Robin Hood Hashing

Pedro Celis

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

ABSTRACT

This thesis deals with hash tables in which conflicts are resolved by open addressing. The initial contribution is a very simple insertion procedure which, in comparison to the standard algorithm, has the effect of dramatically reducing the variance of the number of probes required for a search. This leads to a new search algorithm which requires less than 2.6 probes on average to perform a successful search even when the table is nearly full. Unsuccessful searches require only $O(\ln n)$ probes. Finally, an extension to these methods yields a new, simple way of performing deletions and subsequent insertions. Experimental results strongly indicate little degeneration in performance for both successful and unsuccessful searches.

Acknowledgements

I am deeply grateful to my supervisor, Professor J. Ian Munro, for introducing me to the areas of analysis of algorithms and computational complexity and for his encouragement and friendship throughout the course of my graduate studies. Very special thanks are also due to Professor Per-Åke Larson, also my supervisor, for his guidance, constant support, and friendship. They both suggested the thesis topic, spent a great deal of time with me, and read and reread all my work. Their ideas are present throughout this thesis.

Thanks are also due to the other members of the examining committee, Professors Gaston H. Gonnet, Ian P. Goulden, Jeffrey S. Vitter, and Derick Wood for their helpful criticism. I also thank the Natural Sciences and Engineering Research Council of Canada for its financial support.

On a personal level, my deepest gratitude is to my wife Laura Eugenia, for her love and continuing support, without which my graduate studies could not have been accomplished. We would like to thank our families in Mexico for their constant encouragement, and our daughter Laura Elisa and our many friends in Canada for helping make our stay here a wonderful experience.

Contents

List of Tables

List of Figures

Chapter 1

Overview of Hashing

One of the most natural and indeed important tasks in programming is the implementation of a data structure servicing the operations of insert, delete and find (also called member). Such a structure is often called a dictionary. A hash table provides a convenient way to implement a dictionary.

Ideally, the purpose of a hashing scheme is to be able to determine solely from the identification of a record, called the record key, the exact location where the record is stored. Given a record to be inserted or located, a *key to address transformation* is performed using a *hash function* $h(k) : \mathcal{K} \mapsto \{0, \ldots, n-1\}$ which takes as an argument a key k in the specified universe $\mathcal{K}$ and returns an integer $h(k)$ between 0 and $n-1$, where n is the size of the table. The record is then inserted in the table entry specified by $h(k)$. This causes no problems until a record with key k' has to be inserted and location $h(k')$ is already occupied. In this case we say a *collision* has occurred. Handling collisions is the central issue in hashing and the subject of this thesis.

1.1 Collision Resolution

Collisions are almost certain to occur even if the table is sparsely populated. The famous "birthday paradox" (see for example [FEL68]) asserts that among 23 or more people the probability that at least 2 of them share the same birthday exceeds 1/2. In other words, if we select a random function that maps 23 records into a table of size 365, the probability that no two keys map into the same location is only 0.4927. In general, a hash table of size n is likely (probability $> \frac{1}{2}$) to have at least one collision by the time it contains about $\sqrt{\pi n}$ elements.

There are two popular ways of handling collisions: *chaining* and *open addressing*. The idea of chaining is to keep, for each location, a linked list of the records that *hash* to that location. This implies that each entry in the table must have enough space to contain a record and a link field. There are a number

of interesting tradeoffs and techniques in connection with chaining. Our interest, however, lies in an approach which calls for no additional storage, namely open addressing.

1.2 Open Addressing

Open addressing seems to have first appeared in the literature in [PET57]. The basic idea is to do away with the links entirely and to insert by probing the table in a systematic way. When a collision occurs, one of the colliding records is selected to keep the table location, while the other one continues probing until inserted. The sequence of table entries to be inspected when inserting or searching for a record is called the *probe sequence*. We can augment the hash function with another parameter, the *probe position* or try number, and use it to generate the probe sequence for a record. Thus the hash function becomes $h(k,i)$: $K \times \{1,\ldots,\infty\} \mapsto \{0,\ldots,n-1\}$.

The simplest open addressing hashing scheme, known as *linear probing*, uses the hash function $h(k,i) = (h_1(k) + i - 1) \bmod n$, where $h_1(k)$ is an auxiliary hash function. Another open addressing method, called *double hashing*, uses two independent auxiliary hash functions $h_1(k)$ and $h_2(k)$ to compute $h(k,i) = (h_1(k) + (i-1) * h_2(k)) \bmod n$, where $h_2(k)$ is prime relative to n. Double hashing performs much better than linear probing for high load factors because it reduces the probability of two colliding records having the same remaining probe sequence.

Two other open addressing schemes frequently mentioned in the literature and used as models for analysis are *uniform hashing*, where the hash function provides a random permutation of the numbers $\{0,\ldots,n-1\}$; and *random probing*, where $h(k,i)$ is simply a number chosen at random from $\{0,\ldots,n-1\}$. The difference between these two schemes is that random probing is memoryless, meaning that a location may be probed several times before some other location is probed for the first time. Uniform hashing is conjectured to be optimal in the sense that it minimizes the expected number of collisions during the insertion process [ULL72]. The conjecture has been proved for the asymptotic case [YAO85]. Random probing is simpler to analyse and asymptotically has the same performance as uniform hashing for nonfull tables under most conflict resolution schemes. Random probing and uniform hashing are not usually implemented, since empirical evidence shows that their performance is close to that of double hashing which is less costly to implement. Their interest lies in the fact that they are simpler to analyze and approximate closely the performance of double hashing.

Depending on the ratio of the link field size to the record size, open addressing can yield a better performance than chaining if the space allocated to the links and overflow records is used to increase the table size[1]. The *stan-*

[1]But see [KNU73] sec. 6.4 exercise 13 p. 543 for a way of reducing this and [FELLOW73]

dard search algorithm is to probe locations $h(k,i), i = 1,2,\ldots$ in order until the record is found. The *standard insertion algorithm* is to probe locations $h(k,i), i = 1,2,\ldots$ until an empty location is found. The new record is placed in that location. Figures 1.1 and 1.2 on page 4 show these algorithms. Initially `m = longestprobe = 0`, where `m` is the number of records in the table and `longestprobe` is the longest probe sequence length used by any one of the records stored in the table. The table is filled initially by records having a special key value `empty`. The problem of deletions is address in Chapter 6.

1.3 Comparison of Reordering Schemes

This section reviews several more sophisticated schemes for creating a hash table. The key notion is that records already in the table may be moved as a new one is inserted. Such an insertion algorithm we call a *reordering scheme.* A reordering scheme can be used with any open addressing hashing method but the performance measures presented below are for either random probing or uniform hashing. As we have already noted, the performance of random probing is similar to that of uniform hashing and double hashing.

When comparing hashing schemes, we are interested both in the cost of loading the table and in the "quality" of the table produced, that is, both the efficiency and the efficacy of the insertion technique. We will characterize the quality of the hash table by the behavior of the following random variables: the probe sequence length for a key (psl), which is equivalent to the probe position where the key was placed; the longest probe sequence length (lpsl); the unsuccessful probe sequence length (upsl); and the longest unsuccessful probe sequence length (lupsl). We will compare the expected value (denoted by E[•]), and sometimes the variance (denoted by V[•]), of these random variables for both the case of full and nonfull tables. For nonfull tables these expressions are functions of α, the load factor, defined as $\alpha = m/n$, where m is the number of records in the table and n is its size. Several analyses of hashing schemes, including the one we derive here, have been performed for infinite nonfull tables with load factor α, where $\alpha \leq 1 - \epsilon$, $\epsilon > 0$. Throughout the thesis we refer to this tables as α-full tables.

A very important but often neglected performance measure of a hash table is the longest probe sequence length (lpsl), since it provides a bound on both successful and unsuccessful searches. This value can be used to limit the cost of unsuccessful searches in any open addressing hashing scheme, as was done in Figure 1.2. This elegant but sadly underutilized idea is due to Lyon [LYO78].

For standard uniform hashing with no reordering the following equations can be established [PET57,GON81,GON84]:
for a nonfull table

---

for a discussion.

```
table : array [1..n] of RECORD   { all empty }
n {table size}, m {records inserted}, longestprobe {intially 0} : integer

function insert(Record)
      if m=n then return(FAIL)      { table full }
      k := Key(Record)
      probeposition := 1
      location := H(k, probeposition)
      while table[location] <> empty do
            probeposition := probeposition + 1
            location := H(k, probeposition)
      endwhile
      longestprobe := max( longestprobe, probeposition )
      table[location] := Record
      m=m+1
      return(location)
end function insert
```

Figure 1.1: Standard insertion algorithm

```
function search(k)
      probeposition := 1
      location := H(k, probeposition)
      while probeposition <= longestprobe and table[location] <> empty do
            if key(table[location]) = k then return( location )
            probeposition := probeposition + 1
            location := H(k, probeposition)
      end while
      return(FAIL)      { unsuccessful search }
end function search
```

Figure 1.2: Standard search algorithm

$$\mathrm{E[psl]} = \frac{n+1}{m}\left[H_{n+1} - H_{n-m+1}\right] \approx -\alpha^{-1}\ln(1-\alpha)$$

$$\mathrm{V[psl]} \approx \frac{2}{1-\alpha} + \alpha^{-1}\ln(1-\alpha) - \alpha^{-2}\ln^2(1-\alpha)$$

$$\mathrm{E[lpsl]} = -\log_\alpha n - \log_\alpha(-\log_\alpha n) + O(1)$$

and if we use Lyon's trick

$$\mathrm{E[upsl]} < -\log_\alpha n - \log_\alpha(-\log_\alpha n) + O(1)$$

and for a full table

$$\mathrm{E[psl]} = \ln n + \gamma - 1 + o(1)$$

$$\mathrm{E[lpsl]} = 0.6315... \times n + O(1)$$

$$\mathrm{E[upsl]} = 0.6315... \times n + O(1)$$

where $\gamma = 0.5772156649...$ is Euler's constant.

The total number of probes needed to load a table for this method is simply $m\mathrm{E[psl]}$, which for full tables is equal to $n \ln n + O(n)$.

1.3.1 Ordered Hash Tables

One of the first reordering schemes proposed in the literature was *ordered hash tables* by Amble & Knuth [AMBKNU74]. The scheme was not intended to improve the retrieval of records, but rather to improve the processing of unsuccessful searches. The idea is very simple. Whenever two records collide, the one with the smaller key is stored in the disputed location. When searching in an ordered hash table, the search is unsuccessful whenever a probed location contains a key larger than the search key. As noted, the distribution of the probe position of the keys in the table remains unchanged; only the expected cost of unsuccessful searches is improved, but not its expected worst case. For uniform hashing and a nonfull table and using Lyon's trick we have

$$\mathrm{E[upsl]} = -\alpha^{-1} \ln(1-\alpha) + O\left(\frac{1}{n-m}\right)$$

$$\mathrm{E[lupsl]} < -\log_a n - \log_a(-\log_a n) + O(1)$$

and for a full table

$$\mathrm{E[upsl]} = \ln n + \gamma - 1 + O\left(\frac{1}{n}\right)$$

$$\mathrm{E[lupsl]} = 0.6315... \times n + O(1)$$

The total number of probes required to load a full ordered hash table is the same as for the standard algorithm, namely $n \ln n + n\gamma - n + O(1)$.

1.3.2 Brent's Method

Brent [BRE73] was the first to propose moving stored records to reduce the expected value of the probe sequence length. During an insertion, a sequence of occupied table entries is probed until an empty location is found. Brent's scheme checks whether any of the records in these occupied locations can be displaced to an empty location further in their probe sequence, at a smaller

cost, and the minimum is taken. Figure 1.3 shows graphically how one such insertion of a record R might occur. In the example, instead of inserting the record R in its fifth choice and increasing the total table cost by 5, record R_3 is displaced to its next choice, and then R is placed in its third choice, the place formerly occupied by R_3. The increase in the total table cost is thus reduced from 5 to 4.

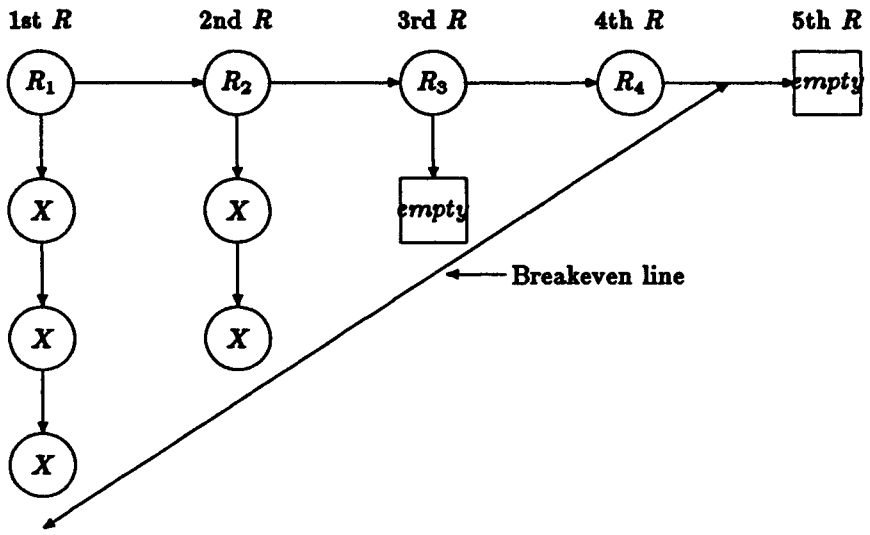

Figure 1.3: Sample insertion in Brent's method

The tables produced by Brent's scheme have a very good E[psl], even when completely filled. For random probing and α-full tables the expected values are

$$\mathrm{E[psl]} = 1 + \frac{\alpha}{2} + \frac{\alpha^3}{4} + \frac{\alpha^4}{15} - \frac{\alpha^5}{18} + \frac{2\alpha^6}{15} + \frac{9\alpha^7}{80} - \frac{293\alpha^8}{5670} - \frac{319\alpha^9}{5600} + \cdots$$

and for full tables

$$\mathrm{E[psl]} \approx 2.4941...$$

So, if the standard search algorithm is used, a record can be retrieved in less than 2.5 probes, on average, regardless of the table size. There has been no successful analysis of the expected values of lpsl and upsl, but it is conjectured [GONMUN79] that they are $\Theta(\sqrt{n})$ for full tables, and this was supported by simulations. Simulations presented in this thesis, support the conjecture that $\Theta(n \ln n)$ time is needed to fill a table.

Brent's method does not require any extra memory to process an insert operation. If the search for an empty location is done on a depth first basis, as suggested in [BRE73], the expected number of times the $h_2(\bullet)$ hash function will

be computed is $\alpha^2+\alpha^5+\alpha^6/3+\cdots$ eventually approaching $\Theta(\sqrt{n})$ for full tables [KNU73]. The disadvantage of searching in this manner is that a number of locations below the breakeven line will be probed. For example the fifth probe position of the record R to be inserted would be probed unnecessarily. The number of additional table positions probed during an insertion is approximately $\alpha^2+\alpha^4+\frac{4}{3}\alpha^5+\alpha^6+\cdots$ [KNU73].

Another way of searching for the closest empty location is to do a level search as recommended in [GON84]. For double hashing, this would imply calling the $h_2(\bullet)$ hash function during an insertion up to $\Theta(n)$ times instead of $\Theta(\sqrt{n})$, which may be preferable to probing the extra locations. A disadvantage of Brent's method is that duplicate keys are not detected by the insertion algorithm, so if duplicate insertion requests may occur, an unsuccessful search should precede each insertion. To keep track of the value of `longestprobe`, the probe position of the stored record that will be displaced must be determined to establish if its new probe position will be the new maximum. The probe position of a stored record can be determined by searching for the location where the record is stored in its probe sequence.

1.3.3 Binary Tree Hashing

Binary tree hashing is the natural generalization of Brent's method. Not only is the record being inserted allowed to displace other records in its probe sequence, but these displaced records may further displace other records in their probe sequences. This is illustrated graphically by Figure 1.4. This method was discovered independently by Mallach [MAL77] and by Gonnet and Munro [GONMUN79].

Since this method is a generalization of Brent's, it is expected to produce better tables at a somewhat higher cost. An approximate model [GONMUN79] yields the following for random probing and α-full tables[2]:

$$\mathrm{E[psl]} = 1+\frac{\alpha}{2}+\frac{\alpha^3}{4}+\frac{\alpha^4}{15}-\frac{\alpha^5}{18}+\frac{2\alpha^6}{105}+\frac{83\alpha^7}{720}+\frac{613\alpha^8}{5760}-\frac{69\alpha^9}{1120}+\cdots$$

and for full tables:

$$\mathrm{E[psl]} \approx 2.13414...$$

These results have been validated using simulation. There is no analysis for the expected values of lpsl and upsl, but Gonnet and Munro, based on simulation results, conjectured them to be about $\lg n+1 \approx 1.44\ln n+1$ for full tables. As with Brent's scheme, the expected cost of loading a full table has not been successfully analysed.

---

[2]The formula for E[psl] for α-full tables is taken from [GON84] and differs slightly from the one in [GONMUN79].

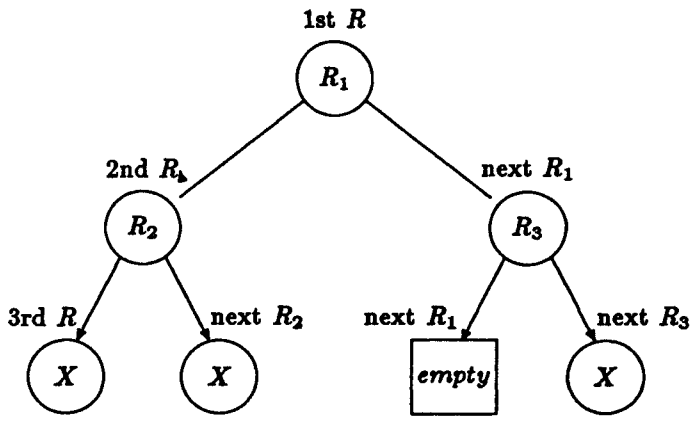

Figure 1.4: Sample insertion using Binary Tree Hashing

The natural order for inspecting the table when searching for an empty location is by levels, as suggested in [MAL77] and [MAD80][3]. However, the amount of memory required to store the tree of locations probed is large, as Mallach noted. Limited simulations done by the author of this thesis indicate that the number of probes into the table appears to be about $.5n^{3/2} \ln n$ and the amount of memory required to store the tree about $.15n^{3/2} \ln n$. What is worse, the variance of these two measures is very high, so the probability of requiring, say, n^2 extra memory locations is significant.

