Searching

Objectives:

- define the searching problem
- discuss searching in lists
- searching in tables using hashing

Definition and terminology

Definition. Assume $k_1, k_2, \ldots, k_n$ is a collection distinct keys and

$$R = \{(k_1, I_1), (k_2, I_2), \ldots, (k_n, I_n)\}$$

is a collection of records (where I_j is the information stored in record (k_j, I_j)) containing those keys.

Given a key value K, **the search problem** is to locate a record (k_j, I_j) such that $K = k_j$.

Successful search - a record with key $k_j = K$ was found

Unsuccessful search - no record with key $k_j = K$ was found

General search approaches

1. Sequential methods

- the records are considered one at a time according to a predefined ordering

2. Direct access methods

- the records are accesses directly based on the value of the search key

3. Indexing methods

- keys are organized into some tree structure which allows fast searching

To describe search methods we shall use the following record structure definition:

```
struct record{
    key   k;
    info  I;
}
```

Searching arrays

The problem will be to find a record with key value K in an array

```
record  A[n];
```

1. Sequential search

```
info SeqSearch (record* A, int n,key K){
    for (i = 0 ; i < n ; i++)
        if (A[i].k == K)
            return A[n].I;
    return NOT_FOUND;
}
```

- sequential each takes $\Theta(n)$ time on the average

2. Binary Search

- if the records are sorted

```
info BinSearch (record* A, int n,key K){
    if (n == 1)
        if (A[0].k == K)
            return A[0].I;
        else
            return NOT_FOUND;
    int mid = n/2;
    if (A[mid].k == K)
        return A[mid].I;
    else
        if (K < A[mid].k)
            return BinSearch(&A[0], mid , K);
        else
            return
              BinSearch(&A[mid+1], n-mid-1, K);
}
```

- takes $\Theta(n)$ time on the avg.

3. Dictionary search

- if the records are sorted and the expected distribution of key values is known
- the location of the key in the key range is translated into an expected value in the array

Example. Assume that the keys are integer numbers uniformly distributed between m_1 and m_2.

A key k should be at index

$$\frac{k - m_1}{m_2 - m_1} \cdot n$$

- is faster than binary search

Searching lists

- the only way to search lists is sequential search
- to improve the performance of sequential search we may dynamically **reorganize the list based on the frequency of accesses** to the different records: *move the more frequently accessed records towards the head of the list*
- if $p_1, p_2, \ldots, p_n$ $(p_i > p_{i+1})$ are the probabilities with which the records are accessed:

$$T(n) = 1p_1 + 2p_2 + \ldots + np_n$$

Example. If each record has the same probability to be searched for:

$$T(n)) = \sum_{i=1}^{n} \frac{i}{n} = \frac{(n+1)}{2}$$

- in most of the cases these probabilities are not known - we try to approximate them
- use self-organizing list - lists that move the records in the list according to the previous accesses

Heuristics for managing self-organizing list:

1. approximate the probabilities **using the number of previous** accesses
   - keep a access counter associated with each record
   - move a record forward if the associated counter becomes greater than the one of the preceding record

Disadvantages

- requires extra space
- changes of frequency poorly handled

2. every time a record is used, **move** it **to the front** of the list
   - it is efficient
   - handles well changes in frequency
   - hard to implement if list is represented in an array
3. every time a record is used, **swap** it **with the record preceding** it (transpose)
   - works well in general
   - there are some patological cases
   - easy to implement with arrays and linked lists

Hashing

- accessing a record in an array by mapping a key value to a position in the array
- the array storing the records is called **hash table**
- the function used to map keys to table entries (slots) si called **hash function**
- a hash function is a function defined over the set of possible key values and takes on values from the set of slots in the hash table (*0* to *n-1* for a table of size *n*)

Hash functions

- the range of the key values has typically much more elements than slots in the hash table
- some keys will map to the same slot (**collision**)
- when defining a hash function the goal is to distribute the keys uniformly over the slots

Example hash functions

1. Hash integer keys by modulo

```
int HashMod (int k){
    return k % TABLE_SIZE;
}
```

2. Mid-square method (numbers)

```
// r = log (TABLE_SIZE)

int HashMidSq (int k){
    unsigned int sq;
    sq = k * k;
    sq = sq << (sizeof(int)*8 - r) / 2;
    sq = sq >> (sizeof(int)*8 - r);
    return sq;
}
```

3. Character folding (strings)

```
int HashFoldCh (char* s){
    int sum = 0;
    for ( ; *s != '\0' ; s++)
        sum += (int) (*s);
    return sum % TABLE_SIZE
}
```

Collision resolution

Collision resolution techniques:

- **open hashing** - collisions are stored outside the hash table
- **closed hashing** - collisions are stored in the hash table

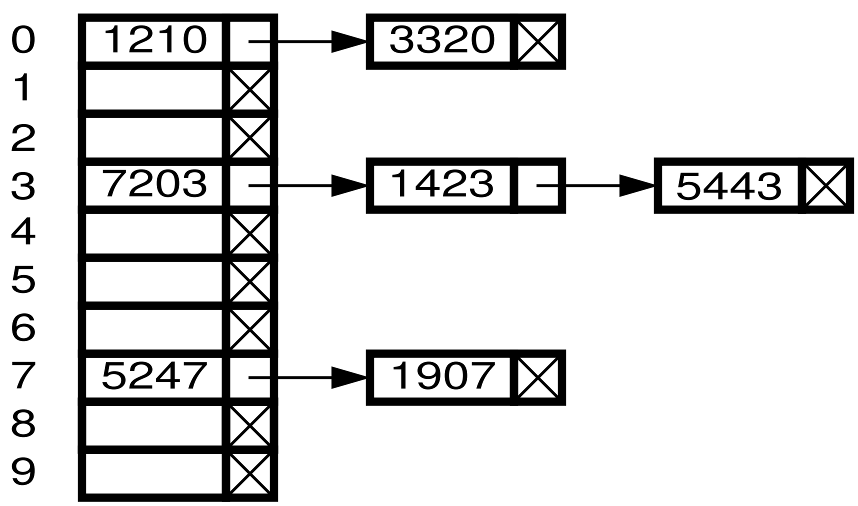
1. **Open hashing** (chaining)

```
struct entry{
    key     k;
    info    I;
    entry* link;
};

entry HashTable[TABLE_SIZE];
```

```
info HashSearch (entry H[]; key K){
   int i;
   entry* pe;
   i = Hash (K);
   if (H[i].k == EMPTY)
      return NOT_FOUND;
   else if (H[i].k == K)
      return H[i].I;
   else
      return SeqSearch (H[k].link, K);
}
```

Example.



2. Closed hashing

- all records are stored in the hash table
- a collision is solved by applying secondary hash functions (probing)

```
void hashInsert (record R){
    int h;
    int curr = h = Hash (R.k);
    for ( int i = 1; H[curr].key != EMPTY ; i++){
        curr = (h + p(R.k, i)) % TABLE_SIZE;
        if (H[curr].k == R.k)
            return EXISTING_KEY;
    }
    H[curr] = R;
}

void hashSearch (record R);
```

Linear probing

$$p(k, i) = i$$

Quadratic probing

$$p(k, i) = i^2$$

Double hashing

$$p(k, i) = i * h_2(k)$$

Analysis of closed hashing

- the cost of insertions and unsuccessful searches is

$$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$