UNIVERSITY OF WATERLOO
COMPUTER SCIENCE DEPARTMENT

Linear Hashing with Linear Probing

Per-Åke Larson

CS-83-38

January, 1984

Linear hashing with linear probing *

Per-Åke Larson

Data Structuring Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario
N2L 3G1

Technical Report CS-83-38

ABSTRACT

A new, simple method for handling overflow records in connection with linear hashing is proposed. The method is based on linear probing and does not rely on chaining. No separate overflow area is required. The expansion sequence of linear hashing is modified to improve the performance. This requires changes in the address computation. A new address computation algorithm and an expansion algorithm are given. The performance of the method is studied by simulation. The overall performance is competitive with that of other variants of linear hashing up to a load factor of 0.80-0.85.

1. Introduction

Linear hashing is a file structure intended for files that grow and shrink dynamically. The technique was originally proposed by Litwin [LI80]. Larson [LA80] and subsequently Ramamohanarao and Lloyd [RL82] developed more general schemes based on a technique called partial expansions. These schemes are able to maintain good retrieval performance and high storage utilization independently of the file size, and without requiring periodic reorganization.

Linear hashing requires some method for handling overflow records. Not all of the methods developed for traditional hashing schemes are applicable, because linear hashing requires that all records hashing to a page can be located. The three schemes mentioned above were all developed assuming that separate chaining is used, that is, records overflowing from a page are placed somewhere in a

* This work was supported by Natural Sciences and Engineering Research Council of Canada, Grant A2460.

separate overflow area and linked into an overflow chain emanating from the home page. In this paper an overflow handling method based on open addressing is presented. The technique used is linear probing, that is, if a record does not fit into its home page, the next higher page is tried, etc. until the first non-full page is found. There is no dedicated overflow area, and no pointers are used.

Using linear probing for handling overflow records has a number of advantages. The retrieval and insertion algorithms are extremely simple. Linear probing preserves locality of reference, thus (potentially) avoiding long disk seeks. Insertions and file expansions can easily be speeded up by allocating more buffer space, reading in several pages at a time. This will be discussed later. Gonnet and Larson [GL82] developed a method that, by using a small amount of extra internal storage, guarantees retrieval of any record in one access. See also [LK83] where implementation aspects are discussed. The method is not, however, able to handle dynamic files. Linear hashing with linear probing gives the basis for a similar one-access method for dynamic files. Those results will be presented in a forthcoming paper.

Several other techniques for handling overflow records in connection with linear hashing have been studied. Mullin [MU81] investigated prime area chaining, that is, chaining is used but overflow records are stored in non-full primary pages, not in dedicated overflow pages. Larson devised [LA82b] and analysed [LA83] a method for combining the overflow area and the primary pages in the same file. He also suggested using several overflow chains per page. Ramamohanarao and Sacks-Davis [RSD83] proposed recursive linear hashing where overflow records from a linear hash file are stored in another (smaller) linear hash file, overflow records from the second level file go into a third level linear hash file, etc.

The rest of the paper is organized as follows. Section 2 gives a brief overview of linear hashing with partial expansion. In section 3 the basic ideas of the new method are presented. The necessary algorithms are given in section 4 and performance results are presented in section 5.

The new method is here presented as a modification of Larson's scheme [LA80], but the same ideas can be applied to the scheme of Ramamohanarao and Lloyd [RL82].

2. Linear hashing with partial expansions

Linear hashing is a technique for gradually expanding (or contracting) the storage area of a hash file. The file is expanded by adding a new page at the end of the file and relocating a number of records to the new page. The basic idea of the method are briefly outlined in this section. More details can be found in [LI80, LA80].

The original scheme proposed by Litwin [LI80] proceeds by first splitting page 0, then page 1, etc. Consider a file consisting of N pages with addresses $0, 1, ..., N-1$. When page j, $j = 0, 1, ..., N-1$, is split the file is extended by one page with address $N+j$. Approximately half of the records whose current address is j are moved to the new page. A pointer p keeps track of which page is the next one to be split. When all the N original pages have been split the file size has doubled. The pointer p is reset to zero and the process

starts over again. A doubling of the file is called a full expansion.

