SC1007 Data Structures and Algorithms **2022/23 Semester 2**

8: Searching

School of Computer Science and Engineering *Nanyang Technological University*

8.1 Searching

Searching algorithms are a set of techniques used to find specific data or elements within a collection of data. In computer science, the most common types of searching algorithms are sequential search, binary search and hash tables. In this lecture, these three algorithms and their complexities will be discussed.

Algorithm 1 Generic Search Algorithm

```
function GenericSearching($data, target$)
    while more possible data to examine are availble do
        Examine one datum
        if the datum = search key then
            return $index$
    return $fail$
```

8.2 Sequential Search

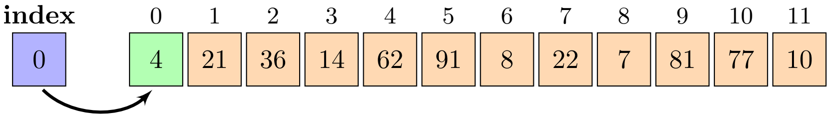
When the given data set is unsorted, we have to check every single element in the data set until either the key is found (successful search) or all elements in the data set have been retrieved once and no match is found (unsuccessful search). The search is called sequential search. It is a **brute-force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

Algorithm 2 Sequential Search

```
function SequentialSearch($data, target$)
    for $i \leftarrow 0$ to length($data$) $- 1$ do
        if $data[i] = target$ then
            return $i$                                  ▷ success
    return $-1$                                         ▷ failure
```

8.2.1 Example: Success Case

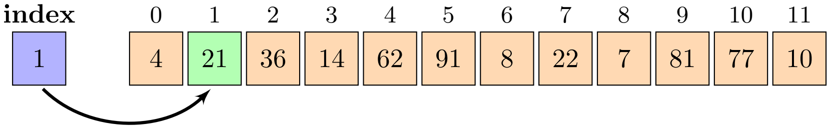
Search key: **14**
Let index =0,



index	0	1	2	3	4	5	6	7	8	9	10	11
0	4	21	36	14	62	91	8	22	7	81	77	10

$4 \neq 14$

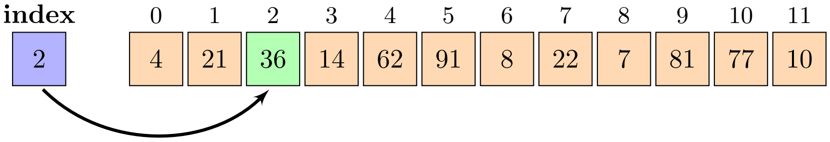
Let index =1,



index	0	1	2	3	4	5	6	7	8	9	10	11
1	4	21	36	14	62	91	8	22	7	81	77	10

$21 \neq 14$

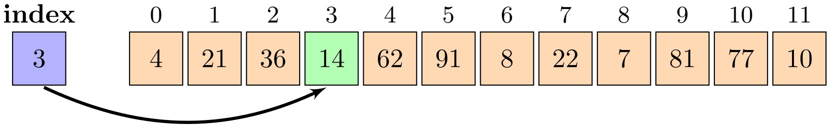
Let index =2,



index	0	1	2	3	4	5	6	7	8	9	10	11
2	4	21	36	14	62	91	8	22	7	81	77	10

$36 \neq 14$

Let index =3, the search key is found



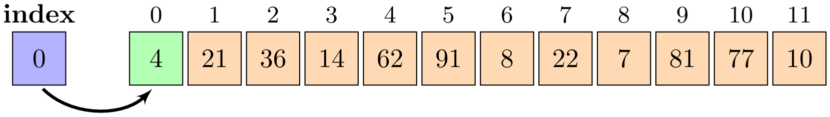
index	0	1	2	3	4	5	6	7	8	9	10	11
3	4	21	36	14	62	91	8	22	7	81	77	10

