely. We will generalize these two laws in the following way. Let n be the total number of authors who published at least one paper in a given journal. The probability that a randomly chosen author contributed exactly i papers will be given by

$$p_i = \frac{1}{\zeta(\theta) i^{\theta}}.$$

The first moment of this distribution corresponds to the average number of papers published by each author; it is given by

$$\mu'_1 = \sum_{i=1}^{\infty} i p_i = \frac{\zeta(\theta-1)}{\zeta(\theta)}.$$

We immediately conclude that this law will only be consistent for $\theta>2$, as has been noted by several other authors; otherwise this first moment will be unbounded, a situation which does not correspond with reality. Note that $n\mu'_1$ denotes the expected number of papers published in a journal which has n contributors.

For $\theta \leqslant 3$, the variance of the distribution under discussion diverges. For $\theta > 3$, the variance is given by

$$\sigma^2 = \frac{\zeta(\theta-2)}{\zeta(\theta)} - \left(\frac{\zeta(\theta-1)}{\zeta(\theta)} \right)^2.$$

The median number of papers by the most prolific author can be approximated by

$$median \approx \left(\frac{n}{\ln(2)\zeta(\theta)(\theta-1)} \right)^{\frac{1}{\theta-1}}.$$

8.4. 80%-20% Rule

The 80%-20% rule was proposed as a probabilistic model to explain certain data processing phenomena. In computing folklore it is usually given as: "80% of the transactions are on the most active 20% of the records, and so on recursively". Mathematically, let $p_1 \geqslant p_2 \geqslant p_3 \geqslant \cdots \geqslant p_n$ be the independent probabilities of performing a transaction on each of the n records. Let $R(j)$ be the cumulative distribution of the p_i's.

i.e.

$$\sum_{i=1}^{j} p_i = R(j); \qquad R(n) = 1.$$

The 80%-20% rule is expressed in terms of the function $R(j)$ by

$$R(n \times 20\%) = 80\%.$$

This rule may be "applied recursively" by requiring that the relation hold for any contiguous subset of p_i's that includes p_1. This requirement yields the necessary condition:

$$R(0.2j) = 0.8R(j).$$

More generally we may consider an $\alpha\%-(1-\alpha)\%$ rule given by

$$R((1-\alpha)j) = \alpha R(j), \qquad \tfrac{1}{2} \leqslant \alpha \leqslant 1. \tag{1}$$

The above functional equation defines infinitely many probability distributions for each choice of α. One simple solution that is valid for all real j is

$$R(i) = \frac{i^{\theta}}{n^{\theta}}$$

where $\theta = \dfrac{\ln(\alpha)}{\ln(1-\alpha)}$. Thus $0 \leqslant \theta \leqslant 1$. This formula for $R(i)$ implies

$$p_i = \frac{i^{\theta}-(i-1)^{\theta}}{n^{\theta}} \tag{2}$$

Note that this probability distribution also possesses the required monotone behavior, i.e. $p_i \geqslant p_{i+1}$.

The parameter θ gives shape to the distribution. When $\theta = 1$ ($\alpha = \frac{1}{2}$) the distribution coincides with the discrete rectangular distribution. The moments and variance of the distribution described by equation (2) are

$$\mu_1' = \sum_{i=1}^{n} i p_i = \frac{\theta n}{\theta+1} + \frac{1}{2} - \frac{\zeta(-\theta)}{n^{\theta}} - \frac{\theta}{12n} + O(n^{-3}).$$

$$\mu_2' = \sum_{i=1}^{n} i^2 p_i = \frac{\theta n^2}{\theta+2} + \frac{\theta n}{\theta+1} + \frac{2-\theta}{6} - \frac{2\zeta(-\theta-1)+\zeta(-\theta)}{n^{\theta}} + O(n^{-1}).$$

$$\mu_3' = \sum_{i=1}^{n} i^3 p_i = \frac{\theta n^3}{\theta+3} + \frac{3\theta n^2}{2(\theta+2)} + \frac{\theta(3-\theta)n}{4(\theta+1)} + O(1).$$

$$\mu_k' = \sum_{i=1}^{n} i^k p_i = \frac{\theta n^k}{\theta+k} + \frac{\theta k n^{k-1}}{2(\theta+k-1)} + \frac{\theta(k-\theta)k n^{k-2}}{12(\theta+k-2)} + O(n^{k-3}) + O(n^{-\theta}).$$

$$\sigma^2 = \frac{\theta n^2}{(\theta+1)^2(\theta+2)} + O(n^{1-\theta}).$$

For large n, the tail of the distribution coincides asymptotically with $p_i \approx i^{\theta-1}$. For the 80%-20% rule, $\theta = 0.138646...$; consequently the distribution which arises from this rule behaves very similarly to the second generalization of Zipf's distribution.

References

[Johnson,1969], [Knuth,1973], [Lotka,1926)], [McWrath,1978)], [Murphy,1973)], [Pope,1975)], [Zipf,1949].

