Data structure and Algorithms

Searching – Giải thuật tìm kiếm

Thanh-Hai Tran

Electronics and Computer Engineering
School of Electronics and Telecommunications

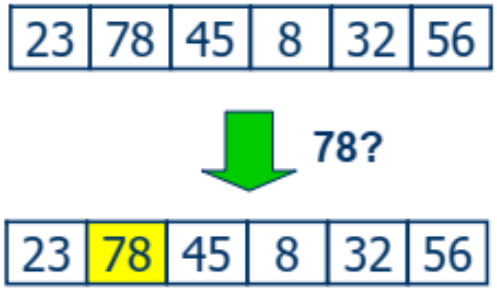
Hanoi University of Science and Technology
1 Dai Co Viet - Hanoi - Vietnam

Outline

- **Introduction**
- **Algorithms for searching a element / key**
  - Comparison based searching (indirect search)
  - Key value based searching (direct search)
- **Algorithms for pattern searching**
  - Brute-force algorithm
  - Knuth-Morris-Pratt (KMP) algorithm
- **References**

Introduction to searching

- **Searching means to find whether a particular value is present in an array or not**
  - If the value is present in the array, then searching is said to be **successful** and the searching process gives the location of that value in the array
  - If the value is not present in the array, the searching process displays an appropriate message and in this case searching is said to be **unsuccessful**



Outline

- **Introduction**
- **Algorithms for searching a element / key**
  - Comparison based searching (indirect search)
  - Key value based searching (direct search)
- **Algorithms for pattern searching**
  - Brute-force algorithm
  - Knuth-Morris-Pratt (KMP) algorithm
- **References**

Linear search (sequential search)

- Linear search, also called as *sequential search*, is a very simple method used for searching an array for a particular value
- It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found
- Linear search is mostly used to search an unordered list of elements (array in which data elements are not sorted)
- Complexity:
  - Best case: O(1) (1 comparision)
  - Worst case: O(n) (n comparisions)
  - Average case : O(n) ((n+1)/2 comparisions)

Linear search

```
int SequentialSearch(int A[], int N, int K) {
    int i=0;
    while (i<N && A[i] != K) i++;
    if (i<N) return i;  //Found
    return -1;          //Not found
}
```

Binary search

- Binary search is a searching algorithm that works efficiently with a sorted list

*Ví dụ: Với dãy khoá

11 23 36 42 58 65 74 87 94 99

a) Nếu X = 23: phép tìm kiếm được thoả mãn và các bước sẽ như sau:

[11 23 36 42 **58** 65 74 87 94 99]

[11 **23** 36 42]

b) Nếu X = 71: phép tìm kiếm không thoả

[11 23 36 42 **58** 65 74 87 94 99]

[65 74 **87** 94 99]

[**65** 74]

[**74**]

Binary search

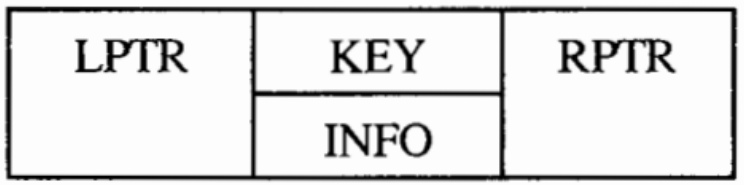
```
int BSearch(int K, int A[], int b, int e) {
    if (b>e) return -1;  //Not found
    int m = (b+e)/2;
    if (K == A[m]) return m;  //found
    else
        if (K<A[m])
            return BSearch(K, A, b, m-1);
        else
            return BSearch(K, A, m+1, e);
}
int BinarySearch(int K, int A[], int N){
    return BSearch(K, A, 0, N-1);
}
```

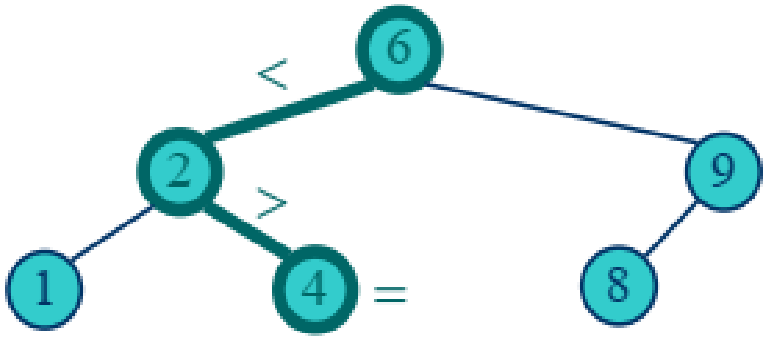
Binary search

- **Complexity: $O(\log_2(n))$**
- **Require the list to be ordered priori**
- **If the list changes a lot (insert, delete) then the cost to re-arrange (Sorting) the list to be taken into account**
- **To overcome the following issues:**
  - Each key is a node
  - List is organized as a Binary Search Tree (BST)

Search using Binary Search Tree

- **Main steps:**
  - If BST = NULL: not found
  - Otherwise:
    - Compare the key at the root node (rk) with the query key (qk):
      - rk = qk: found
      - qk < rk => search in the left subtree
      - qk > rk => search in the right subtree





Search using Binary Search Tree

Algorithm BST-Recursive(T, key)

{T là con trỏ trỏ tới gốc của cây; key là giá trị cần tìm, trả ra con trỏ trỏ tới nút chứa giá trị cần tìm }

1. If (T = NULL) then return NULL;
2. If (key < INFO(T)) return BST-Recursive(LPTR(T), key);
3. Else if (key > INFO(T)) return BST-Recursive(RPTR(T), key);
4. Else return T;

Search in BST and inserting if not found

- Requirement: Find a node having value X in BST, the root node of BST is pointed by T.
- If found: return a pointer q that points to the node having key X
- Otherwise: insert X into BST, move the pointer q to that inserted node and give a notice
- Algorithm: **INSERT_BST(T, X, q)**
  - Step 1: Initialize p = NULL; q = T;
  - Step 2: Search While (q <> NULL)

    case: X < KEY(q): p = q; q = LEFT(q);

    X = KEY(q): return

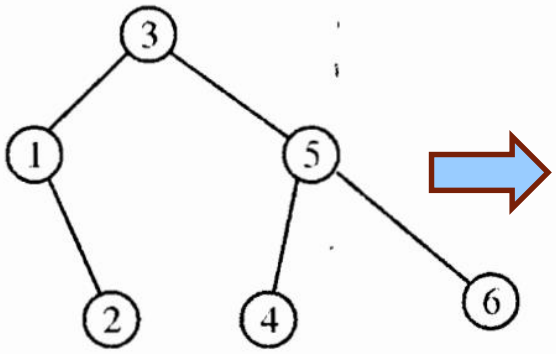
    X > KEY(q): p = q; q = RIGHT(q);