14 == 14

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

8.2.2 Example: Failure Case

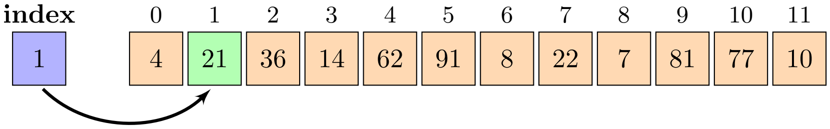
Search key: **72**
Let index =0,



index	0	1	2	3	4	5	6	7	8	9	10	11
0	4	21	36	14	62	91	8	22	7	81	77	10

$4 \neq 72$

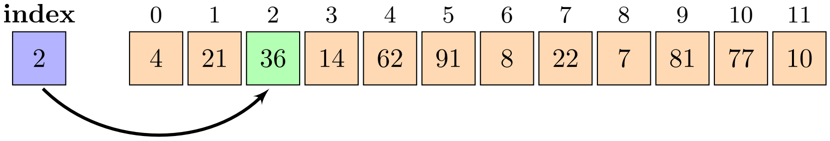
Let index =1,



index	0	1	2	3	4	5	6	7	8	9	10	11
1	4	21	36	14	62	91	8	22	7	81	77	10

$21 \neq 72$

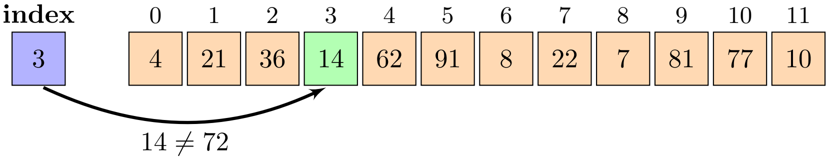
Let index =2,



index	0	1	2	3	4	5	6	7	8	9	10	11
2	4	21	36	14	62	91	8	22	7	81	77	10

$36 \neq 72$

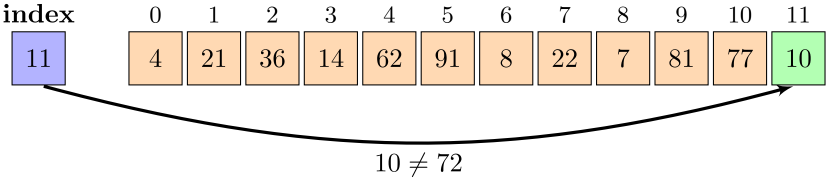
Let index =3,



$$\vdots$$

$$\vdots$$

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.

8.2.3 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 8.2.2.
    - 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

$q = \frac{1}{2}$

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

8.3 Binary Search

In the previous section, we have discussed the time complexity of sequential search. Searching an item from an unsorted array or a linked list will take $\Theta(n)$ on average. To improve the searching performance, the data set need to be sorted out first. Let us assume that the data is sorted in order here and represent in a binary tree. We can use the information of its order to reduce the search work. Binary search is an example which is using **decrease-and-conquer** approach. This algorithm divides a problem into two smaller sub-problems, one of which does not even have to be solved. Thus, the search time of binary search algorithm is reduced significantly.

Binary search first compares a search key with the root node. If they match, then the algorithm stops; otherwise, the same operation is repeated recursively for either the left subtree if the search key is lesser than the middle element or the right subtree if the search key is greater.

Binary Search

```
BTNode* findBSTNode(BTNode *cur, char c){                    -->        T(n)
{
    if (cur == NULL) {
       printf("Not Found\n");
       return cur;
    }

    if(c==cur->item){
       printf("Found\n");
       return cur;
    }                                                        -->           c

    if(c<cur->item)
       return findBSTNode(cur->left,c);                      -->  T((n-1)/2)
    else
       return findBSTNode(cur->right,c);                     -->  T((n-1)/2)
}
```

Listing 1: Recursive Version

- Best-case analysis: 1 comparison against key (the first item is the search key)

- Worst-case analysis: Refer to Listing 1, each recursive function call will visit one of the subtrees. The running time in each call is constant, c. If the current node's item and search key do not match, there will be a recursive call by passing either left subtree or right subtree. When the number of node is 1, the node is not the search key. It will make a recursive call. This is the last recursive call which will print "Not Found" (no match). How many recursive call will be made? Here, we need to assume that the BST is a complete binary tree. The number of recursive call is related to the height of the tree.