10. References

[1] Adel'son-Vel'skii, G.M. and Landis, E.M.: An Algorithm for the organization of information. Doklady Akademia Nauk USSR, 146, No. 2 (1962), pp. 263-266. (3.4.1.3)

[2] Adleman, L., Booth, K.S., Preparata, F. and Ruzzo, W.L.: Improved Time and Space Bounds for Boolean Matrix Multiplication. Acta Inf. Vol 11-1, (1978) pp. 61-70. (7.2)

[3] Aho, A.V., Hopcroft, J.E. and Ullman, J.D.: *The Design and Analysis of Computer Algorithms*. Addison Wesley (1974). (3.4.5.1)

[4] Allen, B. and Munro, J.I.: Self-Organizing Search Trees, J. ACM, Vol. 25-4, (Oct 1978), pp. 526-535. (3.4.1.5,3.1)

[5] Amble, O. and Knuth, D.E.: Ordered Hash Tables. The Computer J. Vol 17-2 (May 1974) pp. 135-142. (3.3.6)

[6] Anderson, M.R. and Anderson, M.G.: Comments on Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets. C.ACM, Vol. 22-2, (Feb 1979) pp. 104-105. (3.3)

[7] Baer, J.L. and Schwab, B.: A Comparison of Tree-Balancing Algorithms. C.ACM, Vol. 20-5, (May 1977) pp. 322-330. (3.4.1.3,3.4.1.4,3.4.1.5)

[8] Bandyopadhyay, S.K.: Comment on Weighted Increment Linear Search for Scatter Tables. C.ACM, Vol. 20-4, (Apr 1977) pp. 262-263. (3.3.3)

[9] Bayer, R. Symmetric Binary B-trees: Data Structure and Maintenance Algorithms. Acta Informatica, Vol 1-4 (1972), pp. 290-306. (3.4.2.2)

[10] Bayer, R. and McCreight, E.: Organization and Maintenance of Large Ordered Indexes. Acta Informatica, Vol 1-3 (1972), pp. 173-189. (3.4)

[11] Bayer, R. and Schkolnick, M.: Concurrency of operations on B-trees. Acta Inf. 9,1 (1977), 1-21. (3.4.2)

[12] Bayer, R. and Unterauer, K.: Prefix Btrees. ACM Trans. Database Syst. 2,1 (Mar 1977), 11-26. (3.4.2)

[13] Bayer, R.: Binary B-trees for virtual memory, Proc. 1971 ACM SIGFIDET Workshop, San Diego, (1971), pp. 219-235. (3.4.2)

[14] Bell, J.R. and Kaman, C.H.: The Linear Quotient Hash Code. C. ACM Vol 13-11 (Nov 1970) pp. 675-677. (3.3.4)

[15] Bell, J.R.: The Quadratic Quotient Method. C. ACM Vol 13-1 (Jan 1970) pp. 107-109. (3.3.5)

[16] Bentley, J.L. and Friedman, J.H.: Data Structures for Range Searching. ACM C. Surveys, Vol. 11-4, (Dec 1979) pp. 397-409. (3.4,3.5)

[17] Bentley, J.L. and Maurer, H.A.: A Note on Euclidean Near Neighbor Searching in the Plane. Inf. Proc. Letters, Vol. 8-3, (Mar 1979) pp. 133-136. (3.5)

[18] Bentley, J.L. and Maurer, H.A.: Efficient Worst-Case Data Structures for Range Searching, Acta Inf. Vol. 13-2, (1980) pp. 155-168. (3.5)

[19] Bentley, J.L.: Decomposable Searching Problems, Inf. Proc. Letters, Vol. 8-5, (June 1979) pp. 244-251. (2)

[20] Bentley, J.L.: Multidimensional Divide-and-Conquer, C.ACM, Vol. 23-4, (Apr 1980) pp. 214-229. (2,3.5)

[21] Bitner, J.R.: *Heuristics that dynamically organize data structures*, SIAM J on Computing, 8, (1979), pp. 82-110. (3.1)

[22] Blake, I.F. and Konheim, A.G.: Big Buckets Are (Are Not) Better! J. ACM, Vol 24-4, (Oct 1977) pp. 591-606. (3.3.3)

[23] Bolour, A.: Optimality Properties of Multiple-Key Hashing Functions, J.ACM, Vol. 26-2, (Apr 1979), pp. 196-210. (3.3,DB)

[24] Brent, R.P.: Reducing the Retrieval Time of Scatter Storage Techniques. C. ACM Vol 16-2 (Feb 1973) pp. 105-109. (3.3.7)

[25] Brown, M.R. and Tarjan, R.E.: A Fast Merging Algorithm, J. ACM, Vol. 26-2, (Apr 1979), pp. 211-226. (3.4.5.1)

[26] Brown, M.R. and Tarjan, R.E.: A Representation for Linear Lists with Movable Fingers, Proceedings of 10th SIGACT S. (May 1978) pp. 19-29. (DR,3.4)

