CX2001 Algorithms **2020/21 Semester 1**

4: Searching

School of Computer Science and Engineering *Nanyang Technological University*

4.1 Searching

Searching problem is one of the classic computational problem to find a search key in a given data set.

Algorithm 1 Generic Search Algorithm

```
1: begin
2: while there is more possible data to examine do
3:     examine one datum
4: if the datum = search key then
5:     return succeed
6: return fail
7: end
```

4.2 Sequential Search

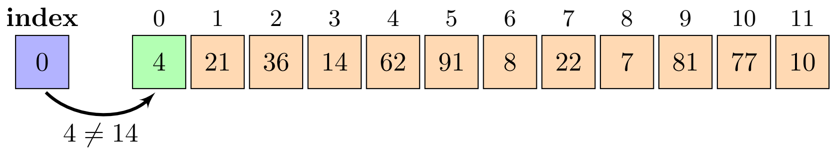
When the given data set is unsorted, we have to check every single element in the data set until either the key is found (successful search) or all elements in the data set have been retrieved once and no match is found (unsuccessful search). The search is called sequential search. It is a **brute-force algorithm**.

Algorithm 2 Sequential Search

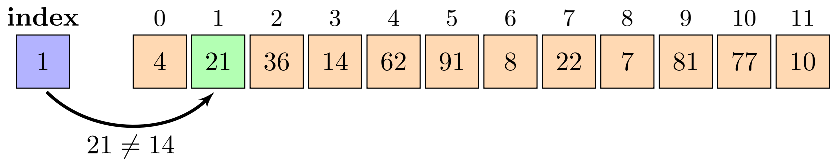
```
1: function seqSearch(int[] Data, int n, int key)
2: begin
3: for $index = 0$ to $n - 1$ do
4:     begin
5:     if $Data[index] == key$ then
6:         return index;                                   {success}
7:     end
8: return -1;                                              {failure}
9: end
```

4.2.1 Example: Success Case

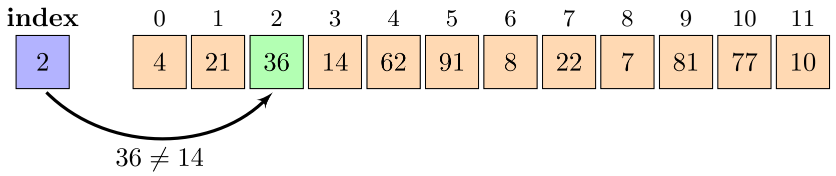
Search key: **14**
Let index =0,



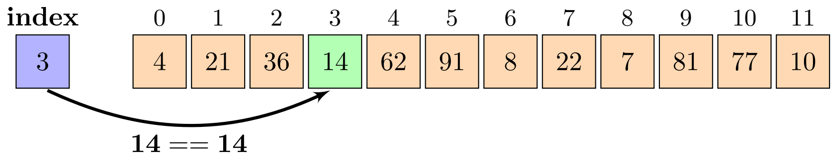
Let index =1,



Let index =2,



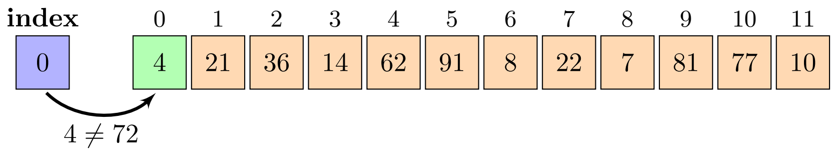
Let index =3, the search key is found



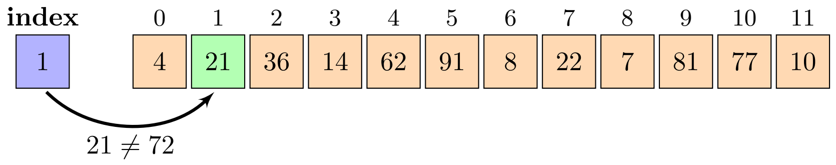
The search key, **14** is found at index =3. **3** return.