Gonnet and Munro [GONMUN79] show how to use an algorithm for the transportation problem, presented in [EDMKAR72], to insert keys into the table. This algorithm has a worst case runtime of $O(n^2 \log n)$ [FRETAR84], but simulations done by the author of this thesis suggest that the expected cost of loading a full table is $\Theta(n^{3/2} \ln n)$, and the amount of extra memory required is $\Theta(n)$, both with a small variance. While this is a great improvement over the natural algorithm, it is still expensive both in time and memory.

As with Brent's method, some additional probes are required to avoid duplicate keys and to keep track of the value of `longestprobe`.

1.3.4 Optimal and Min-Max Hashing

Both Brent's method and binary tree hashing only move stored keys forward in their probe sequences. Arbitrary rearrangements of the stored records must be

[3]The reader is warned that the analysis, conclusions and comments on Mallach's work presented by Madison [MAD80] are incorrect.

allowed to obtain an optimal hash table [POO76,RIV78,GONMUN79]. Poonan was the first to note that the optimal placement of keys in a hash table is a special case of the assignment problem [KÖN31], which can be solved in $O(n^2 \log n)$ time in the worst case [FRETAR84]. Neither the expected cost of finding the optimal hash table nor the expected values of psl, lpsl, and upsl have been determined. For E[psl] and full tables the following bounds exist [GON77,GONMUN79]:

$$1.688382 < \text{E[psl]} < 2.13414...$$

Simulation results indicate that E[psl] $\approx$ 1.82.... If, instead of minimizing the expected value of psl, we minimize the expected value of lpsl (called *min-max* hashing), the bounds become [GON81,RIV78]

$$\ln n + \gamma + o(1) \leq \text{E[lpsl]}$$

$$\text{lpsl} < 4 \lg n \approx 5.77 \ln n \qquad \text{with probability } 1 - \epsilon$$

where ϵ is an arbitrarily small constant and the inequality holds for all $n > n_0(\epsilon)$. This last inequality is not sufficient to prove that E[lpsl] $= O(\ln n)$. As a corollary of a result obtained in Chapter 2, it can be shown that E[lpsl] $= \Theta(\ln n)$ for full min-max hash tables. Both optimal and min-max hash tables, in addition to being expensive to build, require $\Theta(n)$ extra memory during the insertion of a record.

1.4 Summary

In summary, we can say that binary tree, optimal and min-max hashing reduce the expected values of psl and upsl dramatically, but at a very high cost for table creation. The expected number of operations to construct a table using one of these algorithms is high compared to the standard hashing scheme, and also require a nontrivial amount of extra memory. These methods are best suited for applications in which the set of keys is static and known in advance. In that case, the cost of constructing the table can be amortized over a large number of search operations, and the additional memory space required can be released as soon as the table has been created.

Brent's method is perhaps the one that offers the best overall tradeoff. It is simple to program, the time needed for loading a full table seems to be $\Theta(n \ln n)$, it requires no additional memory and has an expected successful search time which is constant. However duplicate keys are not automatically detected during the insertion process, that is, an unsuccessful search is a useful prelude to an insertion. Furthermore, some additional probes into the probe sequence of a stored record to be moved are required to determine if its new probe position is the new maximum. But the greatest disadvantage of Brent's method is the

expected value of lpsl and hence of unsuccessful searches; this is $\Omega(\sqrt{n})$, much worse than the $\Theta(\ln n)$ achieved by binary tree, optimal, and min-max hashing.

Ordered hash tables do not improve the expected value of psl nor lpsl at all. The method improves the expected cost of unsuccessful searches but not its expected worst case. The loading cost is almost the same as for the standard algorithm. If unsuccessful searches are expected to be frequent it should be preferred over the standard algorithm. This method and the standard algorithm are the only two that allow detection of duplicate keys with no additional probes into the table.

As for the standard insertion algorithm, we can say that it is simple to program and efficient to implement, and that the tables it produces have an acceptable expected value of psl for moderate load factors. The main problem with this scheme is that, when the table is full or nearly full, it will take $\Theta(n)$ steps to retrieve some keys and to perform unsuccessful searches. The solution that has been suggested is

> *"A hash table should never be allowed to get that full. When the load factor is about* 0.7 *or* 0.8, *the size of the hash table should be increased, and the records in the table should be rehashed."* [MAU75]

This statement is sometimes interpreted as *"Do not use hashing for real time applications"*. We will see that this is no longer the case.

What is needed is a hashing scheme that combines the best features of each of the methods presented. Such a scheme should:

- Be as simple to program as the standard algorithm,
- Take only $\Theta(n \ln n)$ probes on the average to load a full table,
- Require no additional memory for insertions,
- Perform successful searches in a small number of probes on the average, even if the table is nearly full,
- Have a $\Theta(\ln n)$ expected value of lpsl and upsl.

And of less importance:

- Need no additional probes to detect duplicate entries
- Need no additional probes to keep track of the value of `longestprobe`

The method introduced and analyzed in this thesis, called *Robin Hood hashing*, has all of these characteristics.

1.5 Thesis Outline

In this chapter we have reviewed several reordering schemes that produce very good hash tables; however, the better ones are expensive, in time and memory, to implement. Each of these schemes involves a new insertion algorithm but retains the standard search algorithm. In Chapter 2 we modify the standard insertion algorithm to obtain a new method that we call Robin Hood hashing; we also define a family of hashing schemes that have the same expected value of psl and prove that the expected value of lpsl is $O(\ln m)$ for Robin Hood hashing. In Chapter 3 we study the probability distribution of the random variable psl and its moments for α-full tables. In Chapter 4 we present some modifications to the standard search algorithm to be used on a Robin Hood hash table so as to achieve a small constant expected search time for α-full tables, and loading of a table using $\Theta(n \ln n)$ probes. In Chapter 5 we present the results of simulating several of the methods presented here and Robin Hood hashing. In Chapter 6 we discuss how deletions can be performed in hash tables with open addressing and we present an algorithm to be used with Robin Hood hashing. Simulations results for this new algorithm are also presented. Finally in Chapter 7 we present our conclusions and suggestions for further research.

Chapter 2

The Robin Hood Heuristic

In this chapter we present a new algorithm to insert keys into a hash table. We call this algorithm Robin Hood hashing. We then define a family of reorganisation schemes that share the same expected probe sequence length (E[psl]). Finally we prove that the expected longest probe sequence length (E[lpsl]) is $O(\ln m)$ for a Robin Hood hash table with m records. Searching and the cost of loading are discussed in Chapter 4 and deletions in Chapter 6.

2.1 The Robin Hood Approach to Hashing

In the standard hashing algorithm, as the table fills, the problem may not be so much the average search time, but its expected worst case. An expected value for psl of $\ln n + O(1)$ may be acceptable, but an expected value for lpsl of $\Theta(n)$ is certainly not. In other words, the problem is not so much the mean of psl, but its high variance.

As an attempt to reduce the variance of the psl, consider the following modification to the standard open addressing hashing algorithm: when inserting, if a record probes a location that is already occupied, the record that has traveled longer in its probe sequence keeps the location, and the other one continues on its probe sequence.

Notice that, in deciding which record keeps the location, we are not using any knowledge of the remaining probe sequence of the colliding records, nor any other information regarding the rest of the hash table. This implies that, under random probing, the expected number of additional probes into the hash table required to find an empty location is the same for both records. Therefore the expected value of psl is not affected by this modification to the standard algorithm but its distribution and hence the distribution of the lpsl will change.

Consider a collision of two records R_1 and R_2, and assume that their probe positions are i and j, respectively, with $i < j$. One of the records must be moved

forward in its probe sequence, and we would like to make the decision without any knowledge about the future probe sequences of the records. Which record should be rejected to keep the increase in the variance at a minimum? We have the following three cases:

-1) $i < j <$ E[psl]: moving record R_1 will cause the variance to decrease by a larger amount than moving record R_2.

-2) $i <$ E[psl] $< j$: moving record R_1 will decrease the variance, moving record R_2 would increase it.

-3) E[psl] $< i < j$: moving record R_1 will cause the variance to increase by a smaller amount than moving record R_2.

In all three cases moving the record that has probed the smallest number of locations is the best option.

We call this new scheme *Robin Hood hashing*, since the principle of taking from the rich and giving to the poor is followed. This does not modify the average wealth (psl) per individual (record) but changes its distribution by reducing the imbalance.

Under the standard scheme, a record inserted when the load factor is high is almost guaranteed to probe many locations, but one inserted at a low load factor will usually stay in its first probe position. Robin Hood hashing has a nice time-independence property: the distribution of the psl for a given record is not affected by its position in the insertion sequence. The rules for breaking ties do not affect the distribution of psl, but if ties are always broken in the same direction (say the record with the lower key value keeps the location), exactly the same final table arrangement is obtained for every permutation of the insertion sequence.

Figure 2.1 on page 14 shows the insertion algorithm of Robin Hood hashing. `findposition` is a function that determines the current probe position of a stored record. It returns a zero value if the location is empty. In Chapter 4 we discuss ways of implementing this function; for the moment assume that the function does a standard search for the record counting the number of probes. The use of `totalcost` will also be discussed in Chapter 4.

The standard search algorithm (Fig. 1.2) can be used on a Robin Hood hash table, but better search algorithms are described in Chapter 4.

2.2 A Family of Random Probing Schemes

Consider the family of hashing schemes in which the decision as to which one of the colliding records stays in the location is not based on any knowledge about their future probe sequence. Both the standard hashing algorithm and Robin Hood hashing belong to this family. Other members of this family are ordered hash tables, and signature hashing with variable length separators [GONLAR82].

```
n : integer   {table size}
table : array [1..n] of RECORD   {initially all = empty}
m, totalcost, longestprobe : integer   {intially = 0}

function insert(Record)
      if m=n then return(FAIL)      { table full }
      k := Key(Record)
      probeposition := 0
      while k <> empty do
            probeposition := probeposition + 1
            location := H(k, probeposition)
            totalcost := totalcost + 1
            recordposition := findposition(location)
            if probeposition > recordposition then begin
                  tempRecord := table[location]
                  table[location] := Record
                  Record := tempRecord
                  k := key(Record)
                  longestprobe := max( longestprobe, probeposition )
                  probeposition := recordposition
            end
      endwhile
      totalcost := totalcost + 1
      longestprobe := max( longestprobe, probeposition )
      m := m+1
      return(location)
end function insert
```

Figure 2.1: Robin Hood insertion algorithm

The insertion algorithm for all of the hashing schemes belonging to this family is that of Figure 2.1 except for the condition `probeposition > recordposition` inside the if statement. In the standard algorithm this condition is changed to `recordposition=0` (meaning the location is empty); in ordered hash tables to `k < Key(table[location])` (the key of an empty location is larger than any other); and in signature hashing with variable length separators to `Signature( k, probeposition) < Signature( Key(table[location]), recordposition)` (the signature of an empty location is higher than all other signatures). Any other condition could be used here. The only restriction is that the decision must be made without looking at the remaining probe sequence of the colliding records. Brent's method, binary tree hashing, optimal hashing, and min-max hashing do not satisfy this restriction, and hence are not members of this family.

Lemma 2.1 *Under random probing, and for any hashing scheme in which the decision as to which of the colliding records may stay in the location is not based on any knowledge about their future probe sequence, the expected number of probes required to insert m records into a table of size n is*

$$n[H_n - H_{n-m}]$$

where $H_i = \sum_{r=1}^{i} \frac{1}{r}$ is the i-th harmonic number. The variance of the total number of probes required to insert m records is

$$n^2\left[H_n^{(2)} - H_{n-m}^{(2)}\right] - n[H_n - H_{n-m}]$$

where $H_i^{(2)} = \sum_{r=1}^{i} \frac{1}{r^2}$.

Proof: Since no knowledge about the future probe sequence of the colliding records is used, then the probability that the next location probed is empty is equal to $1 - \frac{i-1}{n}$ regardless of which of the records was rejected. The number of probes needed to perform the i-th insertion is then a geometrically distributed random variable with parameter (probability of success) $1 - \frac{i-1}{n}$, and is independent of the number of probes required for previous insertions. The expected value and variance of the number of probes needed to perform the i-th insertion are $\frac{n}{n-i+1}$ and $\frac{n(i-1)}{(n-i+1)^2}$ respectively. The expected number of probes needed to perform m insertions is simply

$$\begin{aligned}\sum_{r=1}^{m} \frac{n}{n-r+1} &= n \sum_{l=n-m+1}^{n} \frac{1}{l} \\ &= n\left[\sum_{l=1}^{n} \frac{1}{l} - \sum_{l=1}^{n-m} \frac{1}{l}\right] \\ &= n[H_n - H_{n-m}]\end{aligned}$$

and the variance is

$$\begin{aligned}\sum_{r=1}^{m} \frac{n(r-1)}{(n-r+1)^2} &= n \sum_{l=n-m+1}^{n} \frac{n-l}{l^2} \\ &= n^2\left[\sum_{l=1}^{n} \frac{1}{l^2} - \sum_{l=1}^{n-m} \frac{1}{l^2}\right] - n\left[\sum_{l=1}^{n} \frac{1}{l} - \sum_{l=1}^{n-m} \frac{1}{l}\right] \\ &= n^2\left[H_n^{(2)} - H_{n-m}^{(2)}\right] - n[H_n - H_{n-m}] \quad \square\end{aligned}$$

These derivations are fairly simple and this could be the reason why, to the best of our knowledge, they have not been published elsewhere.

Theorem 2.1 *A record inserted using a random probing hashing scheme in which the decision as to which one of the colliding records may stay in the location is not based on any knowledge about their future probe sequences, has an expected probe sequence length of*

$$\mathrm{E}[\mathbf{psl}] = \frac{n}{m}\left[H_n - H_{n-m}\right] = -\alpha^{-1}\ln(1-\alpha) + O\left(\frac{1}{n-m}\right)$$

for a table containing $m < n$ records ($\alpha = \frac{m}{n}$) and

$$\mathrm{E}[\mathbf{psl}] = H_n = \ln n + \gamma + O\left(\frac{1}{n}\right)$$

for a full table.

Proof: The average probe position of the m records inserted is the total number of probes required to insert divided by m. □

Hashing schemes in the above family may have different distributions for the random variable **psl**, but they all have the same E[**psl**].

It is important to distinguish between the variance of the **psl** (probe sequence length) of a record and the variance of the mean **psl** of a table. The latter term is

$$\frac{n^2}{m^2}\left[H_n^{(2)} - H_{n-m}^{(2)}\right] - \frac{n}{m^2}\left[H_n - H_{n-m}\right] < \frac{\pi^2}{6} \approx 1.645$$

for all schemes in the family. The variance of **psl**, however, is in general much larger and depends on the particular scheme.

On page 3 we stated the value of E[**psl**] for the standard method using uniform hashing. For a nonfull table the effect of using uniform hashing instead of random probing is equivalent to increasing the table size by one. For full tables the value of E[**psl**] is reduced by one.

2.3 A Single Final Table

For the standard hashing algorithm, the distribution of the **psl** for a record is a function of its position in the insertion sequence. For example, the first record inserted must be in its first probe position. But if it was inserted when the load factor was α, it is in its k-th position with probability $\alpha^{k-1}(1-\alpha)$.

When using Robin Hood hashing, all records inserted have the same distribution for their **psl**, regardless of their position in the insertion sequence. We prove this by showing that the set of records in the hash table uniquely determines the table arrangement, independent of the order in which the records are inserted, provided that ties are broken in a consistent fashion. By a consistent tie breaker we mean one that always selects the same record (say the one with the smallest key) to keep the location among all the records that are tied with the same probe sequence value, regardless of the order in which the probes occurred.

Theorem 2.2 *Every permutation of the insertion sequence produces the same final Robin Hood hash table, provided that a consistent tie breaker is used.*

Proof: Suppose we have a function $tiebreak(key, probetry)$ that gives a different real value in $[0, 1)$ for each pair $(key, probetry)$. Associate a label with each record probe defined as $label(key, probetry) = probetry + tiebreak(key, probetry)$. Then each location contains the record with the highest label that probed it, regardless of the order in which these probes occurred. Therefore, it will suffice to show that each location receives the same record probes regardless of the insertion sequence.

The record probes that a location receives is a function of the first probe of all records and the set of records rejected from all locations. The set of record probes that hit a location is a transitive closure defined as follows: initially,

$$\mathcal{P}_i = \emptyset$$

Repeat until no $\mathcal{P}_i$ can change

$$\mathcal{P}_i = \mathcal{P}_i \cup \{(R, s+1) | h(R, s+1) = i, [s = 0 \text{ or } \exists l, (R', s') : (R, s) \in \mathcal{P}_l, (R', s') \in \mathcal{P}_l, label(R', s') > label(R, s)]\}$$

The second term of this expression can be read as: the $(s+1)$-st choice of R is i, and either $s = 0$ or R is rejected from its s-th choice. Then $\mathcal{P}_i$ is the set of record probes that location i receives. Each insertion sequence of the records in $\mathcal{R}$ corresponds to a particular order of computing the transitive closure. The sets $\mathcal{P}_i$ are completely defined by the set, $\mathcal{R}$, of records in the table, and the $h(\bullet)$ and $label(\bullet)$ functions, without reference to the order in which the records were inserted. Since a transitive closure is always unique, the set of record probes a location receives is independent of the insertion sequence. □

This unique final table property also applies to some of the other hashing schemes in the family defined in Section 2.2. The above proof could be applied to signature hashing by using the probe signature as the label; and to ordered hash tables, by making the key value the label. Amble & Knuth [AMBKNU74] proved that ordered hash tables always produced a single final table by showing that inserting a set of records with their algorithm was equivalent to inserting the same set in order of increasing key values using the standard insertion algorithm.