Search in BST and inserting if not found

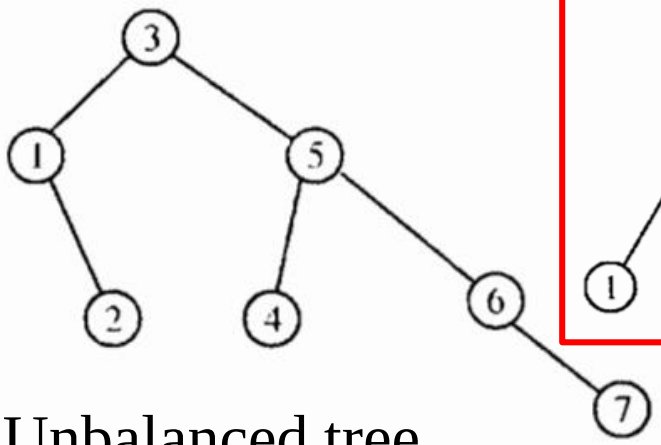
- Algorithm(cont): **INSERT_BST(T, X, q)**
  - Step 1: Initialize p = NULL; q = T;
  - Step 2: Search
  - Step 3: Insert
    - **new (q)** {KEY(q) = X; LEFT(q) = RIGHT(q) = NULL};
    - Case:
      - T == NULL: T = q;
      - X < KEY(p): LEFT(p) = q;
      - Else X > KEY(p): RIGHT(p)= q;
- **Conclusion: to build BST from an unordered list:**
  - Init an empty tree: T = NULL;
  - For (i =0; I < N; i++) **INSERT_BST**(T, x[i], q);

Binary search on BST

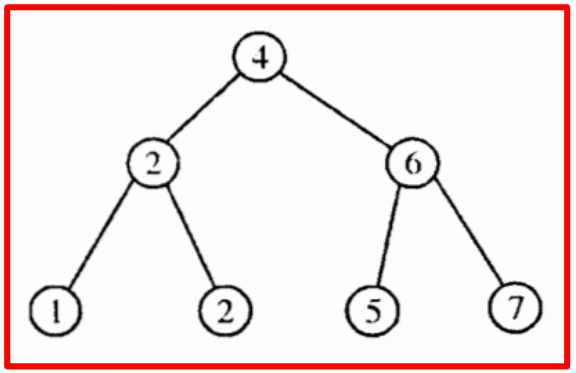
- **BST depends completely on the input list**
  - Balanced binary tree the height of tree is $\lceil log_2(n+1) \rceil$. Then the complexity of search algo: O($\log_2$(n)) (expected)
  - Degenerated tree: (the list is ordered). Then the cost = cost of sequential search = O(n)
- **Question: how to convert to Balanced Binary Tree ?**



Original tree



Unbalanced tree



Balanced tree

Remove a node from BST

- **Requirement:** delete a node having key value X in BST, with root node pointed by the pointer T
- **If the X node is a leaf:** we don't need to find a node to replace, just to assign parent's pointer to that node to NULL;
- **If the X node has only one child (left or right child)**: the node that replaces the X node is the root node of the left subtree / right subtree. The pointer from its parent is updated too.
- **Genera case: the node that replaces X node will be**
  - The extreme right node of the left subtree
  - The extreme left node of the right subtree.

