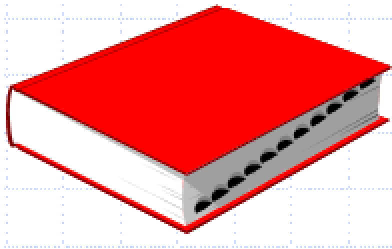
C Programming Basic – week 14

Topics of this week

- Dictionary ADT
- Hash Table
- Hash functions
- Compression maps
- Collision handling
- Exercises

Dictionary ADT



- The dictionary ADT models a searchable collection of key-element items
- The main operations of a dictionary are searching, inserting, and deleting items
- Multiple items with the same key are allowed
- Applications:
  - address book
  - credit card authorization
  - mapping host names (e.g., csci260.net) to internet addresses (e.g., 128.148.34.101)

Dictionary ADT methods

- findElement(k): if the dictionary has an item with key k, returns its element, else, returns the special element NO_SUCH_KEY
- insertItem(k, o): inserts item (k, o) into the dictionary
- removeElement(k): if the dictionary has an item with key k, removes it from the dictionary and returns its element, else returns the special element NO_SUCH_KEY
- size(), isEmpty()
- keys(), elements()

Key-Indexed Dictionaries

Key	Value
1	Intro to CS 1
2	Intro to CS 2
5	Theory of Computation
7	Data Structures
9	Digital Logic

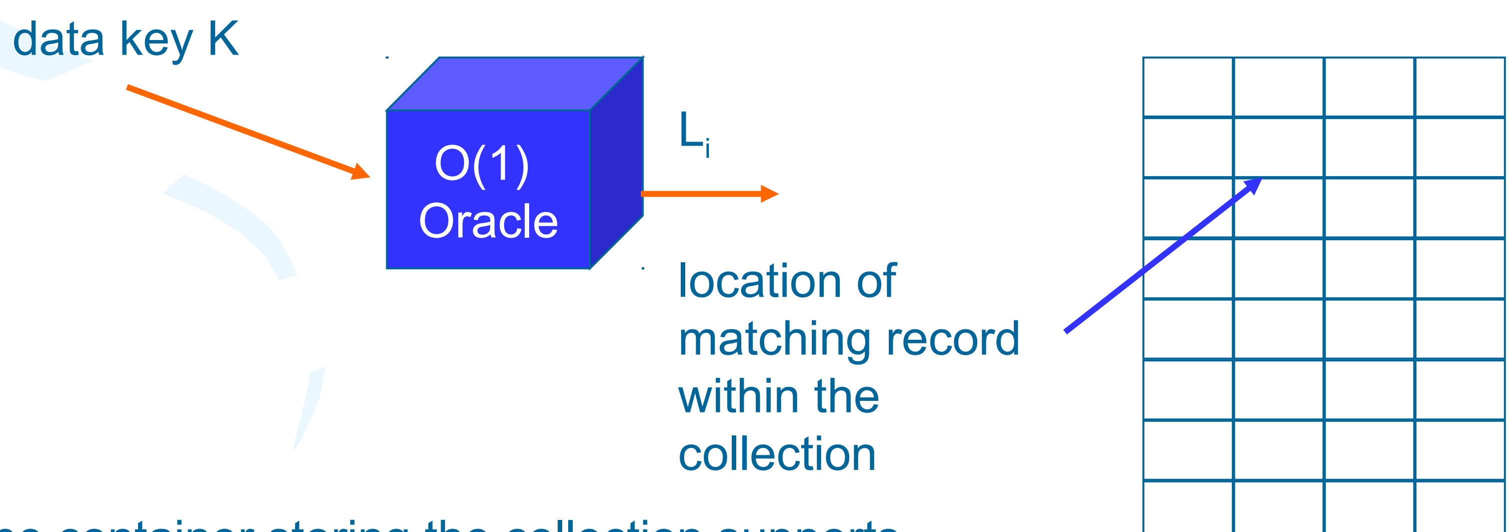
A[]

0	
1	Intro to CS 1
2	Intro to CS 2
3	
4	
5	Theory of Computation
6	
7	Data Structures
8	
9	Digital Logic

Space-efficient only if the cardinality of the set is close to N

Searching without Comparisons

- How could a search algorithm proceed without comparing data elements?
- What if we had some sort of "oracle" that could take the key for a data value and compute, in constant-bounded time, the location at which that key would occur within the data collection?