4.2.2 Example: Failure Case

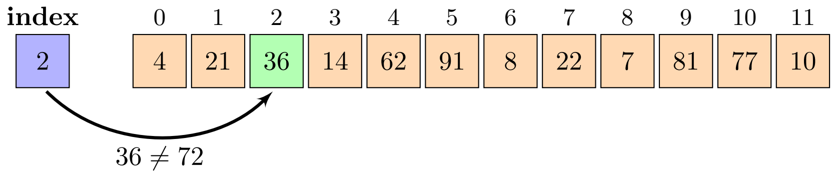
Search key: **72**
Let index =0,



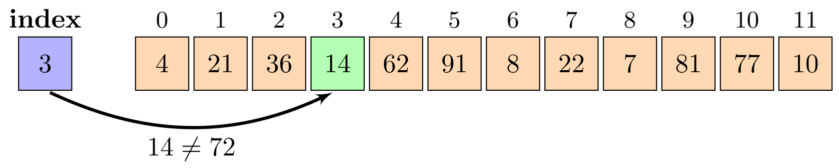
Let index =1,



Let index =2,



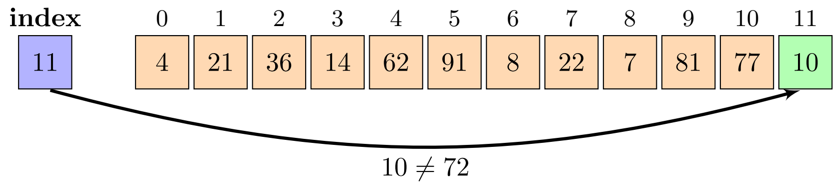
Let index =3,



⋮

⋮

Let index =11,



After searching throughout the given data set, there is no match with the search key, **72**. It is an unsuccessful search. **-1** return.

4.2.3 Complexity Analysis

- Best-case analysis: 1 comparison against key (the first item is the search key)
- Worst-case analysis: n comparisons against key (Either the last item or no item is the search key)
- Average-case Analysis:

- **Key is always in search array:**
  - Let e_i represents the event that the key appears in i^{th} position of array,its probability $P(e_i) = \frac{1}{n}$
  - $T(e_i)$ is the number of comparisons done
  - $0 \leq i \leq n$ and $T(e_i) = i + 1$
  - Since we assume that key definitely is in the array, the average-case analysis can be done as follow

$$\begin{aligned} A_s(n) &= \sum_{i=0}^{n-1} P(e_i)T(e_i) \\ &= \sum_{i=0}^{n-1} (\frac{1}{n})(i+1) \\ &= \frac{1}{n}\sum_{i=1}^{n} i \\ &= \frac{1}{n}\frac{n(n+1)}{2} = \frac{n+1}{2} \end{aligned}$$

- **Key is not always in search array:**

- When the search key is not in the array, the number of comparisons, $\mathbf{A_f(n)}$, is n. Refer to 4.2.2.

- Combine success cases and failure cases with their probability, $P(succ) + P(fail) = 1$:

$$P(succ)A_s(n) + P(fail)A_s(n) = q * (n+1)/2 + (1-q) * n$$

- If there is a 50-50 chance that key is not in the array, $P(succ) = P(fail) = 0.5$.

- The average no of key comparisons is $\frac{3n}{4} + \frac{1}{4}$ or about $\frac{3}{4}$ of entries are examined

In conclusion, both worst-case and average-case complexity are $\Theta(n)$.

4.3 Binary Search

Clearly Search an item from an unsorted array will take $\Theta(n)$ in average. To improve the searching performance, the data set need to be sorted out first. It will be discussed in the later lecture. Next, we assume that the array is sorted in order. We can use the information of its order to reduce the search work. Binary search is an example which is using **divide and conquer** approach. In short, this approach divide a problem into two smaller sub-problems, one of which does not even have to be solved.

Binary search first comparing a search key with the sorted array's middle element. If they match, then the algorithm stops; otherwise, the same operation is repeated recursively for either the upper half of the array if the search key is lesser than the middle element or the lower half of the array if the search key is greater.

Binary Search

```c
int binarySearch (int E[], int first, int last, int k)--> T(n)
{
   if(last < first)
      return -1;
   else {
      int mid = (first + last)/2;                      --> c
      if(k == E[mid])
           return mid;
      else if(k < E[mid])
          return binarySearch(E,first, mid-1,k);       --> T(n/2)
      else
          return binarySearch(E,mid+1, last,k);        --> T(n/2)
   }
}
```

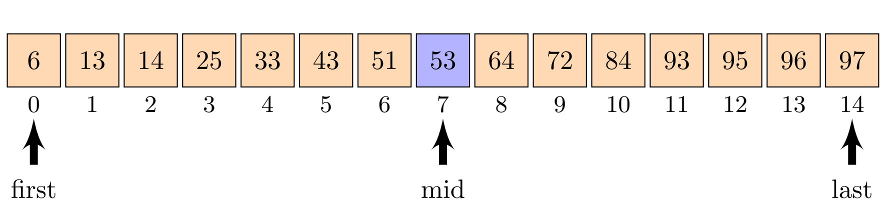
Listing 1: Recursive Version

```c
int binarySearch_iter(int E[], int first, int last, int k)
{
    while (first <= last) {
        int mid = (first+last) / 2;
        if (E[mid] == k)
            return k;
        else if (k < E[mid] )
            last = mid - 1;
        else
            first = mid + 1;
    }
    return -1;
}
```

