Data Structures and Algorithms (10)

Instructor: Ming Zhang
Textbook Authors: Ming Zhang, Tengjiao Wang and Haiyan Zhao

Higher Education Press, 2008.6 (the "Eleventh Five-Year" national planning textbook)

https://courses.edx.org/courses/PekingX/04830050x/2T2014/

Chapter 10. Search

- 10.1 Search in a list
- 10.2 Search in a set
- 10.3 Search in a hash table
- Summary

Search in a Hash Table

- 10.3.0 Basic problems in hash tables
- 10.3.1 Collision resolution
- 10.3.2 open hashing
- 10.3.3 closed hashing
- 10.3.4 Implementation of closed hashing
- 10.3.5 Efficiency analysis of hash methods

Implementation of Closed Hashing

Dictionary

- A special set consisting of elements which are two-tuples (key, value)
  - The keys should be different from each other (in a dictionary)
- Major operations are insertions and searches according to keys
  - **bool hashInsert(const Elem&);**

    // insert(key, value)
  - **bool hashSearch(const Key& , Elem&) const;**

    // lookup(key)

ADT of Hash Dictionaries (attributes)

```
template <class Key , class Elem , class KEComp , class
EEComp> class hashdict
{
private:
    Elem* HT;                   // hash table
    int M;                      // size of hash table
    int currcnt;                // current count of elements
    Elem EMPTY;                 // empty cell
    int h(int x) const ;        // hash function
    int h(char* x)const ;       // hash function for strings
    int p(Key K , int i)        // probing function
```

ADT of Hash Dictionaries (methods)

```
public:
hashdict(int sz , Elem e) {                    // constructor
   M=sz;  EMPTY=e;
   currcnt=0;  HT=new Elem[sz];
   for (int i=0; i<M; i++) HT[i]=EMPTY;
}
~hashdict() { delete [] HT; }
bool hashSearch(const Key& , Elem&) const;
bool hashInsert(const Elem&);
Elem hashDelete(const Key& K);
int size() { return currcnt; }                 // count of elements
};
```

Insertion Algorithm

hash function h, assume k is the given value

- If this address hasn't been occupied in the table, insert the record waiting for insertion into this address
- If the value of this address is equal to K, report "hash table already have this record"
- Otherwise, you can probe the next address of probing sequence according to how to handle collision, and keep doing this.
  - Until some cell is empty (can be inserted into)
  - Or find the same key (no need of insertion)

Code of Hash Table Insertion

```
// insert the element e into hash table HT
template <class Key, class Elem, class KEComp, class EEComp>
bool hashdict<Key, Elem, KEComp, EEComp>::hashInsert(const Elem& e) {
    int home= h(getkey(e));                      // home save the base address
    int i=0;
    int pos = home;                              // Start position of the probing sequence
    while (!EEComp::eq(EMPTY, HT[pos])) {
        if (EEComp::eq(e, HT[pos])) return false;
        i++;
        pos = (home+p(getkey(e), i)) % M;        // probe
    }
    HT[pos] = e;                                 // insert the element e
     return true;
}
```

Search Algorithm

- Similar to the process of insertion
  - Use the same probing sequence
- Let the hash function be h, assume the given value is K
  - If the space corresponding to this address is not occupied, then search fails
  - If not, compare the value of this address with K, if they are equal, then search succeeds
  - Otherwise, probe the next address of the probing sequence according to how to handle collision, and keep doing this.
    - Find the equal key, search succeeds
    - Haven't found when arrive at the end of probing sequence, then search fails

10.3 Search in a Hash Table

```
template <class Key, class Elem, class KEComp, class EEComp>
bool hashdict<Key, Elem, KEComp, EEComp>::
hashSearch(const Key& K, Elem& e) const {
    int i=0, pos= home= h(K);                    // initial position
    while (!EEComp::eq(EMPTY, HT[pos])) {
        if (KEComp::eq(K, HT[pos])) {            // have found
            e = HT[pos];
            return true;
        }
        i++;
        pos = (home + p(K, i)) % M;
    } // while
    return false;
}
```

Deletion

- Something to consider when delete records:
  - (1) The deletion of a record mustn't affect the search later
  - (2) The storage space released could be used for the future insertion
- Only open hashing (separated synonyms lists) methods can actually delete records
- Closed hashing methods can only make marks (tombstones), can't delete records actually
  - The probing sequence would break off if records are deleted. Search algorithm "until an empty cell is found (search fails)"
  - Marking tombstones increases the average search length

Problems Caused by Deletions

0	1	2	3	4	5	6	7	8	9	10	11	12
	K1	K2	K1		K2	K2	K2			K2		

- For example, a hash table of length M = 13, let keys be *k*1 and *k*2, h(*k*1) = 2 , h(*k*2) = 6。
- Quadratic probing
  - The quadratic probing sequence of *k*1: 2、3、1、6、11、11、6、5、12、...
  - The quadratic probing sequence of *k*2: 6、7、5、10、2、2、10、9、3、...
- Delete the record at the position 6, put the element in the last position 2 of *k*2 sequence instead, set position 2 to empty
- search *k*1, but fails (may be put at position 3 or 1 in fact)