Rivest [RIV78] proved that the optimal hash table can always be obtained by using the standard insertion algorithm and a permutation of the insertion sequence. It is interesting to note that under Robin Hood hashing, the final table obtained is not necessarily one that could be obtained by using a different permutation of the insertion sequence and the standard insertion algorithm. Consider for example a table of size 3 and the insertion sequence R_1, R_2, R_3, applied to the Robin Hood insertion algorithm with a tie breaker that decides in favor of the record with the smallest subscript. Let the hash function $H(R, i)$ be as specified in the table below.

	1	2	3
R_1	1	2	3
R_2	1	3	2
R_3	3	1	2

After inserting the 3 records the hash table will be

1	2	3
R_3	R_1	R_2

where each record is in its second probe position. Since no record is in its first probe position this table cannot be obtained using the standard insertion algorithm and a different insertion sequence.

2.4 The Expected Worst Case for Robin Hood Hashing

In the next chapter we will study the distribution of the probe sequence length, psl, of Robin Hood hashing. In this section we derive bounds for the expected value of the longest probe sequence length (E[lpsl]) and prove that for full tables $\mathrm{E[lpsl]} = \Theta(\ln n)$.

Assume we have a set $\mathcal{R} = \{R_1, \ldots, R_m\}$ of m records stored in a hash table. Now define the following functions:

- Let $\sigma: \mathcal{R} \mapsto \{0, \ldots, n-1\}$ represent the table assignment, such that $\sigma(R_i)$ is the table location in which R_i is stored.
- Let $\omega: \mathcal{R} \mapsto \{1, \ldots, \infty\}$ be such that, $\omega(R_i)$ is the position in the probe sequence of R_i of the location in which it is stored.
- Let $\vartheta : \mathcal{R} \times \{0, \ldots, \infty\} \mapsto \mathcal{R}$ be the *backup function*, defined as: $\vartheta(R_i, j) = \sigma^{-1}(h(R_i, \omega(R_i) - j))$, that is, the record occupying the location that record R_i probed j steps before its current location.

Assume that the value of the longest probe sequence length (lpsl) is ℓ, that is, at least one record (say R_{worst}) is in the ℓ-th position of its probe sequence and none occur later. Consider the following intuitive argument: Let $\mathcal{W}_0 = \{R_{worst}\}$. The location R_{worst} probed in its $(\ell-1)$-st choice must contain a record in at least its $(\ell-1)$-st probe. So there are at least two records, R_{worst} and $\vartheta(R_{worst}, 1)$, in at least their $(\ell-1)$-st probes, that is, $\ell-1$ or higher. Let $\mathcal{W}_1 = \{R_{worst}, \vartheta(R_{worst}, 1)\}$. The records in $\mathcal{W}_1$ are all in at least their $(\ell-1)$-st probes. Similarly if both records are moved back we find that at least 4 records, $(R_{worst}, \vartheta(R_{worst}, 1), \vartheta(R_{worst}, 2), \vartheta(\vartheta(R_{worst}, 1), 1))$, are in at least their $(\ell-2)$-nd probes. Let $\mathcal{W}_2 = \{R_{worst}, \vartheta(R_{worst}, 1), \vartheta(R_{worst}, 2), \vartheta(\vartheta(R_{worst}, 1), 1)\}$. Care must be taken in such an intuitive argument since we

are sampling the table with replacement, so the cardinalities of the last two sets are probably but not necessarily 2 and 4.

The preceding intuitive argument can be adapted into a more precise analysis. $\mathcal{W}_i$ will denote a set of records that are in at least their $(\ell-i)$-th probe positions and $\mathcal{U}_{i+1}$, the set of records that are stored in the locations that the records in the set $\mathcal{W}_i$ would hit if moved back one further position. More formally, (note that R_{worst} is an arbitrarily chosen element in its ℓ-th probe position)

$$\mathcal{U}_0 = \{R_{worst}\}$$

$$\mathcal{W}_0 = \{R_{worst}\}$$

$$\mathcal{U}_i = \{R \mid R = \vartheta(R', j) \text{ for some } R' \in \mathcal{U}_{i-j}, 1 \leq j \leq i\}$$

$$\mathcal{W}_i = \mathcal{W}_{i-1} \cup \mathcal{U}_i$$

Each record in the set $\mathcal{W}_i$ is in at least its $(\ell - i)$-th probe position. If none of the locations that we sample when moving back a record were repeated, then the cardinality of the set $\mathcal{W}_i$ would be 2^i. Since we are sampling the locations with replacement, the cardinality of $\mathcal{W}_i$ is a random variable denoted by $\mathbf{w}_i$. We will denote by $\mathbf{u}_i$ the cardinality of the set $\mathcal{U}_i - \mathcal{W}_{i-1}$ which is the number of records that belong to the set $\mathcal{W}_i$ but do not belong to the set $\mathcal{W}_{i-1}$.

We will first find a bound for the expected value of $\mathbf{w}_i$ using occupancy distributions [DAVBAR62, JOHKOT77]. $\mathbf{w}_i$ is equal to $\mathbf{w}_{i-1} + \mathbf{u}_i$. The distribution of $\mathbf{u}_i$ is of the type called classical occupancy with specified boxes (see for example chapter 14 of [DAVBAR62]) where the number of urns is n, among which the number of specified urns is $n - \mathbf{w}_{i-1}$ and the number of balls dropped is $\mathbf{w}_{i-1}$ and we are interested in the number of specified urns that are hit. We now define a new sequence of random variables $\mathbf{v}_i$ such that $\mathrm{E}[\mathbf{v}_i] \leq \mathrm{E}[\mathbf{w}_i]$ for all i. Initially $\mathbf{w}_0 = \mathbf{v}_0 = 1$. Let $\mathbf{v}_i$ be the number of urns hit when $2\mathbf{v}_{i-1}$ balls are dropped at random. If we were to guarantee that the first $\mathbf{v}_{i-1}$ of these balls all went to different urns and the other half were dropped at random, then there would be no difference between the random variables $\mathbf{v}_i$ and $\mathbf{w}_i$. Since this is not the case, the expected value of $\mathbf{v}_i$ is less than the expected value of $\mathbf{w}_i$. The distribution of $\mathbf{v}_i$ is then of the type called classical occupancy. We therefore have

$$\mathrm{E}[\mathbf{v}_i \mid \mathbf{v}_{i-1} = v_{i-1}] = m\left(1 - \left(1 - \frac{1}{m}\right)^{2v_{i-1}}\right)$$

$$\mathrm{V}[\mathbf{v}_i \mid \mathbf{v}_{i-1} = v_{i-1}] = m(m-1)\left(1 - \frac{2}{m}\right)^{2v_{i-1}} + m\left(1 - \frac{1}{m}\right)^{2v_{i-1}} - m^2\left(1 - \frac{1}{m}\right)^{4v_{i-1}}$$

$$= m^2 \left[\left(1 - \frac{2}{m}\right)^{2v_{i-1}} - \left(1 - \frac{1}{m}\right)^{4v_{i-1}} \right]$$

$$+m \left[\left(1 - \frac{1}{m}\right)^{2v_{i-1}} - \left(1 - \frac{2}{m}\right)^{2v_{i-1}} \right]$$

which, using inequality 4.2.29[1] from [ABRSTE70], is bounded by

$$< m^2 \left(e^{-4v_{i-1}/m} - e^{-4v_{i-1}/(m-1)} \right)$$

$$+m \left[\left(1 - \frac{1}{m}\right)^{2v_{i-1}} - \left(1 - \frac{2}{m}\right)^{2v_{i-1}} \right]$$

$$= O(m)$$

To get a bound on the expected value of the longest probe sequence we will first prove the following three lemmas.

Lemma 2.2 $\mathrm{E}[\mathbf{v}_i]$ *is asymptotically equivalent to* $\mathrm{E}[\mathbf{v}_i \mid \mathbf{v}_{i-1} = \mathrm{E}[\mathbf{v}_{i-1}]]$.

Proof: We know that

$$\mathrm{E}[\mathbf{v}_i \mid \mathbf{v}_{i-1}] = m \left(1 - \left(1 - \frac{1}{m}\right)^{2v_{i-1}}\right)$$

Removing the condition we get

$$\mathrm{E}[\mathbf{v}_i] = m \left(1 - \mathrm{E}\left[\left(1 - \frac{1}{m}\right)^{2\mathbf{v}_{i-1}}\right]\right)$$

Using equation 4.2.29 again we get

$$\mathrm{E}\left[e^{-2\mathbf{v}_{i-1}/(m-1)}\right] < \mathrm{E}\left[\left(1 - \frac{1}{m}\right)^{2\mathbf{v}_{i-1}}\right] < \mathrm{E}\left[e^{-2\mathbf{v}_{i-1}/m}\right].$$

We have mentioned that $\mathbf{v}_{i-1}$ has an occupancy distribution. Rényi [RÉN62] establishes that an occupancy distribution converges to a normal distribution as $m \to \infty$ if $e^{b/m}/m \to 0$, where b is the number of balls dropped (This result is also presented on page 318 of [JOHKOT77]). The number of balls dropped in the distribution of $\mathbf{v}_{i-1}$ is $2\mathbf{v}_{i-2} < 2m$, so the condition is satisfied. Then the moment generating function of $\mathbf{v}_{i-1}$ converges to $M_{\mathbf{v}_{i-1}}(t) = \mathrm{E}[e^{t\mathbf{v}_{i-1}}] = e^{\mu t + \frac{1}{2}t^2\sigma^2}$. Using this equation with $t = -2/m$ and $t = -2/(m-1)$, we get

[1] The inequality states that $\exp\left(-\frac{x}{1-x}\right) < 1 - x < \exp(-x)$ for $x < 1$. The way we use it here is $\exp\left(-\frac{xs}{y-x}\right) < \left(1 - \frac{x}{y}\right)^s < \exp\left(-\frac{xs}{y}\right)$, where $0 < x < y$ and $0 < s$.

$$\exp\left(-2\mathrm{E}[\mathrm{v}_{i-1}]/(m-1)+2\sigma^2/(m-1)^2\right) < \mathrm{E}\left[\left(1-\frac{1}{m}\right)^{2\mathrm{v}_{i-1}}\right] < \exp\left(-2\mathrm{E}[\mathrm{v}_{i-1}]/m+2\sigma^2/m^2\right)$$

and since $\sigma^2 = \mathrm{V}[\mathrm{v}_{i-1}] = O(m)$, these bounds become asymptotically

$$e^{-2\mathrm{E}[\mathrm{v}_{i-1}]/(m-1)} < \mathrm{E}\left[\left(1-\frac{1}{m}\right)^{2\mathrm{v}_{i-1}}\right] < e^{-2\mathrm{E}[\mathrm{v}_{i-1}]/m}$$

Using inequality 4.2.29 from [ABRSTE70] on $(1-\frac{1}{m})^{2\mathrm{E}[\mathrm{v}_{i-1}]}$ we obtain the same bounds:

$$e^{-2\mathrm{E}[\mathrm{v}_{i-1}]/(m-1)} < \left(1-\frac{1}{m}\right)^{2\mathrm{E}[\mathrm{v}_{i-1}]} < e^{-2\mathrm{E}[\mathrm{v}_{i-1}]/m}$$

Consequently the difference $\left|\mathrm{E}[(1-\frac{1}{m})^{2\mathrm{v}_{i-1}}] - (1-\frac{1}{m})^{2\mathrm{E}[\mathrm{v}_{i-1}]}\right|$ is bounded and goes to 0 as $m \to \infty$. □.

We can write $\mathrm{E}[\mathrm{v}_i]$ as $m(1-f_i(m))$. Then $f_i(m)$ obeys the following recurrence

$$f_i(m) = \left(1-\frac{1}{m}\right)^{2\mathrm{E}[\mathrm{v}_{i-1}]} = \left(1-\frac{1}{m}\right)^{2m(1-f_{i-1}(m))}$$

$$f_0(m) = 1-\frac{1}{m}$$

$f_i(m)$ represents the fraction of urns not hit by the $2\mathrm{E}[\mathrm{v}_{i-1}]$ balls. It is an increasing function in m and decreasing in i.

Lemma 2.3 $f_j(2^j(m-2)+2) < f_0(m)$.

Proof [By Induction]: Basis: for $j = 1$, we have

$$f_0(2(m-1)) = 1-\frac{1}{2(m-1)}$$

$$f_1(2(m-1)) = \left(1-\frac{1}{2(m-1)}\right)^{4(m-1)\left(1-\left(1-\frac{1}{2(m-1)}\right)\right)} = \left(1-\frac{1}{2(m-1)}\right)^2$$

$$= 1 - \frac{1}{m-1} + \frac{1}{4(m-1)^2}$$

$$< 1 - \frac{1}{m} = f_0(m)$$

Inductive step: assume true for $j-1$ and prove for j. Let $m_j = 2^j(m-2)+2$. The induction hypothesis is

$$f_{j-1}(m_j) < f_0(m_1)$$

Now

$$f_j(m_j) = \left(1 - \frac{1}{m_j}\right)^{2m_j(1-f_{j-1}(m_j))}$$

$$< e^{-2(1-f_{j-1}(m_j))}$$

which by the induction hypothesis is

$$< e^{-2(1-f_0(m_1))}$$

$$< e^{-2\left(1-\left(1-\frac{1}{m_1}\right)\right)}$$

$$= e^{-\frac{1}{m-1}}$$

$$< 1 - \frac{1}{m} = f_0(m) \qquad \square$$

Lemma 2.4 $\sum_{i=0}^{\ell-1} f_i(m) < \frac{2}{3}\ell + \frac{1}{3}\lceil \lg(m-2) \rceil$.

Proof: Let k be the smallest integer that satisfies $m \le m_k = 2^k(m_0-2)+2$, where m_0 is an arbitrarily chosen constant greater than 2. Then $k = \lceil \lg(m-2) - \lg(m_0-2) \rceil$.

$$\sum_{i=0}^{\ell-1} f_i(m) \le \sum_{i=0}^{\ell-1} f_i(m_k)$$

$$= \sum_{i=0}^{k-1} f_i(m_k) + \sum_{i=k}^{\ell-1} f_i(m_k)$$

$$< k + \sum_{i=0}^{\ell-k-1} f_{i+k}(m_k)$$

$$< k + (\ell - k) f_k(m_k)$$

and by the previous lemma

$$< k + (\ell - k) f_0(m_0)$$

$$= k + (\ell - k)\left(1 - \frac{1}{m_0}\right)$$

$$= \ell\left(1 - \frac{1}{m_0}\right) + \frac{1}{m_0}\lceil \lg(m-2) - \lg(m_0 - 2)\rceil$$

and if we let $m_0 = 3$,

$$= \frac{2}{3}\ell + \frac{1}{3}\lceil \lg(m-2)\rceil \qquad \square$$

Theorem 2.3 *The expected value of* lpsl *for a Robin Hood Hash table with m records is bounded by* $\mathrm{E[psl]} < \mathrm{E[lpsl]} < 3\mathrm{E[psl]} + \lceil \lg(m-2)\rceil$

Proof: Define $\mathbf{r}$ to be the sum of probe positions of the records in $\mathcal{W}_{\ell-1}$. From the construction of the sets $\mathcal{W}_i$'s we know that $\mathcal{W}_0 \subseteq \mathcal{W}_1 \subseteq \cdots \subseteq \mathcal{W}_{\ell-1}$. If a record appears for the first time in the set $\mathcal{W}_k$, its probe position is at least $(\ell - k)$, and it appears in exactly $(\ell - k)$ of the ℓ sets $\mathcal{W}_i$'s. It follows that

$$\mathbf{r} \geq \mathbf{w}_0 + \mathbf{w}_1 + \cdots + \mathbf{w}_{\ell-1}$$

and therefore

$$\mathrm{E}[\mathbf{r}|\mathbf{lpsl} = \ell] \geq \sum_{i=0}^{\ell-1} \mathrm{E}[\mathbf{w}_i]$$

$$\geq \sum_{i=0}^{\ell-1} \mathrm{E}[\mathbf{v}_i]$$

$$= \sum_{i=0}^{\ell-1} m(1 - f_i(m))$$

$$= m\left(\ell - \sum_{i=0}^{\ell-1} f_i(m)\right)$$

and by the previous lemma

$$> \frac{m}{3}\left(\ell - \lceil \lg(m-2)\rceil\right).$$

It follows that

$$\mathrm{E}[\mathbf{r}] > \frac{m}{3}\left(\mathrm{E[lpsl]} - \lceil \lg(m-2)\rceil\right).$$

Let $\mathbf{N}$ be the sum of the probe positions of the m records in $\mathcal{R}$ stored in the hash table. Since $\mathcal{V}_{\ell-1}$ is just a subset of $\mathcal{R}$ then $\mathrm{E}[\mathbf{N}] \geq \mathrm{E}[\mathbf{r}]$ and

$$\mathrm{E[N]} \equiv m\mathrm{E[psl]} > \frac{m}{3}\left(\mathrm{E[lpsl]} - \lceil \lg(m-2) \rceil\right).$$

Solving for E[lpsl] we get

$$\mathrm{E[lpsl]} < 3\mathrm{E[psl]} + \lceil \lg(m-2) \rceil.$$

The lower bound holds because psl $\leq$ lpsl for every table and there is a positive probability that psl < lpsl □.

Corollary 2.4 *The expected value of* lpsl *for a full Robin Hood hash table is* $\Theta(\ln n)$.

Proof: Follows from the theorem and the fact that $\mathrm{E[psl]} = \Theta(\ln n)$ for a full Robin Hood hash table. □

Corollary 2.5 *The expected value of* lpsl *for a full min-max hash table is bounded by*

$$\ln n + \gamma + \frac{1}{2} + P(\ln n) + o(1) < \mathrm{E[lpsl]} < 3\ln n + \lceil \lg(n-2) \rceil + 3\gamma + o(1)$$

where $P(x)$ *is a periodic function with period* 1 *and magnitude* $|P(x)| \leq .0001035$, *and by*

$$-\alpha^{-1}\ln(1-\alpha) < \mathrm{E[lpsl]} < -3\alpha^{-1}\ln(1-\alpha) + \lceil \lg(m-2) \rceil$$

for α*-full tables with* m *records and load factor* $\alpha = \frac{m}{n}$.