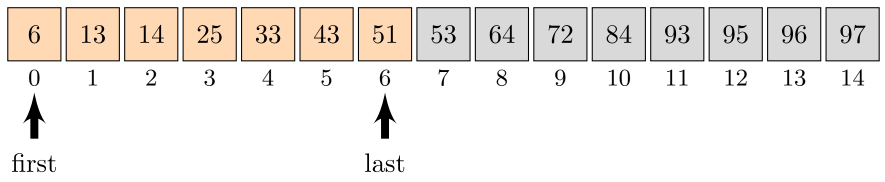
Listing 2: Iterative Version

4.3.1 Example: Success Case

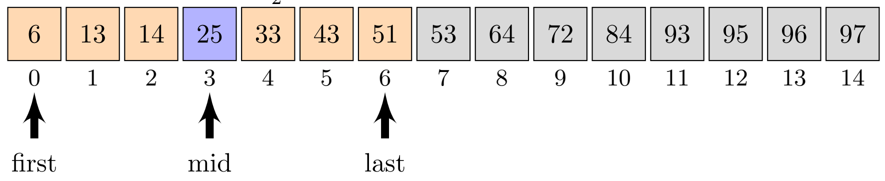
Given a search key, **33**, the **first** index is 0, the **last** index is 14 and the **mid** is $7 \Leftarrow \frac{0+14}{2}$



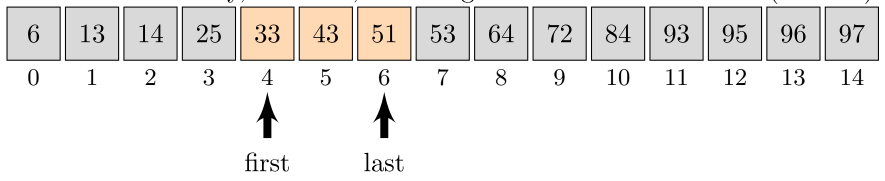
Since the search key, **33** < **53**, we change **last** index from 14 to 6 (**mid**-1)



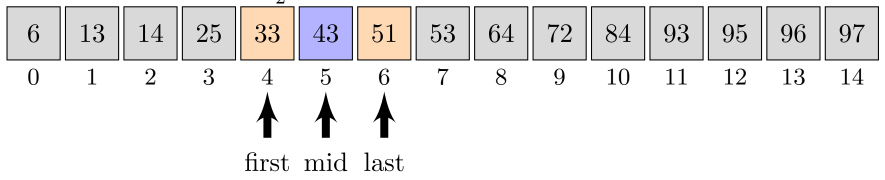
The new **mid** is 3 $\Leftarrow \frac{0+6}{2}$.



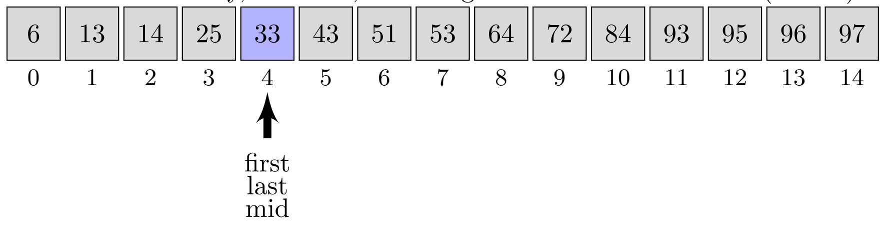
Since the search key, **33** > **25**, we change **first** index from 0 to 4 (**mid**+1)



The new **mid** is 5 $\Leftarrow \frac{4+6}{2}$.



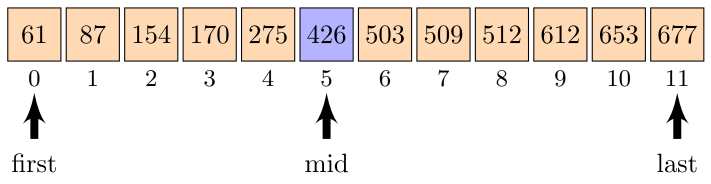
Since the search key, **33** < **43**, we change **last** index from 6 to 4 (**mid**-1)



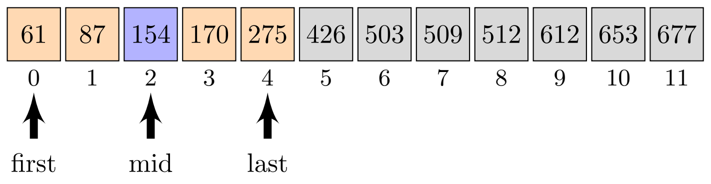
The search key, **33** is found at index, **4**

4.3.2 Example: Failure Case

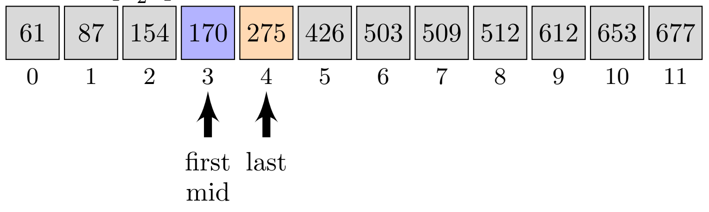
Given a search key, **400**, the **first** index is 0, the **last** index is 11 and the **mid** is 5 $\Leftarrow \frac{0+11}{2}$