Remove a node from BST

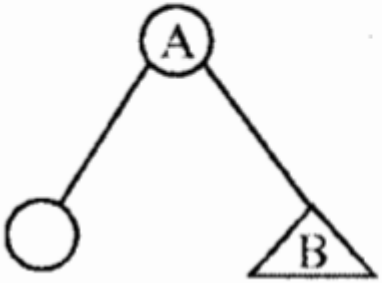
- **Remove the leaf node**

○ Chỉ nút bị loại bỏ

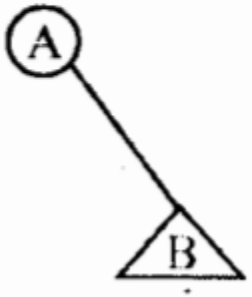
Δ chỉ cây con

Trước

1) Trường hợp nút lá



Sau



Remove a node from BST

- **The node X has one child**

○ Chỉ nút bị loại bỏ

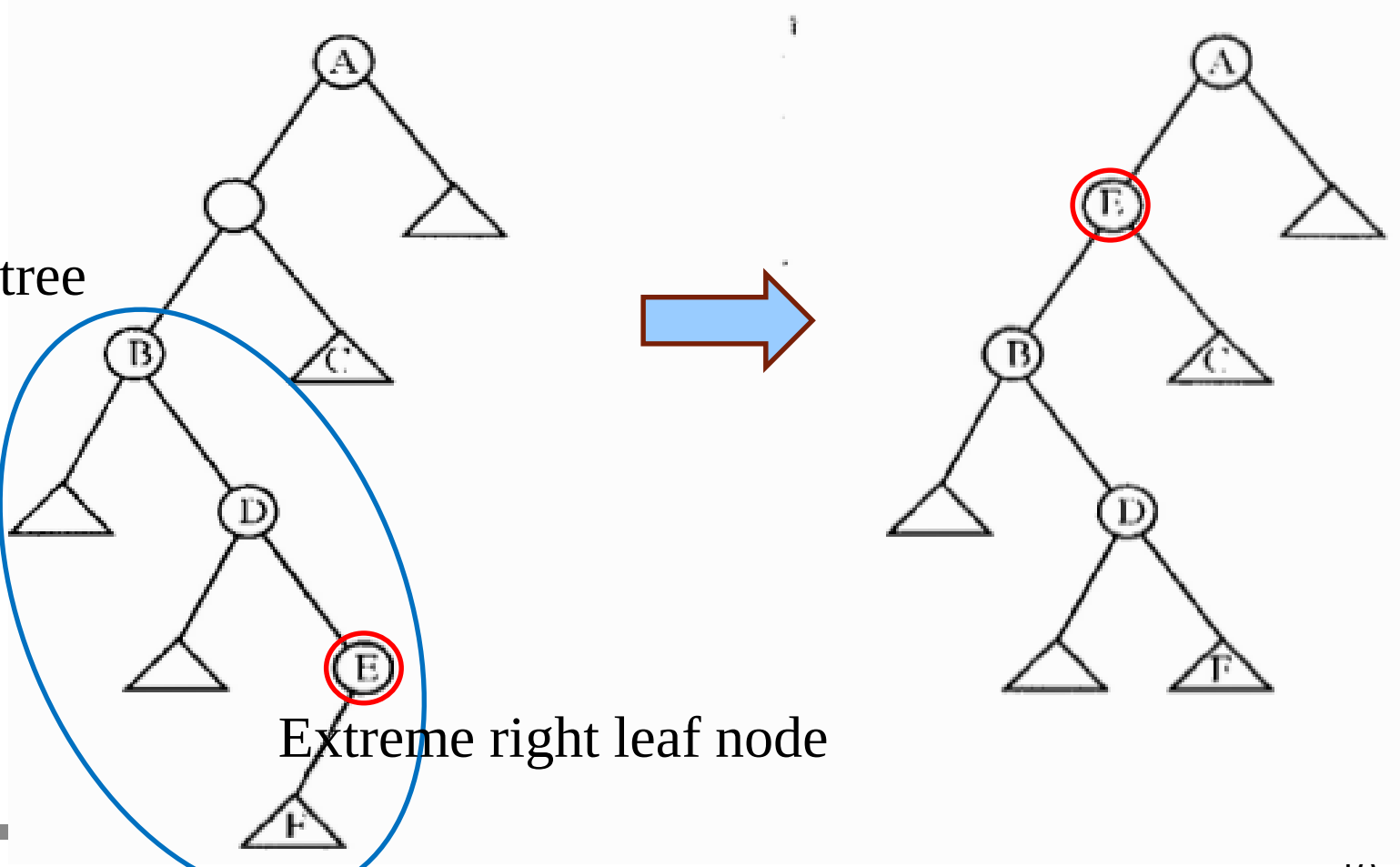
Δ chỉ cây con

Remove a node from BST

- **General case**

○ Chỉ nút bị loại bỏ

Δ chỉ cây con



Remove a node from BST

```
Procedure BSTDEL (Q)
{Trong thủ tục này ta phải hiểu Q là đại diện cho LPTR(R) hoặc
RPTR(R). Giả sử nó chính là LPTR(R) thì câu lệnh Q := RPTR(P) tương
đương với LPTR(R) : = RPTR(P)}
1. {Xử lý trường hợp nút lá và nửa lá}
    P := Q;
    if LPTR(P) = null then begin
            Q := RPTR(P);
            call dispose(P);
                        end;
    if RPTR(P) = null then begin
            Q := LPTR(P);
            call dispose(P);
                        end;
```

Remove a node from BST

```
2. {Xử lý trường hợp tổng quát}
    T := LPTR(P);
    if RPTR(T) = null then begin
            RPTR(T) := RPTR(P);
            call dispose(P);
                        end;
    S := RPTR(T); [Tìm nút thay thế là nút cực phải của cây con trái}
    while RPTR(S) ≠ null do begin
            T := S;
            S := RPTR(T)
                        end;
    RPTR(S) := RPTR(P)
    RPTR(T) := LPTR(S);
    LPTR(S):= LPTR(P);
    Q := S;
    call dispose(P);
3.  return
```

Search using BST

```
Algorithm BST(T, key)
1.  q= T ; {Khởi tạo biến con trỏ để duyệt cây}
2.  while q < > NULL do begin
        if (INFO(q) = key then return q;
        else begin
            if (INFO(q) < key) then q = RPTR(q);
            else q = LPTR(q);
        end.
    end.
3. Return NULL;
```

Direct searching using key

- Linear search has a running time proportional to $O(n)$
- Binary search takes time proportional to $O(\log_2(n))$
- Binary search and binary search trees are efficient algorithms to search for an element
- But what if we want to perform the search operation in time proportional to $O(1)$
- In other words, is there a way to search an array in constant time, irrespective if its size?

Outline

- **Introduction**
- **Algorithms for searching a element / key**
  - Comparison based searching (indirect search)
  - Key value based searching (direct search)
- **Algorithms for pattern searching**
  - Brute-force algorithm
  - Knuth-Morris-Pratt (KMP) algorithm
- **References**

Example

Key	Array of Employees' Records
Key 0 → [0]	Employee record with Emp_ID 0
Key 1 → [1]	Employee record with Emp_ID 1
Key 2 → [2]	Employee record with Emp_ID 2
.....	.....
.....	.....
Key 98 → [98]	Employee record with Emp_ID 98
Key 99 → [99]	Employee record with Emp_ID 99

Example

Key	Array of Employees' Records
Key 00000 ⟶ [0]	Employee record with Emp_ID 00000
.....	.....
Key n ⟶ [n]	Employee record with Emp_ID n
.....	.....
Key 99998 ⟶ [99998]	Employee record with Emp_ID 99998
Key 99999 ⟶ [99999]	Employee record with Emp_ID 99999

