Motivations

- We have to store some records and perform the following
  - **Insert new record**
  - **Search a record by key**
  - **Delete a record**

- What is the optimal data structure that allows fast insertion and searching?
  - Unordered array
    - insert optimal $O(1)$, search slow $O(N)$
  - Ordered array
    - insert slow $O(N)$, search fast $O(\log N)$
  - Linked list
    - insert optimal $O(1)$, search slow $O(N)$
  - Binary Tree (balanced)
    - insert fast $O(\log N)$, search fast $O(\log N)$
  - Hash Table
    - insert optimal $O(1)$, search optimal $O(1)$ …….. ?
Motivations

- More comparisons between **general purpose** data structures

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Search</th>
<th>Insertion</th>
<th>Deletion</th>
<th>Traversal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>—</td>
</tr>
<tr>
<td>Ordered array</td>
<td>$O(\log N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Linked list</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>—</td>
</tr>
<tr>
<td>Ordered linked list</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Binary tree (average)</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Binary tree (worst case)</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Balanced tree (average and worst case)</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Hash table</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>—</td>
</tr>
</tbody>
</table>

- Some disadvantages for Hash tables:
  - They are based on arrays, so difficult to expand once created
  - Performances may degrade significantly if the hash table is too full
  - There is no convenient way in a hash table in any kind of order
- However, hash table is an ideal choice if you do not need to visit items and you can predict in advance the size of the database
**Example:** Access students records by ID (here 1000 students in the database)

<table>
<thead>
<tr>
<th>ID</th>
<th>NAME</th>
<th>SCORE</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>andy</td>
<td>81.5</td>
</tr>
<tr>
<td>001</td>
<td>betty</td>
<td>90</td>
</tr>
<tr>
<td>002</td>
<td>david</td>
<td>56.8</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>303</td>
<td>peter</td>
<td>20</td>
</tr>
<tr>
<td>304</td>
<td>mary</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>998</td>
<td>tom</td>
<td>73</td>
</tr>
<tr>
<td>999</td>
<td>bill</td>
<td>49</td>
</tr>
</tbody>
</table>

**Best Strategy:** use an array of student objects, ID numbers (called keys) can be chosen as the index numbers. O(1) for insert, remove, search

**In real situations, however, keys are not distributed in such orderly fashion**
Example: Access students records by ID (1000 students in the database)

<table>
<thead>
<tr>
<th>ID</th>
<th>NAME</th>
<th>SCORE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0012345</td>
<td>andy</td>
<td>81.5</td>
</tr>
<tr>
<td>0033333</td>
<td>betty</td>
<td>90</td>
</tr>
<tr>
<td>0056789</td>
<td>david</td>
<td>56.8</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9801010</td>
<td>peter</td>
<td>20</td>
</tr>
<tr>
<td>9802020</td>
<td>mary</td>
<td>100</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9903030</td>
<td>tom</td>
<td>73</td>
</tr>
<tr>
<td>9908080</td>
<td>bill</td>
<td>49</td>
</tr>
</tbody>
</table>

Obviously, we cannot create an array of ID of size 10,000,000 where almost all cells will also be empty (→ waste of memory)

<table>
<thead>
<tr>
<th>ID</th>
<th>NAME</th>
<th>SCORE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12345</td>
<td>andy</td>
<td>81.5</td>
</tr>
<tr>
<td>33333</td>
<td>betty</td>
<td>90</td>
</tr>
<tr>
<td>56789</td>
<td>david</td>
<td>56.8</td>
</tr>
<tr>
<td>9908080</td>
<td>bill</td>
<td>49</td>
</tr>
<tr>
<td>9999999</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Goal:** Find a way to squeeze a very large range of keys into a table of much smaller range

We then introduce the function: `int hash(key)`

- key can be an integer, String, etc.
- hash represents a mapping, the method converts a number in a large range into a number that belongs to a smaller manageable range
- A simple approach with key being an integer is to use the modulo operator (%), which finds the remainder of the division

```
smallNumber=largeNumber%smallRange
```

**Example:** `smallRange=1000` and key are students large ID

- `hash(0012345)=345`
- `hash(0033333)=333`
- `hash(0056789)=789`
- `hash(9908080)=80`
### Example:

- To store a record, we compute hash(ID), and store it at the new location in the smaller range array.