[27] Brown, M.R.: Implementation and analysis of binomial queue algorithms, SIAM J. Computing, Vol. 7, No. 3, August 1978. (5.1)

[28] Burge, W.H.: An Analysis of Binary Search Trees Formed from Sequences of Nondistinct Keys, J.ACM, Vol. 23-3, (July 1976) pp. 451-454. (3.4.1)

[29] Burton, W.: Generalized Recursive Data Structures, Acta Inf. Vol. 12-2, (1979) pp. 95-108. (2)

[30] Chou, D.M. and Wong, C.K.: Optimal $\alpha-\beta$ trees with Capacity Constraint, Acta Inf. Vol. 10-3, (1978) pp. 273-296. (3.4.1)

[31] Cichelli, R.J.: A Perfect Hashing Function, Pascal News, 15 (Sep 1979) pp. 56-59. (3.3)

[32] Cichelli, R.J.: Minimal Perfect Hash Functions Made Simple, C. ACM, Vol. 23-1, (Jan 1980), pp. 17-19. (3.3)

[33] Clapson, P.: Improving the Access Time for Random Access Files, C. ACM, Vol 20-3 (Mar 1977) pp. 127-135. (3.3)

[34] Cohen, J. and Roth, M.: On the Implementation of Strassen's Fast Multiplication Algorithm, Acta Inf., Vol. 6 (1976) pp. 341-355. (6.3)

[35] Comer, D. and Sethi, R.: The Complexity of Trie Index Construction, J.ACM, Vol. 24-3, (July 1977) pp. 428-440. (3.4.4)

[36] Comer, D.: The ubiquitous Btree, ACM C. Surveys Vol. 11-2 (June 1979) 121-137. (3.4.2)

[37] Crane, C.A.: Linear lists and Priority Queues as Balanced Binary Trees, STAN-CS-72-259, February 1972. (5.1)

[38] Cremers, A.B. and Hibbard, T.N.: Orthogonality of Information Structures, Acta Inf. Vol. 9-3, (1978) pp. 243-261. (2)

[39] Darlington, J.: A Synthesis of Several Sorting Algorithms, Acta Inf. Vol. 11-1, (1978) pp. 1-30. (2)

[40] Deutscher, R.F., Sorenson, P.G. and Tremblay, J.P.: Distribution dependent hashing functions and their characteristics, Proc. of the Int. Conf. of Management of Data, ACM/SIGMOD, 1975, 224-236. (3.3)

[41] Dobosiewicz, D.: Sorting by Distributive Partitioning, Inf. Proc. Letters, Vol. 7-1, (Jan 1978) pp. 1-6. (4.1)

[42] Driscoll, J.R. and Lien, Y.E.: A Selective Traversal Algorithmm for Binary Search Trees, C.ACM, Vol. 21-6 (June 1978) pp. 445-447. (3.4.1)

[43] Ellis, S.C.: Concurrent search and insertion in 2-3 trees. Tech. Rep. 78-05-01, Dept. of Comp. Science, Univ. of Washington, Seattle, Wash., 1978. (3.4.2.1)

[44] Flajolet, P. and Odlyzko, A.: The Average Height of Binary Trees and Other Simple Trees, submitted. (3.4.1.2)

[45] Flajolet, P., Raoult, R.C. and Vuillemin, J.: The number of Registers Required for Evaluating Arithmetic Expressions, Rap. Recherche 2, U. Paris-Sud, Orsay (Mar 1978). (3.4.1)

[46] Floyd, R.W.: Algorithm 245, C. ACM, Vol. 7 (1964), p. 701. (4.1.5,5.1)

[47] Foster, C.C.: Information Storage and Retrieval Using AVL Trees, In Proc. ACM 20th Nat. Conf., 1965, pp. 192-205. (3.4.1.3)

[48] Francon, J., Viennot G. and Vuillemin J.: Description and Analysis of an Efficient Priority Queue Representation, Proceedings of the 19th Annual Symposium on Foundations of Computer Science (1978), pp1-7. (5.1)

[49] Franta, W.R. and Maly, K.: A Comparison of Heaps and the TL Structure for the Simulation Event Set, C. ACM, Vol. 21-10, (Oct 1978), pp. 873-875. (5.1)

[50] Franta, W.R. and Maly, K.: An Efficient Data Structure for the Simulation Event Set, C.ACM, Vol. 20-8, (Aug 1977) pp. 596-602. (5.1)

[51] Fussenegger, F. and Gabow, H.N.: A Counting Approach to Lower Bounds for Selection Problems, J. ACM, Vol. 26-2, (Apr 1979), pp. 227-238. (5)

[52] Galil, Z. and Megiddo, N.: A Fast Selection Algorithm and the Problem of Optimum Distribution of Effort, J. ACM, Vol. 26-1, (Jan 1979), pp. 58-64. (5)

[53] Ghosh, S., and Senko, M.: File organization On the selection of random access index points for sequential files. J. ACM Vol. 16-4, (Oct 1969) pp. 569-579. (3.4)

