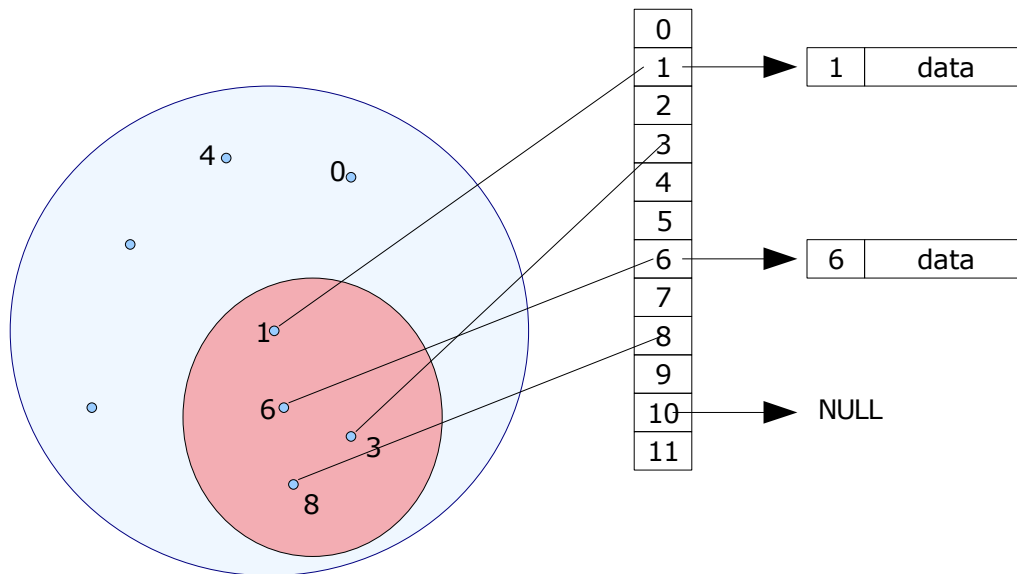


Hash Tables and Hash Functions

Hash tables are effective data structures used to support dictionary operations, *Insert*, *Search* and *Delete*. Searching for an element in a hash table is generally fast, done in $O(1)$ times, although in principle the search could take $O(n)$ time.

Direct Addressing is a simple scheme in which each element has a key, k , drawn from a set of possible keys $U = \{0, 1, \dots, m-1\}$. The Direct Address Table is an array of m pointers, each pointing to the (key, data) object or to NULL if there is no element with that key.



The problem with this scheme is that if the Universe of possible keys is large, we need a large array, most of which may actually be unused.

In direct addressing, key k is held in slot k of an array (in other words, at index k in the array). *Hashing* is a method in which the element with key k is held in slot $h(k)$, where h is a hash function. The function $h(k)$ maps the Universe of keys to a small set $T = \{0, 1, \dots, m-1\}$. Thus, even though the possible set of keys may be very large, the hash function maps them to a manageable set of integers.

Example:

Consider string objects, with keys defined as follows:

key, $k(\text{string } s) = \text{sum of ascii values of the characters in the string.}$

and the hash function:

$h(k) = k \bmod 10$

This maps every string to an integer in the range $[0, 9]$.