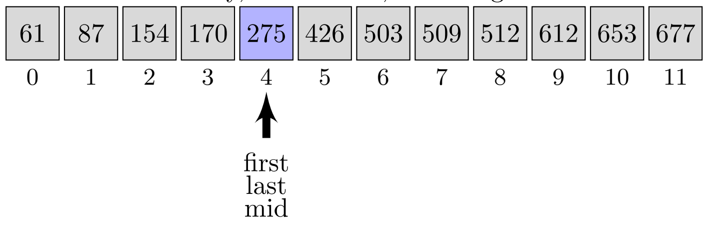
Since the search key, **400** < **426**, we change **last** index from 11 to 4 (**mid**-1)



Since the search key, **400** $>$ **154**, we change **first** index from 0 to 3 (**mid**+1). The new **mid** remain unchanged $\lfloor \frac{3+4}{2} \rfloor = 3$



Since the search key, **400** $>$ **170**, we change **first** index from 3 to 4 (**mid**+1).



Since the search key, **400** $>$ **275**, we change **first** index from 4 to 5 (**mid**+1). Since the **first** index larger than **last**, the search function returns -1 showing that searching fails. **400** does not exist in the given array.

4.3.3 Complexity Analysis

- Best-case analysis: 1 comparison against key (the first item is the search key)
- Worst-case analysis: Refer to Listing 1, each function call will definite the new **mid** as $\lfloor \frac{\textbf{first}+\textbf{last}}{2} \rfloor = 3$ if **last** $\geq$ **first**. The running time is constant, c. If the **mid** element and search key do not match, there will be a recursive call by passing either right or left section. If the size of array, n is even, there are $\frac{n}{2} - 1$ entries in the left section and $\frac{n}{2}$ in the right section. Refer to 4.3.1. If n is odd, there are $\frac{n-1}{2}$ entries in both sections. The worst case is $\frac{n}{2}$. When n is 1, **first** and **last** are the same index. Hence, the new **mid** will be the same index. If this only one entry is not the search key, it will make a recursive call. This is the last recursive call which will return -1 (no match). How many recursive call will be made until n is 1? It is noted that when n is 1, $T(1) = c$ (constant time).

$$\begin{aligned}
T(n) &= T(\frac{n}{2}) + c \\
&= (T(\frac{n}{4}) + c) + c \\
&= (T(\frac{n}{8}) + c) + c + c \\
&\dots \\
&= T(1) + d * c \\
&= (d+1)c
\end{aligned}$$

From the equation above, d is the number of recursive call. We divide n by 2 every call. At d^{th} recursive call, we have

$$\begin{aligned}\frac{n}{2^d} &= 1\\ n &= 2^d\\ d &= \log_2 n\end{aligned}$$

We can observe that the number of iteration, d is less than $\log_2 n$ due to the number of entries can be even or odd number.

$$\begin{aligned}d &\leq log_2 n\\ &= \lfloor \log_2 n \rfloor\end{aligned}$$

Prove: $T(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2(n+1) \rceil \; \forall n \geq 1 \wedge n \in \mathbb{N}$

Let k be some integer number, $k \in \mathbb{N}$ that $k = \lfloor \log_2 n \rfloor$, then

$$\begin{aligned}k &\leq \log_2 n < k+1\\ 2^k &\leq n \qquad < 2^{k+1}\\ 2^k &< n+1 \; \leq 2^{k+1}\end{aligned}$$

For example, if $k = 4$, then we have $16 \leq n < 32$. How about the range of (n+1)? it is $16 < n+1 \leq 32$ because n is an integer number.

$$\begin{aligned}2^k &< n+1 \qquad \leq 2^{k+1}\\ k &< \log_2(n+1) \leq k+1\end{aligned}$$

The last inequality implies that $\lceil \log_2(n+1) \rceil = k+1$ and we define that $k = \lfloor \log_2 n \rfloor$,

$$\lceil \log_2(n+1) \rceil = \lfloor \log_2 n \rfloor + 1$$

$\therefore T(n) = c(\lfloor \log_2 n \rfloor + 1) = c\lceil \log_2(n+1) \rceil$.
The worst-case running time is in $\Theta(\log_2 n)$

- Average-case Analysis:
To analysis the average case, we first consider two scenario, successful search and failed search. By Law of Expectations:

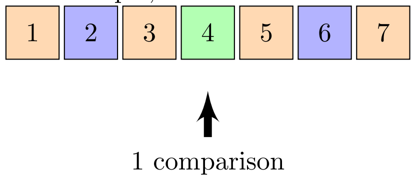
$$A_q(n) = qA_s(n) + (1-q)A_f(n)$$

where q is the probability. Here we consider $n = 2^k - 1$ for simplicity but other values is very close to the following result.
The failed search is the worst case. Thus, its complexity is:

$$A_f(n) = \lceil \log_2(n+1) \rceil = \log_2(n+1) = k$$

The successful search of $n = 2^k - 1$ entries may take from 1 to k comparisons. It depends on the position of the search key. $1, 2, 4, 8, \ldots, 2^{k-1}$ positions require to take $1, 2, 3, 4, \ldots, k$ comparisons respectively.