The key idea is to split the pages in a predetermined sequence. After splitting of a page, it should be possible to locate the records moved to the new page without having to access the old page. The essence of the problem is to design an algorithm which, based solely on the key of the record, determines whether a record remains on the old page or is moved to the new page. It must also achieve the goal that approximately half of the records are moved. There are several solutions to this problem; the interested reader is referred to [LI80, LA80, RL82].

The development of linear hashing with partial expansions was motivated by the observation that linear hashing creates a very uneven distribution of the load over the file. This slows down retrieval and insertions by creating a large number of overflow records. The load of a page that has already been split is expected to be only half of the load of a page that has not yet been split. To achieve a more even load, the doubling of the file (a full expansion) is carried out by a series of partial expansions. If two partial expansions are used, the first one increases the file size to 1.5 times the original size and the second one to twice the original size.

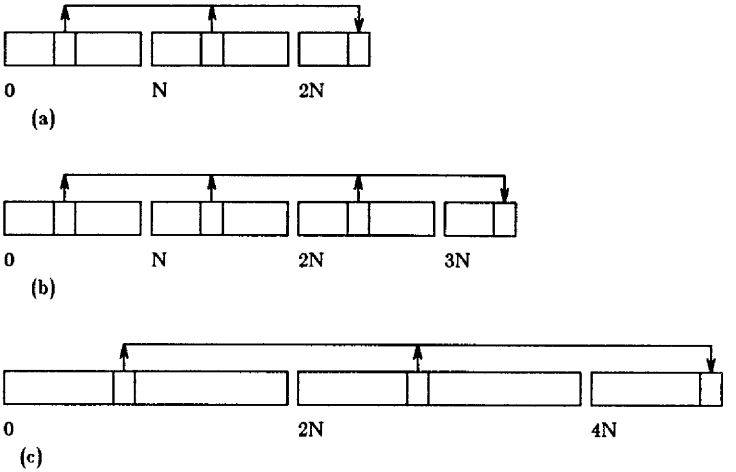

Fig. 1: Illustration of linear hashing with two partial expansions

Assume that two partial expansions per full expansion are used. This is illustrated in Fig. 1. We start from a file of $2N$ pages, logically divided into N groups of two pages each. Group j consists of pages $(j, N + j)$, $j = 0, 1, ..., N - 1$, see Fig. 1(a). To expand the file, group 0 is first expanded, then group 1, etc. by one page. When expanding group j, $j = 0, 1, ..., N - 1$, approximately 1/3 of of the records from page j and $N + j$ are relocated to the new page $2N + j$. When the last group, $(N - 1, 2N - 1)$, has been expanded the file has increased to $3N$ pages. The second partial expansion starts, the only difference being that now groups of three

pages, $(j, N + j, 2N + j)$, are expanded to four pages, see Fig. 1(b). When the second partial expansion is completed, the file size has doubled from $2N$ to $4N$ pages. The next partial expansion reverts to expanding groups of size two, $(j, 2N + j)$, $j = 0, 1, ..., 2N - 1$, see Fig. 1(c). The one after that expands groups of size three, etc. This approach can immediately be extended to any number of partial expansions per full expansion.

To implement linear hashing with partial expansions an address computation algorithm is required. Such an algorithm is given in [LA80]. It computes the (current) home address of a record at any time, given its key. It is designed for any number of partial expansions per full expansion.

The scheme outlined above gives a method for expanding the file one page at a time. In addition, rules for determining *when* to expand (or contract) the file are needed. A set of such rules is called a *control function* because they control the expansion and contraction rate of the file. Several alternatives are possible, but we will here consider only the rule of constant storage utilization. According to this rule the file is expanded whenever the overall storage utilization rises above a threshold α, $0 < \alpha < 1$, selected by the user. When computing the storage utilization the space allocated for overflow records, if any, is also taken into account.

The results reported in [LA82a] show that increasing the number of partial expansions improves the retrieval performance as expected. On the other hand, insertion costs tend to increase. Two partial expansions per full expansion seem to be a good compromise in many situations.