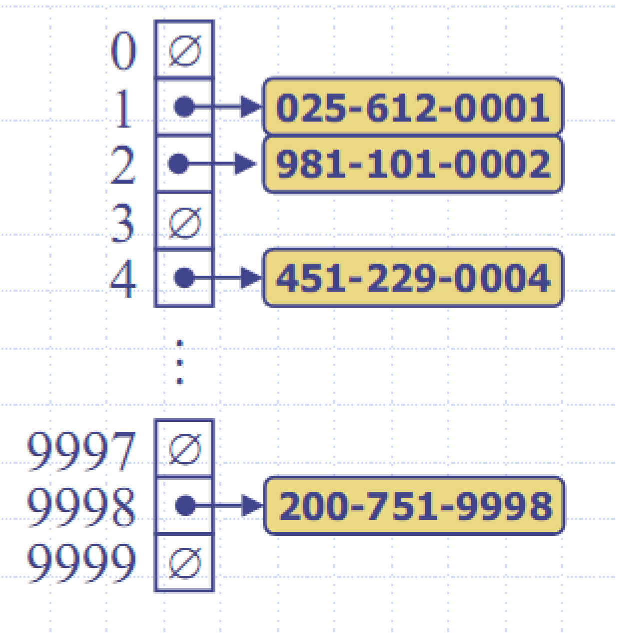
If the container storing the collection supports random access with Θ(1) cost, as an array does, then we would have a total search cost of Θ(1).

Hash Functions and Hash Tables

- An efficient way of implementing a dictionary is a hash table.
- Use an array (or list) of size N (table)
  - Need to spread keys over range [0,N-1]
  - Collisions occur when elements have same key
- Keys are not always integers, nor in range [0,N-1]
- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N
- When implementing a dictionary with a hash table, the goal is to store item (k, o) at index $i = h(k)$

Example

- We design a hash table for a dictionary storing items (SIN, Name), where SIN (social insurance number) is a nine-digit positive integer
- Our hash table uses an array of size N = 10,000 and the hash function
- h(x) = last four digits of x



Hash functions

- A hash function h maps keys of a given type to integers in a fixed interval [0, N – 1]
- Example:

  h(x) = x mod N is a hash function for integer keys

  The integer h(x) is called the hash value of key x
- A hash function is usually specified as the composition of two functions:
- Hash code map:

  h1:keys → integers
- Compression map:

  h2: integers → [0, N – 1]

Hash Code Maps

- Interger cast
  - Bits of the key are interpreted as integer
  - Suitable for keys of length shorter than the number of bits of an integer type
  - Example:
    - ‘A’ -> 65
    - ‘N’ ->78
- Component Sum
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type

$$x = (\underbrace{x_1}_{32\text{ bits}}, \underbrace{x_2}_{32\text{ bits}}, \ldots, \underbrace{x_{n-1}}_{32\text{ bits}}) \Rightarrow h_1(x) = \sum_{i=0}^{n-1} x_i$$

Hash code Maps

- Polynomial accumulation
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)

    $a_0 \; a_1 \; \dots \; a_{n-1}$
  - We evaluate the polynomial

    $p(z) = a_0 + a_1 z + a_2 z^2 + \dots + a_{n-1} z^{n-1}$

    at a fixed value z, ignoring overflows
  - Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)

Exercise 14.1

- Write three function which implements three type of hash code maps above.
- The input key for integer cast and polynomial is a string
- The input key for component sum method is a number of type long.

Compression Map

- The result of the HCM needs to be reduced to a value in [0,N-1]
- Division Method:
  - $h_2(y) = |y| \bmod N$
  - The size N of the hash table is usually chosen to be a prime
- Multiply, Add and Divide (MAD):
  - $h2(y) = |ay + b| \bmod N$
  - a and b are nonnegative integers such that $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value b

Simple implementation of Hash Table

```
#define MAX_CHAR  10
#define TABLE_SIZE  13
typedef struct {
    char key[MAX_CHAR];
    /* other fields */
} element;
element hash_table[TABLE_SIZE];
```