$$\begin{aligned}
T(n) &= T(\frac{n-1}{2}) + c \\
&= T(\frac{(\frac{n-1}{2})-1}{2}) + c + c = T(\frac{n-1-2}{2^2}) + 2c \\
&= T(\frac{(\frac{n-1-2}{2^2})-1}{2}) + 3c = T(\frac{n-1-2-2^2}{2^3}) + 3c \\
&\ldots \\
&= T(\frac{n-(1+2+\ldots+2^{k-2}+2^{k-1})}{2^k}) + kc \\
&= T(\frac{n-2^k+1}{2^k}) + kc
\end{aligned}$$

From the equation above, k is the number of recursive call. At the k^{th} recursive call, we have

$$\begin{aligned}
0 < \frac{n-2^k+1}{2^k} &\leq 1 \\
0 < \frac{n+1}{2^k} - 1 &\leq 1 \\
1 < \frac{n+1}{2^k} &\leq 2 \\
2^k < n+1 &\leq 2^{k+1} \\
k < \log_2(n+1) &\leq k+1 \\
\lceil \log_2(n+1) \rceil &= k+1
\end{aligned}$$

$\therefore T(n) = c_1 + c_2\lceil\log_2(n+1)\rceil$ where c_1 and c_2 are some constant numbers.
The worst-case running time is in $\Theta(\log_2 n)$

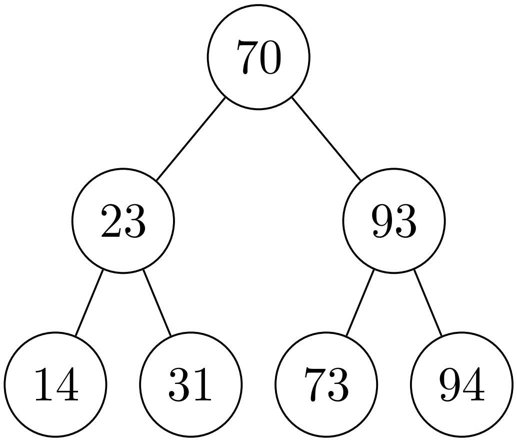
- Average-case analysis:
To analysis the average case, we first consider two scenario, successful search and failed search. By Law of Expectations:

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

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



A Full Binary Tree

In this example, the data are 14, 23, 31, 70, 73 and 93. They are sorted in order and inserted into a binary tree. We can observe that if the search key is at the 4^{th} position (70), we just need one comparison. If the search key is at the 2^{nd} (23) or the 6^{th} (93) 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 Mathematics Revision notes.

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 $\frac{\log_2(n+1)}{n}$ is negligible when n is large.

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

8.3.1 Summary of Binary Search

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

It is noted that we have make an assumption that the tree must be a complete binary tree on the time complexity analysis of the binary search. Having a complete binary search tree, we can achieve our search time in $\mathcal{O}(\log n)$.

If the tree is not a complete binary tree, the search time can be in $\mathcal{O}(n)$. We can consider a linked list as a special case of a binary tree. The root of the tree is the first node. Each node has only one child. In such a binary tree, the binary search has no difference with the sequential search.

Even we have sorted the data in proper order for searching, we still need to construct a complete binary tree or a balanced binary tree. So that, we can have $\mathcal{O}(logn)$ searching time.

8.4 Hashing

A hash table is a data structure that allows efficient lookup, insertion, and deletion of key-value pairs. It works by mapping each key to a unique index in an array, called the hash table. The value associated with the key is then stored at the corresponding index in the array.

It is a typical space-and-time trade-off strategy in algorithm. To achieve search time in $\mathcal{O}(1)$, memory usage have to be increased. In this section, we will discuss the following hashing approach:

- Direct Address Table
- Closed Address Hashing
- Open Address Hashing
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

8.4.1 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 8.2. Motivation of Hashing is to be able assign a unique array index to every possible key that could occur in an application.