3. Linear hashing with linear probing

An overflow handling method based on open addressing must satisfy certain requirements to be applicable to linear hashing. As will become clear from the discussion in this section, linear probing satisfies these requirements. It also offers a number of other advantages.

The cost of expanding a linear hash file is directly affected by the cost of locating all records hashing to a given page. An expansion involves locating all records hashing to a number of existing pages and relocating some of them to the new page. A method generating a large number of possible probe sequences, like double hashing, for example, would make this too costly. Each probe sequence emanating from a page participating in the expansion must be checked. Linear probing generates only one probe sequence from a page. Furthermore, the probe sequence is the same as the physical address sequence. Provided that sufficient buffer space is available, several pages can be read or written at the same time, thus speeding up insertions and expansions.

When expanding the file the existing probe sequences must somehow be extended to include the new page created. It is desirable that this can be done without any actual relocation of records. This is easily achieved by modifying linear probing so as not to wrap around to the first page when reaching the (currently) last page of the file. If there are overflow records from the last page in the current address space, they are allowed to go into the first unused page(s) at the end of the file. In effect the page has, prematurely, been taken into use by receiving overflow records before receiving any "native" records. The next time

the file is expanded it will be within the address space of the file. It is very unlikely that more than one such overflow page would be needed. The effect of this modification is that each record has, in principle, an infinite probe sequence.

Linear hashing with partial expansion extends the file by one page by increasing the size of one group. The expansion sequence discussed in the previous section was group 0, group 1, etc. This particular sequence is not crucial for the method; it merely simplifies the address computation. The only requirement is that the sequence is predetermined and that all groups are eventually expanded.

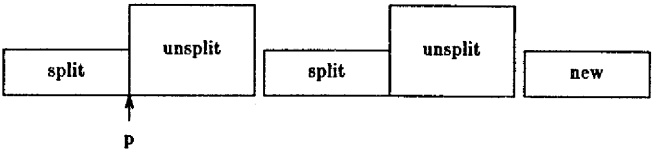

Fig. 2: Illustration of the load distribution, $n_0 = 2$.

The expansion sequence 0, 1, $\cdots$ has a serious drawback when overflow records are handled by linear probing. As illustrated in Fig. 2 for the case of two partial expansions, it creates blocks of consecutive pages with a high load factor (the unsplit pages). In these areas long islands of full pages are very likely to occur. Long islands of full pages slow down insertions, subsequent retrieval and also expansions. They can be avoided, to some extent, by changing the expansion sequence in such a way that the split pages with a lower load are spread more evenly over the file. The expansion sequence 0, 1, $\cdots$ uses a step length of one. We can instead use a larger step length and make a number of sweeps over the groups. If a step length of s, $s \geq 1$, is used, the first sweep would expand groups $0, s, 2s,...$, the second sweep groups $1, s+1, 2s+1,...$ and the last sweep would be $s-1, 2s-1, 3s-1,...$. This achieves the desired effect of spreading the split pages more evenly over the file.

One further modification of the expansion sequence is proposed: to have each sweep go backwards instead of forwards. If the file consists of N groups, the first sweep would be $N-1, N-1-s, N-1-2s,...$, and correspondingly for the other sweeps. This reduces the risk of having a very long island to check during an expansion, because it is likely to have been shortened by previous expansions.

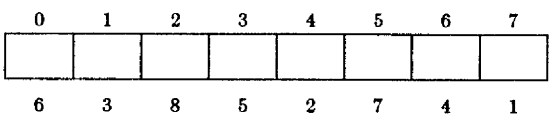

Fig. 3: Expansion sequence when three sweeps are used.

The expansion sequence for a file consisting of 8 groups when 3 sweeps are used is shown in Fig. 3. The first sweep expands groups 7, 4, and 1 (in that order), the

second sweep groups 6, 3, 0 and the last one groups 5 and 2.

The following control function is suggested: the file is expanded whenever the overall load factor increases over a user-selected threshold α, $0 < \alpha < 1$. Because there is no separate overflow area, this rule will result in a storage utilization that, for all practical purposes, is constant and equal to α. The performance analysis in section 5 assumes the use of this rule.