Hash Algorithm via Division

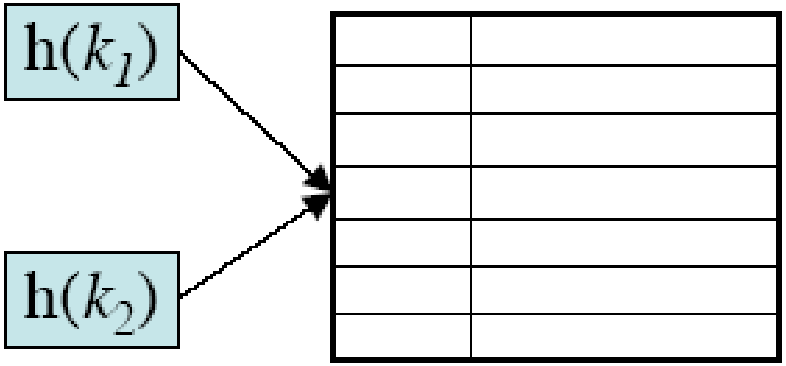
```
void init_table(element ht[])
{
   int i;
   for (i=0; i<TABLE_SIZE; i++)
     ht[i].key[0]=NULL;
}
```

```
int hash(char *key)
{
   return (transform(key)
           % TABLE_SIZE);
}
```

```
int transform(char *key)
{
   int number=0;
   while (*key) number += *key++;
   return number;
}
```

Conflict Resolution

- Collisions - occur when $k_1 \neq k_2$ but $h(k_1) = h(k_2)$
- Results in more complex *insertItem*() and *findElement*() operations
- Conflict Resolution Strategies
  - Closed Addressing (Open Hash Table) - i.e. slots other than h(k) are "closed" and can not be used
  - Open Addressing (Closed Hash Table)- look for another open position in the table



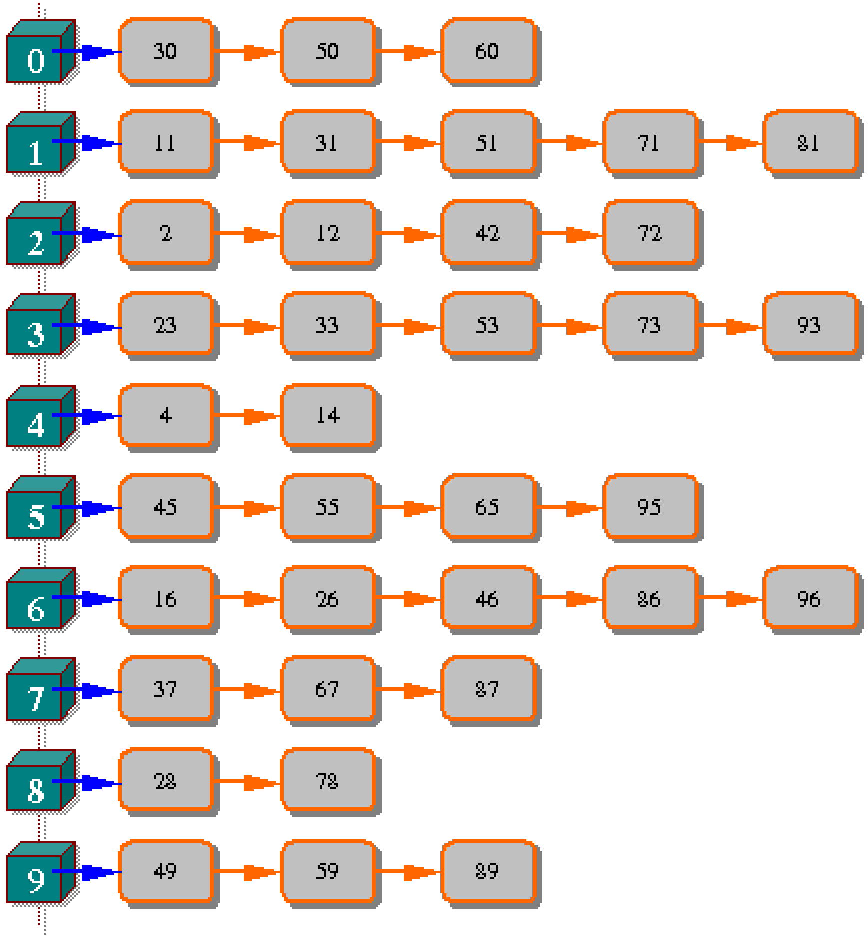
Data structure for Hash Table

- Open Hash Table:
  - Chaining Method
- Closed Hash Table
  - Linear Probing
  - Quadratic Probing
  - Rehashing

Data structure for chaining

- Array of pointers
- Each pointer manage a linked list corresponding to a bucket (address).
- This example shows a chaining hash table with hash function

  N mod 10



Exercise 14.2

- Implement an ADT for chaining hash table providing the following operations:
  - Init
  - Hash function
  - Insert (given key and element)
  - Search, Delete (given key)
  - IsEmpty
  - Clear
  - Traverse