Proof: The lower bounds are from [GON81]. For every set of records, the value of lpsl for a min-max hash table is less than the corresponding value for any other hashing scheme. Therefore, E[lpsl] for min-max hashing is less than E[lpsl] for any other hashing scheme; in particular it is less than the E[lpsl] for Robin Hood hashing. The upper bounds are from Theorem 2.3 above. □

Chapter 3

The Distribution of psl

In Chapter 2 we introduced a simple modification of the insertion procedure of the standard hashing algorithm. We then showed that all records inserted using this heuristic have the same distribution of the probe sequence length, regardless of the position in the insertion sequence. We also proved that the expected position of a record in its probe sequence is $\ln(n) + \gamma + o(1)$ for a full table, and that the expected value of **lpsl** is $\Theta(\ln n)$.

In this chapter we derive the distribution of the probe sequence length for a record in a Robin Hood hash table of infinite size. To study the distribution of the random variable **psl** for an infinite table we will introduce a probability model based on balls and urns. It is important to note that the analysis presented below is for an infinite hash table and that we have not proved that an analysis for finite tables would converge to the same result.

3.1 The Asymptotic Model

Consider the following urn model that corresponds to inserting all elements in the table simultaneously. Assume that we have an infinite number of urns and that we drop at random an infinite number of balls such that the average number of balls per urn is α. Each of these balls is given a label of 1. After the balls have been dropped, for each urn that contains more than one ball, one ball is selected according to some criterion and the rest are marked. All balls are left in the urn. For each marked ball with label 1, we create a new unmarked ball with label 2 and drop it into a random urn. After all the new balls have been distributed, we check the urns and for each urn that contains more than one unmarked ball, we select one (unmarked) ball among those with the highest label and mark the rest. We then create an additional unmarked ball with label $i + 1$ for each newly marked ball with label i and drop it at the urns. We continue in this fashion until each urn has an average of α unmarked

balls, or equivalently until a fraction α of the urns contain an unmarked ball (since there can be at most one unmarked ball per urn).

In this model, urns represent table locations, balls represent probes into the table, ball labels represent probe positions or try numbers, and marking a ball represents rejecting a record from the location. An unmarked ball with label i represents a record placed in its i-th probe position.

Notice that in a finite table, the total number of unmarked balls with a label greater than i (number of records after their i-th probe position) is equal to the total number of balls, either marked or unmarked, with label $i+1$ (number of records that probed their $i+1$-st position). For an infinite file this corresponds to the expected number of unmarked balls with label greater than i per urn being equal to the expected number of balls (marked or unmarked) with label $i+1$ per urn.

We define the following variables:

- Let λ be the expected number of balls (marked or unmarked) per urn.
- Let γ_i be the expected number of balls (marked or unmarked) with label equal to i per urn.
- Let λ_i be the expected number of balls (marked or unmarked) with label less than or equal to i per urn. Then $\lambda_{i+1} = \lambda_i + \gamma_i$.
- Let t_i be the expected number of unmarked balls with label less than or equal to i per urn. The expected (per urn) number of unmarked balls with label greater than i is $\alpha - t_i$. We already mentioned that the expected number of unmarked balls with label greater than i is equal to the expected number of marked or unmarked balls with label $i+1$. Hence $\gamma_{i+1} = \alpha - t_i$.

From the definitions above it follows that $\lambda_1 = \gamma_1 = t_\infty = \alpha$.

Consider an arbitrary but fixed urn. Each ball is labeled before it is dropped, and the urn selected by a ball is independent of its label. Therefore the labels of the balls in the urn, at some stage of the loading process described above, can affect the labels of balls arriving later to the urn only in as much as they can affect the expected (per urn) number of balls created with each label. If we have an infinite number of urns, nothing that occurs in a single urn having a finite number of balls can affect the values of the λ_i's, and with probability 1 the number of balls in the urn is finite.

Another way of viewing this last observation is as follows. The balls in an arbitrary but fixed urn can affect the distribution of the label of another ball hitting the same urn only if that ball was created as a (direct or indirect) consequence of marking those balls. Let us then divide our loading process into two phases. The first phase is the same as the original loading process except that all balls falling into the specified urn are placed in the urn but otherwise ignored. The second phase consists of: a) marking the balls in the specified urn exactly as would have been done in the normal loading process, and b) creating additional balls with the appropriate labels until no new ball needs to be created. The labels of the balls that fall into the specified urn in the

first phase are independent of each other. The labels of the balls created in the second phase are not independent of the labels of the balls in the specified urn. The balls (with any label) created in the second phase can be viewed as a branching process (see for example chapter XII of [FEL68]). Every time a ball is dropped two things can happen: the ball hits an empty urn, in which case no descendant is created, or it hits an urn that contains one unmarked ball, in which case one descendant is created. At the end of the first stage the number of balls in the specified urn is finite with probability 1, and hence, with probability 1 all but a finite number of records have been placed in the table. Therefore the fraction of occupied urns is α, the probability of hitting an urn that contains one unmarked ball is α, and the expected number of direct descendants of a ball is also α. Feller proves (section XII.5 in [FEL68]) that if we start with a finite number of individuals and the expected number of direct descendants of an individual is less than 1, then the total progeny is finite with probability 1. Since the total number of balls created in the second phase is finite and they are dropped among an infinite number of urns then, with probability 1, none of them will fall into the specified urn.

A consequence of these two observations is that the labels of the balls in an urn are independent of each other. Note that the second of these observations does not hold if we have only a finite number of urns, since the effect of a single urn on the λ_i's can be arbitrarily large. For a finite number of urns the labels of the balls in a specified urn are mutually dependent but the larger the number of urns, the weaker the dependency. Neither does it hold if we have a full table, since with probability 1 the number of balls in an urn is infinite. This is the reason why the analysis presented below is valid only for infinite nonfull tables.

We have already studied the distribution of the number of probes required to load the table in Theorem 2.1 (page 14). From that it follows that, $\lambda = -\ln(1-\alpha)$.

3.2 The Distribution

In this section we find the probability distribution of the number of positions probed by a key in an infinite Robin Hood hash table. We do this by establishing a recurrence relation for γ_i.

Larson [LAR83] has proved for infinite α-full tables ($\alpha < 1$), that the distribution of the number of balls that hit an urn is Poisson with parameter λ.

Consider an arbitrary but fixed urn. The urn will contain an unmarked ball with label less than or equal to i, when it has been hit by at least one ball with label less than or equal to i and no ball with label higher than i. Let $q_i(x,s)$ denote the probability that the urn is hit by x balls, s of which have a label higher than i. Then $q_i(x,s)$ is the product of a Poisson and a binomial distribution as follows:

$$q_i(x,s) = \frac{e^{-\lambda}\lambda^x}{x!}\binom{x}{s}\left(\frac{\lambda_i}{\lambda}\right)^{x-s}\left(\frac{\lambda-\lambda_i}{\lambda}\right)^s$$

The Poisson factor gives the probability that the urn receives x balls. $(\lambda - \lambda_i)/\lambda$ is the probability that a ball is labeled higher than i. This probability is independent of the labels of other balls in the same urn. Hence, the probability that s out of x balls have a label higher than i has a binomial distribution. The probability that the urn receives at least one ball with label less than or equal to i and no ball with label higher than i is

$$\begin{aligned} t_i &= \sum_{x=1}^{\infty} q_i(x,0) \\ &= e^{-(\lambda-\lambda_i)} \sum_{x=1}^{\infty} \frac{e^{-\lambda_i}\lambda_i^x}{x!} \\ &= e^{-(\lambda-\lambda_i)}\left(1-e^{-\lambda_i}\right). \end{aligned}$$

Define $p_i(\alpha)$ to be the probability that a record probes at least i locations before being placed in the table. Then $\alpha p_{i+1}(\alpha)$ is the number of unmarked balls with label greater than i per urn. We already mentioned that this is equal to the expected number of marked or unmarked balls per urn with label equal to $i+1$. Therefore, $p_i(\alpha)$ satisfies

$$\alpha p_{i+1}(\alpha) = \gamma_{i+1}$$

and from the relation $\gamma_{i+1} = \alpha - t_i$,

$$\begin{aligned} p_{i+1}(\alpha) &= 1 - \frac{t_i}{\alpha} \\ &= 1 - \frac{e^{-(\lambda-\lambda_i)}\left(1-e^{-\lambda_i}\right)}{\alpha}. \end{aligned}$$

Since $\lambda = -\ln(1-\alpha)$ and $\lambda_i = \gamma_1 + \cdots + \gamma_i = \alpha\left(p_1(\alpha) + \cdots + p_i(\alpha)\right)$, we have

$$p_{i+1}(\alpha) = 1 - \left(\frac{1-\alpha}{\alpha}\right)\left(e^{\alpha(p_1(\alpha)+\cdots+p_i(\alpha))} - 1\right).$$

We summarize this in the following theorem.

Theorem 3.1 *In the asymptotic model for an infinite Robin Hood hash table with load factor α ($\alpha < 1$), the probability $p_i(\alpha)$ that a record is placed in the i-th or further position in its probe sequence is equal to*

$$p_1(\alpha) = 1$$

$$p_{i+1}(\alpha) = 1 - \left(\frac{1-\alpha}{\alpha}\right)\left(e^{\alpha(p_1(\alpha)+\cdots+p_i(\alpha))} - 1\right).$$

α	Robin Hood Hashing	Standard Method
0.7	0.527	2.906
0.8	0.700	6.428
0.9	0.983	16.207
0.95	1.230	35.739
0.99	1.618	194.328
0.9999	1.872	1992.110
1.0^-	1.883	∞

Table 3.1: Variance at various load factors

The expected value of this distribution can be obtained by noting that

$$\mathrm{E[psl]} = p_1(\alpha) + p_2(\alpha) + \cdots$$

which can be solved from the equation

$$p_\infty = 0 = 1 - \left(\frac{1-\alpha}{\alpha}\right)\left(e^{\alpha \mathrm{E[psl]}} - 1\right),$$

yielding

$$\mathrm{E[psl]} = -\alpha^{-1}\ln(1-\alpha),$$

which agrees with the expected value of psl proven in Theorem 2.1.

The variance of this distribution is given by

$$V(\alpha) = \sum_{i=1}^{\infty} \left(i + \alpha^{-1}\ln(1-\alpha)\right)^2 \left(p_i(\alpha) - p_{i+1}(\alpha)\right).$$

The variance for the standard method was given on page 3. Table 3.1 shows the variance of psl at various load factors for the standard method and for Robin Hood hashing. The reduction is quite dramatic. The value at $\alpha = 1^-$ was obtain by linear numerical extrapolation. Figure 3.1 shows how the variance increases with the load factor and Figure 3.2 shows the relation between E[psl] and V[psl].

The variance can be computed to any specified tolerance (ignoring the effects of roundoff errors caused by finite precision arithmetic) using the following procedure. Let $r_i(\alpha) = \alpha\left(p_i(\alpha) + \cdots + p_\infty(\alpha)\right)$ and let $q_i(\alpha) = p_i(\alpha) - p_{i+1}(\alpha)$ denote the probability that a record is placed in its i-th probe position. Then

$$\sum_{i=1}^{\infty} r_i(\alpha) = \alpha \sum_{i=1}^{\infty}\sum_{j=i}^{\infty} p_j(\alpha)$$

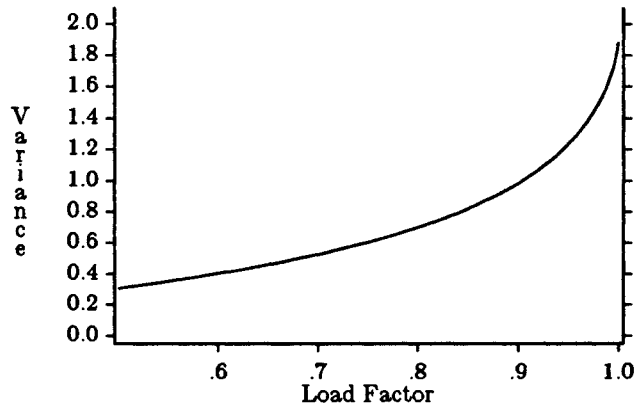

Figure 3.1: V[psl] vs. α

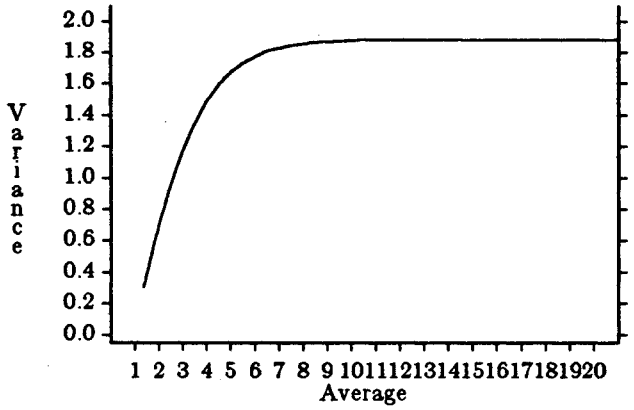

Figure 3.2: V[psl] vs. E[psl]

$$= \alpha \sum_{i=1}^{\infty} i\, p_i(\alpha)$$

$$= \alpha \sum_{i=1}^{\infty} i \sum_{j=i}^{\infty} q_j(\alpha)$$

$$= \alpha \sum_{i=1}^{\infty} \frac{i(i+1)}{2} q_i(\alpha)$$

Using this we can write the variance formula in terms of $r_i(\alpha)$ as

$$V(\alpha) = \frac{2}{\alpha}\sum_{i=1}^{\infty} r_i(\alpha) - \mathrm{E}[\mathrm{psl}] - \mathrm{E}[\mathrm{psl}]^2$$

$$= \frac{2}{\alpha}\sum_{i=1}^{\infty} r_i(\alpha) + \frac{\ln(1-\alpha)}{\alpha} - \frac{\ln^2(1-\alpha)}{\alpha^2}.$$

We can establish a recurrence relation for $r_i(\alpha)$ from the recurrence relation for $p_i(\alpha)$ as follows:

$$p_i(\alpha) = \frac{1}{\alpha} - \frac{1-\alpha}{\alpha} e^{\alpha(p_1(\alpha)+\cdots+p_{i-1}(\alpha))}$$

$$= \frac{1}{\alpha} - \frac{1-\alpha}{\alpha} e^{\alpha \mathrm{E}[\mathrm{psl}]-r_i(\alpha)}$$

$$= \frac{1}{\alpha}\left(1 - e^{-r_i(\alpha)}\right),$$

which implies

$$\alpha p_i(\alpha) = 1 - e^{-r_i(\alpha)}$$

$$r_i(\alpha) - r_{i+1}(\alpha) = 1 - e^{-r_i(\alpha)}.$$

Hence a recurrence relation for $r_i(\alpha)$ is

$$r_{i+1}(\alpha) = r_i(\alpha) - 1 + e^{-r_i(\alpha)}, \qquad i > 1,$$

where $r_1(\alpha) = \alpha \mathrm{E}[\mathrm{psl}] = -\ln(1-\alpha)$. Equation 4.2.37 from [ABRSTE70] states that for $x \leq 1.5936$, $e^{-x} < 1 - \frac{x}{2}$. Therefore, if $r_i(\alpha) \leq 1.5936$,

$$r_{i+1}(\alpha) \quad = r_i(\alpha) - 1 + e^{-r_i(\alpha)} \quad < r_i(\alpha) - 1 + 1 - \frac{r_i(\alpha)}{2} \quad = \frac{r_i(\alpha)}{2},$$

from which it follows that

$$r_{i+k}(\alpha) < \frac{r_i(\alpha)}{2^k}, \qquad k > 0,$$

and hence,

$$\sum_{j=i+1}^{\infty} r_j(\alpha) < r_i(\alpha).$$

Therefore, if we stop the summation when the last term added is less than ϵ ($\epsilon \leq 1.5936$), the sum of the remaining terms is less than ϵ. In Table 3.2 we show the expected value and the variance of psl for values of α close to 1. Ignoring the effect of roundoff errors the variance is accurate to 10 decimal digits.

Roundoff errors will not present a serious problem in the computation of $V(\alpha)$ since the recurrence for $r_i(\alpha)$ is in a sense self-correcting. It is easy to prove that if the absolute error of $r_i(\alpha)$ is δ_i and $|\delta_i| \leq \frac{1}{2}$ then $|\delta_i| > |\delta_{i+1}| > \ldots$. Since $r_i(\alpha)$ converges very rapidly to zero the total number of terms added as well as the accumulation of absolute errors will be small. For example, computing the last row in Table 3.2 required only the first 34 terms of the sequence $r_i(\alpha)$.

Table 3.2 serves to illustrate an interesting point. For load factors close to 1, increasing α has the effect of shifting the distribution to the right without changing much its form. If α is increased such that the expected value is increased by exactly 1, all the central moments of the distribution remain basically the same. In the process of shifting the distribution to the right, a small oscillation will occur in the values of the central moments.