4. Algorithms

In this section algorithms needed to implement the basic file operations are discussed. It is assumed that the expansions sequence discussed in the previous section is used and that overflow records are handled by linear probing. Only two algorithms will be given in detail: an algorithm for address computation and an algorithm for file expansion. Insertion, retrieval and deletion are basically the same as for traditional linear probing and will be discussed only briefly. A linear hash file with linear probing is defined by a number of parameters, and its current state by a number of state variables:

Parameters

b	page size in number of records
N	original number of groups
n_0	number of partial expansions per full expansion
s	number of sweeps per partial expansion (or step length used)

State variables

cpx	current partial expansion. Initial value: $cpx = 1$
sw	current sweep, $1 \leq sw \leq s$. Initial value: $sw = 1$
p	next group to be expanded. Initial value: $p = N - 1$
n	number of pages in an unexpected group, $n_0 \leq n \leq 2n_0 - 1$. Initial value: $n = n_0$
$maxadr$	highest address in current address space. Initial value: $maxadr = n_0\, N - 1$
$lstpg$	highest page in use, $lstpg \geq maxadr$. Initial value: $lstpg = maxadr$

Two of the state variables are not absolutely necessary: $maxadr$ can be computed from n_0, N, cpx, sw and p, and n can be computed from cpx and n_0.

The address computation algorithm given below makes use of two hashing functions. The first one, denoted by h, is a normal hashing function, $0 \leq h(K) \leq n_0\, N - 1$, and is used for distributing the records over the original file (of size $n_0\, N$ pages). The second one, denoted by D, returns a sequence of values, $D(K) = (d_1(K), d_2(K),...)$, where the values d_i are uniformly distributed in (0,1]. The value $d_i(K)$ is used to determine whether to relocate the record

with key K to the new page during the *ith* partial expansion. Assume that the *ith* partial expansion expands each group from n to $n + 1$ pages. To achieve a uniform distribution of the load over the file (at the end of the expansion) approximately $1/(n + 1)$ of the records should be relocated to the new part of the file. This can be achieved by relocating a record if $d_i(K) \leq 1/(n + 1)$ (changing its address), otherwise not. The group size during the *ith* expansion is $n = n_0 + (i - 1) \bmod n_0$. The address computation algorithm given below is based on this idea. The hashing function D can easily be implemented by a random number generator to which the key is provided as the seed.

```
address (K)

    ha: = h(K); fsz: = n0*N;
    ngrps: = N;

    for x: = 1 to cpx do begin
        if dx(K) ≤ 1/(n0 + (x − 1) mod n0) then begin
            k: = ngrps − 1 − (ha mod ngrps);
            swp: = k mod s;
            swpl: = ngrps div s;
            fsw: = swp*swpl + min (swp, ngrps mod s);
            npg: = fsz + fsw + (k div s) + 1;
            if npg ≤ maxadr then ha: = npg;
        end;
        fsz: = fsz + ngrps;
        if (x − 1) mod n0 = n0 − 1 then ngrps: = 2 * ngrps;
    endloop;
    return (ha);
end {address};
```

The current address of a record is computed by tracing all the address changes up to the current partial expansion. If the record was moved during the xth expansion, that is, if the condition of the first if-statement evaluates to true, the address of the new page must be computed. This is done by adding the file size when the xth expansion started (fsz), the number of pages created by fully completed sweeps (fsw) and by the current sweep (the term (k **div** s) + 1). If npg $\leq$ *maxadr* the new address is within the current address range. This test can fail only when x = *cpx*.

Once the home address has been computed, insertion or retrieval of a record is done in the same way as for traditional linear probing. The only difference is that the probe sequence of a record does not wrap around when reaching the last page of the file. If the last page is reached during an insertion and it is full, then the next page is (prematurely) taken into use and *lstpg* is increased by one.

It is assumed that the file is expanded as soon as the overall load factor rises above the threshold α. An algorithm for expanding the file by one page is given at the end of this section. File expansion necessitates some (local)

rearrangement of records. When records are moved to the new page, space will be freed up in the old part of the file. To retain searchability these holes must, whenever possible, be filled by moving overflow records back to, or at least closer to, their home pages.

