REALIZATION OF OPEN ADDRESSING HASH TABLE IN THE CHAINED ALLOCATED MEMORY

Valentina Dyankova and Rositza Hristova

Abstract: *In this article, we examine a realization of an open addressing hash table in the chained allocated memory, giving us the opportunity to decrease the number of linear probing when a given element has not been inserted in the table*

Keywords: *open addressing hash table, collision, search for an element, deleting of an element*

Introduction

The extraction of the particular piece or pieces of data from a previously stored huge volume of data is a fundamental operation called searching. It is an undividable part of a set of real tasks. Usually, the goal of searching is to gain access to the data, which is contained in an element, and to move on the next process or task. Applications of the searching method are widely distributed and they include different sets of operations. Hash table is a fundamental and widely used data structure implementing fast searching algorithms. At the realization point of the structure, different approaches could be used to adapt in a better way the requirements of the speed efficiency, used memory, etc.

In the current article, we examine a realization of an open addressing hash table in the chained allocated memory. It gives us the opportunity to reduce significantly the number of linear probing when we determine the fact that a particular element is not included in the hash table as well as speed up the resulting in success process of searching.

Open Addressing Hash Table Concepts

A hash table represents an aggregation of elements, each of which has a key (identification part) and a body (data part). The value of the key identically differ it from the rest of the elements. From an organizational and processing view, the values of the elements' data part in the hash table are not of primary importance. That is the reason why we do not examine a concrete defined type of the proposed hash table. The access to an element in the hash table is implemented by transforming the element's key in its address. Consequently, searching could be presented as an image $hash : K \rightarrow A$, called hash table. Because of the representation of the hash table in the memory, the transformation of the key is taken down to transforming the index of the array. In this way, if N is the total number of elements in the array, then $hash : K \rightarrow$ *{0, 1, 2, ..., N -1}*

Choosing a well-functioning and efficient hash table is a guarantee of uniformly distributed elements of the hash table in the array, but that is not a purpose of this article. The classic hash function -- $hash(k) = k\%N$, where $k \in K$ and k is the element key – will be used without considering the fact that elements' keys are natural numbers.

In this way, the power of set K is greater than the possible number of elements in the array and this might cause the situation where two elements with different keys, $k_1 \neq k_2$, pretend to occupy the same location in the array, $hash(k_1) = hash(k_2)$. Such elements are called synonyms, and the phenomenon – collision.

The problem of solving collisions has different solutions, but from all of them, we will examine the method of linear open addressing. In this method, if an element pretends to be placed at a position that has been already occupied by another element then the array is scanned in order for an open position. Sequential search would have been realized with the function $rehash(i) = (i+1)\%N$. The effect that takes place when solving the collisions in the open addressing table is called primary clustering. The elements that cause the collision (with the same value of the hash function) are located sequentially with respect to the order of their entries. Then the elements that are about to be placed in the array also cause collisions since their original place in the array has been already taken.

The case in which elements with different assigned values from the hash function pretend to the same index of the array is called secondary clustering. The gathered roles of elements in the hash table are called clusters.

The basic operations with hash tables and standard algorithms for their implementation are:

Searching for an element with key k – the array index a_1 under which an element must be found is calculated with the help of the hash function $hash(k) = a_1$; but if there is another element with the same index, the calculation for a position is done by the statement $a_2 = rehash(a_1)$. Positioning of a_2 is similar. This process of linear probing continues till: an element with key k is found (successful end); an element with a key equal to null is found (unsuccessful end), or an element with a particular key does not exist (unsuccessful end) when the hash table has been scanned.

Inserting an element with a key k – if it has been determined that an element with such key is not in the hash table, then it is inserted.

Deleting of an element with key k – the element's key is given a null value at position i after a success is returned from executing the operation, searching of an element with key k, and determining its location at position i.