Here's a program that reads in a text file, and computes the hash code of each word in it according to the function defined above.

```
/* Hash1.cpp */  
  
#include<iostream>  
#include<fstream>  
#include<string>
```

```

using namespace std;

int hash1(string s){
    int sum=0;
    char c;
    for(int i=0;i<s.length();i++){
        c=s[i];
        sum+=c;
    }
    return sum % 10;
}

int main(){
    ifstream IN;
    ofstream OUT,OUT2;
    string s;
    int hashCode, freq[10]={0,0,0,0,0,0,0,0,0,0};
    char c;
    IN.open("20-para-lorem.txt");
    OUT.open("h1.txt");
    OUT2.open("h2.txt");
    while(!IN.eof()){
        IN>>s;
        hashCode=hash1(s);
        OUT2<<s<<" "<<hashCode<<endl;
        freq[hashCode]++;
    }
    for(int i=0;i<10;i++){
        OUT<<i<<" "<<freq[i]<<endl;
    }
    IN.close();
    OUT.close();
    OUT2.close();
}

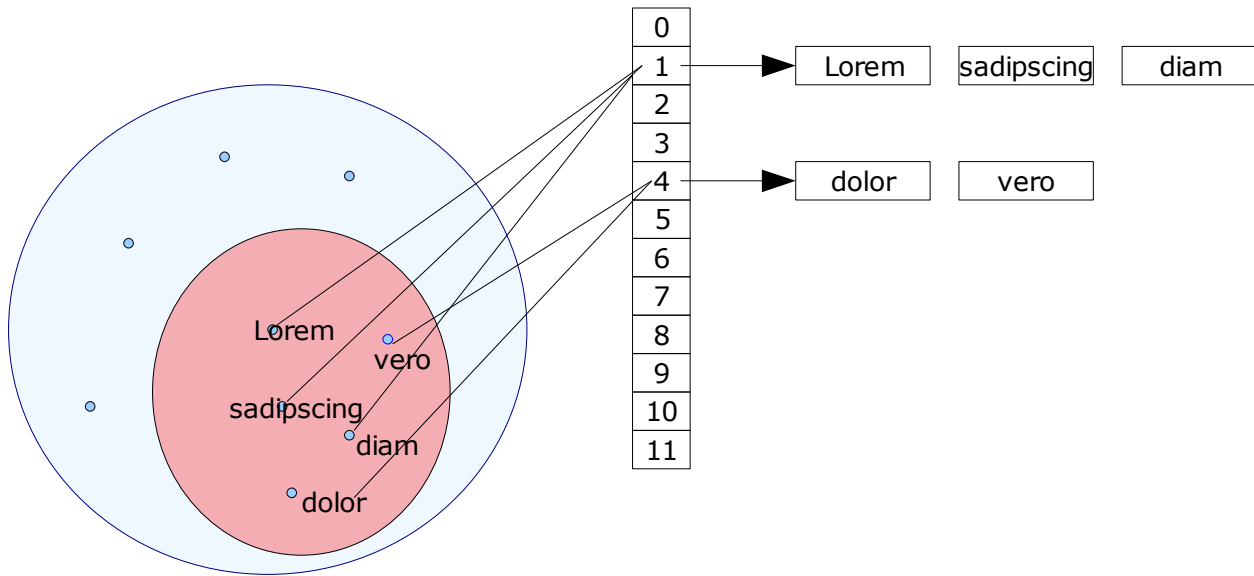
```

For a randomly generated text file ("20-para-lorem.txt") with 1794 words, the frequency distribution of hash codes and sample codes for a few words are shown below:

Code	Frequency
0	124
1	266
2	100
3	144
4	162
5	218
6	188
7	274
8	220
9	98

Word	Code
Lorem	1
ipsum	8
dolor	4
sit	6
amet,	7
consetetur	0
sadipscing	1
elit,	8
sed	6
diam	1

Obviously, in the above example more than one word has the same hash code. For example the words "Lorem", "sadipscing" and "invidunt" each has hash code 1. This is called collision. Various techniques are used to deal with collision, perhaps the simplest of which is called "chaining". In this scheme, whenever there is a collision, that word is added to a linked list. The storage structure in this case is an array of 10 pointers, each of which points to the head of a linked list.



To search for a word in this structure, first compute the hash code value, and search through the linked list corresponding to that value. In the above example, at most 274 searches are needed to locate a word.

Obviously, the running times for insertion, searching and deletion operations depend, on the average, on the length of the linked lists. A commonly used measure in this context is the load factor, defined as the ratio of the number of elements to be stored over the # of slots. In the above example the load factor is $1794/10=179.4$. This is the average length of a list in the hash table.

A *good* hash function is one that avoids collision, and uniformly maps to the available slots $T=[0,m-1]$. A *perfect* hash function is one that has a $O(1)$ worst case search time.