Consider a page participating in an expansion and denote its address by pg. All records whose (current) home address is pg must be checked because some of them will be moved to the new page. To find these records page pg is first checked, then pg + 1, etc. up to and including the first non-full page. The expansion algorithm scans over this area twice. The first scan collects every record that is not stored on its home page. The set of collected records will include those that are to be moved to the new page, and records that possibly will be moved closer to their home pages. The collected records are temporarily stored in a record pool until reinserted during the second scan over the area. In addition to the record itself, its home address is also stored. To avoid writing, pages are not modified during the first scan.

The second scan goes over the same area as the first scan, restoring the records collected during the first scan. Whenever there is an empty slot on a page, the algorithm attempts to fill that slot with a record from the record pool. The record selected is the one with the lowest home address.

The above process is repeated for every old page participating in the expansion. The records remaining in the record pool are those that are to be moved to the new page. They are inserted in the last part of the algorithm.

A page in the file consists of b record slots and each slot consists of three parts: a status field, the key of the record and the rest of the record. The status field indicates whether the slot is empty or full. As written the algorithm uses only one buffer with the same size and format as a page. However, expansions can be significantly speeded up by using more buffer space, reading and writing several pages when accessing the file. The effect of additional buffer space will be studied further in the next section.

The structure of the record pool is intentionally left unspecified. The actual implementation will (mainly) depend on the amount of internal storage available. If main memory space is abundant, all records in the record pool can reside in main memory, otherwise some or all of them will have to be temporarily stored on disk. The size of the record pool is one of the variables studied in the next section.

```
expand (p);

gr: = p; np: = n;
lvl: = ⌊(cpx − 1)/n0⌋;
ngr: = N * 2 ** lvl;

{update state variables}
maxadr: = maxadr + 1;
p: = p − s;
if p < 0 then begin
    sw: = sw + 1; p: = ngr − sw;
    if sw > s then begin
        cpx: = cpx + 1; n: = n + 1;
        sw: = 1; p: = ngr − 1;
        if n ≥ 2 * n0 then begin
            n: = n0; p: = 2 * ngr − 1;
        end;
    end;
end;

for i: = 1 to np do begin
    pg: = gr + (i − 1) * ngr;

    {collect all records to be relocated and store them in rcrdpool}
    cp: = pg − 1; lmdf: = pg − 1;
    repeat
        cp: = cp + 1;
        read page cp into buffer;
        cnt: = 0;
        for j: = 1 to b do
            if buffer.slot[j].status=full then begin
                cnt: = cnt + 1;
                adr: = address(buffer.slot[j].key);
                if adr ≠ cp then begin
                    lmdf: = cp;
                    insert (buffer.slot[j], adr) into rcrdpool;
                end;
            end;
    until cnt < b or cp = lstpg;
```

```
{reinsert all records whose address is ≤ lmdf}
for cp: = pg to lmdf do begin
    read page cp into buffer;
    for j: = 1 to b do begin
        if buffer.slot[j].status=full and address(buffer.slot [j].key) ≠ cp
        then buffer.slot[j].status=empty;

        if buffer.slot[j].status=empty and not empty(rcrdpool)
        then begin
            {note: the records in rcrdpool are assumed to be sorted
            in ascending order on home address}
            adr:=[home address of first record in rcrdpool];
            if adr ≤ cp then begin
                buffer.slot[j]:=[first record in rcrdpool];
                buffer.slot[j].status:=full;
                delete first record from rcrdpool;
            end;
        end;
    end {j-loop};
    write buffer into page cp;
end  {cp-loop};

end {i-loop};

{all records remaining in rcrdpool have home address = maxadr}

cp: = maxadr − 1;
repeat
    cp: = cp + 1;
    if cp ≤ lstpg
        then read page cp into buffer
        else for j: = 1 to b do buffer.slot[j].status:=empty;

    j: = 0;
    while j < b and not empty (rcrdpool) do begin
        j: = j + 1;
        if buffer.slot[j].status=empty then begin
            buffer.slot[j]:=[first record in rcrdpool];
            buffer.slot[j.]status:=full;
        end;
    endloop;
    write buffer into page cp;
until empty (rcrdpool);

if lstpg < cp then lstpg: = cp;

end {expand};
```

5. Performance

In order to study the performance of the new method and the effects of parameter changes, a simulation model was built. Results obtained by a series of simulation experiments are presented and discussed in this section. The following performance measures are considered: average number of accesses for successful and unsuccessful searches, average number of accesses for insertion of a record and average size of the record pool used during an expansion.