For example, $n = 2^3 - 1 = 7$ entries



In this example, we can observe that if the search key is at the 4th position, we just need one comparison. If the search key is at the 2nd or the 6th position, we need two comparisons etc.
We assume that the probability of each position be searched is equal, i.e. $\frac{1}{n}$. The complexity of successful search is:

$$
\begin{aligned}
A_s(n) &= \frac{1}{n}\sum_{t=1}^{k} t2^{t-1} \\
&= \frac{(k-1)2^k + 1}{n} \\
&= \frac{[\log_2(n+1) - 1](n+1) + 1}{n} \\
&= \log_2(n+1) - 1 + \frac{\log_2(n+1)}{n}
\end{aligned}
$$

The derivation can be found in 4.5.

The average-case time complexity can be obtain by substitute $A_s(n)$ and $A_f(n)$ into $A_q(n)$:

$$
\begin{aligned}
A_q(n) &= qA_s(n) + (1-q)A_f(n) \\
&= q[\log_2(n+1) - 1 + \frac{\log_2(n+1)}{n}] + (1-q)(\log_2(n+1)) \\
&= \log_2(n+1) - q + q\frac{\log_2(n+1)}{n} \\
&= \Theta(log_2(n))
\end{aligned}
$$

q is the probability which is always ≤ 1 and $\frac{\log_2 n}{n}$ is negligible when n is large.

Therefore, Binary Search does approximately $\log_2(n+1)$ comparisons on average for n entries.

4.3.4 Summary of Binary Search

- Divide and conquer strategy
- Examine data item in the middle and search recursively on single half
- Average and worst time complexities are both $\Theta(log(n))$

4.4 Hashing

- Direct Address Table

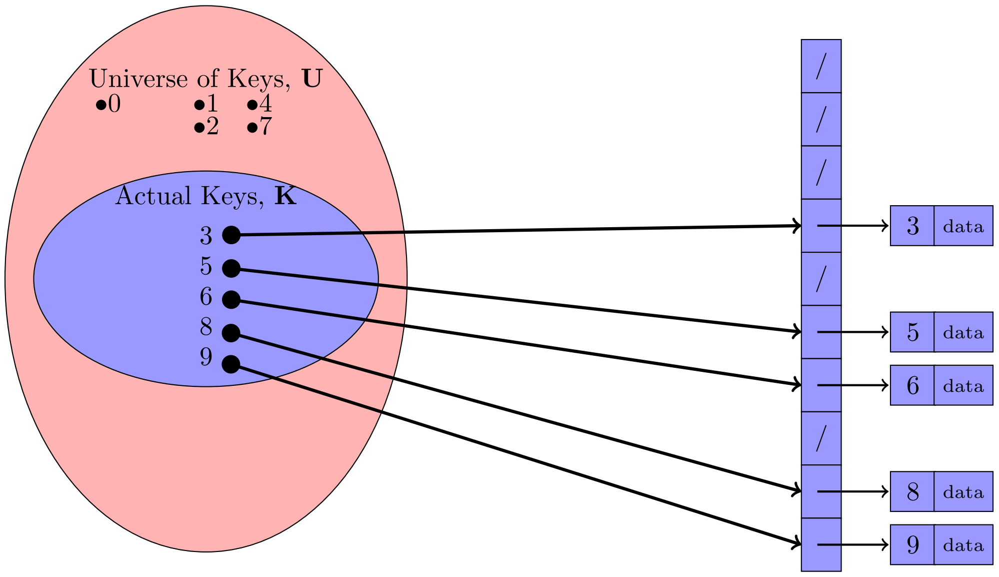


Figure 4.1: Direct Address Table

- Closed Address Hashing
- Open Address Hashing
  - Linear Probing
  - Double Hashing

4.4.1 Direct Address Table

Suppose that the set of actual keys is $K \subseteq \{0, 1, 2, \ldots, m-1\}$ and keys are distinct. We can define an array, $T[0 \ldots m-1]$ or **direct-address table**:

$$T[x.k] = \begin{cases} x, & \text{if } k \in K \wedge x.k = k \text{ ;} \\ NIL, & \text{otherwise.} \end{cases}$$

Thus, operations take $\mathcal{O}(1)$ time.

The direct address tables are impractical when the range of keys can be very large ($m \gg |K|$. e.g. 64-bit numbers (range of $m \approx 18.45 \times 10^{18}$) To overcome the large range issue, we can use a **hash function**, $h(key)$ to map the universe, **U** of all keys into hash table slots, $\{0, 1, 2, \ldots, m-1\}$. However, multiple keys may be mapped to the same slots. It is known as **collision**. See Figure 4.2. Motivation of Hashing is to be able assign a unique array index to every possible key that could occur in an application.

- Key space may be too large for an array on the computer while only a small fraction of the key values will appear.
- The purpose of hashing is to translate an extremely large key space into a reasonably small set of integers.
- A hash function f: key space $\longrightarrow$ hash codes.