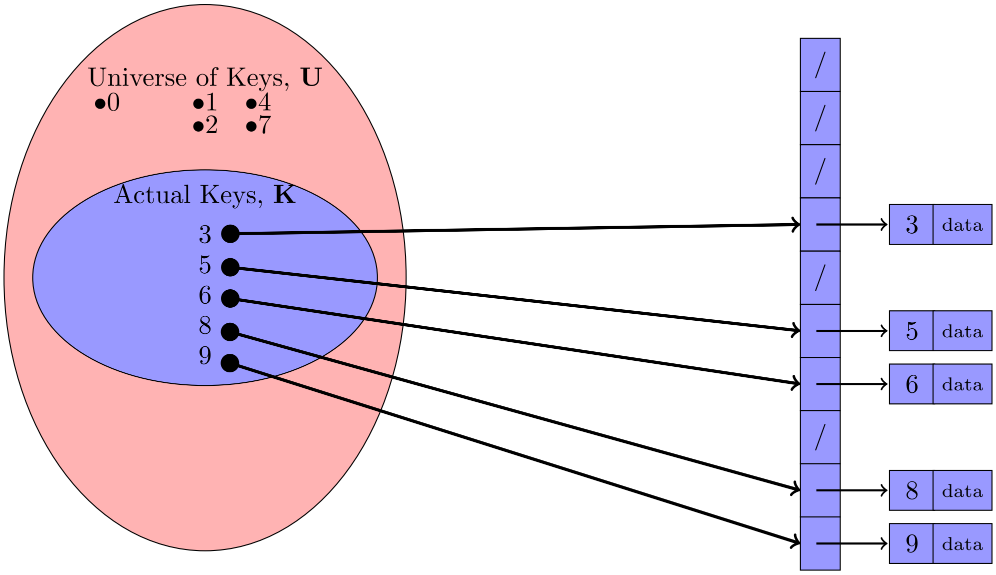


Figure 8.1: Direct Address Table

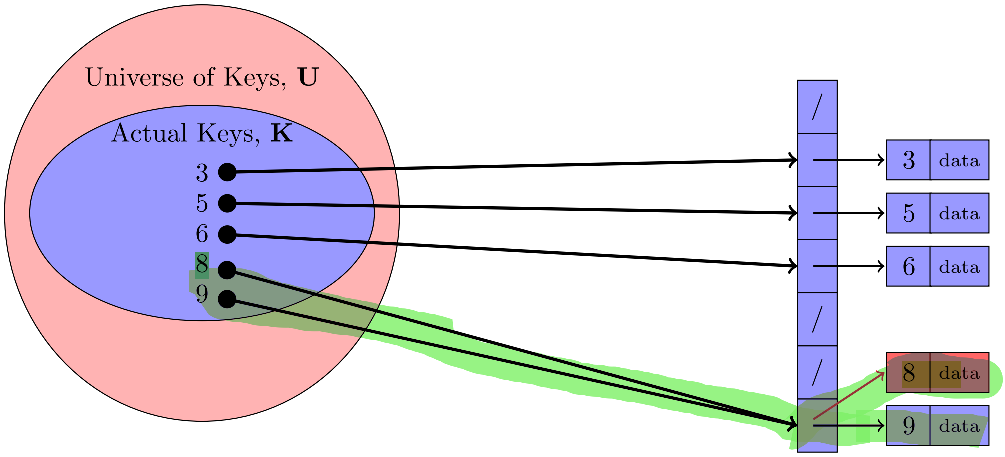


Figure 8.2: Collision Issue in Hash Table

- 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.

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 $hash(k) = 0$.

8.4.2 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**:

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

   Modulo Arithmetic

   - 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

```

8.4.3 How to handle collisions?

To handle collision issue in hash table, we can use

- Closed Address Hashing
- Open Address Hashing

8.4.4 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 $\alpha = \frac{n}{h}$ is the **load factor** of the hash table.

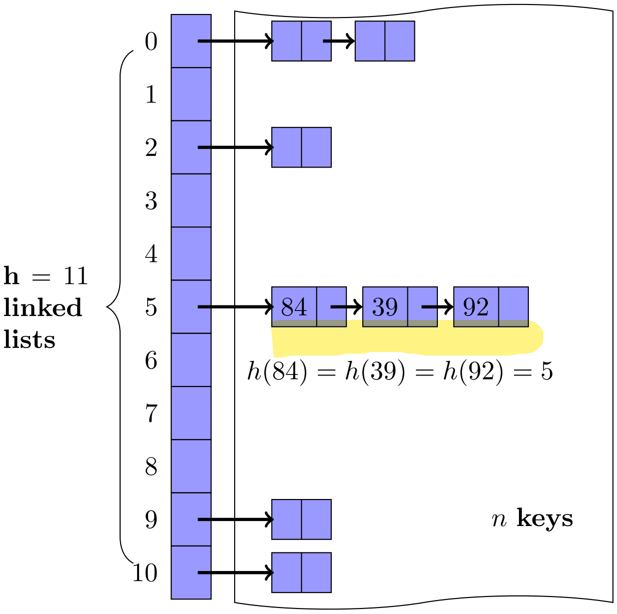


Figure 8.3: Closed Addressing Hash Table

- In closed address hashing, there will be α 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

8.4.4.1 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, α
- Thus the expected number of comparisons in an unsuccessful search is α.

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

8.4.5 Open Address Hashing

- To store all elements in the hash table
- The load factor $\alpha=n/h$ is never greater than 1
- When collision occurs, probe is required for an alternative slot
- The address is open (not fixed)
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

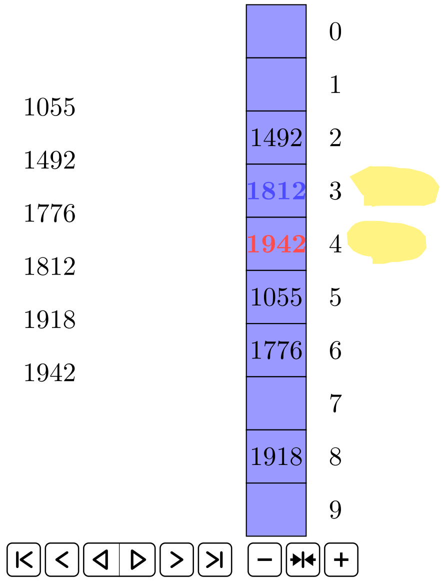


Figure 8.4: Open Addressing via Linear Probe

8.4.5.1 Linear Probing

- Suppose the hash function, $H'(k) = k \bmod h$

- If $H'(k)$ is an empty slot, key k will be hashed to $H'(k)$.

- Otherwise, we need to probe the next empty slot by taking $(H'(k) + 1) \bmod h$

- Repeating the probing function,

  $$H(k, i) = (k + i) \bmod h$$

  for $i = 1, 2, \ldots, h - 1$ until an empty slot is found

- Consider the linear probing policy for storing the following keys:

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

  The linear probing hash function, $H(k, i) = (k + i) \bmod 10$

8.4.5.2 Searching A Key In A Hash Table With Probing New Slots

Algorithm 3 Searching a Key