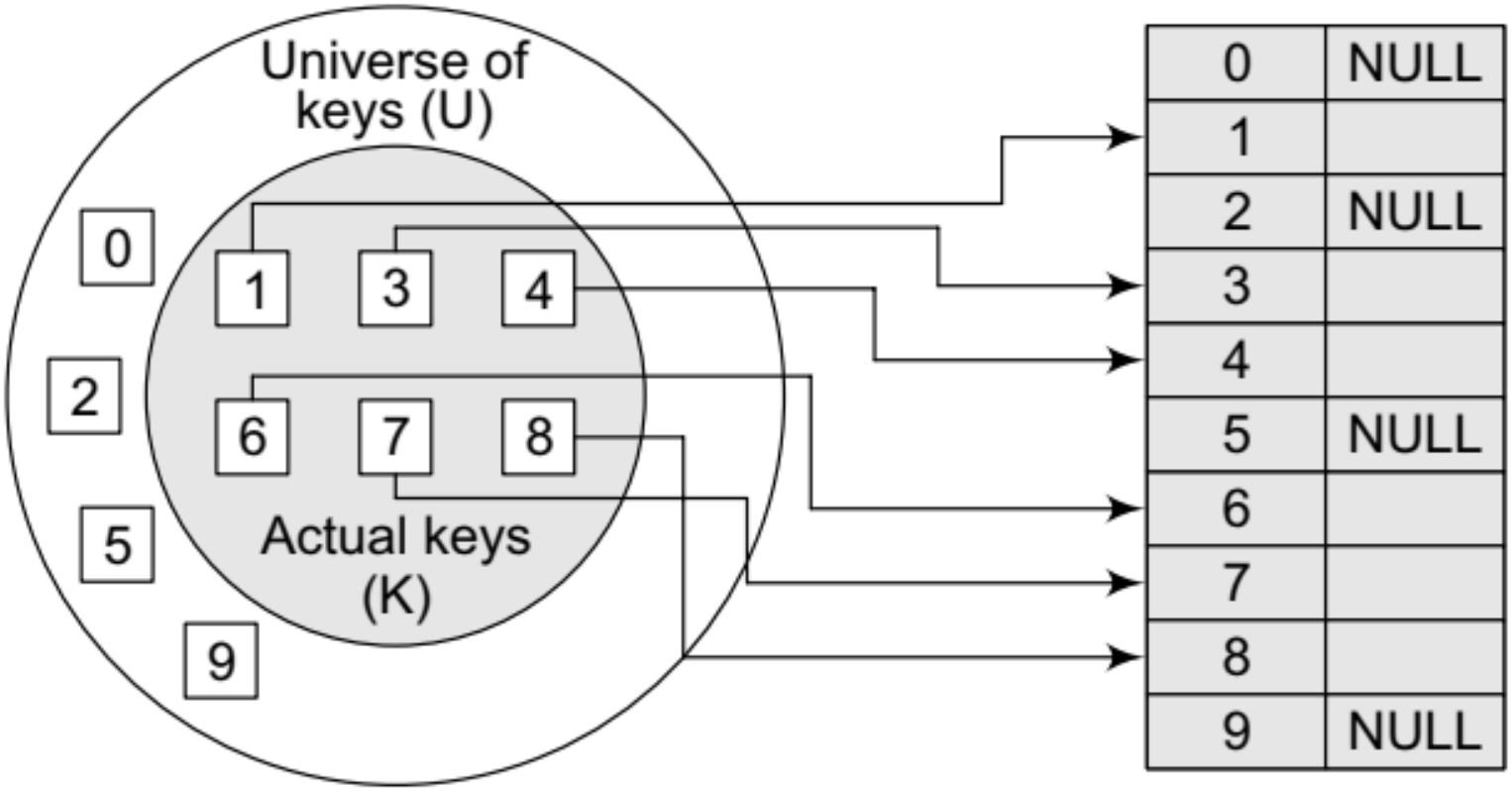
Solution ?

- The elements are not stored according to the *value* of the key.
- So in this case, we need a way to convert a five-digit key number to a two-digit array index.
- We need a function which will do the transformation.
- In this case, we will use the term *hash table* for an array
- The function that will carry out the transformation will be called a *hash function.*

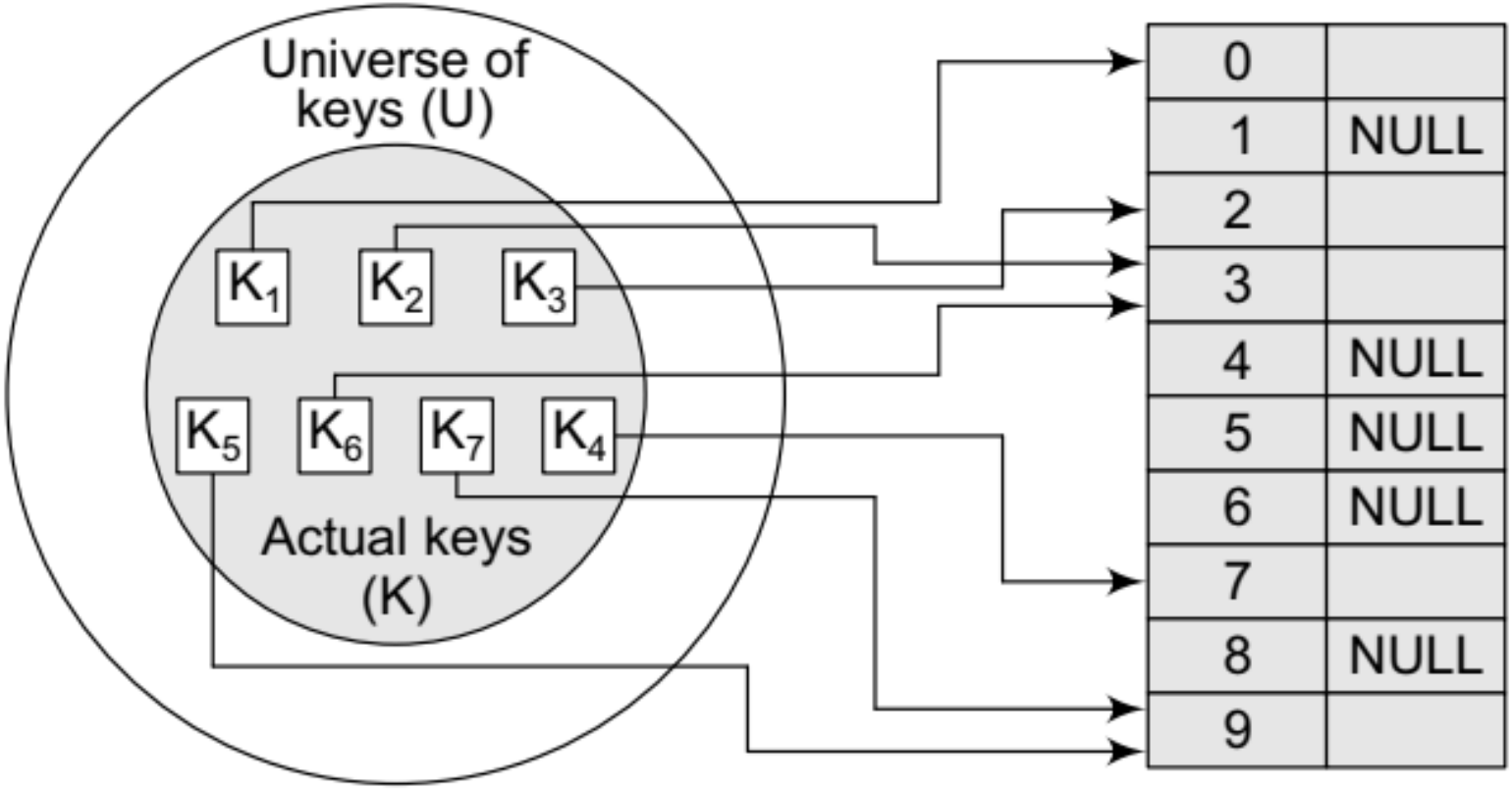
Hash tables

- Hash table is a data structure in which keys are mapped to array positions by a hash function
- In the example discussed here we will use a hash function that extracts the last two digits of the key.
- We map the keys to array locations or array indices
- A value stored in a hash table can be searched in O(1) time by using a hash function which generates an address from the key