It is obvious that the performance will vary cyclically where a cycle corresponds to one full expansions. The performance behaviour over a full expansion is illustrated in Fig. 4 to Fig. 7. The parameters for the example file are: page size 20, storage utilization 0.8, and 2 partial expansions per full expansion. The results plotted are the averages from 100 runs. Two different cases are shown: 2 sweeps (solid line) and 4 sweeps (dotted line) per partial expansion.

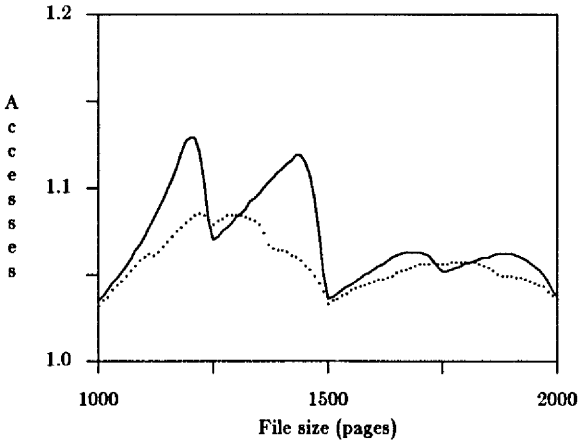

Fig. 4: Successful search

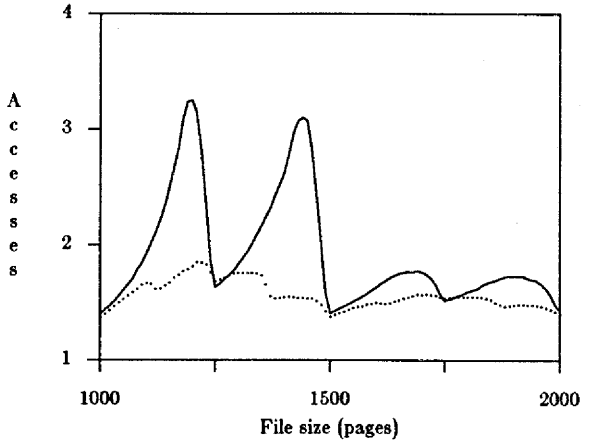

Fig. 5: Unsuccessful search

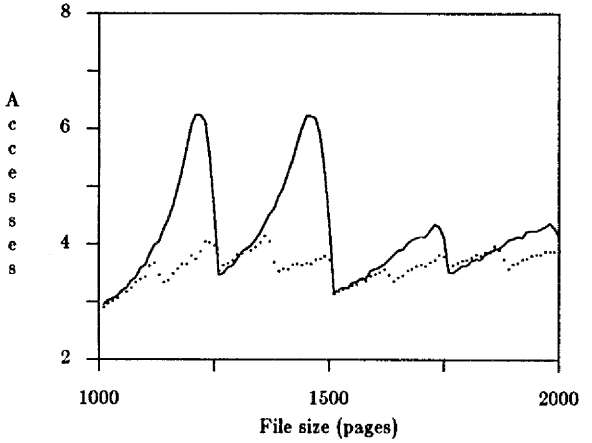

Fig. 6: Insertion costs (including expansion costs)