Tombstones

- Set a special mark bit to record the cell status of the hash table
  - Be occupied
  - Empty
  - Has been deleted
- The mark to record the status of has been deleted is called tombstone
  - Which means it was occupied by some record ever
  - But it isn't occupied now

Deletion Algorithms with Tombstones

```
template <class Key, class Elem, class KEComp, class EEComp>Elem
hashdict<Key,Elem,KEComp,EEComp>::hashDelete(const Key& K)
{   int i=0,  pos = home= h(K);                  // initial position
    while (!EEComp::eq(EMPTY, HT[pos]))  {
        if (KEComp::eq(K, HT[pos])){
            temp = HT[pos];
            HT[pos] = TOMB;                      // set up tombstones
             return temp;                        // return the target
        }
        i++;
        pos = (home + p(K, i)) % M;
    }
    return EMPTY;
}
```

Insertion Operation with Tombstones

- If a cell marked as a tombstone is met at the time of insertion, can we insert the new record into this cell?
  - In order to avoid inserting two same keys
  - The process of search should carry on along the probing sequence, until find a real empty cell

An Improved Version of Insertion Operation with Tombstones

```
template <class Key, class Elem, class KEComp, class EEComp>
bool hashdict<Key, Elem, KEComp, EEComp>::hashInsert(const
Elem &e)  {
    int insplace, i = 0, pos = home = h(getkey(e));
    bool tomb_pos = false;
    while (!EEComp::eq(EMPTY, HT[pos])) {
        if (EEComp::eq(e, HT[pos])) return false;
        if (EEComp::eq(TOMB, HT[pos]) &&  !tomb_pos)
        {insplace = pos;  tomb_pos = true;}            // The first
        pos = (home + p(getkey(e), ++ i)) % M;
    }
    if (!tomb_pos) insplace=pos;                        // no tombstone
      HT[insplace] = e;    return true;
}
```

Efficiency Analysis of Hash Methods

- Evaluation standard: the number of record visits needed for insertion, deletion, search
- Insertion and deletion operation of hash tables are both based on search
  - Deletion: must find the record at first
  - Insertion: must find until t the tail of the probing sequences, which means need a failed search for the record
    - For the situation without consideration about deletion, it is the tail cell.
    - For the situation with consideration about deletion, also need to arrive at the tail to confirm whether there are repetitive records

Important Factors Affecting Performance of Search

- Expected cost of hash methods is related to the load factor
- α= N/M
    - When α is small, the hash table is pretty empty, it’s easy for records to be inserted into empty base addresses.
    - When α is big, inserting records may need collision resolution strategies to find other appropriate cells
- With the increase of α, more and more records may be put further away from their base addresses

Analysis of Hash Table Algorithms (1)

- The probability of base addresses being occupied is α
- The probability of the i-th collision occurring is

$$\frac{N(N-1)\cdots(N-i+1)}{M(M-1)\cdots(M-i+1)}$$

- If N and M are both very large, then it can be expressed approximately as

$$(N/M)^i$$

- The expected value of the number of probing is 1, plus occurring probability of each the i-th ($i \geq 1$) collision, which is cost of inserting, :

$$1+\sum_{i=1}^{\infty}(N/M)^i = 1/(1-a)$$

Analysis of Hash Table Algorithms (2)

- A cost of successful search (or deletion) is the same as the cost of insertion
- With the increase of the number of records of hash tables, α also get larger and larger
- We can get the average cost of insertion (the average of the cost of all the insertion) by computing the integral from 0 to current value of α