<table>
<thead>
<tr>
<th>ID</th>
<th>NAME</th>
<th>SCORE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>bill</td>
<td>49</td>
</tr>
<tr>
<td>333</td>
<td>betty</td>
<td>90</td>
</tr>
<tr>
<td>345</td>
<td>andy</td>
<td>81.5</td>
</tr>
<tr>
<td>789</td>
<td>david</td>
<td>56.8</td>
</tr>
<tr>
<td>999</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- To search for a student, we just need to peek at the location hash(targetID).
Some desirable properties of a hash function

- Simple and quick to calculate
- Even distribution for the return indexes

However, there is no guarantee that two keys or more won't hash to the same array index

This situation is called a collision

It is not possible to avoid collisions while hashing, however there exist two main strategies for addressing this problem

1- Open addressing
2- Separate Chaining
Open Addressing: Definition

- First we need to consider an array with a size greater than the number of expected data items (x2 size).
- If no collision happens, the array will be half-full.
- Using open addressing, if the new data can't be placed at the index calculated by the hash function (if the cell is already occupied), it must find another location in the array.
- There exist 3 main methods of open addressing which depend on the way to determine the next vacant cell:
  - Linear probing (easy, but problem of clustering may appear)
  - Quadratic probing (effective)
  - Double Hashing (preferred)
Open Addressing: Linear Probing

- We search sequentially for vacant cells:
  
- **Insertion example:**
  - compute hash value for a given key e.g. 345
  - If 345 cell is occupied when we try to inset a new data item, we go to 346, and then 347 and so on, until we find an empty cell

- **Search example:**
  - Compute hash value for a given key e.g. 345
  - Start probing at 345, and increment the probe until
    - the key of a new probe cell equal to the search key
    - the new probe cell is empty (search has failed)

- **Comments on Delete:**
  - Deleted items should be marked with a special key (example: -1)
  - The insert method should consider the deleted cell as empty
  - The search method should treat the deleted cell as filled (and search further along)
Open Addressing: Linear Probing

- More examples on search
- **Todo:** Java applet hash.html
class HashTable{
    private DataItem table;
    private int size;

    public HashTable(int size){
        this.size=size;  table = new DataItem[size];
    }

    public int hash(int key){return  key%size; }

    public void insert(DataItem item){
        int h = hash(item.getKey()); // key
        while (table[h]!=null &

        && table[h].getKey()!=-1) { // until empty spot or -1
            h=(h+1)%size; //if occupied,increment by 1
        }
        table[h]=item; // enter item into table
    }

    public DataItem find(int key){
        int h = hash(key); // compute hash of item
        while (table[h]!=null &

        && table[h].getKey()!=key) { // find matching item or null
            h=(h+1)%size; // if no match, increment by 1
        }
        return table[h]; // return item or null
    }
}
Open Addressing: Linear Probing

- Execution example (complete code Textbook p535)

    Enter size of hash table: 12
    Enter initial number of items: 8

    Enter first letter of show, insert, delete, or find: s
    Table: 108 13 0 ** ** 113 5 66 ** 117 ** 47

    Enter first letter of show, insert, delete, or find: f
    Enter key value to find: 66
    Found 66

    Enter first letter of show, insert, delete, or find: i
    Enter key value to insert: 100
    Enter first letter of show, insert, delete, or find: s
    Table: 108 13 0 ** 100 113 5 66 ** 117 ** 47

    Enter first letter of show, insert, delete, or find: d
    Enter key value to delete: 100
    Enter first letter of show, insert, delete, or find: s
    Table: 108 13 0 ** -1 113 5 66 ** 117 ** 47

Key values run from 0 to 119 (12 times 10, minus 1). The ** symbol indicates that a cell is empty. The item with key 100 is inserted at location 4 (the first item is numbered 0) because 100%12 is 4. Notice how 100 changes to –1 when this item is deleted.
Open Addressing: Linear Probing- limitations

- As the hash table gets more full, clustering grows larger; It can result in very long probe length and degrade performance from $O(1)$ to $O(M)$, $M$ being the width of the cluster.

- The load factor is defined as the ratio between the number of items in the table and the size of the table. If load increases, the probe length $P$ grows larger. After some calculations (see Knuth, and textbook p567), one can obtain some formula for $P$ relative to successful or unsuccessful searches.

- It is critical to design a hash table to ensure that it never becomes more than half or at most two-third full.

- Expanding the array is also problematic (cannot use a direct copy), it involves a time consuming rehashing process.