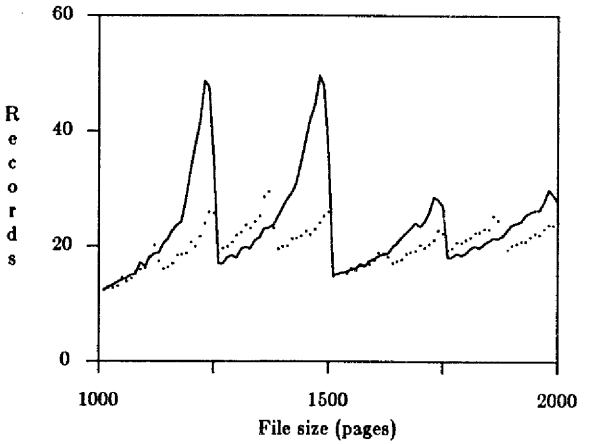

Fig. 7: Size of the record pool

Notice that the insertion costs in Fig. 6 include both the accesses required to insert a record and the accesses required for file expansion. The size of the record pool during an expansion (increasing the file by one page) equals the maximum number of records kept in the pool at any time during the expansion. The results plotted in Fig. 7 are then the averages over a number of expansions.

The full expansion from 1000 to 2000 pages depicted in Figs. 4 to 7 consists of two partial expansions: the first one from 1000 to 1500 pages and the second one from 1500 to 2000 pages. Both retrieval costs and insertion costs are lower during the second partial expansion. Each "hump" in the graphs represents one sweep. A local maximum occurs close to the end of each sweep. Increasing the number of sweeps, in this case from 2 to 4, clearly improves the performance. It reduces the overall average and, perhaps more importantly, it also reduces the variations.

Table 1 shows how increasing the number of sweeps improves the performance. The figures are averages over a full expansion. As seen from the table the performance first improves and then slowly deteriorates again. Around 5 sweeps per partial expansion appears to be optimal.

Sweeps	Successf. search	Unsuccessf. search	Insertion (total)	Size of record pool
1	1.48	9.66	16.43	91.6
2	1.07	1.92	4.19	23.6
3	1.06	1.65	3.77	21.1
4	1.06	1.59	3.67	20.7
5	1.06	1.59	3.67	20.7
6	1.06	1.61	3.69	21.0
8	1.07	1.66	3.77	21.7
10	1.07	1.70	3.82	22.1

$b = 20$, $\alpha = 0.8$, $n_0 = 2$

Table 1: Average performance as a function of the number of sweeps.

Table 2 shows the (overall) average performance for a few combinations of page size, storage utilization and number of partial expansions. The number of sweeps is 5 in all cases. Increasing the page size clearly improves the performance, especially if the required storage utilization is high. This effect is typical for linear probing. Unless very large pages are used, a target storage utilization of 0.9 seems too high. Retrieval costs are still reasonable, but insertion costs are rather high. A storage utilization of at most 0.8, or perhaps 0.85 for large pages, is a more realistic goal. Up to this level the retrieval performance of the current method is comparable to that of other versions of linear hashing [LA82a, LA83, MU81, RL82]. A comparison of insertion costs for different versions is more difficult, because the results reported in [LA82a, LA83, MU81, RL82] omitted some of the costs involved.

n_0	α	Successful search			Unsuccessful search		
		$b=10$	$b=20$	$b=40$	$b=10$	$b=20$	$b=40$
2	0.7	1.06	1.02	1.01	1.40	1.17	1.07
	0.8	1.14	1.06	1.03	2.22	1.60	1.32
	0.9	1.51	1.25	1.13	9.93	5.49	3.42
3	0.7	1.05	1.01	1.00	1.36	1.13	1.03
	0.8	1.12	1.05	1.02	2.10	1.49	1.22
	0.9	1.40	1.18	1.08	6.81	3.85	2.38

n_0	α	Insertion cost (total)			Size of record pool		
		$b=10$	$b=20$	$b=40$	$b=10$	$b=20$	$b=40$
2	0.7	4.30	2.94	2.41	8.7	14.3	25.9
	0.8	6.13	3.67	2.77	14.6	20.7	34.8
	0.9	22.0	9.87	5.59	53.2	55.2	70.2
3	0.7	4.90	3.12	2.44	9.3	14.7	26.1
	0.8	6.89	3.84	2.75	15.5	21.2	34.7
	0.9	18.9	8.20	4.50	45.3	46.4	59.0

$s = 5$ Buffer space: 1 page

Table 2: Average performance over a full expansion

In the current method it is advantageous to read or write several consecutive pages in one access, provided that the necessary buffer space is available. This is a consequence of using linear probing. Doing so is particularly attractive during expansions when several consecutive pages must be scanned and reorganized. Table 3 shows how the insertion costs are affected by using more buffer space during expansions. The figures are averages over a full expansion. The improvement is considerable; using 3 buffer pages instead of one cuts the expansion costs by half.