[54] Gonnet, G.H. and Munro, J.I.: Efficient Ordering of Hash Tables. SIAM J. on Computing. Vol 8-3 (Aug 1979) pp. 463-478. (3.3.7,3.3.8)

[55] Gonnet, G.H. and Rogers, L.D.: An Algorithmic and complexity analysis of the heap as a data structure. Res. Rep. CS 75-20, U. of Waterloo, Waterloo, Canada, 1975. (5.1)

[56] Gonnet, G.H. and Rogers, L.D.: The Interpolation-Sequential Search Algorithm. Information Processing Letters, Vol 6-4, (Aug 1977), pp. 136-139. (3.2.3)

[57] Gonnet, G.H., Munro, J.I. and Suwanda, H.: Toward Self-Organizing Linear Search, Proc. 20th IEEE F.O.C.S. (Oct 1979) pp. 169-174. (3.1.2,3.1.3)

[58] Gonnet, G.H., Rogers, L.D. and George, A.: An Algorithmic and Complexity Analysis of Interpolation Search. Acta Informatica, Vol. 13-1 (Jan 1980) pp. 39-46. (3.2.2)

[59] Gonnet, G.H.: Average Lower Bounds for Open Addressing Hash Coding. Proc. Theoretical Comp. Science, Waterloo, (Aug 1977) pp. 159-162. (3.3)

[60] Gonnet, G.H.: Open Addressing Hashing with Unequal Probability Keys. To appear in the J. of Computer and System Sciences. (3.3.1)

[61] Gonnet, G.H.: *Interpolation and Interpolation-Hash Searching*. PhD Thesis. University of Waterloo, Waterloo. (1977). (3.2.2)

[62] Guibas, L., McCreight, E., Plass, M., and Roberts, J.: A new representation for linear lists. Proc. 9th ACM Symp. Theory of Computing, ACM, New York, 1977, 49-60. (3.4)

[63] Guibas, L.J. and Sedgewick, R.: A Dichromatic Framework for Balanced Trees. 19th Annual Symposium on Foundations of Computer Science. 1978 pp. 8-21 (3.4)

[64] Guibas, L.J. and Szemeredi, E.: The Analysis of Double Hashing. J.C.S.S. Vol 16-2 (Apr 1978) pp. 226-274. (3.3.4)

[65] Guibas, L.J.: The Analysis of Hashing Techniques that Exhibit k-ary Clustering. J. ACM Vol 25-4 (Oct 1978) pp. 544-555. (3.3)

[66] Gupta, U.: Bounds on Storage for Consecutive Retrieval, J. ACM, Vol. 26-1, (Jan 1979), pp. 28-36. (DB)

[67] Halatsis, C. and Philokypru, G.: Pseudo Chaining in Hash Tables. C. ACM Vol 21-7 (July 1978) pp. 554-557. (3.3)

[68] Held, G. and Stonebraker, M.: B-trees re-examined. C.ACM 21,2 (Feb.1978), 139-143. (3.4.2)

[69] Hendricks, W.J.: *An account of self-organizing systems*. SIAM J on Computing 5(1976), pp.715-723. (3.1.2,3.1.3)

[70] Hirschberg, D.S.: Fast Parallel Sorting Algorithms, C.ACM, Vol. 21-8 (Aug 1978) pp. 657-661. (4)

[71] Hoare, C.A.R.: Algorithm 63 and 64, C.ACM, Vol. 4-7, (July 1961) pp. 321-322. (4.1.3)

[72] Hoare, C.A.R.: Quicksort, Computer Journal, Vol. 5-4, (Apr 1962) pp. 10-15. (4.2.3)

[73] Horibe, Y. and Nemetz, T.O.H.: On the Max-Entropy Rule for a Binary Search Tree, Acta Inf. Vol. 12-1, (1979) pp. 63-72. (3.4.1.5)

[74] Horowitz, E., and Sahni, S.: Fundamentals of Data Structures , Computer Science Press, Inc., 1976. (3.4)

[75] Horvath, E.C.: Stable Sorting in Asymptotically Optimal Time and Extra Space, J.ACM, Vol. 25-2, (Apr 1978) pp. 177-199. (4.1.4.3)

[76] Hwang, K. and Yao, S.B.: Optimal Batched Searching of Tree Structured Files in Multiprocessor Computer Systems, J.ACM, Vol. 24-3, (July 1977) pp. 441-454. (3.4)

[77] Johnson, D.B. and Gonzalez, T.F.: Sorting Numbers in Linear Expected Time and Constant Extra Space. Proceedings of the 16th Allerton Conference, University of Illinois, (1978). (4)

[78] Johnson, D.B. and Kashdan, S.D.: Lower Bounds for Selection in X+Y and Other Multisets, J. ACM, Vol. 25-4, (Oct 1978), pp. 556-570. (5)

[79] Johnson, N.L. and Kotz, S.: *Discrete Distributions*. Houghton Mifflin, Boston, 1969. (8)