Problem Solving

In the classic literature, the question about deleting an element from an open addressing table is either not mentioned [Амерал, 2001], [Рейнголд, 1980], [Мейер, 1982], or mentioned in one of the following ways:

- The implementation of the operation is possible, but it is very complex [Амерал, 2001].
- Marking of an element by the obvious method breaks the chain of synonyms [Наков, 2002], [Шишков, 1995].
- Entering a specification with tree conditions: the element is filled; the element is empty, and it has never been filled; the element is empty, but has been previously filled [Шишков,1995], [Смит,2001], [Sedgewick,1998].
- The chained synonyms in the array are rearranged, so that the elimination of an element has no effect on the searching or inserting algorithms [Шишков, 1995].
- Secondary hashing all elements between the deleted element and the next available position [Sedgewick,1998].

Two implementations of deleting an element are examined -- the obvious method, [Азълов, 1995] and secondary hashing method, [Sedgewick, 1998].

So, the way the searching operation has been given to us and our previous comments arise the following questions:

- How does deleting of an element from the chained linear probes reflect on element, which is a member of this chain?
- Does the case that a null key in consecutive linear probes is being reached give us the opportunity to state that the element in interest is not found?
- Is it necessary to search the table until a never filled element is reached in order to be determined that the element in interest does not exist?

Example

An illustration of the raised question is the following example: in the hash table of size 13 are inserted the elements with keys

- *14 (hash(14)=14%13=1);*
- *16 (hash(16)=16%13=3);*
- *29 (hash(29)=29%13=3, rehash(3)=4%13=4);*
- *55 (hash(55)=55%13=3, rehash(3)=4%13=4, rehash(4)=5%13=5);*
- *21 (hash(21)=21%13=8);*
- *35 (hash(35)=35%13=9);*
- *49 (hash(49)=49%13=10);*
- *50 (hash(50)=50%13=11).*

0	1	2	3	4	5	6	7	8	9	10	11	12
0	14	0	16	29	55	0	0	21	35	49	50	0

Applying the widely used in the literature algorithm for searching elements with keys *14, 20, 55, 42, 48*, the following results and questions are obtained:

- About *14:* index 1 is received by using the hash function. It stores the key that we are looking for;
- About *20:* index 7 is received by using the hash function. It stores the null key. This shows that the place is empty and the result of searching is unsuccessful.
- About *55:* Applying the hash function, we receive index 3, where is stored key $16 \neq 55$.. Following the classical algorithm (linear probing is executed until an element with a given key is reached or en empty space in the array), we apply the method of linear probing twice to receive the key *55*.
- About *42:* Applying the hash function, we receive index 3, where is stored key $16 \neq 42$. Exercising the method of linear probing three times, we reach key 0. Searching ends up unsuccessful.
- About *48:* Applying the hash function, index 9 is received. It stores a key $35 \neq 48$. Using the method of linear probing three times we receive key 0. It is an unoccupied space and searching ends up unsuccessful. This case raises the following question: Is it possible at the time of receiving index 9 to conclude that searching exits with failure. The latter follows from the fact that collision did not occur at position 9 as we continue to insert the elements. So there is no doubt that the element we are searching for will not appear in the consecutive linear probing for this position.

An illustration of the questions 2.1 and 2.2 is the situation when we search for element with key *55* after we have previously deleted element with key *29.* Locations of the elements after the deleting are:

0	1	2	3	4	5	6	7	8	9	10	11	12
0	14	0	16	0	55	0	0	21	35	49	50	0

Then, applying the searching algorithm for element with key *55* and going through the elements, index 4 is reached. It stores key 0 i.e. the location is empty or according to the classical algorithm (searching until a given key is found or an empty location is reached), searching would have finished unsuccessful. This contradicts the fact that collision occurred at position 9 and if it has not contained the key of our interest, then we continue with the linear probing until we reach the key of interest (successful end) or to position at which no collision has occurred (unsuccessful end)

Solution to the Problem

The basic problem to which we offer a solution in this article is the implementation of a searching algorithm for an element inserted in open addressing hash table. The algorithm reduces the number of linear probing as long as a particular element has not been inserted in the table. Excluding an element from the table has no harmful effects on the algorithms for searching and inserting an element in the inner chains of synonyms when collisions are being solved. On the other hand, the use of the inefficient “garbage collector” is not necessary.