Direct map


Universe of keys (U)
0
1
2
3
4
5
6
7
8
9
Actual keys (K)
0 NULL
1
2 NULL
3
4
5 NULL
6
7
8
9 NULL

Direct map – collision problem



- The main goal of using a hash function is to reduce the range of array indices that have to be handled.
- Thus, instead of having U values, we just need K values, thereby reducing the amount of storage space required

Hash functions

- A hash function is a mathematical formula which, when applied to a key, produces an integer which can be used as an index for the key in the hash table
- The main aim of a hash function is that elements should be relatively, randomly, and uniformly distributed
- It produces a unique set of integers within some suitable range in order to reduce the number of collisions
- A good hash function can only minimize the number of collisions by **spreading** the elements uniformly throughout the array

Properties of a Good Hash Function

- **Low-cost:** The cost of executing a hash function must be small, so that using the hashing technique becomes preferable over other approaches
- **Determinism:** A hash procedure must be deterministic. This means that the same hash value must be generated for a given input value
- **Uniformity:** A good hash function must map the keys as evenly as possible over its output range**.** The property of uniformity also minimizes the number of collisions

Hash functions

- Division method
- Multiplication method
- Mid-Square method
- Folding method

Division method

- It is the most simple method of hashing an integer x
- This method divides x by M and then uses the remainder obtained.
- In this case, the hash function can be given as **:**

$$h(x) = x \bmod M$$

```
int const M = 97; // a prime number
int h (int x)
{ return (x % M); }
```

- **Example:** Calculate the hash values of keys 1234 and 5462. h(1234) = 1234%97 = 70, h(5462) = 16
- **How to choose M ?:** choose M* to be a largest prime number < M

Multiplication method

- *Step 1*: Choose a constant A such that 0 < A < 1.
- *Step 2*: Multiply the key k by A
- *Step 3*: Extract the fractional part of kA
- *Step 4*: Multiply the result of Step 3 by the size of hash table (m)

$$h(k) = \lfloor m\ (kA \bmod 1) \rfloor$$

- Knuth has suggested that the best choice of A is

$$(\sqrt{5} - 1)/2 = 0.6180339887$$

Multiplication method

- **Example:** Given a hash table of size 1000, map the key 12345 to an appropriate location in the hash table
- **Solution:** We will use A = 0.618033, m = 1000, and k = 12345

$$
\begin{aligned}
h(12345) &= \lfloor\ 1000\ (12345 \times 0.618033 \bmod 1)\ \rfloor \\
&= \lfloor\ 1000\ (7629.617385 \bmod 1)\ \rfloor \\
&= \lfloor\ 1000\ (0.617385)\ \rfloor \\
&= \lfloor\ 617.385\ \rfloor \\
&= 617
\end{aligned}
$$

Mid-Square method

- *Step 1*: Square the value of the key. That is, find k^2.
- *Step 2*: Extract the middle r digits of the result obtained in Step 1.
- In the mid-square method, the same r digits must be chosen from all the keys. Therefore, the ash function can be given as

$$h(k) = s$$

- where s is obtained by selecting r digits from k^2

Mid-square method

- **Example**: Calculate the hash value for keys 1234 and 5642 using the mid-square method. The hash table has 100 memory locations
- **Solution:**
  - Note that the hash table has 100 memory locations whose indices vary from 0 to 99.
  - This means that only two digits are needed to map the key to a location in the hash table, so r = 2

When $k = 1234$, $k^2 = 1522756$, $h(1234) = 27$

When $k = 5642$, $k^2 = 31832164$, $h(5642) = 21$

Observe that the 3rd and 4th digits starting from the right are chosen.

Folding method

- *Step 1*: Divide the key value into a number of parts. That is, divide k into parts k1, k2, ..., kn, where each part has the same number of digits except the last part which may have lesser digits than the other parts
- *Step 2*: Add the individual parts. That is, obtain the sum of k1 + k2 + ... + kn. The hash value is produced by ignoring the last carry, if any
- Note that the number of digits in each part of the key will vary depending upon the size of the hash table.

Folding method

- **Example:** Given a hash table of 100 locations, calculate the hash value using folding method for keys 5678, 321, and 34567
- **Solution:** Since there are 100 memory locations to address, we will break the key into parts where each part (except the last) will contain two digits. The hash values can be obtained as shown below

key	5678	321	34567
Parts	56 and 78	32 and 1	34, 56 and 7
Sum	134	33	97
Hash value	34 (ignore the last carry)	33	97

Collisions

- Two records cannot be stored in the same location
  - Key:
    - 123, 321, 234, 432, 345, 543, 678...
  - m = 7, m* = 7
  - Address function:

    h(k) = k mod m* = k mod 7
  - Collision:
    - h(543) = h(123)
    - h(678) = h(321)

k	h(k)
123	4
321	6
234	3
432	5
345	2
543	**4**
678	**6**

Address	k
0	
1	
2	345
3	234
4	**123**
5	432
6	**321**

- The two most popular methods of resolving collisions
  - Open addressing
  - Chaining

Collision resolution by open addressing

- Once a collision takes place, open addressing or closed hashing computes new positions using a probe sequence and the next record is stored in that position
- In this technique, all the values are stored in the hash table. The hash table contains two types of values:
  - *sentinel values* (e.g., –1)
  - And *data values*
- The presence of a sentinel value indicates that the location contains no data value at present but can be used to hold a value

Linear probing - idea

- The simplest approach to resolve a collision is linear probing.
- In this technique, if a value is already stored at a location generated by h(k), then the following hash function is used to resolve the collision

$$h(k, i) = [h'(k) + i] \bmod m$$

- where m is the size of hash table, **h'(k) =(k mod m)** and I is the probe number that varies from 0 to m-1.
- Linear probing is known for its simplicity. When we have to store a value, we try the lots: [h'(k)] mod m, [h'(k)+1] mod m and so on until a vacant location is found.