Declaration

Data structure declaration

```
#define B ...  // size of hash table
typedef ... KeyType;  // int
typedef struct Node
{
   KeyType Key;
   // Add new fields if it is necessary
   Node* Next;
};
typedef Node* Position;
typedef Position Dictionary[B];
Dictionary D;
```

Initiate a Hash Table

```
void MakeNullSet()
{
  int i;
  for(i=0;i<B;i++)
   D[i]=NULL;
}
```

Search an element in the hash table

```
int Search(KeyType X)  {
  Position P;
  int Found=0;
  //Go to bucket at H(X)
  P=D[H(X)];
  //Traverse through the list at bucket
  H(X)
  while((P!=NULL) && (!Found))
    if (P->Key==X) Found=1;
   else P=P->Next;
 return Found;
}
```

Insert an element

```
void InsertSet(KeyType X)
{
   int Bucket;
   Position P;
   if (!Member(X, D)) {
        Bucket=H(X);
        P=D[Bucket];
        //allocate a new node at D[Bucket]
        D[Bucket] = (Node*)malloc(sizeof(Node));
        D[Bucket] ->Key=X;
        D[Bucket] ->Next=P;
   }
}
```

Delete an element

```
void DeleteSet(ElementType X){
    int Bucket, Done;
    Position P,Q;
    Bucket=H(X);
    // If list has already existed
    if (D[Bucket]!=NULL) {
    // if X at the head of the list
    if D[Bucket]->Key==X)
    {
        Q=D[Bucket];
        D[Bucket]=D[Bucket]->Next;
        free(Q);
    }
```

```
else { // Search for X
    Done=0;
    P=D[Bucket];
    while ((P->Next!=NULL) && (!Done))
        if (P->Next->Key==X)
            Done=1;
        else P=P->Next;
    if (Done) { // If found
        // Delete P->Next
        Q=P->Next;
        P->Next=Q->Next;
        free(Q);
    }
}
}
}
```

Emptiness

Verify if a bucket is empty

```
int emptybucket (int b){
   return(D[b] ==NULL ? 1:0);
}
```

Verify if the table is empty

```
int empty( ){
   int b;
   for (b = 0; b<B;b++)
   if(D[b] !=NULL)  return 0;
   return 1;
}
```

Clear a bucket

```
void clearbucket (int b){
  Position p,q;
  q = NULL;
  p = D[b];
  while(p !=NULL){
    q = p;
    p=p->next;
    free (q);
  }
  D[b] = NULL;
}
```

Clear the hash table

```
void clear( )
{
  int b;
  for (b = 0; b<B ; b++)
  clearbucket(b);
}
```

Traverse a bucket

```
void traversebucket (int b)
{
   Position p;
   p= D[b];
   while (p !=NULL)
   {
      // Assume that the key is of int type
      printf("%3d", p->key);
      p= p->next;
   }
}
```

Traverse the table

```
void traverse()
{
  int b;
  for (b = 0;n<B; b++)
  {
    printf("\nBucket %d:",b);
    traversebucket(b);
  }
}
```

Exercise 14.3

- You assume to make an address book of mobile phone.
- You declare a structure which can hold at least “name,” “telephone number,” and “e-mail address”, and make a program which can manage about 100 these data.
- (1) Read about 10 from an input file, and store them in a hash table which has an “e-mail address” as a key. Then confirm that the hash table is made. In this exercise, the hash function may always return the same value.
- (2)Define the hash function properly, and make the congestion occur as rare as possible

Linear Probing (linear open addressing)

- Compute f(x) for identifier x
- Examine the buckets

  ht[(f(x)+j)%TABLE_SIZE]

  $0 \le j \le$ TABLE_SIZE

  - The bucket contains x.
  - The bucket contains the empty string
  - The bucket contains a nonempty string other than x
  - Return to ht[f(x)]

Linear Probing - example

0	49**
1	58**
2	69**
3	
4	
5	
6	
7	
8	18
9	89

With linear probing *f(i) = i*.

Here is a hash table of size *T = 10*, where the entries 89, 18, 49, 58, and 69 have been inserted. The hash function is *h(key) = key%10.*

Throughout this talk we use a table size T = 10, although in practice it should be prime.

Exercise 14.4

- Implement an ADT Hash Table with linear probing method.