The current article gives a solution to this problem, as data (recording occurring of a collision at a particular place when we inserted the elements in the hash table) is stored for each position. This leads to the idea of using alternative array *ph* from Boolean values:

$$ph[i] = \begin{cases} true \\ false \end{cases}$$

Then, the location of the examined elements above will modify both arrays:

- *14 (hash(14)=14%13=1):*

	0	1	2	3	4	5	6	7	8	9	10	11	12
t	0	14	0	0	0	0	0	0	0	0	0	0	0
ph	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*

- *16 (hash(16)=16%13=3):*

	0	1	2	3	4	5	6	7	8	9	10	11	12
t	0	14	0	16	0	0	0	0	0	0	0	0	0
ph	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*

- *29 (hash(29)=29%13=3; rehash(3)=4%13=4);*

	0	1	2	3	4	5	6	7	8	9	10	11	12
t	0	14	0	16	29	0	0	0	0	0	0	0	0
ph	*false*	*false*	*false*	*true*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*

- *55 (hash(55)=55%13=3, rehash(3)=4%13=4, rehash(4)=5%13=5):*

	0	1	2	3	4	5	6	7	8	9	10	11	12
t	0	14	0	16	29	55	0	0	0	0	0	0	0
ph	*false*	*false*	*false*	*true*	*true*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*

- *21 (hash(21)=21%13=8):*

	0	1	2	3	4	5	6	7	8	9	10	11	12
t	0	14	0	16	29	55	0	0	21	0	0	0	0
ph	*false*	*false*	*false*	*true*	*true*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*

- *35 (hash(35)=35%13=9):*

	0	1	2	3	4	5	6	7	8	9	10	11	12
t	0	14	0	16	29	55	0	0	21	35	0	0	0
ph	*false*	*false*	*false*	*true*	*true*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*

- *49 (hash(49)=49%13=10):*

	0	1	2	3	4	5	6	7	8	9	10	11	12
t	0	14	0	16	29	55	0	0	21	35	49	0	0
ph	*false*	*false*	*false*	*true*	*true*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*

- *50 (hash(50)=50%13=11):*

	0	1	2	3	4	5	6	7	8	9	10	11	12
t	0	14	0	16	29	55	0	0	21	35	49	50	0
ph	*false*	*false*	*false*	*true*	*true*	*false*	*false*	*false*	*false*	*false*	*false*	*false*	*false*

At these current states of both arrays, searching of an element with a key *48* will lead to application of the hash function before key *48 - hash(48)=48%13=9.* This location is not occupied by the element with the key of interest and $ph[i] = false$, so that applying the method of linear probing is not necessary and the conclusion for an unsuccessful searching end could be drawn.

Program Implementation

Implementation of the proposed solution (language C++|):

```
const int nilkey=0;
template<class T>
struct element {int key; T info;};
template<class T>
class hashtable
{ private:
       int tabsize;
       int free;
       element* t;
       bool* ph;
  public:
       hashtable();
       hashtable (int n);
       bool is_full();
       int search (int k);
       void insert (element e);
       void del (int k);
};
template<class T>
hashtable<T>::hashtable()
{ tabsize = 0; free=0; }
template<class T>
hashtable<T>::hashtable(int n)
{ tabsize = n; free=n;
  t = new element[tabsize];
  ph = new bool[tabsize];
  for (int i=0; i < tabsize; i++)
      { t[i].key=nilkey; ph[i]=false; }
}
```

```
int h(int k)  { return k%tabsize; }
int r(int i)  { return (i+1)%tabsize; }
template<class T>
bool hashtable<T>::is_full()
{ return free==0; }
template<class T>
int hashtable<T>::search( int k)
{ bool b=false; int i=h(k); int j=i;
  while ( t[i].key!=k && ph[i] && !b )
        { i=r(i); b = i==j; }
  if (t[i].key==k) return i;
  else return –1;
}
template<class T>
void hashtable<T>::insert( element E)
{ int i=search(E.key);
  if ( i<0 && !is_full() )
       { i=h( E.key);
         while    (    t[i]!=nilkey    )    {
ph[i]=true; i=r(i); }
         t[i]=E; free--;
}      }
template<class T>
void hashtable<T>::del( int k)
{ int n=search(k);
  if (n>=0) { t[n].key=nilkey; free++; }
}
```

