Data structures and Algorithms
Searching

Pham Quang Dung

Hanoi, 2012

Outline

1. Sequential searches
2. Binary searches
3. Binary search trees
4. AVL trees
5. String searching
6. Map and Hashing

Sequential searches

- Input: a list $A[1..n]$ and an item x
- Output: position i such that $A[i] = x$

```
int sSearch(int A[], int L, int R, int x){
  for(int i = L; i <= R; i++)
    if(A[i]==x)
      return i;
  return -1;
}
```

Outline

Binary search

- Input: a list $A[1..n]$ in non-decreasing order and an item x
- Output: position i such that $A[i] = x$

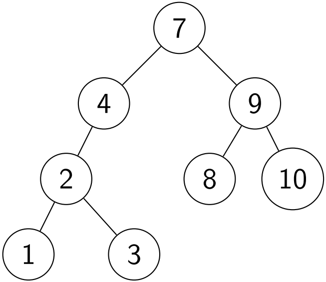
```
int bSearch(int A[], int L, int R, int x){
  int l = L;
  int r = R;
  while(l <= r){
    int mid = (l+r)/2;
    if(A[mid] == x)
      return mid;
    if(A[mid] > x)
      r = mid-1;
    else
      l = mid+1;
  }
  return -1;
}
```

Outline

1. Sequential searches
2. Binary searches
3. **Binary search trees**
4. AVL trees
5. String searching
6. Map and Hashing

Binary search trees (BST)

- Each node has unique key
- Keys of nodes of left subtree of a node v are less than the key of v
- Keys of nodes of right subtree of a node v are greater than or equal to the key of v



Binary search trees

```
struct Node{
  int key;
  Node* leftChild;
  Node* rightChild;
};

Node* root;
```

BST

- makeNode(int v): create a node with key = v, return the new created node
- insert(Node* ptr, int v): insert a new node with key = v to the BST pointed by ptr
- search(Node* ptr, int v): search a node having key = v, return this node if found
- findMin(Node* ptr): return the node having smallest key of the BST pointed by ptr
- del(Node* ptr, int v): remove a node having key = v

BST

```
Node* makeNode(int x){
  Node* p = new Node;
  p->key = x;
  p->leftChild = NULL;
  p->rightChild = NULL;
  return p;
}
```

BST

```
Node* insertNode(Node* r, int x){
  if(r == NULL){
    r = makeNode(x);
  }else if(r->key > x){
    r->leftChild = insertNode(r->leftChild,x);
  }else if(r->key <= x){
    r->rightChild = insertNode(r->rightChild,x);
  }
  return r;
}
```

BST

```
Node* search(Node* r, int x){
  if(r != NULL){
    if(r->key == x)
      return r;
    else if(r->key > x)
      return search(r->leftChild,x);
    else
      return search(r->rightChild,x);
  }
  return NULL;
}
```

BST

```
Node* findMin(Node* r){
  if(r == NULL) return NULL;
  Node* lmin = findMin(r->leftChild);
if(lmin == NULL) return r;
if(lmin->key < r->key)
  return lmin;
else
  return r;
}
```

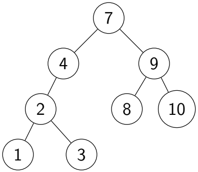
BST

```
Node* del(Node* r, int x){
  Node* tmp;
  if(r == NULL) printf("Not Found\n");
  else if(x < r->key) r->leftChild = del(r->leftChild,x);
  else if(x > r->key) r->rightChild = del(r->rightChild,x);
  else{// x = r->key, remove r means that make the smallest node of the right tree of r the root of the new tree
    if(r->leftChild != NULL && r->rightChild != NULL){
      tmp = findMin(r->rightChild);
      r->key = tmp->key;
      r->rightChild = del(r->rightChild,tmp->key);
    }else{
      tmp = r;// keep this node for freeing memory
      if(r->leftChild == NULL)        r = r->rightChild;
      else if(r->rightChild == NULL)        r = r->leftChild;
      delete tmp;
    }
  }
  return r;
}
```

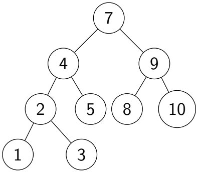
Outline

AVL trees

- AVL is a BST (G. M. Adelson-Velskii and E. M. Landis, 1962)
  - the height of the left child differs from the height of the right child by at most 1 (balance property)
  - left and right subtrees are both AVL
- Modification (insertion, deletion of nodes) on AVL must conserve the balance property
- Assumption: the keys of all nodes are different (e.g., cannot construct an AVL for 8, 8, 9)



a. BST (not AVL)