Linear probing - example

- Consider a hash table of size 10. Using linear probing, insert the keys 72, 27, 36, 24, 63, 81, 92, and 101 into the table
- Let h'(k) = k mod m, m = 10
- Initially, the hash table can be given as:

0	1	2	3	4	5	6	7	8	9
–1	–1	–1	–1	–1	–1	–1	–1	–1	–1

- Step 1-6:

0	1	2	3	4	5	6	7	8	9
0	81	72	63	24	–1	36	27	–1	–1

- Step 7: k = 92 h(92,0)= (92 mod 10 + 0) mod 10 = 2
  - But T[2] is occupied.
  - Then try i = 1 => h(92,1) = 3 => occupied
  - Then try i = 2 => h(92,2) = 4 => occupied
  - Then try i = 3 => h(92,3) = 5 => vacant

Searching a value using linear probing

- the array index is re-computed and the key of the element stored at that location is compared with the value that has to be searched
- If a match is found, then the search operation is successful. The search time is O(1)
- If the key does not match, then the search begins a sequential search of the array that continue until:
  - the value is found, or
  - the search function encounters a vacant location in the array, indicating that the value is not present, or
  - the search function terminates because it reaches the end of the table and the value is not present
- In worst case, the search operation may have to make n–1 comparisons => O(n)

Pros and cons

- Linear probing finds an empty location by doing a linear search in the array beginning from position h(k)
- algorithm provides good memory caching through good locality of reference
- But, it results in clustering, and thus there is a higher risk of more collisions where one collision has already taken place
- The performance of linear probing is sensitive to the distribution of input values.
- **Solution:** quadratic probing

Other probing

- **Quadratic probing**

$$h(k, i) = [h'(k) + c_1 i + c_2 i^2] \bmod m$$

- **Double probing**

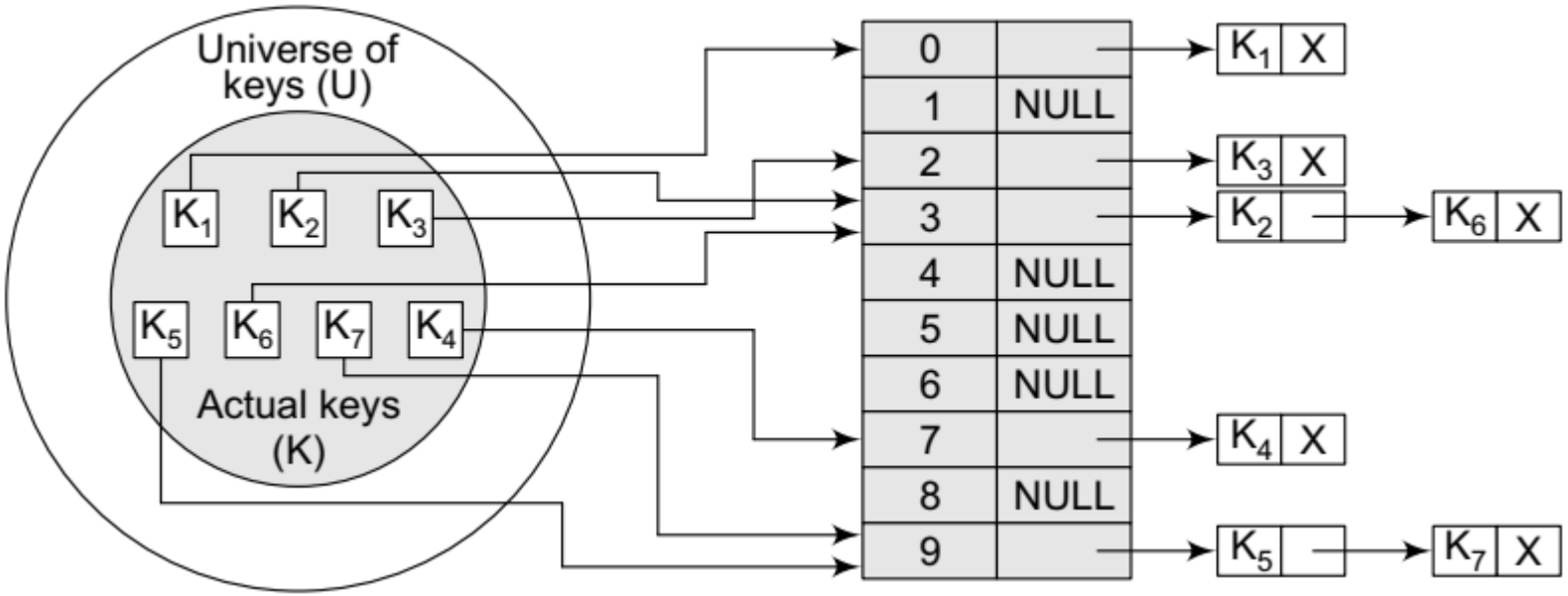
$$h(k, i) = [h_1(k) + i h_2(k)] \bmod m$$

Rehashing

- When the hash table becomes nearly full, the number of collisions increases, thereby degrading the performance of insertion and search operations.
- In such cases, a better option is to create a new hash table with size double of the original hash table
- All the entries in the original hash table will then have to be moved to the new hash table.
- Rehashing seems to be a simple process, it is quite expensive and must therefore not be done frequently

Collision resolution by chaining

- In chaining, each location in a hash table stores a pointer to a linked list that contains all the key values that were hashed to that location