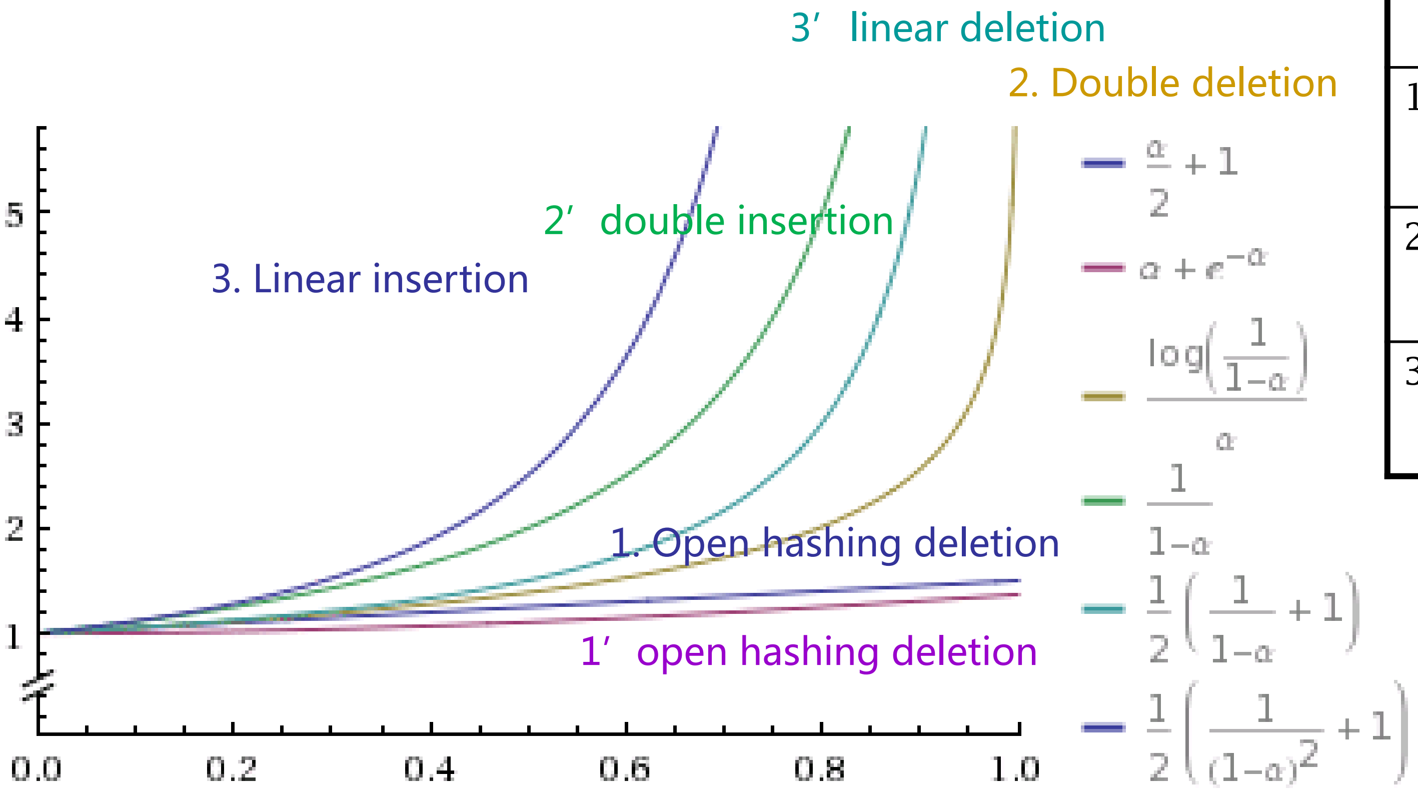
$$\frac{1}{a}\int_0^a \frac{1}{1-x}dx = \frac{1}{a}\ln\frac{1}{1-a}$$

Hash Table Algorithms Analysis (table)

No.	Collision resolution strategy	Successful search (deletion)	Failed search (insertion)
1	Open hashing	$1+\frac{\alpha}{2}$	$\alpha+e^{-\alpha}$
2	Double hashing	$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$	$\frac{1}{1-\alpha}$
3	Linear probing	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$

Hash Table Algorithms Analysis (diagram)

- ASLs of using different way to resolve collision in hash tables



No.	Collision resolution strategy	Successful search (deletion)	Failed search (insertion)
1	Open hashing	$1+\frac{\alpha}{2}$	$\alpha+e^{-\alpha}$
2	Double hashing	$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$	$\frac{1}{1-\alpha}$
3	Linear probing	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$

Conclusion of Hash Table Algorithms Analysis

- Normally the cost of hash methods is close to the time of visiting a record. It is very effective , greatly better than binary search which need log *n* times of record visit
  - Not depend on n, only depend on the load factor $\alpha=n/M$
  - With the increase of α, expected cost would increase too
  - When α ≤ 0.5, The excepted cost of most operations is less than 2 (someone say 1.5)
- The practical experience indicates that the critical value of the load factor α is 0.5 (close to half full)
  - When the load factor is bigger than this critical value, the performance would degrade rapidly

Conclusion of Hash Table Algorithms Analysis (2)

- If the insertion or deletion of hash tables is complicated, then efficiency degrades
  - A mass of insertion operation would make the load factor increases.
    - Which also increase the length of synonyms linked chains, and also increase ASL
  - A mass of deletion would increase the number of tombstones.
    - Which increase the average length from records to their base addresses
- In the practical application, for hash tables with frequent insertion or deletion, we can perform rehashing for hash tables regularly
  - Insert all the records to another new table
    - Clear tombstones
    - Put the record visited most frequently on its base address

Thinking

- Can we mark the status of empty cell and having been deleted as a special value, to distinguish them from "occupied" status?
- Survey implementation of dictionary other than hash tables.

Data Structures and Algorithms

Thanks

the National Elaborate Course (Only available for IPs in China)
http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

Ming Zhang, Tengjiao Wang and Haiyan Zhao

Higher Education Press, 2008.6 (awarded as the "Eleventh Five-Year" national planning textbook)