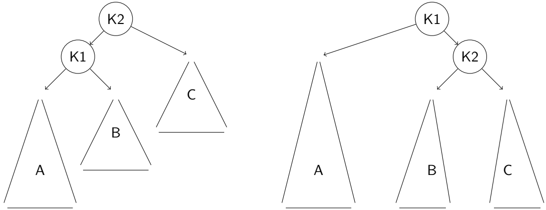
b. AVL

AVL tree - recovery the balance property

- A BST T whose left and right subtrees are AVL
- Perform rotation actions so that the resulting BST is an AVL
- After the insertion or deletion of a node on an AVL
  - balance property of the AVL may loss
  - the height of any subtree change at most 1
  - identify which subtrees loosing balance property

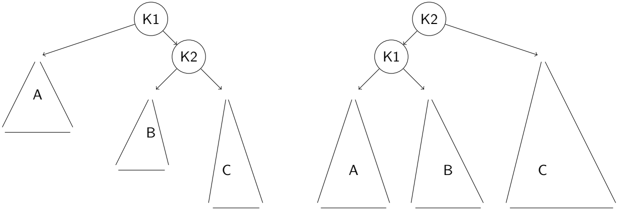
Recovery balance property

Case 1 (perform right rotation at K2)



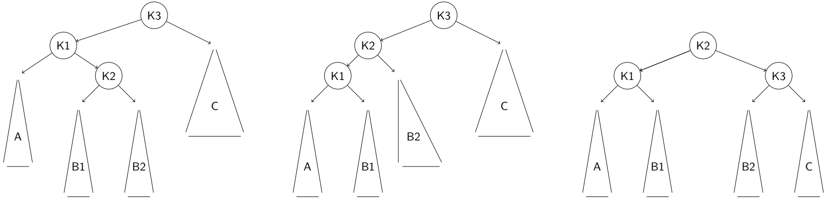
Recovery balance property

Case 2 (perform left rotation at K1)

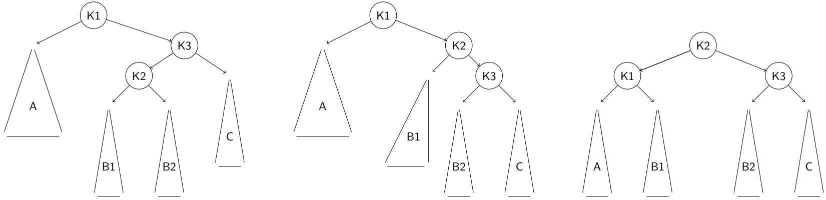


Recovery balance property

Case 3 (perform left rotation at K1 and then right rotation at K3)



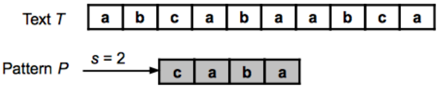
Case 4 (perform right rotation at K3 and then left rotation at K1)



Outline

String searching

- String matching problem: find one or all occurrences of a pattern in a given text
- Applications
  - information retrieval
  - Text editors
  - computational biology (DNA sequences)
- Formal formulation
  - A text is an array $T[1..n]$ and a pattern is an array $P[1..m]$ $(m \neq n)$
  - $T[i], P[j] \in$ a finite alphabet $\sum$ (e.g., $\sum = \{0, 1\}$ or $\sum = \{a, \ldots, z\}$)
  - We say that pattern P **occurs with shift s** in T if $0 \leq s \leq n - m$ and $T[s + 1..s + m]$=$P[1..m]$



String searching algorithms

- Naive
- Boyer-Moore
- Rabin-Karp
- Knuth-Morris-Pratt (KMP)