[80] Jonassen, A. and Dahl, O.: Analysis of an algorithm for Priority Queue Administration, BIT 15 (1975), 409-422. (5.1)

[81] Karlton, P.L., Fuller, S.H., Scroggs, R.E. and Kaehler, E.B.: Performance of Height-Balanced Trees, CACM, Vol. 19-1, 1976, pp. 23-28. (3.4.1.3)

[82] Kemp, R.: The Average Number of Registers Needed to Evaluate a Binary Tree Optimally, Acta Inf. Vol 11-4, (1979) pp. 363-372. (3.4.1)

[83] Knott, G.D.: A Numbering System for Binary Trees, C.ACM, Vol. 20-2, (Feb 1977) pp 113-115. (3.4.1)

[84] Knott, G.D.: Hashing Functions. The Computer J. Vol 18-3. (Aug 1975) pp. 265-278. (3.3)

[85] Knuth, D.E.: Structured Programming with Go To Statements, ACM C. Surveys, Vol. 6-4, (Dec 1974) pp 261-301. (3.1.1,3.4.1.1,4.1.4.1.3)

[86] Knuth, D.E.: The Art of Computer Programming, Vol. 1, Fundamental Algorithms, Addison-Wesley, Reading, Mass. (1973). (5.1)

[87] Knuth, D.E.: *The Art of Computer Programming: Sorting and Searching*, Vol. III, Addison-Wesley, Don Mills, Ont. 1973. (3.1,3.3,3.4.4,5.1.8)

[88] Konheim, A.G. and Weiss, B.: An Occupancy Discipline and Applications. SIAM J. Applied Math. Vol 14 (1966). pp 1266-1274. (3.3.3)

[89] Kosaraju, S.R.: Insertions and Deletions in One-Sided Height-Balanced Trees, C. ACM, Vol. 21-3. (Mar 1978) pp.226-227. (3.4.1.3)

[90] Kruijer, H.S.M.: The Interpolated File Search Method. Informatie, 16-11 (Nov 1974), pp. 612-615. (3.2)

[91] Kwong, Y.S. and Wood, D.: T-trees A variant of B-trees. Tech. Rep. 78-CS-18, Computer Science Dept., McMaster Univ., Hamilton, Ont., 1978. (3.4.2)

[92] Lee, D.T. and Wong, C.K.: Worst-Case Analysis for Region and Partial Region Searches in Multidimensional Binary Search Trees and Balanced Quad Trees, Acta Inf. Vol. 9-1, (1977) pp. 23-29. (3.5)

[93] Lee, K.P.: A Linear Algorithm for Copying Binary Trees Using Bounded Workspace, C.ACM, Vol. 23-3, (Mar 1980) pp. 159-162. (3.4.1)

[94] Lipton, R.J., Rosenberg, A.L. and Yao, A.C.: External Hashing Schemes for Collection of Data Structures, J. ACM, Vol. 27-1, (Jan 1980), pp. 81-95. (3.3)

[95] Lomet, D.B.: Multi-Table Search for B-Tree Files, RC-7461, IBM T.J. Watson Research Center. (3.4.2)

[96] Lotka, A.J.: The Frequency Distribution of Scientific Production. *J of the Washington Academy of Sciences*. Vol 16-12. (1926). pp. 317-323. (8)

[97] Luccio, F. and Pagli, L.: Power Trees, C.ACM, Vol. 21-11, (Nov 1978) pp. 941-947. (3.4.1.3)

[98] Lum, V.Y., Yuen, P.S.T. and Dodd, M.: Key-to-address transform techniques: a fundamental performance study on large existing formatted files. C.ACM, Vol.14,4 (1971),228-239. (3.3)

[99] Lum, V.Y.: General performance analysis of key-to-address transformation methods using an abstract file concept. C.ACM, Vol.16,10 (1973), 603-612. (3.3)

[100] Lyon, G.: Hashing with Linear Probing and Frequency Ordering. J. Res. Nat. B. S. Vol 83-5 (Sep 1978) pp. 445-447. (3.3.3)

[101] Lyon, G.: Packed Scatter Tables, C. ACM, Vol. 21-10, (Oct 1978), pp. 857-865. (3.3.7)

[102] MacLaren, M.D.: Internal Sorting by Radix Plus Sifting. J.ACM, Vol. 13-3, (July 1966) pp 404-411. (4)

[103] MacVeigh, D.T.: Effect of Data Representation on Cost of Sparse Matrix Operations, Acta Inf. Vol. 7 (1977) pp. 361-394. (DR)

[104] Mallach, E.G.: Scatter Storage Techniques. The Computer J. Vol 20-2 (May 1977) pp. 137-140. (3.3.7)

[105] Maly, K.: A Note on Virtual Memory Indexes. C.ACM, Vol. 21-9 (Sep 1978) pp. 786-787. (3.4.2)

[106] Manacher, G.K.: Significant Improvements to the Hwang-Lin Merging Algorithm, J. ACM, Vol. 26-3, (July 1979), pp. 434-440. (4.3)