α	E[psl]	V[psl]
0.999664076800000	8.001315823171590	1.858139243052232
0.999798446080000	8.511169078813680	1.866079401322557
0.999879067648000	9.021370218606829	1.871527896538105
0.999927440588800	9.531796488394859	1.875199517231543
0.999956464353280	10.042367691591898	1.877691681261172
0.999973878611968	10.553031774223938	1.879338720554571
0.999984327167181	11.063755138536856	1.880452782253256
0.999990596300308	11.574516205192676	1.881172428971951
0.999994357780185	12.085301173608965	1.881662617015195
0.999996614668111	12.596101251430808	1.881969228803765
0.999997968800867	13.106910855958161	1.882183027132164
0.999998781280520	13.617726453164405	1.882310200529520
0.999999268768312	14.128545811978501	1.882403238711074
0.999999561260987	14.639367527359419	1.882454476688046
0.999999736756592	15.150190716472672	1.882494905033230
0.999999842053955	15.661014825681323	1.882515173687757
0.999999905232373	16.171839508448106	1.882532293661257
0.999999943139424	16.682664548281976	1.882540811550456
0.999999965883654	17.193489809878379	1.882547039215162
0.999999979530193	17.704315209740438	1.882551897182168
0.999999987718116	18.215140694818758	1.882552445419591
0.999999992630869	18.725966233239235	1.882556986109655
0.999999995578522	19.236791804066193	1.882554050563478
0.999999997347113	19.747617395165321	1.882559604296574
0.999999998408268	20.258443000532866	1.882554143979192
0.999999999044961	20.769268617700754	1.882561185802651
0.999999999426976	21.280094224137862	1.882553669736858
0.999999999656186	21.790919851274725	1.882562303706443
0.999999999793711	22.301745445239809	1.882553019490445
0.999999999876227	22.812571044804227	1.882563180358687
0.999999999925736	23.323396704852990	1.882552359244798
0.999999999955442	23.834222327948908	1.882563897004573
0.999999999973265	24.345047951303751	1.882551762771584
0.999999999983959	24.855873920875710	1.882564479259102
0.999999999990375	25.366699544487133	1.882551265572360
0.999999999994225	25.877525168158416	1.882564931761678
0.999999999996535	26.388350791866402	1.882550886543640
0.999999999997921	26.899179085802461	1.882565252280429
0.999999999998753	27.410000259208050	1.882550636515376
0.999999999999252	27.920818465773641	1.882565437145715
0.999999999999551	28.431656451540576	1.882550521555459
0.999999999999731	28.942492377092564	1.882565484104902
0.999999999999838	29.453352340925230	1.882550544643252

Table 3.2: Expected value and Variance of psl for α close to 1

Chapter 4

New Search Algorithms

We have presented a simple modification to the standard insertion algorithm and analysed its effect on the random variables psl and lpsl. If we use this modified insertion algorithm together with the standard search algorithm then, for full tables, the expected number of probes for successful, worst successful and unsuccessful searches is $\Theta(\ln n)$. Each collision in the insert procedure requires the evaluation of the function `findposition` to determine the probe position of the record encountered. If this function determines the probe position of the record by doing a successful search with the standard algorithm counting the number of probes needed, then the cost of loading a full table will be $\Theta(n \ln^2 n)$. Fortunately, we can improve the performance of Robin Hood hashing by modifying the search algorithm and the `findposition` function.

4.1 Speeding up Searching.

In Chapter 2 we saw that it is very unlikely for any record to have a very long probe sequence, since there would have to be a host of other records with very long probe sequences. Similarly, it is very unlikely for a record to have a very short probe sequence. For example, the probability of a record being in probe position 1 is $\frac{1-\alpha}{\alpha}(e^{\alpha}-1)$, which goes to 0 as the table becomes full.

As the load factor increases, the average probe position for a full table increases at a logarithmic rate, but the variance appears to remain bounded by a constant. This means that most of the probability mass is found around the mean. Figure 4.1 demonstrates graphically this clustering effect.

This figure suggests the following question: If the probability that a record is stored in the first location of its probe sequence is so small, why probe there first? This is the key observation leading to the proposed search heuristic.

As the reader might have guessed already, a faster way to search for a record is to start at a location close to the mean position, and then move away from it

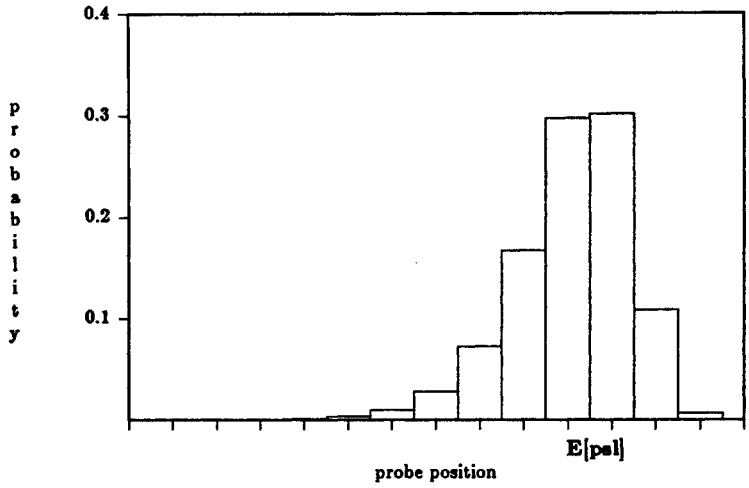

Figure 4.1: Probability distribution of psl for a nearly full table

in both directions (possibly at different rates). There are many search sequences that fit this description but let us refer to them collectively as *mean centered* approaches to searching. More precisely, a mean centered search heuristic is one that finds a key that is l positions above (or below) the mean within cl probes. We then have the following:

Theorem 4.1 *Any mean centered approach for searching an infinite α-full Robin Hood hash table has an expected search cost of $O(1)$.*

Proof: The number of probes made before reaching a location that is l steps away from $t = \lfloor \mathrm{E[psl]} \rfloor$ is $O(l)$. The cost of searching will then be

$$\mathrm{E[search\ cost]} = \sum_{l=0}^{\infty} O(l) \Pr\{|\mathrm{psl} - t| = l\}$$

$$= O\left(\sum_{l=1}^{\infty} \Pr\{|\mathrm{psl} - \mathrm{E[psl]}| \geq l\}\right)$$

$$\leq O\left(\sum_{l=1}^{\infty} \frac{\mathrm{V[psl]}}{l^2}\right)$$

$$= O\left(\frac{V[psl]\pi^2}{6}\right)$$

Since $V[psl] \leq 1.883$ for $\alpha \leq 1 - \epsilon$, we conclude

$$E[\text{search cost}] = O(1) \qquad \square$$

4.2 Organ-Pipe Searching

A natural question to ask is: How should a Robin Hood hash table (or any hash table for that matter) be searched in order to minimize the expected number of probes into the table? Referring back to Figure 4.1, it is clear that if we search first the probe position having the highest associated probability, then the next highest, and so on, we have an optimum search strategy. We call this technique *organ-pipe search*. From the previous theorem, we know that the expected cost for organ-pipe search will be $O(1)$.

The actual number of probes required can be estimated as follows: compute using the formulas of Theorem 3.1 (page 29) the asymptotic probability distribution; compute from that the expected cost of using this search heuristic. As $\alpha \to 1$, the search cost appears to be bounded by 2.57. Figure 4.2 shows how the expected search cost of organ-pipe searching increases for different load factors.

To do organ-pipe searching, one must determine the most likely location for a record, the next most likely, and so on. The easiest way to do this is to keep counters of how many records there are in each probe position, and then use these counters when searching.

Some additional memory will be required to store the counters. In Section 2.4 we proved that less than about $4.44 \ln n + 3\gamma$ different probe positions are expected to contain all the records, so only $\Theta(\ln n)$ counters are needed. From the simulation results presented in the next chapter, it appears that far fewer ($1.15 \ln n + 2.5$) are really needed. For all intents and purposes, 30 integers should be sufficient.

To reduce the overhead of the search algorithm, we will assume that the number of records observed at each probe position always follows an organ-pipe pattern, as the probability distribution does. This does not always hold since it is possible to find situations where, for example, one record is stored in probe position 1 and none in probe position 2. However, doing a true organ-pipe search will not reduce the expected number of probes significantly and will add to the overhead of the algorithm.

A modified insertion algorithm maintaining the counters needed by an organ-pipe search is shown in Figure 4.4, and the algorithm for organ-pipe search is given in Figure 4.3. Initially, `tallest` is equal to 1; `longestprobe` is equal to 0; `m` is equal to 0, and the array `count` is filled with zeros except for `count[0]` and `count[1]` which are -1. On entry to the search procedure the values of

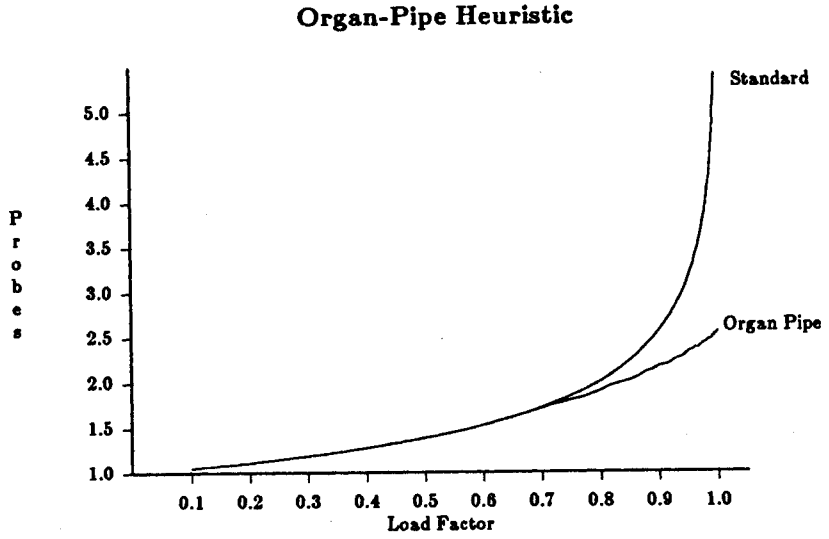

Figure 4.2: Expected Search Cost for Organ-Pipe Searching

```
function search(k)
    upposition := tallest
    uplocation := H(k, upposition)
    downposition := upposition-1
    downlocation := H(k, downposition)
    for i:=1 to longestprobe do
        if count[upposition] > count[downposition] then
            if key(table[uplocation]) = k then return( uplocation )
            upposition := upposition + 1
            uplocation := H(k, upposition)
        else
            if key(table[downlocation]) = k then return( downlocation )
            downposition := downposition - 1
            downlocation := H(k, downposition)
    end for
    return(FAIL)    { unsuccessful search }
end function search
```

Figure 4.3: The Organ-Pipe Search Heuristic

```
table : array [1..n] of RECORD   {all empty}
count : array [0..30] of integer   {all 0 except count[0]=count[1]= -1}
n, m, tallest {intially 1}, longestprobe {intially 0}: integer

function insert(Record)
    if m=n then return(FAIL)    { table full }
    count[longestprobe+1] := 0
    k := Key(Record)
    probeposition := 0
    while k <> empty do
        probeposition := probeposition + 1
        location := H(k, probeposition)
        recordposition := findposition(location)
        if probeposition > recordposition then begin
            count[probeposition] := count[probeposition]+1
            count[recordposition] := count[recordposition]-1
            tempRecord := table[location]
            table[location] := Record
            Record := tempRecord
            k := Key(Record)
            longestprobe := max( longestprobe, probeposition )
            probeposition := recordposition
        end
    endwhile
    while count[tallest+1]>count[tallest] do tallest := tallest + 1
    count[probeposition] := count[probeposition]+1
    count[longestprobe+1] := -1
    m := m+1
    return(location)
end function insert
```

Figure 4.4: Robin Hood insertion keeping counters

count[0] and count[longestprobe+1] are equal to -1 to force all probes to be in the range from 1 to longestprobe.

4.3 Smart Searching

Close inspection of the algorithm for organ-pipe search reveals that every probe into the table involves two comparisons. If the hash table is in core, it might be slightly faster not to use the counters. Furthermore, we can save the $\Theta(\ln n)$ extra memory.

Consider the following search heuristic which we call *smart searching*. Let

t denote the average probe position of the records in the table truncated to the nearest integer. Then, when searching, consider the locations in the order $t, t+1, t-1, t+2, t-2, \ldots$, until the range from 1 to `longestprobe` is covered. Figure 4.5 shows this algorithm.

Since smart search is a mean centered approach for searching, the expected number of probes to find an element in an infinite α-full table is bounded by a constant. It is somewhat higher than that of organ-pipe search, but each probe involves one key comparison and no overhead (besides that required for the loop), so it may be slightly faster, depending on the cost of doing key comparisons.

The expected number of probes required for a search using this heuristic can be estimated in a similar fashion: first compute the asymptotic probability distribution, then the cost of searching given this distribution. Figure 4.6 shows how the expected cost of doing smart searching varies for different load factors. As $\alpha \rightarrow 1$, this value appears to be bounded by 2.84.

4.4 A Smart `findposition`

In Figure 2.1 the function `findposition` was used for the first time. This function computes the probe position of a record already in the table.

At that point we said that this could be done by examining the sequence of locations generated by the hash functions for the key until we find the location where the record is currently stored. If we examine the probe sequence in the order $1, 2, 3, \ldots$, the total cost to load a full table is $\Theta(n \ln^2 n)$ because when a record is moved from probe position i to probe position $i+1$, i steps are required to determine that the other record is in a probe position greater than i.

If double hashing is the method being used to generate the probe sequence then the value of the function can be obtained by performing a division in a finite field. This operation is not usually available in hardware and the number of operations required in a software implementation is $\Theta(\ln n)$. Hence the total cost to load a full table would be $\Theta(n \ln^2 n)$.

We can use the same trick here and examine the sequence in a mean centered order. Both heuristics mentioned before are applicable. Notice that we do not have to probe the hash table, we need only "probe" the probe sequence by inspecting the values generated by the hash function. Now `findposition` will require only a constant number of steps on average to determine the probe position of a stored record.

Hence loading a Robin Hood hash table up to a load factor $\alpha < 1$ takes, on the average, the same number of probes as for the standard method, and each probe costs $O(1)$. We therefore have the following result.

Theorem 4.2 *The expected cost of loading a Robin Hood hash table up to a load factor $\alpha < 1$, is within a constant factor of the cost of loading using the standard method.*

```
table : array [1..n] of RECORD
n, m, totalcost, longestprobe : integer

function search(k)
      meanposition := trunc(totalcost/m)
      downposition := meanposition
      upposition := downposition + 1
      while( downposition >= 1 and  upposition <= longestprobe ) do
            downlocation := H(k, downposition)
            if key(table[downlocation]) = k then return( downlocation )
            uplocation := H(k, upposition)
            if key(table[uplocation]) = k then return( uplocation )
            downposition := downposition-1
            upposition := upposition+1
      end while
      while( downposition >= 1 ) do
            downlocation := H(k, downposition)
            if key(table[downlocation]) = k then return( downlocation )
            downposition := downposition-1
      end while
      while ( upposition <= longestprobe ) do
            uplocation := H(k, upposition)
            if key(table[uplocation]) = k then return( uplocation )
            upposition := upposition+1
      end for
      return(FAIL)      { unsuccessful search }
end function search
```

Figure 4.5: The Smart Searching Heuristic

4.5 Summary

In this chapter we have shown how a Robin Hood hash table can be searched to substantially reduce the expected search time and have discussed two algorithms for doing so. We then explained how to use the same idea to determine the probe position of a record in the table in a constant number of steps on the average.

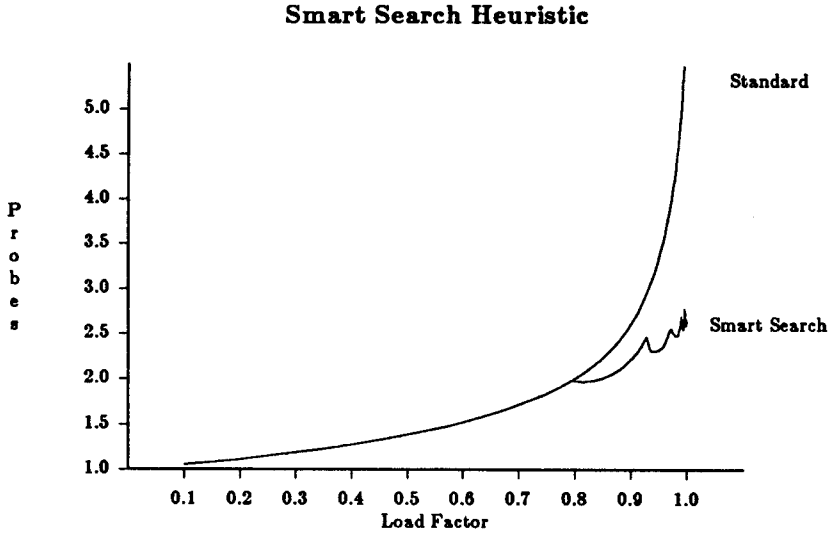

Figure 4.6: Expected Search Cost for Smart Searching

Chapter 5

Simulation Results

In this chapter we present the results of some rather extensive simulations. There are several reasons for performing the simulation experiments. The most important one is to validate the results of the analysis in the previous chapters and show that they can be used to predict the performance of double hashing combined with the Robin Hood heuristic. Secondly, we would like to know the performance of full tables and the expected value of the longest probe sequence length (E[lpsl]), which could not be determined theoretically. A third reason is to compare execution times. Unfortunately the execution times reported are not very accurate, as explained below, and the cost of performing operations on the keys (key comparisons and computation of hash functions) affects the conclusions that can be drawn from the timing estimates; nonetheless, a comparison of execution times is interesting.

5.1 Simulation Experiments

Simulations of the standard hashing, Brent's, Robin Hood with organ-pipe searching, and Robin Hood with smart search reordering schemes were performed. Double hashing was used to generate the probe sequence. Keys were pairs of 31-bit integers generated by a pseudorandom number generator described on page 30 of [KNU69]. The hash functions used were $h_1(K_1, K_2) = K_1 \bmod n$ and $h_2(K_1, K_2) = K_2 \bmod (n-2)+1$, where K_1 and K_2 are the two integers forming the key and n, the table size, is a prime number.

Simulations were done for table sizes of 1021, 4093, 16273, 65537, and 262139. Statistics were collected at load factors as close as possible for these table sizes to 60%, 70%, 80%, 90%, and 100%. For each file size, 210 simulation experiments were performed. The simulations required more than a hundred hours of cpu time on a pair of Vax-11/780's.

The following five performance measures were recorded: average number of

probes to insert; average number of probes to search; average time to insert; average time to search; and the longest probe sequence length. 95% confidence intervals for each of these performance measures were also computed. Whether or not the theoretically predicted value lies within the 95% confidence interval is indicated with a $\surd$ or $\times$ in the tables summarizing the results.