```
function SEARCH($k$)
    $code \leftarrow hash(k)$
    $loc \leftarrow code$
    $ans \leftarrow \emptyset$
    while $H[loc] \neq \emptyset$ do
        if $H[loc].key == k$ then
            $ans \leftarrow H[loc]$
            break
        else
            $loc \leftarrow probe(loc)$
            if $loc == code$ then
                break
```

Three outcomes of searching:

1. key found at $H[loc]$ – Success
2. position empty – Fail
3. all slots are searched

8.4.5.3 Limitations of Linear Probe

Linear probing is a simple probing new slot method. Unfortunately, it has some limitations. As we can imagine, in open address hashing, searching becomes expensive when the load factor α approaches to 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.

8.4.5.4 Quadratic Probing

Instead of linearly probing the next empty slot in the hash table, we can probe the slot quadratically by the following probe function:

$$H(k,i) = (k + c_1 i + c_2 i^2) \bmod h$$

However, c_1, c_2 and h need to be carefully selected to ensure that all hash slots can be probed. For $h = 2^n$, a good choice for the constants are $c_1 = c_2 = \frac{1}{2}$.

Although quadratic probing can avoid primary clustering, it still face another phenomenon known as **secondary clustering**. If two keys have the same initial probe position, their probe sequences will be the same. It implies that the later inserted keys will take longer search time.

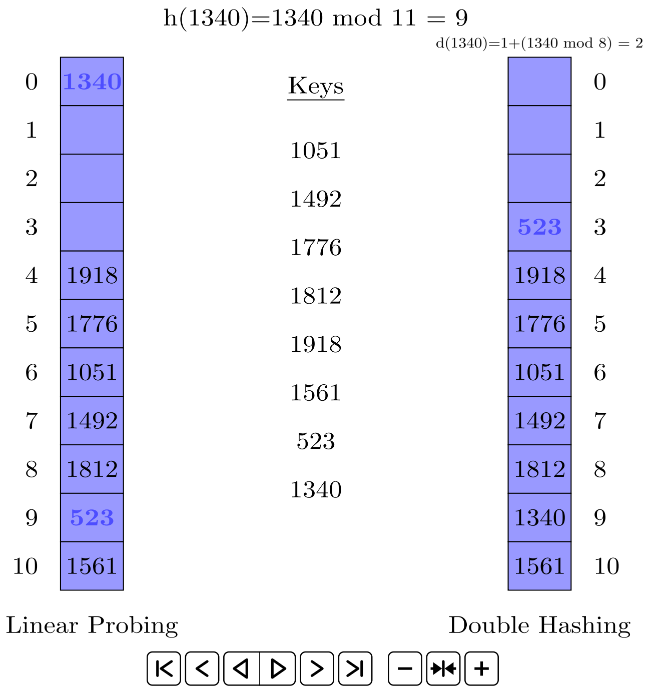


Figure 8.5: Open Addressing via Linear Probe and Double Hashing

8.4.5.5 Double Hashing

It is another better way to alleviate the collision issue. Its probing to the new slot is a more random method. Suppose the hash function, $H(k,i)$

$$H(k,i) = (k + iD(k)) \bmod h$$

where $D(k)$ be another hash function and $i \in [0, h-1]$.

If $H(k,0)$ slot is occupied, a probe method will be applied to find the new slot.

The hash table size, h, 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.

8.4.5.6 Example: Linear Probing VS Double Hashing

Given the following keys:

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

- Linear Probing: $H(k,i) = (k+i) \bmod 11$ $i = 0, 1, 2, \ldots, 10$
- Double Hashing: $H(k,i) = (k + iD(k)) \bmod 11$, where $D(k) = 1 + (k \bmod 8)$ and $i = 0, 1, 2, \ldots, 10$

8.4.5.7 Time Complexity

The time complexity of open addressing methods as follow: Linear Probing:

- Successful Search: $\frac{1}{2}(1+\frac{1}{1-\alpha})$
- Unsuccessful Search: $\frac{1}{2}(1+(\frac{1}{1-\alpha})^2)$

Quadratic Probing (*approximate any open addressing method which causes secondary clusters)

- Successful Search: $1-\ln(1-\alpha)-\frac{\alpha}{2}$
- Unsuccessful Search: $\frac{1}{1-\alpha}-\alpha-\ln(1-\alpha)$

Double Hashing:

- Successful Search: $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$
- Unsuccessful Search: $\frac{1}{1-\alpha}$

The proof can be found in [Knuth97]. In this course, you do not need to memorize these time complexities but you need to know that they are functions of $\frac{1}{1-\alpha}$. What does it mean? Think about it.

8.4.5.8 Rehashing The Hash Table

We can observe that the multiple probing is always required when most of slots are occupied (α is approaching to 1). it is unrelated to the probing methods. Therefore, we need to rehash the hash table when it reaches the load factor threshold.

8.4.5.9 Deletion A Key Under Open Addressing

Removing a key from an open-addressing hash table is a tedious task. Instead of deleting the key, we leave the key in the table but make a maker like 'obsolete' or 'tombstone' to indicate that it is deleted. It can be overwritten by a new key when it is inserted to the slot.

We may do a "garbage collection" when we have done a large number of deletions. It can improve the insertion and searching time.

8.5 Appendix

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

References

[Knuth97] Knuth, Donald E. "The Art of Computer Programming, Volume 3 (3rd Ed.): Sorting and Searching," *Addison Wesley Longman Publishing Co., Inc.*, 1997.