[107] Manacher, G.K.: The Ford-Johnson Sorting Algorithmm is Not Optimal, J. ACM, Vol 26-3, (July 1979), pp. 441-456. (4.1)

[108] Maruyama, K. and Smith, S.E.: Analysis of Design Alternatives for Virtual Memory Indexes, C.ACM, Vol. 20-4 (Apr 1977) pp. 245-254. (3.4.6)

[109] Maurer, W.D. and Lewis, T.E.: Hash table methods. ACM C. Surveys Vol. 7-1, (Mar 1975) 5-19. (3.3)

[110] Maurer, W.D.: An Improved Hash Code for Scatter Storage. C. ACM Vol 11-1 (Jan 1968) pp. 35-38. (3.3)

[111] McCabe, J.: *On serial files with relocatable records*. Operations Res. 12 (1965), pp. 609-618. (3.1.2)

[112] McCreight, E.: Pagination of B*trees with variable-length records. C.ACM 20.9 (Sep 1977), 670-674. (3.4.3)

[113] McKellar, A.C. and Wong, C.K.: Dynamic Placement of Records in Linear Storage, J.ACM, Vol 25-3, (July 1978), pp. 421-434. (3.1)

[114] McWrath, W.E.: Relationships between hard/soft, pure/applied, and life/non life disciplines and subject book use in a university library. *Information Processing and Management*. Vol. 14-1. (1978). pp. 17-28. (8)

[115] Mendelson, H. and Yechiali, U.: Performance Measures for Ordered Lists in Random-Access Files. J. ACM, Vol. 26-4, (Oct 1979), pp. 654-667. (3.3)

[116] Mendelson, H., Pliskin, J.S. and Yechiali, U.: Optimal Storage Allocation for Serial Files, C.ACM, Vol. 22-2, (Feb 1979) pp. 124-130. (DB)

[117] Miller, R. and Snyder, L.: Multiple access to Btrees. Proc. Conf. Inform. Sci. and Syst., March 1978. (3.4.2)

[118] Miller, R., Pippenger, N., Rosenberg, A. and Snyder, L.: Optimal 2-3 trees. IBM Res. Rep. RC 6505, 1977. (3.4.2.1)

[119] Morris, R.: Scatter Storage Techniques. C. ACM Vol 11-1 (Jan 1968) pp 38-44. (3.3)

[120] Motzkin, D.: The Use of Normal Multiplication Tables for Information Storage and Retrieval, C.ACM, Vol. 22-3, (Mar 1979) pp. 193-207. (DB)

[121] Munro, J.I. and Spira, P.M.: Sorting and Searching in Multisets, SIAM J on Computing, Vol. 5-1 (Mar 1976) pp. 1-8. (4)

[122] Murphy, L.J.: Lotka's Law in the Humanities. *J. of the American Society of Information Science*. Vol. 24-6. (1973). pp. 461-462. (8)

[123] Nat, M. van der.: On Interpolation Search, C. ACM, Vol. 22-12, (Dec 1979), p. 681. (3.2.2)

[124] Nievergelt, J. and Reingold, E.M.: Binary Search Trees of Bounded Balance, SIAM J. Computing, Vol. 2-1, (1973), 33-43. (3.4.1)

[125] Nievergelt, J.: Binary Search Trees and File Organization, ACM C. Surveys, Vol. 6-3, (Sep 1974), pp. 195-207. (3.4.1)

[126] Odlyzko, A.M.: Periodic Oscillations of Coefficients of Power Series that Satisfy Functional Equations, to appear in Advances in Mathematics. (3.4.2)

[127] Ottmann, Th. and Stucky, W.: Higher Order Analysis of Random 1-2 Brother Trees, Institut fur Angewandte Informatik, U. Karlsruhe, 84 (Aug 1979). (3.4.2)

[128] Ottmann, Th., Six, H.W. and Wood, D.: Right Brother Trees, C.ACM, Vol. 21-9 (Sep 1978) pp. 769-776. (3.4.1.3)

[129] Pan, V.Y.: Optimal Methods for the Evaluation of Polynomials over the Fields C and R, IBM Res. Report RC7754, (May 1979). (6.4)

[130] Peltola, E. and Erkio, H.: Insertion Merge Sorting, Inf. Proc. Letters, Vol. 7-2, (Feb 1978) pp. 92-99. (4.2)

[131] Perl, Y. and Reingold, E.M.: Understanding the Complexity of Interpolation Search, Information Processing Letters (Dec 1977), Vol. 6-6, pp. 219-221. (3.2)

[132] Perl, Y., Itai, A. and Avni, H.: Interpolation Search - A Log Log N Search, C.ACM, Vol. 21-7, (July 1978), pp. 550-553. (3.2)

[133] Peterson, W.W.: Addressing for Random-Access Storage, IBM Journal of Research and Development, Vol. 1-4 (Apr 1957), pp. 130-146. (3.2.3.3)