Naive algorithm

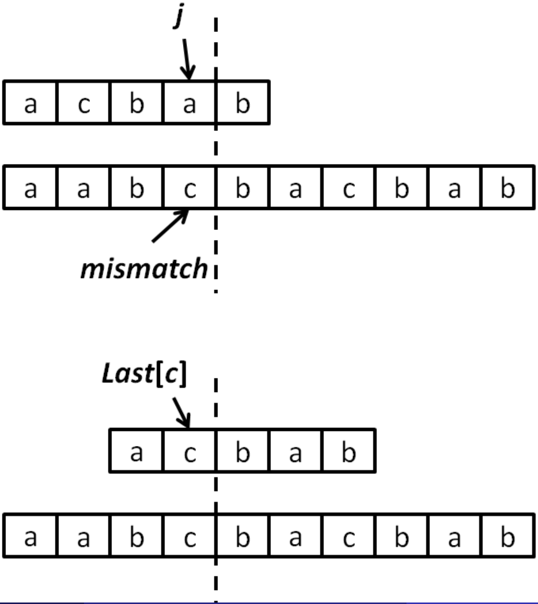
Algorithm 1: NaiveSM(P, T)

```
foreach s = 0..n-m do
    i ← 1;
    while i ≤ m and P[i] = T[i + s] do
        i ← i + 1;
    if i ≥ m then
        Output(s);
```

Boyer-Moore algorithm

- Left to right shift
- Right to left scan
- Use information gained by preprocessing P in order to skip as many alignment as possible
- Bad character shift rule
  - $last[c]$: the right-most occurrence of c in P
  - When mismatch: shift P right by $max\{j - last[c], 1\}$ where j is the position of mismatch character of P

Boyer-Moore algorithm



Boyer-Moore algorithm

```
void computeLast(){
  for(int c = 0; c < 256; c++)
   last[c] = 0;
  for(int i = m; i >= 1; i--){
    if(last[P[i]] == 0)
      last[P[i]] = i;
  }
}
void BoyerMoore(){
  int s = 0;
  while(s <= n-m){
    int j = m;
    while(j > 0 && T[j+s] == P[j])   j--;
    if(j == 0){
      Output(s);
      s = s + 1;
    }else{
      int k = last[T[j+s]];
      s = s + max(j-k,1);
    }
  }
```

Rabin-Karp algorithm

- Convert the pattern $P[1..m]$ to a number:

$$p = P[1] * d^{m-1} + P[2] * d^{m-2} + \cdots + P[m] * d^0$$

  where each character $P[i]$ is viewed as a nonnegative integer $< d$, and d is the size of the alphabet

- Using Horner's rule:

$$p = P[m] + d * (P[m-1] + d * (\cdots + d * P[1]) + \dots)$$

- Convert $T[s+1..s+m]$ to the integer

$$t_s = T[s+1] * d^{m-1} + \cdots + T[s+m]$$

- **Note**: t_{s+1} can easily be computed from t_s as follows: $t_{s+1} = (t_s - T[s+1] * d^{m-1}) * d + T[s+m+1]$

Rabin-Karp algorithm

- Drawback: when m is large, then the computation of p and t_s does not take constant time
- Solution: Compute p and t_s modulo a suitable number q
  - Still problem: $p \equiv t_s (mod\ q)$ does not mean that $p = t_s$, we have to check $P[1..m]$ and $T[s+1..s+m]$ character by character to see if they are really identical
- Worst-case time is $\mathcal{O}(mn)$ where $P = a^m$ and $T = a^n$

Knuth-Morris-Pratt (KMP) algorithm

- Comparison: from left to right
- Shift: more than one position
- Preprocessing the pattern
  - Pattern $P[1..m]$
  - $\pi[q]$ is the length of the longest prefix of $P[1..m]$ which is also the **strictly** suffix of $P[1..q]$

Example

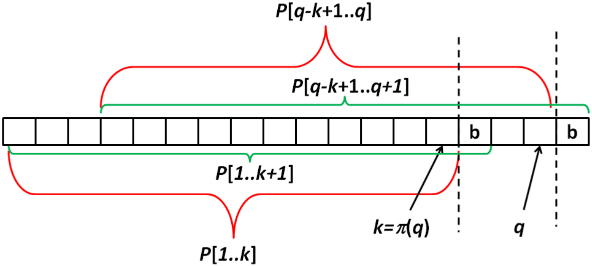
q	1	2	3	4	5	6	7	8	9	10
$P[q]$	a	b	a	b	a	b	a	b	c	a
$\pi[q]$	0	0	1	2	3	4	5	6	0	1