The load on the computing systems used to perform the simulations affects the cpu time reported to the user. The simulation experiments were done under different system loads and on different configurations, so the figures reported in milliseconds should be approached with some skepticism.

5.2 Results for Robin Hood Hashing

	≈ 60%		≈ 70%		≈ 80%	
n	*pred*	*simulation*	*pred*	*simulation*	*pred*	*simulation*
1021	1.525	1.525±.005$\surd$	1.717	1.714±.006$\surd$	2.006	2.004±.008$\surd$
4093	1.527	1.529±.003$\surd$	1.720	1.723±.004$\surd$	2.011	2.016±.005$\surd$
16273	1.527	1.527±.001$\surd$	1.720	1.720±.002$\surd$	2.012	2.011±.002$\surd$
65537	1.527	1.528±.001$\surd$	1.720	1.720±.001$\surd$	2.012	2.012±.001$\surd$
262139	1.527	1.527±.000$\surd$	1.720	1.720±.000$\surd$	2.012	2.012±.001$\surd$

	≈ 90%		100%	
n	*predicted*	*simulation*	*approx*	*simulation*
1021	2.546	2.531±.012×	7.507	7.526±.183$\surd$
4093	2.556	2.557±.006$\surd$	8.895	9.026±.205$\surd$
16273	2.558	2.557±.003$\surd$	10.275	10.423±.171$\surd$
65537	2.558	2.559±.002$\surd$	11.668	11.659±.176$\surd$
262139	2.558	2.558±.001$\surd$	13.054	13.115±.170$\surd$

Table 5.1: Robin Hood: Number of probes to insert (E[psl])

Validation of the model of Robin Hood hashing is necessary for two reasons: the analysis assumed random probing, even though double hashing is the preferred method in practice; the formulas obtained are for tables of infinite size, hence we want to see how accurately they predict the performance of finite tables.

Tables 5.1, 5.3, and 5.4 show the predicted and experimentally observed values for the number of probes per record needed for loading a table using the Robin Hood heuristic, and for searching it using the organ-pipe and smart search heuristics, respectively. Table 5.2 shows the predicted and experimental values of the variance of psl.

As mentioned in Chapter 4, the function `findposition` "probes" only the probe sequence of the stored record and not the hash table. These "probes" are not counted in the figures presented in Table 5.1. The predicted values

	≈ 60%		≈ 70%		≈ 80%	
n	*pred*	*simulation*	*pred*	*simulation*	*pred*	*simulation*
1021	.4024	.4007±.0042√	.5256	.5222±.0046√	.6984	.6943±.0062√
4093	.4029	.4036±.0022√	.5266	.5264±.0027√	.6999	.7002±.0033√
16273	.4030	.4037±.0010√	.5266	.5268±.0011√	.7000	.6993±.0015√
65537	.4031	.4037±.0006√	.5266	.5268±.0007√	.7000	.7002±.0009√
262139	.4031	.4031±.0003√	.5266	.5266±.0003√	.7001	.7001±.0004√

	≈ 90%		100%	
n	*predicted*	*simulation*	*approx*	*simulation*
1021	.9794	.9657±.0080×	1.8282	1.8179±.0160√
4093	.9821	.9800±.0043√	1.8634	1.8635±.0077√
16273	.9826	.9811±.0021√	1.8761	1.8775±.0044√
65537	.9828	.9830±.0010√	1.8805	1.8815±.0022√
262139	.9828	.9826±.0005√	1.8819	1.8813±.0011√

Table 5.2: Robin Hood: Variance of probe sequence length (V[psl])

in Table 5.1 were computed using the formulas of Theorem 2.1 and those in Tables 5.2, 5.3, and 5.4 from the probability distribution of Theorem 3.1. The reader is again reminded that the actual load factors are as close as possible to the table heading value but do differ fractionally, hence the values in the *predicted* column do indeed differ. For the case of full tables the probability distribution of psl was approximated by that of an α-full table with load factor $\alpha = 1 - \frac{1}{n}$. This approximation appears to give acceptable results.

The predicted values of the mean and the variance of psl agree with the values observed in the simulations. Only in one of the 20 (5%) sets of experiments the moments of the distribution are outside the 95% confidence interval and most of the remaining predictions are well within the confidence intervals.

The estimates for the number of probes required to search are based on an asymptotic model. For α-full tables the accuracy of the estimates improves with the table size, as might be expected. For full tables the approximations are mostly outside the confidence intervals, but only by less than .15 probes for smart search and less than .01 probes for organ-pipe searching. We conclude that for all practical purposes these estimates are adequate.

The number of probes to load a Robin Hood hash table is the same when using the organ-pipe or the smart search heuristics, but the cpu time may differ since different heuristics are used in `findposition`. Tables 5.5 and 5.7 show the insertion time per record using each of these heuristics. Tables 5.6 and 5.8 show the search time.

The motivation for introducing smart search was that it might be faster than organ-pipe searching. Even though the number of probes is higher, each probe will require one less comparison. It appears that loading using smart search is

	≈ 60%		≈ 70%		≈ 80%	
n	*pred*	*simulation*	*pred*	*simulation*	*pred*	*simulation*
1021	1.526	1.525±.005√	1.718	1.706±.005×	1.903	1.900±.009√
4093	1.527	1.529±.003√	1.720	1.719±.003√	1.905	1.907±.005√
16273	1.527	1.527±.001√	1.720	1.720±.002√	1.905	1.905±.002√
65537	1.527	1.528±.001√	1.720	1.720±.001√	1.906	1.906±.001√
262139	1.527	1.527±.000√	1.720	1.720±.000√	1.906	1.906±.001√

	≈ 90%		100%	
n	*predicted*	*simulation*	*approx*	*simulation*
1021	2.169	2.150±.007×	2.549	2.568±.053√
4093	2.172	2.164±.004×	2.546	2.576±.056√
16273	2.172	2.168±.002×	2.552	2.552±.002√
65537	2.172	2.172±.001√	2.556	2.553±.001×
262139	2.172	2.172±.001√	2.543	2.552±.001×

Table 5.3: Organ-Pipe Search: number of probes

	≈ 60%		≈ 70%		≈ 80%	
n	*pred*	*simulation*	*pred*	*simulation*	*pred*	*simulation*
1021	1.526	1.525±.005√	1.718	1.714±.006√	1.968	1.964±.005√
4093	1.527	1.529±.003√	1.720	1.723±.004√	1.968	1.973±.002×
16273	1.527	1.527±.001√	1.720	1.720±.002√	1.968	1.974±.002×
65537	1.527	1.528±.001√	1.720	1.720±.001√	1.968	1.968±.001√
262139	1.527	1.527±.000√	1.720	1.720±.000√	1.968	1.968±.000√

	≈ 90%		100%	
n	*predicted*	*simulation*	*approx*	*simulation*
1021	2.214	2.201±.010×	2.612	2.741±.022×
4093	2.218	2.218±.006√	2.650	2.760±.023×
16273	2.220	2.218±.003√	2.696	2.763±.020×
65537	2.220	2.221±.001√	2.613	2.777±.021×
262139	2.220	2.219±.001√	2.831	2.761±.020×

Table 5.4: Smart Search: number of probes

slightly faster (by about 20%) than when using organ-pipe searching, although this could also be attributed to the cost of keeping track of the counters in the later method. The search time per key for organ-pipe searching also appears higher (by about 20%) than the corresponding time for smart search.

n	$\approx 60\%$	$\approx 70\%$	$\approx 80\%$	$\approx 90\%$	100%
1021	.1239±.0022	.1467±.0025	.1817±.0027	.2487±.0031	.9167±.0270
4093	.1233±.0007	.1470±.0011	.1833±.0012	.2534±.0015	1.1407±.0315
16273	.1229±.0004	.1461±.0004	.1826±.0005	.2531±.0007	1.3301±.0252
65537	.1315±.0002	.1565±.0003	.1957±.0003	.2717±.0004	1.6095±.0281
262139	.1360±.0003	.1617±.0003	.2019±.0004	.2796±.0006	1.8440±.0244

Table 5.5: Organ-Pipe: average time (msecs) to insert a record

n	$\approx 60\%$	$\approx 70\%$	$\approx 80\%$	$\approx 90\%$	100%
1021	.0975±.0019	.1000±.0015	.1036±.0015	.1092±.0009	.1199±.0017
4093	.0963±.0006	.1006±.0005	.1050±.0006	.1109±.0005	.1204±.0013
16273	.0977±.0003	.1022±.0004	.1069±.0003	.1130±.0003	.1220±.0003
65537	.1031±.0002	.1080±.0002	.1130±.0002	.1198±.0002	.1295±.0002
262139	.1067±.0002	.1121±.0003	.1172±.0002	.1245±.0003	.1348±.0003

Table 5.6: Organ-Pipe: average time (msecs) for a successful search

The second reason for performing simulation experiments was to determine the value of E[lpsl]. In Chapter 2 we proved that $E[psl] < E[lpsl] < 3E[psl] + 1.443 \ln m$, which for full tables means that $\ln n + O(1) < E[lpsl] < 4.443 \ln n + O(1)$. Gonnet [GON81] proved that $\ln n + O(1)$ is a lower bound for E[lpsl] for min-max hashing (and hence for all hashing schemes) using random probing.

The length of the longest probe sequence is presented in Table 5.9 and Figure 5.1. For a fixed load factor, the longest probe sequence length grows slowly when the table size is increased. Figure 5.1 seems to indicate that the growth is, at most, logarithmic in the table size.

For full Robin Hood tables, the E[lpsl] seems to be about $1.15 \ln n + 2.5$. This is better than the conjecture of $1.44 \ln n + 1$ for binary tree hashing [GONMUN79]. From the simulations of min-max hashing presented in [GONMUN79] it would appear that the expected length of the longest probe sequence for that method is about $1.15 \ln n + 0.5$; however, the file sizes used there are too small (101 and 499) to give such a conjecture much credence.

n	$\approx 60\%$	$\approx 70\%$	$\approx 80\%$	$\approx 90\%$	100%
1021	.1133±.0020	.1312±.0021	.1601±.0026	.2117±.0030	0.7291±.0224
4093	.1151±.0010	.1340±.0011	.1633±.0012	.2189±.0013	0.9162±.0225
16273	.1143±.0004	.1327±.0004	.1621±.0005	.2181±.0006	1.0674±.0188
65537	.1297±.0003	.1508±.0003	.1834±.0003	.2461±.0005	1.3447±.0307
262139	.1231±.0002	.1427±.0002	.1736±.0002	.2326±.0003	1.4211±.0176

Table 5.7: Smart Searching: average time (msecs) to insert a record

n	$\approx 60\%$	$\approx 70\%$	$\approx 80\%$	$\approx 90\%$	100%
1021	.0798±.0016	.0819±.0017	.0872±.0014	.0890±.0014	.0965±.0012
4093	.0814±.0006	.0849±.0006	.0890±.0006	.0910±.0005	.0992±.0006
16273	.0814±.0002	.0847±.0002	.0901±.0002	.0916±.0002	.1001±.0003
65537	.0883±.0002	.0919±.0002	.0977±.0002	.0998±.0002	.1099±.0003
262139	.0906±.0001	.0945±.0001	.1005±.0001	.1028±.0001	.1135±.0003

Table 5.8: Smart Searching: average time (msecs) for a successful search

n	$\approx 60\%$	$\approx 70\%$	$\approx 80\%$	$\approx 90\%$	100%
1021	3.629±.065	4.000±.013	4.329±.064	5.105±.041	10.443±.187
4093	3.967±.024	4.062±.033	4.800±.054	5.329±.064	12.133±.208
16273	4.014±.016	4.262±.060	5.000±.000	5.771±.057	13.819±.172
65537	4.029±.023	4.614±.066	5.000±.000	6.000±.000	15.181±.178
262139	4.098±.040	4.967±.024	5.022±.020	6.000±.000	16.815±.179

Table 5.9: Robin Hood: longest probe sequence length

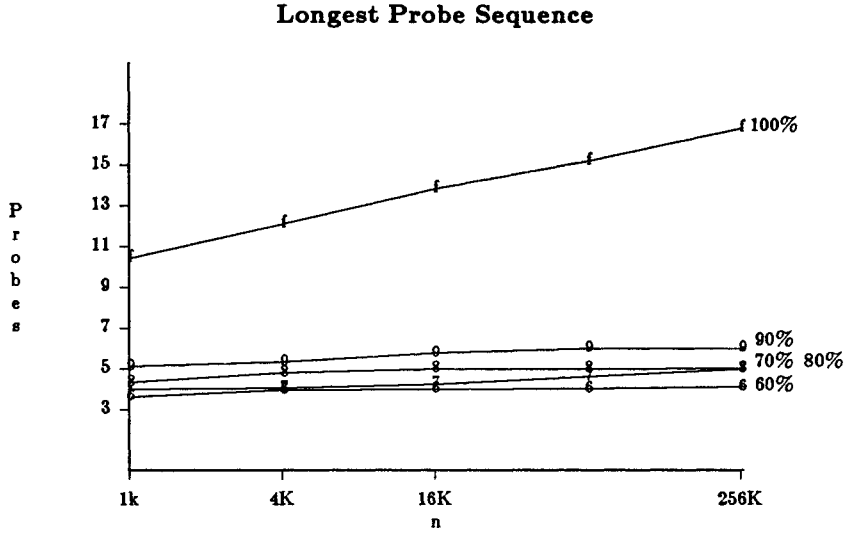

Figure 5.1: Longest Probe Sequence Length for Robin Hood Hashing

5.3 The Standard Method

Brent's method and standard double hashing were also simulated to compare their performance with that of Robin Hood hashing. These three methods are the only open addressing schemes that permit loading a full table in $\Theta(n \ln n)$ time. The other three schemes discussed, binary tree, optimal and min-max hashing appear to require $\Omega(n\sqrt{n})$ time to load a full table. Tables 5.10, 5.11, 5.12, and 5.13 show the simulation results for the standard method. The difference in the average time to insert and search a record is probably due to the to the overhead of keeping track of the value of **longestprobe**.

n	$\approx 60\%$	$\approx 70\%$	$\approx 80\%$	$\approx 90\%$	100%
1021	1.524±.005	1.714±.006	2.006±.007	2.540±.012	6.542±.078
4093	1.528±.003	1.724±.004	2.014±.004	2.558±.006	7.894±.081
16273	1.527±.001	1.719±.002	2.009±.002	2.556±.003	9.311±.074
65537	1.527±.001	1.720±.001	2.011±.001	2.558±.002	10.686±.079
262139	1.527±.000	1.720±.000	2.012±.001	2.559±.001	12.024±.085

Table 5.10: Standard method: average number of probes to insert or search

n	$\approx 60\%$	$\approx 70\%$	$\approx 80\%$	$\approx 90\%$	100%
1021	.0686±.0019	.0705±.0023	.0754±.0029	.0840±.0028	.1475±.0033
4093	.0666±.0006	.0697±.0007	.0748±.0007	.0841±.0007	.1734±.0016
16273	.0669±.0002	.0703±.0002	.0753±.0002	.0847±.0002	.1998±.0014
65537	.0735±.0002	.0772±.0002	.0830±.0002	.0935±.0002	.2507±.0016
262139	.0753±.0001	.0792±.0001	.0853±.0001	.0965±.0001	.2901±.0018

Table 5.11: Standard method: average number of msecs to insert a record

An interesting observation is that for full tables, the average number of probes required by Robin Hood hashing to insert a record is about 1 more than the corresponding value for the standard method. We proved in section 2.2 that these two schemes require the same average number of probes to insert if random probing is used. However random probing is memoryless, but the simulations were done using double hashing which is not.

The issue here is the difference between random probing and uniform hashing (or double hashing) for the standard method. Our simulations for the standard method do match the analysis for uniform hashing (page 3). Under the Robin Hood heuristic, the probe sequence length for each element is virtually the same. This, of course, minimizes the expected number of repeated probe values in the portion of the probe sequence encountered, subject to the constraint that the

expected value of the sum of the psl of all the records is $n \ln n + \gamma + o(1)$. Hence we are not at all surprised by the virtually complete agreement between Robin Hood under random probing and under double hashing.

n	$\approx 60\%$	$\approx 70\%$	$\approx 80\%$	$\approx 90\%$	100%
1021	.0534±.0020	.0578±.0017	.0623±.0013	.0721±.0016	.1381±.0020
4093	.0535±.0007	.0584±.0006	.0633±.0005	.0739±.0005	.1645±.0016
16273	.0540±.0002	.0581±.0002	.0642±.0002	.0748±.0003	.1925±.0013
65537	.0607±.0001	.0653±.0001	.0718±.0002	.0836±.0002	.2438±.0016
262139	.0628±.0001	.0677±.0001	.0745±.0001	.0867±.0001	.2842±.0018

Table 5.12: Standard method: average number of msecs to search

Robin Hood hashing takes more time to insert than the standard method mainly because of the overhead involved in the `findposition` function. The difference depends on the load factor, but goes from 50% more for a 60% load factor to six times more for a full table.

The number of probes required to search starts to be lower for organ-pipe search than for the standard algorithm at a load factor of about 80%. This is the point at which the heuristic begins to differ from the standard search algorithm, as a comparison of Tables 5.3 and 5.4 with Table 5.1 shows. In spite of the higher number of probes required to search, the search time for full tables is only 43% more for the standard method at $n = 1021$ and 250% more at $n = 262139$, when compared with smart searching. This is because each probe requires less overhead. This suggests that perhaps a good combination is to use the Robin Hood insertion algorithm with the standard search algorithm when the load factor is below 90% and with smart search when it is above.

n	$\approx 60\%$	$\approx 70\%$	$\approx 80\%$	$\approx 90\%$	100%
1021	9.438±.252	12.991±.396	20.281±.724	36.424±1.372	649.04±25.11
4093	12.105±.331	16.552±.453	24.904±.645	47.152±1.403	2574.2±106.6
16273	14.343±.258	19.495±.421	29.900±.668	56.791±1.233	10374.±449.
65537	16.695±.298	23.181±.420	35.710±.692	70.191±1.485	41918.±1688.
262139	19.019±.288	26.829±.437	41.591±.689	81.367±1.513	164230.±7153.