[134] Pippenger, N.: On the Application of Coding Theory to Hashing, IBM J. Res. Development, Vol 23-2 (Mar 1979) pp. 225-226. (3.3)

[135] Pooch, U.W. and Nieder, A.: A Survey of Indexing Techniques for Sparse Matrices, ACM C. Surveys, Vol. 5-2, (June 1973) pp. 109-133. (DR)

[136] Pope, A.: Bradford's Law and the Periodical Literature of Information Sciences, *J. of the Association of Information Sciences*, Vol. 26-4, (1975), pp. 207-213. (8)

[137] Porter, T. and Simon, I.: Random Insertion into a Priority Queue structure, IEEE Transactions SE-1 (Sep 1975), 292-298. (5.1)

[138] Power, L.R.: Internal Sorting Using a Mimimal Tree Merge Strategy, ACM-TOMS, Vol. 6-1 (Mar 1980) pp. 68-79. (4.2)

[139] Price, C.E.: Table Lookup Techniques, Computing Surveys, Vol. 3-2 (June 1971) pp. 56-58. (3.2)

[140] Proskurowski, A.: On the Generation of Binary Trees, J. ACM, Vol. 27-1, (Jan 1980), pp. 1-2. (3.4.1)

[141] Raiha, K.J. and Zweben, S.H.: An Optimal Insertion Algorithm for One-Sided Height-Balanced Binary Search Trees, C.ACM, Vol. 22-9, (Sep 1979) pp. 508-512. (3.4.1.3)

[142] Reingold, E.M.: A note on 3-2 trees, Fibonacci Quarterly Vol. 17-2, pp. 151-157 (Apr 1979). (3.4.2.1)

[143] Rivest, R.: *On self-organizing sequential search heuristics*, C.ACM, 19, (1976), pp. 63-67. (3.1)

[144] Rivest, R.L.: Optimal Arrangement of Keys in a Hash Table, J. ACM Vol 25-2 (Apr 1978) pp. 200-209. (3.3.8)

[145] Robson, J.M.: The Height of Binary Search Trees, Australian Computer J. Vol 11-4 (Nov 1979) pp. 151-153. (3.4.1.1)

[146] Rosenberg, A.L. and Snyder, L.: Minimal comparison 2-3 trees, SIAM J. Comput. Vol. 7-4, (Nov 1978) pp. 465-480. (3.4.2.1)

[147] Rosenberg, A.L. and Snyder, L.: Time- and Space-Optimality in B-Trees, Res. Rep. #167, (Aug 1979), Dept. of Computer Science, Yale University. (3.4.2)

[148] Rosenberg, A.L. and Stockmeyer, L.J.: Hashing Schemes for Extendible Arrays, J.ACM, Vol. 24-2, (Apr 1977) pp. 199-221. (3.3)

[149] Rosenberg, A.L. and Stockmeyer, L.J.: Storage Schemes for Boundedly Extendible Arrays, Acta Inf. Vol. 7 (1977) pp. 289-303. (DR)

[150] Rosenberg, A.L., Wood, D. and Galil, Z.: Encoding Tree-Like Data Structures in Trees, IBM Res. Report RC7353 (Oct 1978). (DR)

[151] Rosenberg, A.L.: Data Encodings and their Costs, Acta Inf. Vol. 9-3, (1978) pp. 273-292. (DR)

[152] Rosenberg, A.L.: Encoding Data Structures in Trees, J.ACM, Vol 26-4, (Oct 1979), pp. 668-689. (3.4)

[153] Rotem, D. and Varol, Y.L.: Generation of Binary Trees from Ballot Sequences, J. ACM, Vol. 25-3, (July 1978), pp. 396-404. (3.4.1)

[154] Samadi, B.: B-trees in a system with multiple views. Inf. Process. Lett. Vol. 5-4, (Oct 1976) pp. 107-112. (3.4.2)

[155] Santoro, N.: Extending the Four Russians' Bound to General Matrix Multiplication, Inf. Proc. Letters, Vol. 10-2, (Mar 1980) pp. 87-88. (6.3)

[156] Saxe, J.B. and Bentley, J.L.: Transforming Data Structures to Dynamic Structures, Dept of Computer Science, C.M.U. (Sep 1979). (2)

[157] Saxe, J.B.: On the Number of Range Queries in k-Space, Discr App Math, Vol. 1-3, (1979) pp. 217-225. (3.5)

[158] Schonhage, A.: Fast Multiplication of Polynomials Over Fields of Characteristic 2, Acta Inf. Vol. 7 (1977) pp. 395-398. (6)

[159] Sedgewick, R.: Implementing Quicksort Programs, C. ACM, Vol. 21-10, (Oct 1978), pp. 847-856. (4.1.3)

[160] Sedgewick, R.: The Analysis of Quicksort Programs, Acta Inf., Vol. 7 (1977) pp. 327-355. (4.1.3)

[161] Severance, D.G. and Carlis, J.V.: A Practical Approach to Selecting Record Access Paths, ACM C. Surveys, (Dec 1977) pp. 259-272. (DB)