Knuth-Morris-Pratt (KMP) algorithm - preprocessing

```
void computePI(){
  pi[1] = 0;
  int k = 0;
  for(int q = 2; q <= m; q++){
    while(k > 0 && P[k+1] != P[q])
      k = pi[k];
    if(P[k+1] == P[q])
      k = k + 1;
    pi[q] = k;
  }
}
```

Denote $k = \pi[q]$

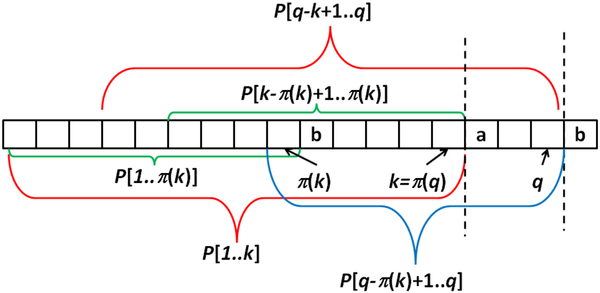
- If $P[q+1] = P[k+1]$, then $\pi[q+1] = \pi[q] + 1$



Knuth-Morris-Pratt (KMP) algorithm - preprocessing

Denote $k = \pi[q]$

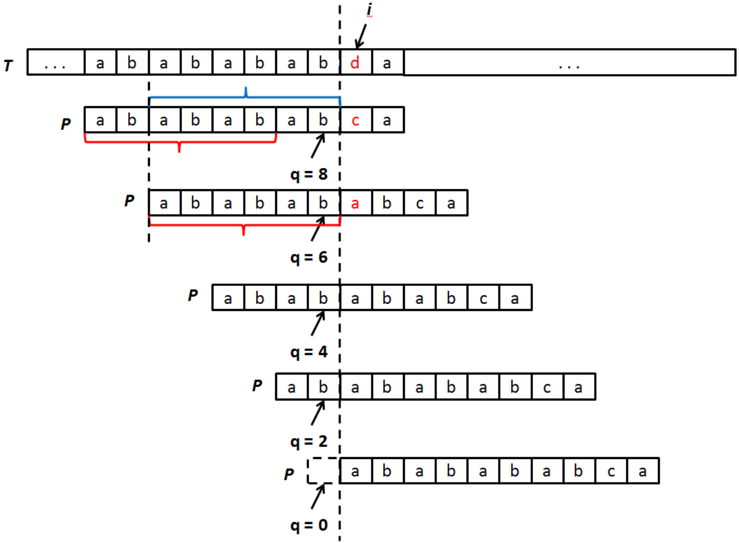
- if $P[q+1] \neq P[k+1]$ and $P[q+1] = P[\pi[k]+1] = b$:
  - $P[1..k] = P[q-k+1..q] \Rightarrow P[k-\pi[k]+1..k] = P[q-\pi[k]+1..q]$
  - Moreover, $P[k-\pi[k]+1] = P[1..\pi[k]$, so $P[1..\pi[k]] = P[q-\pi[k]+1..q]$,
  - Hence $P[1..\pi[k]+1] = P[q-\pi[k]+1..q+1]$, this means $\pi[q+1] = \pi[k]+1$



Knuth-Morris-Pratt (KMP) algorithm

```
void kmp(){
  int q = 0;
  for(int i = 1; i <= n; i++){
    while(q > 0 && P[q+1] != T[i]){
      q = pi[q];
    }
    if(P[q+1] == T[i])
      q++;
    if(q == m){
      cout << "match at position " << i-m+1 << endl;
      q = pi[q];
    }
  }
}
```

Knuth-Morris-Pratt (KMP) algorithm



Outline

1. Sequential searches
2. Binary searches
3. Binary search trees
4. AVL trees
5. String searching
6. Map and Hashing

Map ADT

- Table stores key-value pairs
- Keys cannot be duplicated
- Operations
  - size()
  - empty()
  - get(k): return the item having key k
  - put(k, v): put a key-value pair (k, v) into the table
  - remove(k): remove item having key k from the table
- Operations should be performed efficiently without sorting items of the table

Map implementation

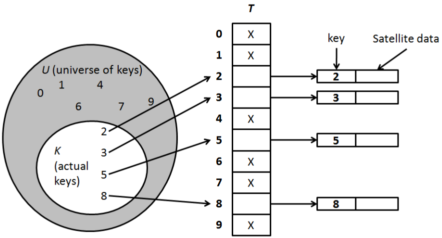
- Arrays
- Linked-lists
- Binary search trees
- **Hash tables** (focus of this topic)