Table 5.13: Standard method: Longest probe sequence length

The observed values of the longest probe sequence length are high for the standard method. We see this as the main drawback of the standard algorithm. The values obtained are in agreement with the expected values of lpsl for uniform hashing [GON81] of $E[\text{lpsl}] = .6315... \times n + O(1)$ for full tables, and $E[\text{lpsl}] = -\log_\alpha n - \log_\alpha(-\log_\alpha n) + O(1)$ for α-full tables.

5.4 Brent's Method

Brent's method was also simulated. The results of the simulation are shown in Tables 5.14, 5.15, 5.16, 5.17 and 5.18. To insert records into the table, a level search was performed for the empty location. Brent's original suggestion [BRE73] was to perform a search in a depth first manner, but this would increase the number of locations probed to insert each record.

n	$\approx 60\%$	$\approx 70\%$	$\approx 80\%$	$\approx 90\%$	100%
1021	1.645±.007	1.896±.008	2.278±.011	2.991±.016	8.992±.187
4093	1.649±.004	1.901±.005	2.282±.006	3.001±.008	10.610±.179
16273	1.648±.002	1.898±.002	2.280±.003	3.002±.004	12.085±.182
65537	1.648±.001	1.898±.001	2.281±.001	3.003±.002	13.559±.176
262139	1.648±.000	1.899±.001	2.282±.001	3.004±.001	14.929±.156

Table 5.14: Brent's method: average number of probes to insert

n	$\approx 60\%$	$\approx 70\%$	$\approx 80\%$	$\approx 90\%$	100%
1021	1.365±.003	1.466±.003	1.599±.004	1.802±.004	2.439±.008
4093	1.368±.002	1.468±.002	1.601±.002	1.805±.002	2.468±.004
16273	1.368±.001	1.468±.001	1.601±.001	1.805±.001	2.484±.002
65537	1.368±.000	1.467±.000	1.601±.000	1.805±.001	2.491±.001
262139	1.368±.000	1.468±.000	1.601±.000	1.806±.000	2.496±.001

Table 5.15: Brent's method: average number of probes to search

n	$\approx 60\%$	$\approx 70\%$	$\approx 80\%$	$\approx 90\%$	100%
1021	.1554±.0019	.1660±.0022	.1814±.0023	.2104±.0026	.4611±.0082
4093	.1563±.0005	.1672±.0006	.1833±.0007	.2136±.0007	.5361±.0076
16273	.1584±.0002	.1692±.0002	.1857±.0002	.2171±.0002	.6111±.0079
65537	.1681±.0002	.1798±.0003	.1978±.0003	.2317±.0003	.7232±.0086
262139	.1716±.0003	.1837±.0003	.2022±.0004	.2370±.0004	.8123±.0079

Table 5.16: Brent's method: average number of msecs to insert

Notice that the number of probes needed to insert using Brent's method is higher than for Robin Hood hashing, but the cost in milliseconds is lower at high load factors. In checking that a location in the table was empty, we only did a comparison on the first integer of the key. If key comparisons were

n	$\approx 60\%$	$\approx 70\%$	$\approx 80\%$	$\approx 90\%$	100%
1021	.0475±.0016	.0518±.0017	.0541±.0014	.0584±.0010	.0689±.0010
4093	.0496±.0004	.0524±.0003	.0553±.0003	.0598±.0003	.0720±.0003
16273	.0513±.0001	.0538±.0001	.0572±.0001	.0616±.0001	.0745±.0001
65537	.0573±.0001	.0606±.0001	.0635±.0001	.0685±.0001	.0832±.0001
262139	.0597±.0001	.0626±.0001	.0662±.0001	.0715±.0002	.0869±.0002

Table 5.17: Brent's method: average number of msecs to search

n	$\approx 60\%$	$\approx 70\%$	$\approx 80\%$	$\approx 90\%$	100%
1021	4.533±.083	5.390±.100	6.543±.110	8.924±.168	40.476±1.856
4093	5.233±.090	6.243±.110	7.552±.127	10.352±.165	78.724±3.518
16273	5.729±.083	6.957±.104	8.576±.118	11.867±.170	154.11±8.05
65537	6.424±.085	7.629±.091	9.514±.130	13.071±.194	305.67±14.50
262139	6.990±.080	8.452±.101	10.501±.134	14.391±.163	590.04±25.73

Table 5.18: Brent's method: Longest probe sequence length

more expensive (for example if keys were character strings) this relation could be reversed.

No checking for duplicate keys was performed during the insertions. Checking for duplicates would increase the number of probes required for Brent's method since an unsuccessful search (its weakest point) would precede each insertion. Checking for duplicates does not increase the number of probes required by Robin Hood hashing.

Searching using Brent's method requires about .06 probes less and about 30% less time than using smart searching on the average for full tables. The weakest point of Brent's method is the E[lpsl] (and hence unsuccessful searches) which appears to be $\Theta(\sqrt{n})$ for full tables and $\Theta(\ln m)$ for α-full tables.

5.5 Summary

The main conclusion we can draw from this chapter is that the results from the theoretical analysis can indeed be used to predict the performance of Robin Hood hashing combined with double hashing. From the simulations it appears that for full tables E[lpsl] $\approx 1.15 \ln n + 2.5$ for Robin Hood hashing and E[lpsl] $= 1.15\sqrt{n} + O(1)$ for Brent's method. For nonfull tables these values appear to be $o(\ln m)$ and $\Theta(\ln m)$ respectively.

Smart search appears to be faster than organ-pipe searching even when keys are two integers long. The time to insert a record appears to the lowest for the

standard method, and the time to retrieve a record appears to be the lowest for Brent's method. But in both cases, the execution time for Robin Hood hashing is not much worse and the longest probe sequence length is many times better.

Chapter 6

Deletions

In this chapter we discuss the standard way of handling deletions in hash tables with open addressing and show how it applies to Robin Hood hashing. We then slightly modify the Robin Hood insertion and search algorithms to improve their performance when deletions may have occurred. The study presented in this chapter is explorative in nature, a full analysis has not been attempted. All conclusion are based on simulations.

6.1 Deletions in Hashing with Open Addressing

It is surprising to notice that the obvious way to delete a record does not work for a hash table with open addressing. We cannot simply remove the record and mark its location as empty. The problem arises because a search, using the algorithm in Figure 1.2 on page 4, for a record will fail if the desired record was rejected from that location during insertion.

In general, deletions can be handled by marking with a special code the table entry that contains the record to be deleted. There will then be three kinds of table entries: empty, occupied and deleted. When inserting, deleted table entries are treated as if they were empty, but they are treated as if they were occupied during searches.

It is fairly easy to see that if the standard insertion and search algorithms (modified to handle deleted entries correctly) are used, then after a sufficiently large number of deletions followed by insertions the E[**psl**] $= (1-\alpha)^{-1}$ and E[**upsl**] $= n$. $(1-\beta)^{-1}$ is the expected number of probes to insert a record when the ratio of occupied entries is β, but all records in the table were inserted when this ratio was α. Since all records in the table are now either occupied or deleted, and the value of `longestprobe` can only increase, unsuccessful searches take n probes. Hence the performance deteriorates to something worse than

the performance of linear probing (which is E[psl] $\approx \frac{1}{2}\left(1+(1-\alpha)^{-1}\right)$ and E[upsl] $\approx \frac{1}{2}\left(1+(1-\alpha)^{-2}\right)$).

Peterson [PET57] and Larson [LAR83] also discuss the effect of deletions on hashing with open addressing. As far as we know, there have been no "positive" results published on doing deletions and subsequent insertions on hash tables with open addressing. By "positive" we mean substantial improvements over E[psl] $= O\left(\frac{1}{1-\alpha}\right)$ and E[upsl] $= O(n)$ in the steady state.

6.2 Deletions in Robin Hood Hashing

Deletions in a Robin Hood hash table can be performed by marking the table entry as empty and using the algorithms of Figures 2.1, 4.4, 4.3 and 4.5. This is because in these algorithms we don't stop a search when an empty location is found. However, simulation results show that this will cause the distribution of the psl to spread out, with a resulting increase in the search cost.

Our main concern for an efficient algorithm to perform deletions and subsequent insertions in a hash table is, as before, the variance (and not the expected value) of psl. With this in mind, we propose the following modifications to the algorithms for Robin Hood hashing: to delete a record, mark the table entry as deleted but keep the key value; when inserting, a deleted element is displaced if and only if it would be displaced if it were not flagged as deleted; when a deleted element is displaced, it is discarded and the insertion is complete.

We would expect this to make the expected value of psl increase without bound, but keep the variance bounded by a small constant. The expected value will increase without bound because once a location contains a record at probe position i, then in the future it can only contain records that are at or past probe position i.

We expect the variance to remain bounded, and hence most records to be at a probe position close to the average probe position. Our intuition is that, when a record is at a probe position which is small compared to the average, it is unlikely that it will be placed in that location even if it contains a deleted key. If its probe position is large, it is very likely that the record will be placed in that location even if the location is occupied.

Unfortunately, we have not been able to analyze the behavior of this algorithm. However extensive simulations strongly indicate that the variance of psl after a large number of updates remains bounded by a small constant, and is never greater than that of a full table in which no deletions have occurred. Therefore, the cost of a successful search (we explain the algorithm below) after an arbitrary number of insertions and deletions into the table remains bounded by the cost of a successful search in a full table with no deletions. Unsuccessful searches are similarly bounded.

We have already mentioned that the average probe position grows without bound. This would imply that the cost of doing an insertion also grows without

bound, since each key has probed positions 1 to about the average probe position. We can reduce and bound this cost by keeping track of the value of the smallest probe position among the records (deleted or otherwise) in the table. The insertion procedure then starts at a probe position equal to this value, since a placement before that position is not possible. Empty locations are treated as containing a deleted record in probe position 0.

The most efficient way of keeping track of the smallest probe position is to have counters of how many records are at each probe position. This is similar to the approach of the algorithm in Figure 4.4, but now we also count deleted entries. Only $\Theta(\ln n)$ counters are needed and we conjecture, based on the simulations of the previous and present chapters, that about $1.15 \ln n + 2.5$ different probe positions will contain all the records in the table.

In Figure 6.1 we show a modified version of the algorithm of Figure 2.1 to perform an insertion into a Robin Hood table when there may be deleted elements in the table. The new variables used are `dshortestprobe` and the array `dcount`, which we use solely to keep track of the value of `shortestprobe`. Initially, `dshortestprobe` is equal to zero and the array `dcount` is equal to zero, except `dcount[0]` which is equal to n. `findposition` returns 0 for an empty record (as before) and for a deleted record it returns the probe position where it was before being deleted[1]. The only modification required to the smart search heuristic of Figure 4.5 is to change the condition `downposition>=1` for `downposition>=dshortestprobe` in the while loops.

In Figure 6.2 we present a modified version of the algorithm of Figure 4.4. The difference between the arrays `dcount` and `count` is that the first one also counts deleted records (empty records are considered as deleted in probe position zero) while `count` does not. The array `dcount` is used only to keep track of `dshortesprobe` and is not used by the search algorithm. The sum of the values in the array `dcount` is equal to n, but the sum of the values in the array `count` is equal to $m - 2$. Initially the elements in `dcount` are all equal to 0 except `dcount[0]` which is equal to n; the elements of `count` are equal to 0 except `count[0]` and `count[1]` which are equal to -1; `tallest` is equal to 1. To delete a record from the table, the record must be marked as deleted, `count[i]` must be decremented and `shortestprobe` and `longestprobe` must be updated if necessary. The only changes required to the organ-pipe search heuristic of Figure 4.3 of page 37 are: the starting value of `i` in the for-loop should be `shortestprobe` (instead of 1) and the array `count` should be indexed modulo its size.

---

[1] Actually, it is better to return the probe position minus one for deleted records, therefore breaking ties against the deleted element.

```
table : array [1..n] of RECORD   {all empty}
dcount : array [0..csize-1] of integer;   {all 0 except dcount[0]=n}
totalcost, shortestprobe, longestprobe : integer   {all 0}
n, m : integer

function insert(Record)
      if m=n then return(FAIL)      { table full }
      k := Key(Record)
      probeposition := shortestprobe
      while (k <> empty) and (k <> deleted) do
            probeposition := probeposition + 1
            location := H(k, probeposition)
            totalcost := totalcost + 1
            recordposition := findposition(location)
            if probeposition > recordposition then begin
                  tempRecord := table[location]
                  table[location] := Record
                  Record := tempRecord
                  increment dcount[probeposition mod csize]
                  decrement dcount[recordposition mod csize]
                  k := key(Record)
                  longestprobe := max( longestprobe, probeposition )
                  probeposition := recordposition
            end
      endwhile
      while (dcount[shortestprobe mod csize] = 0) do increment shortestprobe
      m := m+1
      return(location)
end function insert
```

Figure 6.1: Robin Hood insertion algorithm when deletions may have occured

6.3 Simulation Results

We simulated Robin Hood hashing with deletions and insertions for different table sizes and load factors. Algorithm 6.2 was used for insertions and the organ-pipe algorithm for searching. Figure 6.3 shows the expected number of probes to find a key after a number of replacement operations (a deletion followed by an insertion) were applied to the table. The cost of searching increases at first, but stabilizes after a large number of replacements have been made. We conjecture that the probability distribution of the psl reaches a steady state when there are no empty locations left in the table, only occupied or deleted ones.

After the steady state has been reached the cost of searching seems to depend only on the load factor α and not on the table size, and is never greater than the

```
table : array [1..n] of RECORD   {all empty}
count : array [0..csize-1] of integer
             {all 0 except count[0]=-1 and count[1]=-1}
dcount : array [0..csize-1] of integer  {all 0 except dcount[0]=n}
shortestprobe, longestprobe, dshortestprobe, dlongestprobe: integer
n, m, tallest : integer

function insert(Record)
    if m=n then return(FAIL)    { table full }
    count[longestprobe+1] := 0
    count[shortestprobe-1] := 0
    k := Key(Record)
    probeposition := dshortestprobe
    while (k <> empty) and (k <> deleted) do
        probeposition := probeposition + 1
        location := H(k, probeposition)
        recordposition := findposition(location)
        if probeposition > recordposition then begin
            increment count[probeposition mod csize]
            decrement count[recordposition mod csize]
            increment dcount[probeposition mod csize]
            decrement dcount[recordposition mod csize]
            tempRecord := table[location]
            table[location] := Record
            Record := tempRecord
            k := Key(Record)
            longestprobe := max( longestprobe, probeposition )
            probeposition := recordposition
        end
    endwhile
    increment count[probeposition mod csize]
    dlongestprobe := max( dlongestprobe, longestprobe )
    while count[(tallest+1) mod csize]>count[tallest mod csize]
                                    increment tallest
    while count[ smallest mod csize] = 0 increment smallest
    while count[ dsmallest mod csize] = 0 increment dsmallest
    count[longestprobe+1] := -1
    count[smallestprobe-1] := -1
    m := m+1
    return(location)
end function insert
```

Figure 6.2: Robin Hood insertion keeping counters when deletions may have occured

cost of searching a full table where no replacements have been made (< 2.6).

Fig 6.4 shows the expected size of the interval of probe positions that contain all the records in the table. The size of this interval represents the cost of performing unsuccessful searches when we use either the smart search or organ-pipe heuristic. The size of the interval grows at first when replacements are done and stabilizes after a large number of them have been performed. However, the value of the interval when the steady state is reached depends not only on the load factor but also on the size of the file. For a fixed load factor the interval grows logarithmically with the file size. In any case, the size of the interval seems to be less than about $1.15 \ln n + 2.5$.

Fig 6.5 shows the expected distance of the average psl from the smallest probe position among the records in the file. This represents the expected number of probes needed in a successful search if the standard search algorithm is used. The average cost of an insertion is equal to the average position above the minimum of the records in the table plus the expected number of probes required to hit a deleted location. From this we conjecture that the cost of an insertion after the steady state has been reached is $\Theta(\ln n + (1-\alpha)^{-1})$.

6.4 Summary

In this chapter we presented a variation of the insertion algorithms to be used when deletions may occur. Simulation experiments indicate that the hash tables produced by this new algorithm retain good expected costs for both successful and unsuccessful searches. We conjectured that the cost of doing insertions into a Robin Hood table with load factor α, is $O(\ln n + (1-\alpha)^{-1})$.

As far as we know, these are the only algorithms for hash tables with open addressing in which the cost of searching remains low even if deletions may have occurred. However, the cost of performing an insertion can be high if α is close to 1. This will also be true for any other replacement algorithm since $\Theta((1-\alpha)^{-1})$ probes into the table are needed to find any deleted location. Proving that the variance of psl remains bounded after an arbitrary number of insertions and deletions have been performed remains an open problem.