[162] Severance, D.G.: Identifier Search Mechanisms: A Survey and Generalized Model, ACM C. Surveys, Vol. 6-3, (Sep 1974) pp. 174-194. (3)

[163] Sheil, B.A.: Median Split Trees: A Fast Lookup Technique for Frequently Ocurring Keys, C.ACM, Vol. 21-11 (Nov 1978) pp. 947-958. (3.4.1.5)

[164] Shneiderman, B.: Jump Searching: A Fast Sequential Search Technique, C. ACM, Vol. 21-10, (Oct 1978), pp. 831-835. (3.1.5)

[165] Snyder, L.: On B-Trees Re-Examined, C.ACM, Vol. 21-7 (July 1978) p 594. (3.4.2)

[166] Solomon, M. and Finkel, R.A.: A Note on Enumerating Binary Trees, J. ACM, Vol. 27-1, (Jan 1980), pp. 3-5. (3.4.1)

[167] Sprugnoli, R.: Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets, C.ACM, Vol 20-11, (Nov 1977) pp. 841-850. (3.3)

[168] Strong, H.R., Markowsky, G. and Chandra, A.K.: Search Within a Page, J. ACM, Vol. 26-3, (July 1979), pp. 457-482. (DB.3.4.5)

[169] Tanenbaum, A.: *Simulations of dynamic sequential search algorithms*, C.ACM Vol. 21-9, (Sep 1978), pp. 790-791. (3.1)

[170] Tarjan, R.E. and Yao, A.C-C.: Storing a Sparse Table, C.ACM, Vol. 22-11, (Nov 1979) pp. 606-611. (DR)

[171] Todd, S.: Algorithm and Hardware for a Merge Sort Using Multiple Processors, IBM J Res. Development, Vol. 22-5 (Sep 1978) pp. 509-517. (4.2.1)

[172] Trabb, L.: Set Representation and Set Intersection, Stanford Rep. 78-861, (Dec 1978). (3.4.4,3.3)

[173] Ullman, J.D.: A Note on the Efficiency of Hashing Functions. J. ACM Vol 19-3 (July 1972) pp. 569-575. (3.3)

[174] Ulrich, E.G.: Event Manipulation for Discrete Simulations Requiring Large Number of Events, C.ACM, Vol. 21-9 (Sep 1978) pp. 777-785. (5.1)

[175] Unterauer, K.: Dynamic Weighted Binary Search Trees, Acta Inf. Vol. 11-4, (1979) pp. 341-362. (3.4.1.4)

[176] Vaishnavi, V.K., Kriegel, H.P. and Wood, D.: Height Balanced 2-3 Trees, Computing, Vol 21, (1979) pp. 195-211. (3.4.2.1)

[177] Vuillemin, J.: A Data Structure for Manipulating Priority Queues, C. ACM, Vol. 21-4 (Apr 1978) pp. 309-314. (5.1)

[178] Vuillemin, J.: A Unifying Look at Data Structures, C.ACM, Vol. 23-4, (Apr 1980) pp. 229-239. (2.3.5,3.4.1,3.1.1)

[179] Williams, J.W.J.: Algorithm 232, C. ACM, Vol. 7 (1964), pp. 347-348. (4.1.5,5.1)

[180] Wirth, N. Algorithms + Data Structures = Programs (New Jersey: Prentice-Hall, 1976). (3.4)

[181] Woodall, A.D.: A Recursive Tree Sort, Computer Journal, Vol. 14-1, (1971) pp. 104-104. (4.1)

[182] Yao, A.C-C. and Yao, F.F.: Lower Bounds on Merging Networks, J.ACM, Vol 23-3, (July 1976) pp. 566-571. (4.3)

[183] Yao, A.C. and Yao, F.F.: The Complexity of Searching an Ordered Random Table. Proceedings of the Symposium on Foundations of Computer Science, Houston, (Oct 1976), 173-176. (3.2)

[184] Yao, A.C.: On Random 2-3 Trees, Acta Informatica, Vol. 9-2, (1978), pp. 159-170. (3.4.2.1)

[185] Zaki, A. and Baer, J.L.: A Comparison of Query Costs in AVL and 2-3 Trees, Tech. Rep. 78-02-01 (Aug 1979) Dept. Computer Science, U. of Washington. (3.4.1.3,3.4.2.1)

[186] Zipf, G.K.: *Human Behaviour and the Principle of Least Effort*. Cambridge MA. Addison-Wesley. 1949. (8)

[187] Zobrist, A.L. and Carlson, F.R.: Detection of Combined Occurences, C.ACM, Vol. 20-1, (Jan 1977) pp. 31-35, also Vol. 20-9 (Sep 1977) pp. 678-680. (3.6)

[188] Zweben, S.H. and McDonald, M.A.: An Optimal Method for Deletions in One-Sided Height-Balanced Trees, C.ACM, Vol. 21-6 (June 1978) pp. 441-445. (3.4.1.3)