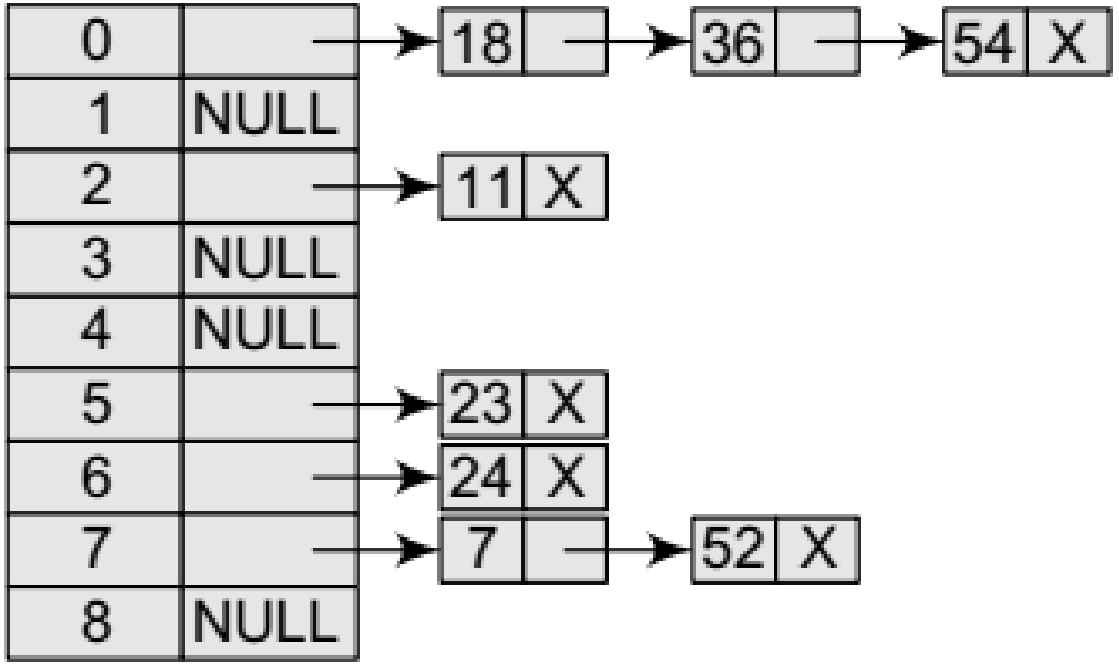


Operations on chained hash table

- Searching for a value in a chained hash table is as simple as scanning a linked list for an entry with the given key.
- Insertion operation appends the key to the end of the linked list pointed by the hashed location.
- Deleting a key requires searching the list and removing the element.
- Costs:
  - Inserting a key in a chained hash table is O(1)
  - Deleting and searching a value is given as O(m) where m is the number of elements in the list of that location
  - Worst case, searching a value may take a running time of O(n), where n is the number of key values stored in the chained hash table

Example

- Insert the keys 7, 24, 18, 52, 36, 54, 11, and 23 in a chained hash table of 9 memory locations. Use h(k) = k mod m



Pros and Cons

- It remains effective even when the number of key values to be stored is much higher than the number of locations in the hash table
- **If** the number of keys to be stored **increases,** the performance of a chained hash table does degrade gradually (linearly)
- Its performance does not degrade when the table is more than half full
- This technique is absolutely free from clustering problems and thus provides an efficient mechanism to handle collisions
- **But:** chained hash tables inherit the disadvantages of linked lists **(pointer storage, traversing)**

Real world applications of Hashing

- CD Databases
- Drivers Licenses/Insurance Cards
- Sparse Matrix
- File signatures
- Game Boards
- Graphics

Outline

- **Introduction**
- **Algorithms for searching a element / key**
  - Comparison based searching (indirect search)
  - Key value based searching (direct search)
- **Algorithms for pattern searching**
  - Brute-force algorithm
  - Knuth-Morris-Pratt (KMP) algorithm
- **References**

Introduction to pattern search

- Problem: Given a document V consists of n characters (v_0, v_1,…, v_{n-1}) and a pattern P of m characters (p_0, p_1,…, p_{m-1}). Search the first occurrence of P in V.
- Algorithms:
  - Brute-force: O(m x n).
  - KMP: O(m + n).

Example

- Ex1:
  - Pattern P has m characters, T has n character
  - T: “the rain in spain stays mainly on the plain”
  - P: “ain ”
- Applications:
  - Information retrieval
  - Document editing
  - Bio signal processing (ADN)

Brute-force algorithm

- The Brute Force algorithm compares the pattern to the text, one character at a time, until unmatching characters are found

```
TWO ROADS DIVERGED IN A YELLOW WOOD
ROADS
```

```
TWO ROADS DIVERGED IN A YELLOW WOOD
 ROADS
```

```
TWO ROADS DIVERGED IN A YELLOW WOOD
  ROADS
```

```
TWO ROADS DIVERGED IN A YELLOW WOOD
   ROADS
```

```
TWO ROADS DIVERGED IN A YELLOW WOOD
    ROADS
```

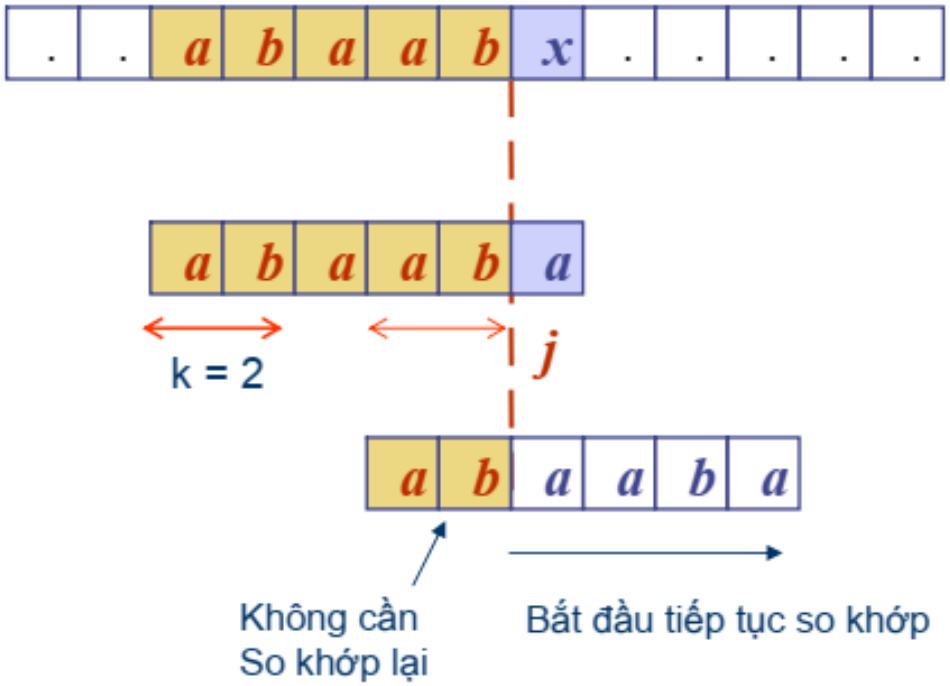
Brute-force algorithm

```
int  BFSearch(char V[N],
char P[M] ) {
/*Ham tra ve vi tri tim thay dau
tien, tra ve -1 neu khong tim
thay*/
    if (N<M) return -1;
    int  i, j;
    i=j=0;
    do {
          while (j<M &&
V[i+j]==P[j]) {
                j++;
        }
        if (i<=N-M && j<M ) {
            i++;
            j=0;
        }
    } while (i<=N-M && j<M) ;
    if (j==M) return i;
    else return -1;
}//end BFSearch
```

Knuth-Morris-Pratt (KMP) algorithm

- The Knuth-Morris-Pratt (KMP) string searching algorithm differs from the brute-force algorithm by keeping track of information gained from previous comparisons
- A failure function (*f*) is computed that indicates how much of the last comparison can be reused if it fails
- Notations:
  - Pattern P has m characters
  - Sub-Pattern P[i..j] of P consists of character indexed from i to j
  - Prefix of P is: P[0..i]
  - Suffix of P is P[i..m-1]
- **Ex: P = abacade**
  - P' = aba is prefix of P
  - P'' = ade is một suffix of P