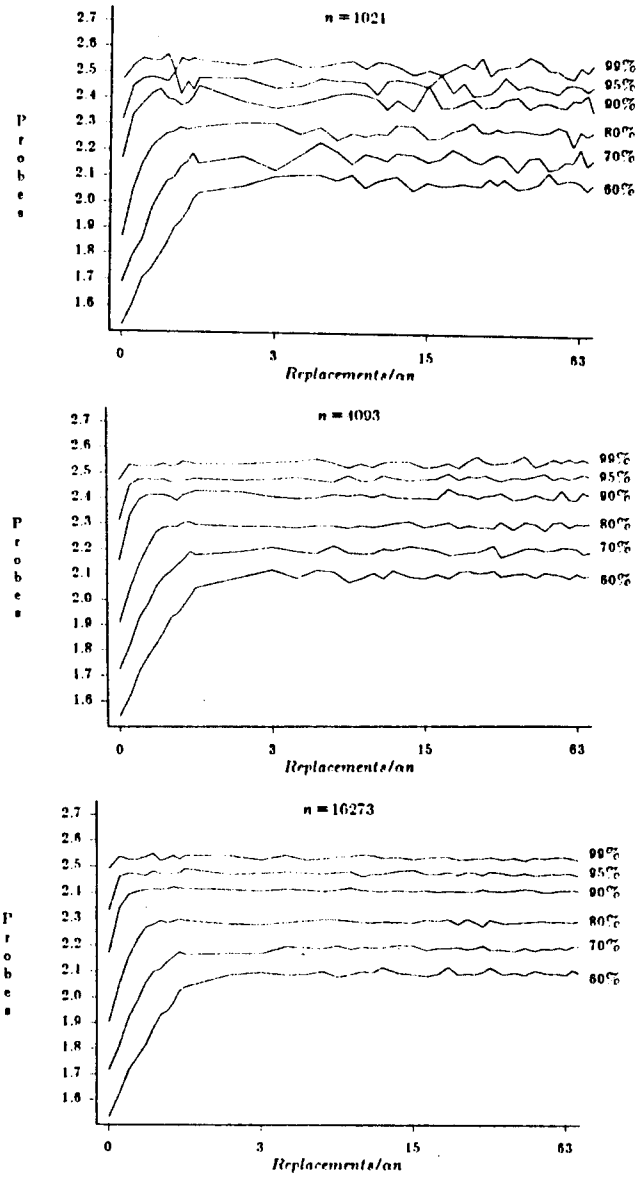

Figure 6.3: Average Cost of Successful Searches after Deletions

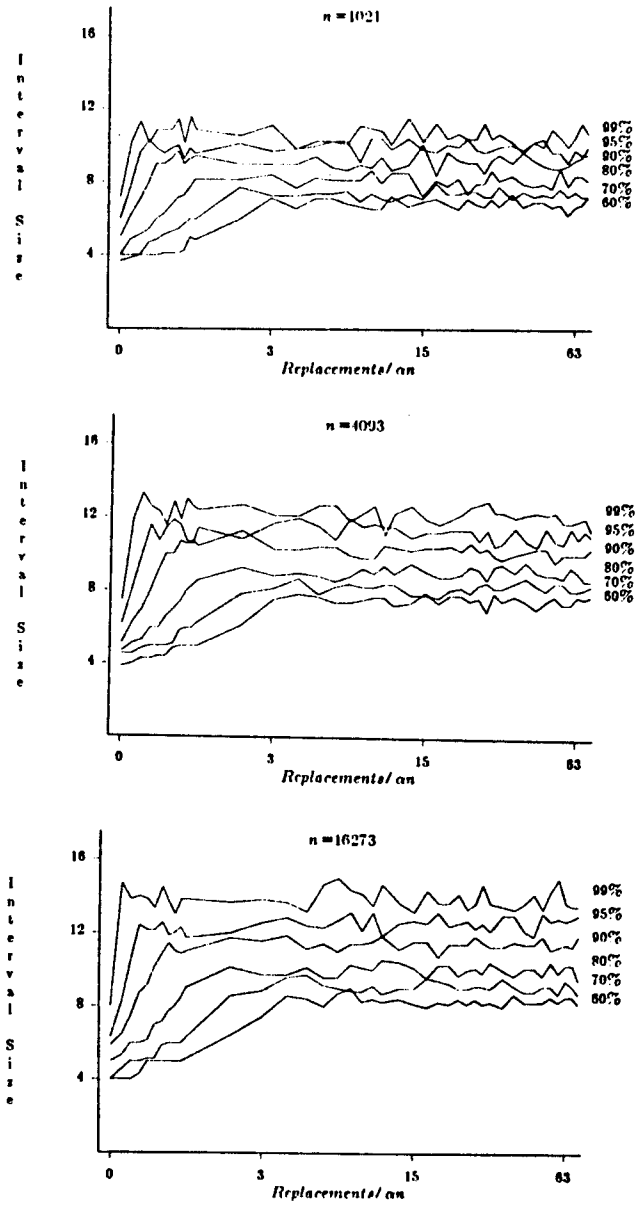

Figure 6.4: Average Cost of Unsuccessful Searches after Deletions

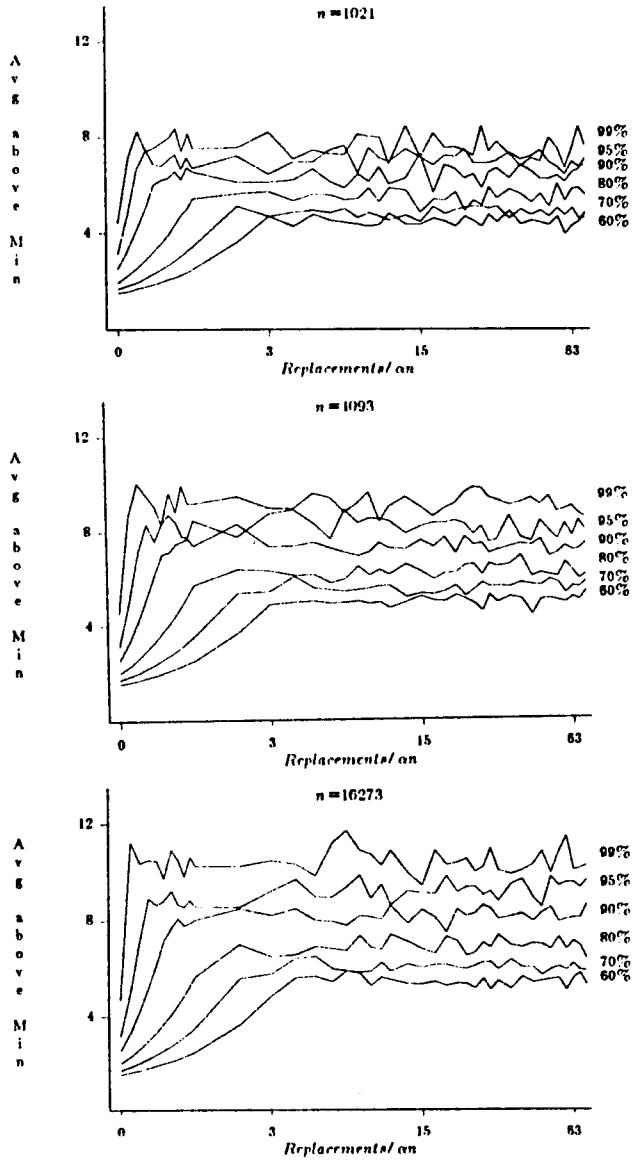

Figure 6.5: Average Probe Position above Minimum after Deletions

Chapter 7

Conclusions and Further Research

In this chapter we present our conclusions and some ideas for further research.

7.1 Conclusions

We consider the main contribution of this thesis to be the introduction of a new reordering scheme which we have called Robin Hood hashing. This new scheme is obtained by rather simple modifications to the standard insertion and search algorithms.

We were able to prove that the expected value of the probe sequence length (E[psl]) is $O(\ln n)$ and that the expected value of the longest probe sequence length (E[lpsl]) is also $O(\ln n)$ even for a full table. An asymptotic model giving the probability distribution of the probe position of a record was developed and taking the limit numerically we showed that the variance (V[psl]) is less than or equal to 1.883 for any $\alpha \leq 1 - \epsilon$, $\epsilon > 0$. The strongest feature of this new method is that the variance is reduced to a small constant . This enables us to perform successful searches in a constant number of steps on average and unsuccessful searches in $O(\ln n)$ steps.

The standard method, optimal hashing and min-max hashing minimize respectively the number of probes required to insert, perform a successful search and perform an unsuccessful search. Robin Hood hashing comes within a small constant factor of each of these three lower bounds simultaneously.

Furthermore, we presented modifications to our algorithms for handling deletions and subsequent insertions, and indicated using simulations that the cost of successful and unsuccessful searches remains low. As far as we know, no previous work has shed encouraging light on this problem for the case in which conflicts are resolved by open addressing.

Method	Insert Cost	Successful	Unsuccessful
Standard	$\ln n + O(1)$	$\ln n + O(1)$	$\approx .6315... \times n$
Brent's	$\Theta(\ln n)$†	≈ 2.49‡	$1.15\sqrt{n} + O(1)$†
Binary Tree	$\Omega(\sqrt{n})$†	≈ 2.13‡	$1.44 \ln n + O(1)$†
Optimal	$\Omega(\sqrt{n})$†	≈ 1.82†	$\Theta(\ln n)$†
Min-Max	$\Omega(\sqrt{n})$†	≈ 1.83†	$\Theta(\ln n)$† $< 2.44 \ln n + O(1)$
Smart Search	$\ln n + O(1)$	≈ 2.78‡	$1.15 \ln n + O(1)$†
Organ-Pipe		≈ 2.56‡	$< 2.44 \ln n + O(1)$

† *Conjectures based on simulation*
‡ *Obtained by numerical extrapolation*

Table 7.1: Comparison of Hashing Schemes for full tables

7.2 Further Research

Tables 7.1 and 7.2 summarize what is currently known about the performance of hashing with open addressing. The cost measures are given in number of probes. The insertion cost is equal to the total number of probes required to insert divided by the total number of keys inserted. Simulation studies show that $\Theta(n)$ extra memory is required by binary tree, optimal and min-max hashing during insertions. Organ-pipe search requires $\Theta(\ln n)$ extra memory to keep track of the observed frequencies. The search cost for Brent's, binary tree and Robin Hood hashing have been analyzed only for nonfull tables, and the numbers presented in the table are numerical extrapolations to $\alpha = 1$.

From Table 7.1 it is clear that simulation has been useful in answering many questions in this area. The obvious open problems are to prove that the conclusions based on the simulation experiments are correct. From Table 7.2 one gets the impression that the case of nonfull tables has been somewhat neglected by previous simulation studies. Even though the analysis for full tables provides a bound on the performance of the hashing scheme in question, a study of nonfull tables is also important since in reality most hash tables are not allowed to become full.

Regarding Robin Hood hashing we see the following as the main open problems:

- Finding E[lpsl]. We have proved bounds of E[psl] < E[lpsl] < 3E[psl] + $\lceil \lg m \rceil$. The bounds are reasonably tight only for full tables.
- Finding the probability distribution of psl for full tables.
- Finding the steady-state distribution and the moments of psl when deletions and subsequent insertions occur. We have provided simulation results only.

Method	Insert Cost	Successful	Unsuccessful
Standard	$-\frac{\ln(1-\alpha)}{\alpha}$	$-\frac{\ln(1-\alpha)}{\alpha}$	$-\log_\alpha n$ $-\log_\alpha(-\log_\alpha n)$ $+O(1)$
Brent's	$\Theta\left(-\frac{\ln(1-\alpha)}{\alpha}\right)$ †	$1+\frac{\alpha}{2}+\frac{\alpha^3}{4}+\frac{\alpha^4}{15}-$ $\frac{\alpha^5}{18}+\frac{2\alpha^6}{15}+\frac{9\alpha^7}{80}-$ $\frac{293\alpha^8}{5670}-\frac{319\alpha^9}{5600}+$ $\cdots$	$\Theta(\ln m)$†
Binary Tree	?	$1+\frac{\alpha}{2}+\frac{\alpha^3}{4}+\frac{\alpha^4}{15}-$ $\frac{\alpha^5}{18}+\frac{2\alpha^6}{15}+\frac{83\alpha^7}{720}-$ $\frac{613\alpha^8}{5760}-\frac{69\alpha^9}{1120}+\cdots$	?
Optimal	?	?	?
Min-Max	?	?	?
Smart Search / Organ-Pipe	$-\frac{\ln(1-\alpha)}{\alpha}$	No explicit formula but can be computed for any $\alpha < 1$	$o(\ln m)$† $< -\alpha^{-1}\ln(1-\alpha)$ $+\lceil \lg m \rceil$

† *Conjectures based on Simulation*

Table 7.2: Comparison of Hashing Schemes for nonfull tables

Throughout this thesis we have concentrated on the use of the Robin Hood heuristic for internal hash tables and most of our concerns have included execution time (e.g. the motivation for the introduction of the smart search heuristic). But Robin Hood hashing may also be a good alternative for external dictionaries, especially since it appears that deletions can be handled with no significant degradation of performance. In that case the main concern is the number of external probes. There are several modifications to our algorithms that seem appropriate when they are intended to be used for external files. One is to modify the organ-pipe heuristic to further reduce the expected number of probes required for a successful search. This could be done by stopping the search on the upper end of the probe sequence when we probe a location that contains a record with a smaller probe number, or if it is empty. Another modification is to have an internal table that contains for each bucket the value of the maximum probe position (or the maximum and the minimum) among the records in the bucket. About 4 (8) bits per bucket should be enough in most practical cases. An analysis of the distribution and the moments for psl when used with buckets would also be required, as well as an analysis of the expected value of lpsl.

Bibliography

[ABRSTE70] **Abramowitz, M. and I. Stegun,** *Handbook of Mathematical Functions,* Dover Publications, Inc., New York

[AMBKNU74] **Amble, O. and D.E. Knuth,** "Ordered Hash Tables," *The Computer Journal,* Vol. 17, No. 2, pp.135-147, May 1974

[BRE73] **Brent, R.P.,** "Reducing the Retrieval Time of Scatter Storage Techniques," *Communications of the ACM,* Vol. 16, No. 2, pp.105-109, February 1973

[DAVBAR62] **David, F.N. and D.E. Barton,** *Combinatorial Chance,* Charles Griffin & Company Limited, London 1962

[EDMKAR72] **Edmonds, J. and R.M. Karp,** "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems", *Journal of the ACM,* Vol. 19, No. 2, pp.248-264, April 1972

[ERDRÉN61] **Erdös, P. and A. Rényi,** "On a Classical Problem of Probability Theory", *Magy. Tud. Akad. Mat. Kutató Int. Közl.,* Vol. 6 pp.215-220, 1961

[FELLOW73] **Feldman, J.A. and J.R. Low,** "Comment on Brent's Scatter Storage Algorithm", *Communications of the ACM,* Vol. 16, No. 11, pp.703, November 1973

[FEL68] **Feller, W.,** *An Introduction to Probability Theory and its Applications, Vol. I,* John Wiley & Sons, New York, 1968

[FRETAR84] **Fredman, M.L. and R.E. Tarjan,** "Fibonacci Heaps and Their Use in Improved Network Optimization Algorithms", *Proc. 25th Annual IEEE Symposium on Foundations of Computer Science,* pp.338-346, October 1984

[GON77] **Gonnet, G.H.,** "Average Lower Bounds for Open Addressing Hash Coding", *A Conference on Theoretical Computer Science,* pp.159-162, University of Waterloo Waterloo, Ontario, August 1977

[GONMUN79] **Gonnet, G.H. and J.I. Munro,** "Efficient Ordering of Hash Tables", *SIAM Journal on Computing,* Vol. 8, No. 3, pp.463-478, August 1979 (a preliminary version was presented at the 9th ACM STOC May 1977)

[GON81] **Gonnet, G.H.,** "Expected Length of the Longest Probe Sequence in Hash Code Searching", *Journal of the ACM,* Vol. 28, No. 2, pp.289-304, April 1981

[GONLAR82] **Gonnet, G.H. and P.-Å. Larson,** "External Hashing with Limited Internal Storage", *Technical report CS-82-38,* Computer Science Dept., Univ. of Waterloo, October 1982

[GON84] **Gonnet, G.H.,** *Handbook of Algorithms and Data Structures,* Addison-Wesley, Reading Massachusetts 1984

[JOHKOT77] **Johnson, N.L. and S. Kotz,** *Urn Models and their Application,* John Wiley & Sons, New York 1977

[KÖN31] **König, D.** "Graphok és Matrixok", *Matematikai és Fizikai Lapok,* Vol. 38, pp.116-119, 1931

[KNU69] **Knuth, D.E.,** *The Art of Computer Programming, Vol. II: Seminumerical Algorithms,* Addison-Wesley, Reading Massachusetts, 1969

[KNU73] **Knuth, D.E.,** *The Art of Computer Programming, Vol. III: Sorting and Searching,* Addison-Wesley, Reading Massachusetts, 1973

[LAR83] **Larson, P.-Å.,** "Analysis of Uniform Hashing", *Journal of the ACM,* Vol. 30, No. 4, pp.805-819, October 1983

[LYO78] **Lyon, G.E.,** "Packed Scatter Tables", *Communications of the ACM,* Vol. 21, No. 10, pp.857-865, October 1978

[MAD80] **Madison, J.A.T.,** "Fast Lookup in Hash Tables with Direct Rehashing", *The Computer Journal,* Vol. 23, No. 2, pp.188-189, May 1980

[MAL77] **Mallach, E.G.,** "Scatter Storage Techniques: A Unifying Viewpoint and a Method for Reducing Retrieval Times", *The Computer Journal,* Vol. 20, No. 2, pp.137-140, May 1977

[MAU75] **Maurer, W.D. and T.E. Lewis,** "Hash Table Methods", *ACM Computing Surveys,* Vol. 7, No. 1, pp.5-19, March 1975

[PET57] **Peterson, W. W.,** "Addressing for Random-Access Storage", *IBM Journal of Research and Development* Vol. 1, No. 2, pp.130-146, April 1957

[POO76] **Poonan, G.,** "Optimal Placement of Entries in Hash Tables", *ACM Computer Science Conference (Abstract only),* Vol. 25, 1976, (Also DEC Internal Tech. Rept. LRD-1, Digital Equipment Corp., Maynard Mass)

[RÉN62] **Rényi, A.,** "Three New Proofs and a Generalization of a Theorem of Irving Weiss", *Magy. Tud. Akad. Mat. Kutató Int. Közl.,* Vol. 7 pp.200-209, 1962

[RIV78] **Rivest, R.L.,** "Optimal Arrangements of Keys in a Hash Table", *Journal of the ACM,* Vol. 25, No. 2, pp.200-209, April 1978

[ULL72] **Ullman, J.D.,** "A Note on the Efficiency of Hash Functions", *Journal of the ACM,* Vol. 19, No. 3, pp.569-575, July 1972

[YAO85] **Yao, A.C.,** "Uniform Hashing is Optimal", *Journal of the ACM,* Vol. 32, No. 3, pp.687-693, July, 1985