Solution: Data structure

```
#define NULLKEY –1
#define M 100 // size of hash table
struct node
{
   int key;
};
//Declare hash table as an array
struct node hashtable[M];
int NODEPTR;
int N = 0;
```

Hash function and initialization

```
int hashfunc(int key)
{
  return(key% 10); // or any number
}

void initialize( )
{
int i;
for(i=0;i<M;i++)
hashtable[i].key=NULLKEY;
N=0;
//so nut hien co khoi dong bang 0
}
```

Check the state of table

```
int full( ){
return (N==M-1 ? 1: 0);
}


int empty( ){
return (N==0 ? 1: 0);
}
```

Search

```
int search(int k){
  int i;
  i=hashfunc(k);
  while(hashtable[i].key!=k &&
  hashtable[i].key !=NULLKEY){
  //rehash :fi(key)=f(key)+1) % M
     i=i+1;
     if(i>=M) i=i-M;
  }
  if(hashtable[i].key==k) // found
     return i;
  else // not found
  return M;
}
```

Insert

```
int insert(int k){
  int i, j;
  if(full()){
     printf("\n Hash table is full. Can not insert
     the key %d ",k);
     return;
  }
  i=hashfunc(k);
  while(hashtable[i].key !=NULLKEY){
  // Rehash
     i ++;
     if(i>M) i= i-M;
  }
  hashtable[i].key=k;
  N=N+1;
  return i;
}
```

Remove a key

```
void remove(int i){
   int j, r, a, cont=1;
   do {
       hashtable[i].key = NULLKEY;
       j = i;
       do {
           i=i+1;
           if(i>=M) i=i-M;
           if(hashtable[i].key == NULLKEY)cont = 0;
           else {
               r = hashfunc(hashtable[i].key);
               a = (j<r && r<=i)||(r<=i && i<j)||(i<j && j<r);
           }
       } while (cont && a);
       if(cont) hashtable[j].key=hashtable[i].key;
   } while(cont);
}
```

Quadratic Probing

- Linear probing tends to cluster
  - Slows searches
- designed to eliminate the primary clustering problem of linear (but some secondary clustering)
- uses a quadratic collision function i.e. $f(i) = i^2$
- no guarantee of finding an empty cell if table is > half full unless table size is prime

Exercise 14.5

- Implement an ADT Hash Table with quadratic probing method.

Search

```
int search(int k) {
  int i, d;
  i = hashfunc(k);
  d = 1;
  while(hashtable[i].key!=k && hashtable[i].key !
  =NULLKEY){
  //Quadratic probing
  i = (i+d*d) % M;
  d = d+1;
  }
  if(hashtable[i].key==k) // found
     return i;
  else // not found
  return M;
}
```

Insert

```
int insert(int k){
   int i, d;
   if(full()){
      printf("\n Hash table is full. Can not insert
      the   key %d ",k);
      return;
   }
   i=hashfunc(k); d = 1;
   while(hashtable[i].key !=NULLKEY){
      //Quadratic probing
      i = (i+d*d) % M;
      d = d+1;
   }
   hashtable[i].key=k;
   N=N+1;
   return i;
}
```

Double Hashing

- Double hashing uses a secondary hash function $h_2(k)$ and handles collisions by placing an item in the first available cell of the series
  $(i + h_2(k)) \bmod N$
- The secondary hash function $\boldsymbol{h_2(k)}$ cannot have zero values
- The table size $\boldsymbol{N}$ must be a prime to allow probing of all the cells

- Common choice of compression map for the secondary hash
- function: $h_2(k) = q - k \bmod q$
- where
  - $q < N$
  - q is a prime

Exercise 14.6

- Implement an ADT Hash Table with rehashing method, using two following hash functions:

- **f1(key)= key % M**
- **f2(key) =(M-2)-key %(M-2)**

Hash functions

```
int hashfunc(int key)
{
  return(key%M);
}
//Secondary function
int hashfunc2(int key)
{
  return(M-2 – key%(M-2));
}
```

Search

```
int search(int k) {
  int i, j ;
  i = hashfunc (k);
  j = hashfunc2 (k);
  while(hashtable[i].key!=k &&
  hashtable[i].key !=NULLKEY){
  //Rehashing
     i = (i+j) % M ;
  }
  if(hashtable[i].key==k) // found
     return i;
  else // not found
  return M;
}
```

Insert

```
int insert(int k){
   int i, j;
   if(full()){
      printf("\n Hash table is full. Can not insert  the
      key %d ",k);
      return M;
   }
   if (search (k) < M){
      printf ("This key exist in the hash table") ;
      return M ;
   }
   i = hashfunc(k); j = hashfunc2(k) ;
   while(hashtable[i].key !=NULLKEY){
      //Rehashing
      i = (i+j) % M;
   }
   hashtable[i].key=k;
   N=N+1;
   return i;
}
```