Result Analysis

The number of linear probing in the open addressing table at the time the search is done depends on:

1. The ratio $\alpha = M/N$, where M is the number of the stored elements in the table vs. N , the total number of elements in the table. In the incomplete table (small α), it is expected most of the searches to end up in several probing. In contrast, when the table is almost complete (α has a value close to *1*) , searching could require a big number of linear probes.
2. The way of generating clusters in the hash table. The observation shows that the average number of linear probing resulted in unsuccessful searching is proportional to squares of clusters' length. The successful searches are always cheaper (less in number probes) than the unsuccessful ones.

The grades on number of linear probing resulted in successful search (1) and unsuccessful one (2) are given by D. Knut [Кнут, 1978] when the following stages are performed:

- Grades are pessimistic and based on the fact that an element's key k can appear at any moment.
- Grades loose their precision when α is close to *1*.

$$S(\alpha) = \frac{1}{2}(1 + \frac{1}{1-\alpha}) \quad (1) \qquad\qquad U(\alpha) = \frac{1}{2}(1 + \frac{1}{(1-\alpha)^2}) \quad (2)$$

Table 1 gives us different values of α in order: average number of probes necessary to successfully find an element using the formula (1) that calculates $S(\alpha)$; the average value received by implementing the classic algorithm in the program and the average value received by the given algorithm in this article.

N	*α*	0.25	0.50	0.66	0.75	0.83	0.90	0.95
	S(α)	1.167	1.5	1.971	2.5	3.441	5.5	10.5
1009	Classic algorithm	1.132	1.325	1.566	2.217	3.153	5.31	8.739
1009	Proposed implementation	1.132	1.325	1.566	2.217	3.153	5.31	8.739
10007	Classic algorithm	1.126	1.419	1.682	2.095	2.751	4.168	7.047
10007	Proposed implementation	1.126	1.419	1.682	2.095	2.751	4.168	7.047
100003	Classic algorithm	1.121	1.367	1.725	2.123	2.751	4.318	8.257
100003	Proposed implementation	1.121	1.367	1.725	2.123	2.751	4.318	8.257

Table 1.

We see that when the keys are equally distributed, the results are expected to be better than the average possible. We have proved that both methods' efficiency is the same.

Using formula (2), the grade we receive is more expensive (average number of probing is greater) than the grade received by using formula (1). The suggested algorithm improves the expected number of probes when the element with a particular value for us is not in the hash table. Table 2 represents the different values of α in order: average number of probes necessary for unsuccessful searching in the hash table and calculated using formula (2); the received result is interpreted as a mean value of the program implementation using the classic approach and the mean value generated by the examined algorithm in this article.

N	*α*	0.25	0.50	0.66	0.75	0.83	0.90	0.95
	U(α)	1.389	2.5	4.825	8.5	17.80	50.5	200.5
1009	Classic algorithm	1.368	2.324	4.169	7.655	19.858	39.836	177.844
1009	Proposed implementation	1.045	1.289	2.231	5.236	16.123	34.688	163.653
10007	Classic algorithm	1.362	2.40	4.071	7.335	16.929	34.466	119.056
10007	Proposed implementation	1.033	1.417	2.13	4.409	12.368	26.784	97.504
100003	Classic algorithm	1.381	2.354	4.322	7.127	14.018	39.283	175.473
100003	Proposed implementation	1.042	1.367	2.403	4.247	9.207	30.851	151.717

Table 2.

We see from Table 2 that the suggested implementation influences the number of comparisons when searching is unsuccessful. When α is closed to $1/2$ (that is preferable loading of the open addressing hash table), the obtained result is comparable with $S(\alpha)$.