Knuth-Morris-Pratt (KMP) algorithm



Knuth-Morris-Pratt (KMP) algorithm

Algorithm *prefixFunction*(*P*)

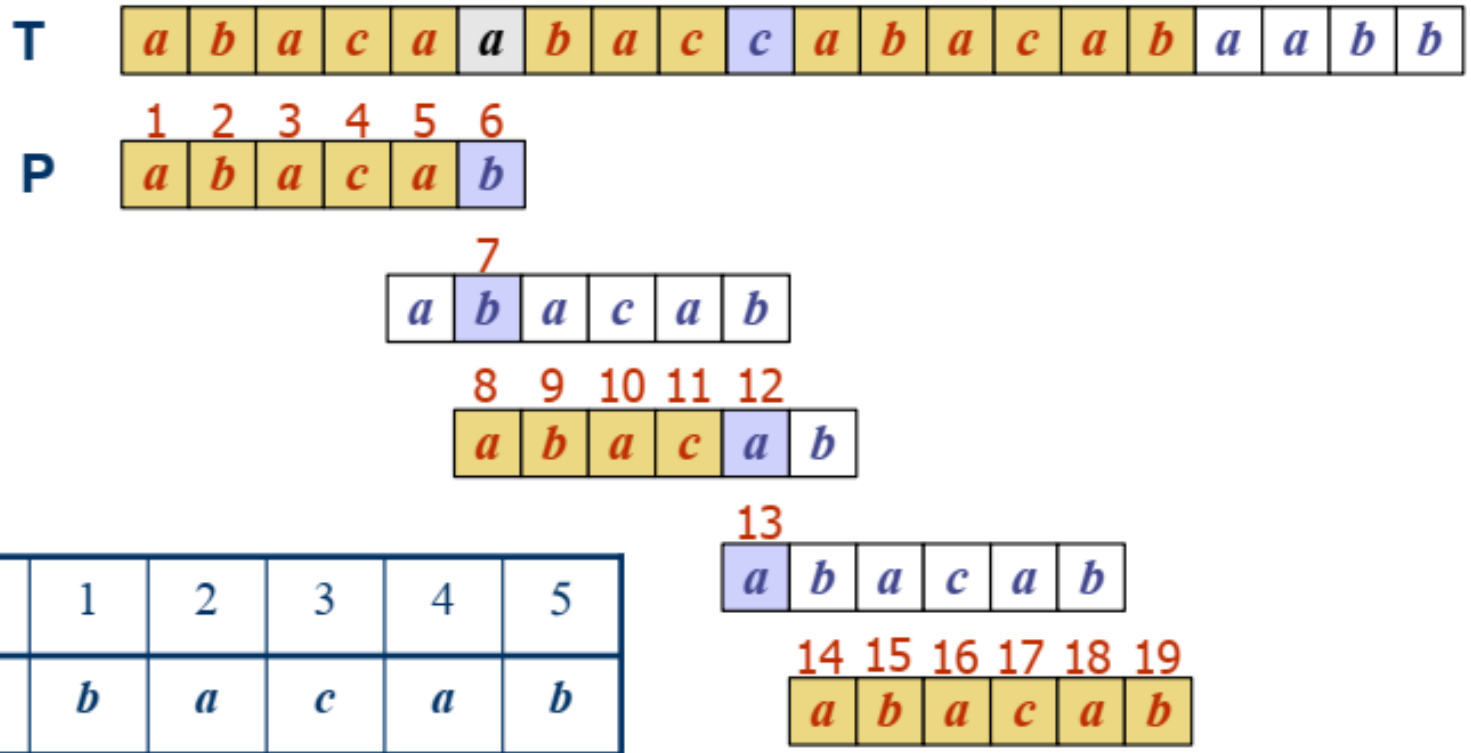
```
{ m là độ dài của P}
    p[0] = 0; i = 1; j = 0;
    while i < m do
    begin
        while ( j > 0 && P[i] != P[j]) do
                j = p[j];
        if (P[i] = P[j]) then  j = j + 1;
        p[i] = j;
        i = i + 1;
    end.
```

i	0	1	2	3	4	5
P[*i*]	*a*	*b*	*a*	*a*	*b*	*a*
p(*i*)	0	0	1	1	2	3

Knuth-Morris-Pratt (KMP) algorithm

```
Algorithm KMP(T, P)
    p ← prefixFunction(P);
    j = 0; i = 0 ;
    while ( i < n)  do
    begin
        while ( j >= 0 && T[i] != P[j]) do
                    j = p[j];
        if (T[i] = P[j]) then  j = j + 1;
        if (j = m ) then output(i-m);
        i = i + 1;
    end.
```

Knuth-Morris-Pratt (KMP) algorithm



j	0	1	2	3	4	5
$P[j]$	a	b	a	c	a	b
$p(j)$	0	0	1	0	1	2

Knuth-Morris-Pratt (KMP) algorithm

```
int  KMPSearch(char V[N], char P[M] ) {
/*Ham tra ve vi tri tim thay dau tien, tra ve -1 neu khong tim thay*/
    if (N<M) return -1;
    int  i, j;
    i=j=0;
    do {
        while (j<M && V[i+j]==P[j]) {
            j++;
        }
        if (i<=N-M && j<M ) {
            j=Overlap(P, j);
            if (j==0) i++;
        }
    } while (i<=N-M && j<M) ;
    if (j==M) return i;
    else return -1;
}//end KMPSearch
```

Exercises

- Calculate hash values of keys: 1892, 1921, 2007, 3456 using different methods of hashing
- What is collision ? Explain various techniques to resolve a collision. Which technique do you think better and why ?
- Consider a hash table with size = 10, using linear probing, insert the key 27, 72, 63, 42, 36, 18, 29, 101 into the table

Exercises

4. Implement a list of students, each student is a record with following information: ID (value of 4 digits), name. We would like to directly search the student according to his ID, if the number of students N = 500, and table size M = 550.

1. Choose a suitable methods (open addressing / chained hashing table)
2. Define list structure, hash table
3. Define function to store a student
4. Define a function to search a student given his ID

References

- Chương 10 – Tìm kiếm - Cấu trúc dữ liệu và Giải thuật – Đỗ Xuân Lôi
- Bài giảng – Nguyễn Thanh Bình - ĐTVT
- Bài giảng – Đỗ Bích Diệp
- Chapter 14 - Searching and Sorting – Data structure in C – Rema Thareja - 2014