Hashing

- A different approach to searching from the comparison-based methods
- Hashing tries to reference an item in the table directly based on its key without navigating through the table and comparing the search key with the keys of all items
- Hashing transform a key into a table address
- Two approaches
  - Direct-address tables
  - Hash tables

Direct-address tables

- Simple technique
- Work well when the universe U of keys is small
- Suppose each key is taken from $U = \{0, \ldots, m-1\}$ where m is not too large
- Use an array (**direct-address table**) $T[0..m-1]$
  - Each slot k corresponds to a key k

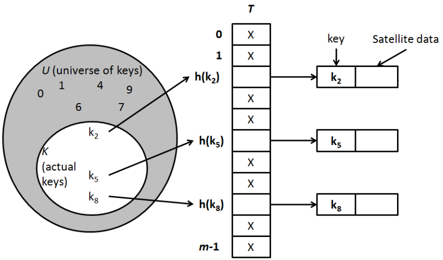


Hash table

- Each element with key k is stored in slot $h(k)$ of the hash table (called **hash function**)

$$h : U \to \{0, 1, \ldots, m-1\}$$

- The size m of hash table is typically much less than the size of U
- **Collision** when two keys k_1, k_2 hash the same slot: $h(k_1) = h(k_2)$
  - Solution by chaining
  - Solution by open addressing



Chaining

- Place all elements that hash to the same slot into the same linked list
- Slot i of the table stores a pointer to the head of the linked list

Algorithm 2: Put(k,v)

$x.key \leftarrow k$;
$x.value \leftarrow v$;
Insert x at the head of list $T[h(k)]$;

Algorithm 3: Get(k)

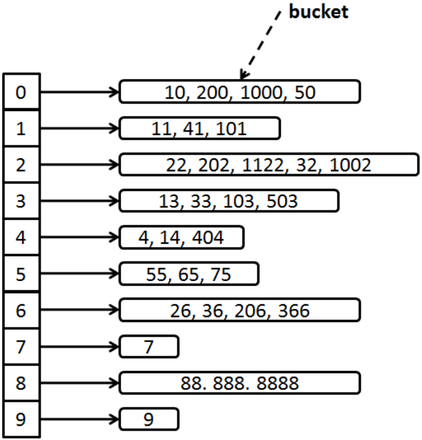
Search an element with key k in list $T[h(k)]$;
return x;

Algorithm 4: Remove(k)

$x \leftarrow \text{Get}(k)$;
Delete x from list $T[h(k)]$;

Hashing: example

- keys: integer
- m is chosen to be 10
- $h(k) = k \bmod 10$
- Collison: two keys that hash to the same value (e.g., 22, 202)



Analysis of hashing with chaining

- We define a **load factor** $\alpha = \frac{n}{m}$
- Assumption of **simple uniform hashing**
- Average running time of unsuccessful search is $\Theta(1 + \alpha)$ (proof is detailed in "Introduction to Algorithms" book)
- Average running time of successful search is $\Theta(1 + \alpha)$ (proof is detailed in "Introduction to Algorithms" book)
- Put and get actions take $\mathcal{O}(1)$ if the lists are implemented as doubly linked lists

Open addressing

- All elements are stored in table itself
- When searching for an element: examine tables slots
- No list and no elements stored outside the table, avoid pointers together
- The hash table can fill up, no further insertion can be made

Open addressing - insertion

- Sucessively examine (**probe**) the table until an empty slot is found to put the key
- Instead of fixing the order $0, 1, m-1$ ($\Theta(n)$ search time), the sequence of slots probed depends the key being inserted
- Extended hash function:
  $h : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$
- The probe sequence for key k is $\langle h(k,0), h(k,1), \ldots, h(k, m-1)\rangle$ which is a permutation of $0, 1, \ldots, m-1$

Open addressing - insertion

Algorithm 5: Put(k, v)

$x.key \leftarrow k$;
$x.value \leftarrow v$;
$i \leftarrow 0$;
while $i < m$ **do**
    $j \leftarrow h(k, i)$;
    if $T[j] = NULL$ **then**
        $T[j] \leftarrow x$;
        return j;
    $i \leftarrow i + 1$;
Error "hash table overflow";

Open addressing - search

Algorithm 6: Get(k)

```
i ← 0;
while i < m do
    j ← h(k, i);
    if T[j].key = k then
        return T[j];
    i ← i + 1;
return NULL;
```