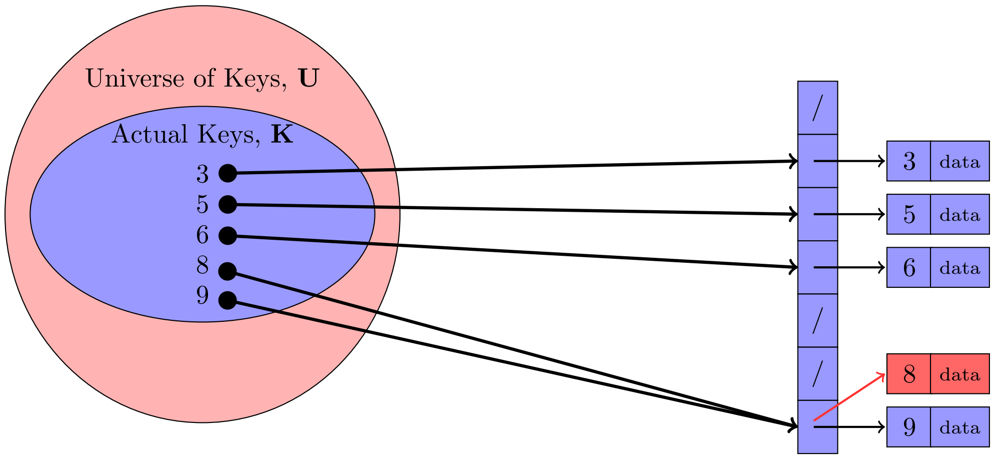


Figure 4.2: Collision Issue in Hash Table

Example: A hash table of 200 entries.
A possible hash function is

$$hash(k) = k \bmod 200$$

When multiple keys are mapped to the same hash code, a collision occurs. e.g $k = 200$ and $k = 400$ are mapped to $hash(k) = 0$.

4.4.2 What hash function to use?

- A hash function **MUST** return a value within the hash table range.
- It should achieve an even distribution of the keys that actually occur across the range of indices.
- It should be easy and quick to compute.
  - If we know nothing about the incoming key distribution, evenly distribute the key range over the hash table slots while avoiding obvious opportunities for clustering.
  - If we have knowledge of the incoming distribution, use a distribution-dependant hash function.

Here we introduce three types of hash functions:

1. **The Division Method**:

$$f(k) = k \bmod m$$

   - The return value is the last four bits of k, $f(k) = k \bmod 16$
   - Avoid to use power of 10 for decimal numbers as keys
   - The best table size is often a prime number not too close to exact powers of 2 for “real” data.
   - Real data may not always evenly distribute
2. **The Folding Method**:
   - All bits contribute to the result

- Partition the key into several parts and combine the parts in a convenient way (e.g. addition or multiplication).
- Example 1: Sum the key value, take modulus of the sum. The sum must be large enough compared to the quotient, M.

```
int h(char x[10])
 {  int i, sum;
  for (sum=0, i=0; i < 10; i++)
    sum += (int) x[i];
  return(sum % M);
 }

```

- Example 2: Mid-square method: Square the key value, take the middle r bits (from the result) for a hash table of 2^r slots

3. **Multiplicative Congruential method**: (pseudo-number generator)

   Step 1: Choose the hash table size, h.

   Step 2: Choose the multiplier, a.

$$a = 8\lfloor\frac{h}{23}\rfloor + 5$$

   Step 3: Define the hash function, f

$$f(k) = (a * k) \bmod h$$

```
>> h = 31
>> a= 8*floor(h/23) +5

a =

    13

>> k = 1:15

k =

   1     2     3     4     5     6     7     8     9    10    11    12    13    14    15

>> fk = mod((a*k),h)

fk =

  13    26     8    21     3    16    29    11    24     6    19     1    14    27     9