Following tables 3 and 4 indicate the statistics resulted by counting $U(\alpha)$. Counting is done in three ways:

N	α	0.25	0.50	0.66	0.75	0.83	0.90	0.95
1009	Classic algorithm	1.366	2.008	2.834	3.968	5.668	9.854	21.72
1009	Proposed implementation	1	1	1	1	1	1	1
10007	Classic algorithm	1.322	2.043	2.923	3.963	5.72	10.622	20.245
10007	Proposed implementation	1	1	1	1	1	1	1
100003	Classic algorithm	1.347	2.003	3.009	4.159	5.783	10.355	20.889
100003	Proposed implementation	1	1	1	1	1	1	1

Table.3

a) Table 3 shows the statistics obtained when there are no collisions in the hash table. In contrast, determining of a missing key with the suggested algorithm is performed with one comparison. This much is necessary for a successful searching of an element. Look, that the values of α are closed to *1*, which is acceptable to this kind of hash table, is important to the improvement of the classic algorithm.

b) Table 4 contains statistics showing that the set of keys K exceeds 10 times the size of the hash table N and the keys are normally distributed (the average length of the chains of elements is less than 10 elements). In this way, finding of a missing key with the suggested algorithm of loading the table $\alpha < 0.85$ (preferable for open addressing tables) is accomplished by less than $1/3$ linear probing.

N	α	0.25	0.50	0.66	0.75	0.83	0.90	0.95
1009	Classic algorithm	1.388	2.466	4.396	6.206	13.624	32.008	158.326
1009	Proposed implementation	1.034	1.366	2.492	3.778	9.004	22.446	151.369
10007	Classic algorithm	1.377	2.472	4.407	8.255	15.096	46.891	174.926
10007	Proposed implementation	1.052	1.461	2.48	5.55	10.206	36.688	143.522
100003	Classic algorithm	1.390	2.413	4.606	7.88	15.509	45.809	196.747
100003	Proposed implementation	1.049	1.427	2.702	4.929	10.819	36.568	182.90

Table.4

Conclusion

Suggested implementation of hash table in the consecutive allocated memory can be used with any hash functions and any way of processing collusions. In any case, the number of probing is reduced drastically when the result of searching an element is unsuccessful. The operations, adding or deleting of an element, directly or indirectly perform searching of an element and their speed is also improved. Additional advantage is that the use of "garbage collector" is not necessary in deleting an element. The proposed application of the algorithm for linear searching is preferable to applications using frequent execution of the operation unsuccessful search and capricious of used memory.

Literature

[Азълов, 1995] П.Азълов. Програмиране. Основен курс. Изд. АСИО, София, 1995.
[Амерал, 2001] Л.Амерал. Алгоритми и структури от данни в С++. ИК СОФТЕХ, 2001.
[Careson] D.Careson. Software Design Using C++. An Online Book, http://cis.stvincent.edu/swd
[Кнут, 1978] Д.Кнут. Искусство програмирования для ЭВМ, т.3. Сортировка и поиск. Изд. Мир, Москва, 1978.
[Мейер, 1982] Б.Мейер, К.Бодуэн. Методы программирования. т.2. Изд. Мир, Москва, 1982.
[Наков, 2002] П.Наков, П.Добриков. Програмиране=++Алгоритми; Изд.TopTeam Co., София,2002
[Sedgewick, 1998] R.Sedgewick. Algorithms in C, Part 4: Searching, Adddison Wesley, 1998
[Смит, 2001] Т.М.Смит. Принципи и методи на програмирането с PASCAL. Изд. Техника, София, 2001
[Рейнголд, 1980] Э.Рейнголд,Ю.Нивергельт,Н.Део. Комбинаторные алгоритмы. Теория и практика. Изд.Мир, М., 1980
[Шишков, 1995] Д.Шишков и др. Структури от данни. Изд.Интеграл, Добрич, 1995

Authors' Information

Valentina S. Dyankova – e-mail: vspasova@yahoo.com
Rositza P. Christova – e-mail: r.hristova@fmi.shu.bg.net
Shumen University "Ep. Konstantin Preslavsky", Faculty of Mathematics and Informatics; Str. "Universitetska" 115, Shumen, 9712; Bulgaria