Open addressing

- **Uniform hashing** assumption: Each key is equally likely to have any of $m!$ permutation of $\{0, 1, \ldots, m-1\}$ as its probe sequence
- Three common techniques for probe sequence computation
  - Linear probing
  - Quadratic probing
  - Double hashing
- All of three techniques guarantee that $h(k, 0), h(k, 1), \ldots, h(k, m-1)$ is a permutation of $0, 1, \ldots, m-1$ for each key k
- None of three techniques fulfills the assumption of uniform hashing

Open addressing

- Linear probing: $h(k, i) = (h'(k) + i)$ **mod** m where h' is an ordinary hash function
- Quadratic probing: $h(k, i) = (h'(k) + c_1 i + c_2 i^2)$ **mod** m where h' is ordinary hash function and $c_1, c_2 \neq 0$ are constants
- Double hashing (one of the best method for open addressing): $(h(k, i) = (h_1(k) + ih_2(k))$ **mod** m where h_1, h_2 are auxiliary hash functions
  - The value of $h_2(k)$ must be relatively prime to the hash table size m for the entire hash table to be searched (see "Introduction to Algorithms" for more detail)
  - Some approaches:
    - Let m be a power of 2 and design $h_2(k)$ such that it always returns an odd value
    - Let m be a prime and design $h_2(k)$ such that it always returns a positive integer less than m (e.g., $h_2(k) = 1 + k$ **mod** $(m - 1)$)

Open addressing - analysis

- Inserting an element into an open-address hash table with load factor $\alpha = \frac{n}{m}$ requires at most $\frac{1}{1-\alpha}$ probes on average, assuming uniform hashing

Universal hashing

- If the chosen hash function is fixed, then there might be n keys that hash to the same slot, yielding an average retrieval time of $\Theta(n)$
- Solution: universal hashing
  - Select the hash function randomly from a carefully designed class of functions at the beginning of the execution
  - Randomization guarantees that no single input will always evoke worst-case behavior
  - Good average running time

Definition

Let $\mathcal{H}$ be a finite collection of hash functions that map a given universe U of keys into the range $\{0, 1, \ldots, m-1\}$. Such collection is said to be **universal** if for each pair of distinct keys k, $l \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(k) = h(l)$ is at most $\frac{\mathcal{H}}{m}$. In other words, for a given pair of distinct keys k, l, we pick a hash function from $\mathcal{H}$, the probability that $h(k) = h(l)$ is at most $\frac{1}{m}$.

Universal hashing

Theorem

Suppose that a hash function h is chosen randomly from a universal collection of hash functions and has been used to hash n keys into a table T of size m using chaining to resolve collisions. If key k is not in the table, then the expected length $E[n_{h(k)}]$ of the list that k hashes to is at most the load factor $\alpha = \frac{n}{m}$. If the key k is in the table, then the expected length $E[n_{h(k)}]$ of the list that k hashes to is at most $1 + \alpha$

Proof.

See "Introduction to Algorithms" book □

Universal hashing - example

- Choose a prime number p large enough so that every possible key k is in the range $0, \ldots, p-1$, inclusive. Let $\mathbb{Z}_p = \{0, \ldots, p-1\}$, and $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$
- $p > m$ by assumption that the size of universe U is greater than the number of slots of the hash table
- Define hash function $h_{a,b}(k) = ((ak + b) \textbf{ mod } p) \textbf{ mod } m, \forall a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p$
- $\mathcal{H}_{p,m} = \{h_{a,b} : a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$

Theorem

$\mathcal{H}_{p,m}$ *defined above is* **universal**

Proof.

See "Introduction to Algorithms" book $\square$

Perfect hashing

- Hashing can provide excellent **worst-case** performance when the set of keys is **static**: once the keys are stored in the table, they never change
- **Perfect hashing**: $\mathcal{O}(1)$ memory access are required to perform a search in the worst-case
- Idea: Two-levels hashing with universal hashing at each level
  - First level: selected carefully from a family of universal hash functions
  - Second level uses hash tables instead of linked lists: Choose carefully hash function h_j for hash table S_j of slot j in order to guarantee that there are no collisions at the second level
    - Set the size m_j of hash table S_j to n_j^2 where n_j is the number of keys hashing to slot j

Perfect hashing