The results in Table 3 are based on the assumption that additional buffer space is used only during expansions. If additional buffer space is used during insertions as well, the costs can be reduced even further. For the example file the following results are obtained: using 2 buffer pages reduces the total insertion costs to approximately 3.4 accesses and 4 pages results in approximately 2.9 accesses.

Buffer pages	Insertion	Expansion	Total
1	2.97	1.21	4.19
2	2.95	0.76	3.70
3	3.00	0.60	3.60
4	2.97	0.53	3.50
5	2.96	0.50	3.45

$b = 20 \quad s = 2$

$\alpha = 0.8 \quad n_0 = 2$

Table 3: Effects of using additional buffer space during expansions.

6. Conclusions

A new method for handling overflow records in connection with linear hashing has been presented and its performance analysed by means of sumulation. The method is based on linear probing and does not require a dedicted overflow area nor chaining. The expansion sequence of linear hashing was modified to avoid creating large clusters of full pages. This significantly improved the overall performance.

The algorithms needed to implement the basic file operations are quite simple. An address computation algorithm and an algorithm for expanding the file by one page were given. Retrieval, insertion and deletion can be done in the same way as for traditional linear probing.

The overall performance of the new method is competitive with that of other variants of linear hashing, up to a load factor of 0.80-0.85. One of the advantages of using linear probing is that all the basic file operations can be speeded up simply by using more buffer space, reading or writing several pages in one access. Linear probing also preserves locality of reference, thus avoiding long seeks to some extent.

The reason for changing the expansion sequence of linear hashing was to rapidly spread out pages with a low load over the file. This prevents long clusters

of full pages from forming. The expansion sequence proposed is one possible implementation of this idea. It has the advantage of being simple, but some other sequence may very well achieve a better overall performance. However, the margin for performance improvements is rather narrow.

References

[GL82] Gonnet, G.H. and Larson, P.-Å.: External hashing with limited internal storage, Technical Report CS-82-38, University of Waterloo, 1982.

[LA80] Larson, P.-Å.: Linear hashing with partial expansions, In Proc. 6th Conf.. Very Large Data Bases (Montreal, Canada), ACM, New York, 1980, pp. 224-232.

[LA82a] Larson, P.-Å.: Performance analysis of linear hashing with partial expansions, ACM Trans. Database Systems, 7, 4 (1982), 566-587.

[LA82b] Larson, P.-Å.: A single-file version of linear hashing with partial expansions, In Proc. 8th Conf. Very Large Data Bases (Mexico City, Mexico), VLDB Endowment, California, 1982, pp. 300-309.

[LA83] Larson, P.-Å.: Performance analysis of a single-file version of linear hashing, The Computer Journal (to appear).

[LK83] Larson, P.-Å. and Kajla, A.: File organization: Implementation of a method guaranteeing retrieval in one access, Comm. of the ACM (to appear).

[LI80] Litwin, W.: Linear hashing: A new tool for file and table addressing, In Proc. 6th Conf. Very Large Data Bases (Montreal, Canada), 1980, pp. 212-223.

[MU81] Mullin, J.K.: Tightly controlled linear hashing without separate overflow storage, BIT, 21, 4 (1981), 389-400.

[RL82] Ramamohanarao, K. and Lloyd, J.K.: Dynamic hashing schemes, The Computer J. 25, 4 (1981), 478-485.

[RSD83] Ramamohanarao, K. and Sacks-Davis, R.: Recursive linear hashing, Technical Report 83/1, Dept. of Computing, Royal Melbourne Institute of Technology, Melbourne, Australia, 1983.