```

4.4.3 How to handle collisions?

To handle collision issue in hash table, we can use

- Closed Address Hashing
- Open Address Hashing

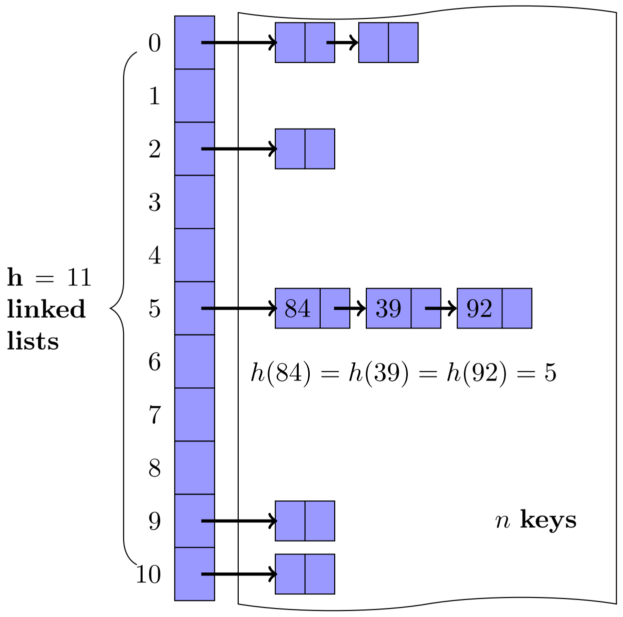


Figure 4.3: Closed Addressing Hash Table

4.4.4 Closed Address Hashing

- Maintains the original hashed address
- Records hashed to the same slot are linked into a list
- The address is closed (fixed). Each key has a corresponding fixed address
- Also called **chained hashing**
- Initially, all entries in the hash table are empty lists.
- All elements with hash address i will be inserted into the linked list $H[i]$.
- If there are n records to store in the hash table, then $\frac{n}{h}$ is the **load factor** of the hash table.
- In closed address hashing, there will be $\frac{n}{h}$ number of elements in each linked list on average.
- During searching, the searched element with hash address i is compared with elements in linked list $H[i]$ sequentially

4.4.4.1 Analysis of Chained Hashing

The **worst case** behaviour of hashing happens when all elements are hashed to the same slot. In this case

- The linked list contains all n elements
- An unsuccessful search will do n key comparisons

- A successful search, assuming the probability of searching for each item is $1/n$, will take

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

- In this analysis, checking if link list is NULL is not counted as a key comparison

The **average case**: If we assume that any given item is equally likely to hash into any of the h slots, an **unsuccessful** search on average does n/h key comparisons.

- An unsuccessful search means searching to the end of the list.
- The number of comparisons is equal to the length of the list.
- The average length of all lists is the load factor, n/h
- Thus the expected number of comparisons in an unsuccessful search is n/h.

Assume that any given item is equally likely to hash into any of the h slots, a **successful** search on average does $\Theta((1 + n/h))$ comparisons.

- We assume that the items are inserted at the end of the list in each slot.
- The expected no. of comparisons in a successful search is 1 more than the no. of comparisons done when the sought after item was inserted into the hash table.
- When the i^{th} item is inserted into the hash table, the average length of all lists is $(i - 1)/h$. So when the i^{th} item is sought for, the no. of comparisons is

$$(1 + \frac{i-1}{h})$$

- So the average number of comparisons over n items is:

$$\begin{aligned}
\frac{1}{n}\sum_{i=1}^{n}(1 + \frac{i-1}{h} &= \frac{1}{n}\sum_{i=1}^{n}(1) + \frac{1}{nh}\sum_{i=1}^{n}(i-1) \\
&= 1 + \frac{1}{nh}\sum_{i=0}^{n-1} i \\
&= 1 + \frac{n-1}{2h}
\end{aligned}$$

- Therefore, a successful search on average does $\Theta(1 + n/h)$ comparisons
- If n is proportional to h, i.e. $n = \mathcal{O}(h)$, then $n/h = \mathcal{O}(h)/h = \mathcal{O}(1)$.
- Thus, each successful search with chained hashing takes **constant time** averagely

4.4.5 Open Address Hashing

- To store all elements in the hash table
- The load factor n/h is never greater than 1
- Collision is handled by **rehashing**, a process to look for an alternative slot.
- the address is open (not fixed)
    - linear probing: the simplest rehashing method
    - Double Hashing

4.4.5.1 Linear Probing

- Suppose the hash function, $h(k)$

$$h(k) = k \bmod m$$

- Key k is hashed to slot $h(k)$ which is non-empty slot
- To rehash, we take j = $h(k)$ + i for $i = 1, 2, \ldots, m-1$
- Repeating the rehash function, $h'(j,i) = j + i \bmod m$ until an empty slot is found
- Consider the linear probing policy for storing the following keys:

$$1055, 1492, 1776, 1812, 1918, 1942$$

  The hash function,

$$h(k) = k \bmod 10$$

- the rehash function is

$$h'(j,1) = (k+1) \bmod m$$

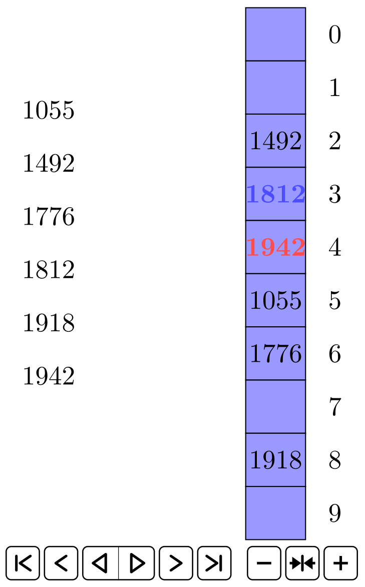


Figure 4.4: Open Addressing via Linear Probe

4.4.5.2 Searching a key in a hash table with rehashing

Algorithm 3 Searching a Key

```
1: function Search (k
2: begin
3: code ← hash(k)
4: loc ← code
5: ans ← ∅
6: while H[loc] ≠ ∅ do
7:     if H[loc].key==k then
8:         begin
9:         ans ← H[loc]
10:        break
11:        end
12:    else
13:        begin
14:        loc ← rehash(loc)
15:        if loc==code then
16:            break
17:        end
18: end
```

