Hash Map

- Mapping keys to values
  - Assume that keys are unique

- Components
  - Keys
  - Hash function
  - Buckets

```
<table>
<thead>
<tr>
<th>keys</th>
<th>hash function</th>
<th>buckets</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Smith</td>
<td>01 521-8976</td>
<td>01</td>
</tr>
<tr>
<td>Lisa Smith</td>
<td>02 521-1234</td>
<td>02</td>
</tr>
<tr>
<td>Sandra Dee</td>
<td>03</td>
<td>03</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>14 521-9655</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>
```
Motivation

- **Associative Array**
  - A collection of (Key, Value)
- **Examples**
  - Student Grades (Name, Grade)
    - (“John”, 90), (“Tom”, 92), (“Joe”, 85)
    - GradeOf(“John”) returns 90
  - Years when a war started (Year, War)
    - (1776, “Revolutionary”), (1861, “Civil War”), (1939, “WW2”)
    - WarStarted(1861) returns “Civil War”
    - WarStarted(1990) returns NULL

Motivation

- What if using Arrays? (for large volume of data)
  - Maximum size must be known in advance
  - The key can be a index of the array
    - The key might not be a number
    - Most of the array elements would be empty
Motivation

- Array of a structure
  - A structure that stores key and value
    - Adding elements: Add to the end
    - Removing elements: Find the element moving next elements after the element
    - Finding: search from the beginning of the array

Motivation

- Linked List
  - Adapt Memory Size
  - Structure

```c
struct Node {
    int key;
    char *value;
    struct Node *pNext
};
```
Motivation

- Search a key
  ```
  for (p=head; p!=NULL; p=p->next)
  {
    if (p->key == A_KEY)
      return p;
  }
  
- Memory Efficient
- Searching takes time

Hash Tables

- Hash Table
  - Fixed-size array: each element points at a linked list
    ```
    struct Node *HashTable[TABLESIZE];
    ```
Hash Tables

- **Hash Function**
  - Mapping each key to an array index
  - E.g., for an integer key $h$
    - Hash function: $i = h \mod \text{TABLESIZE}$
    - HashTable[$i$] is a head of a linked list

  ![Hash Table Diagram](image)

- **Array of size 5 with hash function “h % 5”**
  - “1776 % 5” is 1
  - “1861 % 5” is 1
  - “1939 % 5” is 4

  ![Hash Table Diagram](image)
Hash Tables

- How to decide the size of Hash Tables?
  - If the size of Hash Table is small, the average of buckets would be large
    - Slow look-ups
    - Memory Efficient
  - If the size of Hash Table is large, the average of buckets would be small
    - Fast Look-ups
    - Memory Inefficient
  - Good distributing buckets is important

Hash Function

- Choose a good Hash Function for the good distribution
  - Simple schemes normally don’t produce good distribution
    - E.g., number of characters % TABLESIZE
    - Sum the ASCII values of all characters % TABLESIZE
  - Reasonably good hash functions
    - Weighted sum of characters $x_i$ in the string
      - $\sum a_i x_i \%$ TABLESIZE
      - Best if $a_i$ and TABLESIZE are relatively prime
      - E.g., $a = 65599$, TABLESIZE = 1024
Hash Function

- Although computation is expensive to compute $a_i$ for each value of $i$, it is reasonably
  - $((x[0] \times 65599 + x[1]) \times 65599 + x[2]) \times 65599 + x[3])$
  - ...

  ```
  unsigned hashfunc(char *x) {
    int i; unsigned int h = 0;
    for (i=0; x[i]; i++)
      h = h * 65599 + x[i];
    return (h % 1024);
  }
  ```

Hash Map

- Structure
  - struct Node{
    char *key;
    char *value;
    struct Node *pNext;
  };

- Hash Table
  - struct Node *HashTable[TABLESIZE];

- Functions
  - Hash function: unsigned hashfunc(char *key)
  - Look up with key: struct Node *lookup(char *key)
  - Install entry: struct Node *install(char *key, char *value)
Hash Map

- **Lookup function**

```c
struct Node *lookup(char *key)
{
    struct Node *p;
    for (p = HashTable[hashfunc(key)]; p!=NULL; p=p->next)
        if (strcmp(key, p->key) == 0)
            return p; // found
    return NULL; // not found
}
```

Hash Map

- **Add a new Node if none exists, or overwrite the old value**

```c
struct Node *install(char *key, char *value)
{
    struct Node *p;
    if ((p = lookup(key)) == NULL){
        // Next slide
    } else
        free(p->value);
p->value = malloc(strlen(value) + 1);
strcpy(p->value, value);
return p;
}
```
Hash Map

- Allocate memory for the new struct and the key

```c
p = malloc(sizeof(struct Node));
p->key = malloc(strlen(key) + 1);
strcpy(p->key, key);
// add to front of linked list
unsigned i = hashfunc(key);
p->next = HashTable[i];
HashTable[i] = p;
```

Recall: Bitwise for Mod function

- Mod function is expensive (x % c)
  - We can utilize a bitwise operator if c is a power of 2
  - E.g.,
    - 53 = 32 + 16 + 4 + 1
    - \[0\ 0\ 1\ 1\ 0\ 1\ 0\ 1\]
    - 53%16 is 5, we can consider only the last four bits of the number
Bitwise for Mod function

- Bitwise operator
  - $53 \& (16-1) = 5$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>53</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp; 15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>