Three outcomes of searching:

1. key found at h(k) – Success

2. position empty – Fail

3. probe table (downwards subject to its mod. table size) until key found or empty slot met or whole table searched

4.4.5.3 Limitations of Linear Probe

Linear probing is a simple rehashing method. Unfortunately, it has some limitations. As we can imagine, in open address hashing, searching is expensive when the load factor approaches 1. For linear probing, the problem can happen earlier, if keys are hashed to nearby places in the hash table. We call this phenomenon as **Primary clustering**, which is characterized by long runs of occupied slots. The search time and insertion time will be increased.

4.4.5.4 Double Hashing

Double hashing: It is another better way to alleviate the collision issue. Its rehashing to the new slot is a more random method.
Suppose the hash function, $h(k)$

$$h(k) = k \bmod m$$

If h(k) slot is occupied, a rehash method will be applied to find the new slot.
let $d(k)$ be another hash function.

$$rehash(h, d) = (h(k) + i * d(k)) \bmod m \quad i = 1, 2, \ldots, m-1$$

The hash table size, m, should be a prime number. Using a prime number makes it impossible for any number to divide it evenly, so the probe sequence will eventually check every slot. it is

4.4.5.5 Example: Linear Probing VS Double Hashing

Given the following keys:

$$1051, 1492, 1776, 1812, 1918, 1561, 523, 1340$$

- Linear Probing: $h(k) = k \bmod 11$ and $rehash(j, i) = (j + i) \bmod 11$, $i = 1, 2, \ldots, 10$
- Double Hashing: $h(k) = k \bmod 11$, $d(k) = 1 + (k \bmod 8)$ and $rehash(j, i) = (j + i * d) \bmod 11$, $i = 1, 2, \ldots, 10$

4.4.5.6 Resize the hash table

We can observe that the multiple rehashing is always required when most of slots are occupied. it is unrelated to the rehashing methods. Therefore, we need to resize the hash table when it reaches a maximum load factor.

4.4.5.7 Deletion a key under open addressing

A slot becomes empty when an element is deleted: should be marked as 'obsolete' or 'tombstone' instead, so that searching will not stop there.

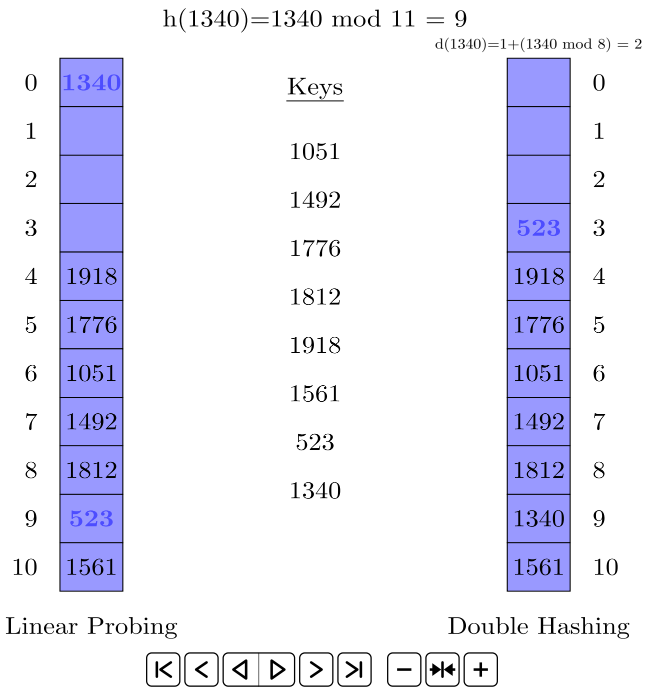


Figure 4.5: Open Addressing via Linear Probe and Double Hashing

4.5 Appendix

$$\begin{aligned}
\sum_{t=1}^{k} t2^{t-1} &= 1\cdot 1 + 2\cdot 2 + 3\cdot 4 + 4\cdot 8 + \ldots + k\cdot 2^{k-1} \\
2\sum_{t=1}^{k} t2^{t-1} &= \qquad 1\cdot 2 + 2\cdot 4 + 3\cdot 8 + \ldots + (k-1)\cdot 2^{k-1} + k\cdot 2^{k} \\
(2-1)\sum_{t=1}^{k} t2^{t-1} &= -1\cdot 1 - 1\cdot 2 - 1\cdot 4 - 1\cdot 8 - \ldots - 1\cdot 2^{k-1} + k\cdot 2^{k} \quad \triangleright \text{ eq. 2 - eq. 1} \\
\sum_{t=1}^{k} t2^{t-1} &= -2^{k} + 1 + k\cdot 2^{k} \quad \triangleright \text{ geometric series} \\
&= 2^{k}(k-1) + 1
\end{aligned}$$