搜索 [二分搜索]

```
left ← 1 , right ← n
repeat
    mid ← (left + right)/2
    if (t = A[mid]) then
        return "True"
    else if (t < A[mid]) then
        right ← mid-1
    else
        left ← mid+1
until left > right
return "False"
```

O(log n)

随机数搜索

```
do
    r ← random(1,n)
until A[r] = 0
return r
```

O(n)

排序

[n²] Selection-Sort

```
for integer i ← 1 to n-1
    k ← i
    for int j ← i+1 to n
        if A[k] > A[j]
            k ← j
    swap A[i], A[k]
```

[n²] Insertion Sort (A[1...n])

```
for int i ← 1 to n
    for int j ← i to 1 (j-1>0)
        if A[j-1] > A[j] then
            Swap A[j-1], A[j]
```

[n²] Bubble-Sort (A[1...n])

```
for int i ← 1 to n-1
    for int j ← 2 to n
        if A[j-1] > A[j] then
            swap A[j-1], A[j]
```

Merge_Sort (A, n) [nlogn]

```
if n > 1
    p ← ⌊n/2⌋
    B[1...p] ← A[1...p]
    C[1...n-p] ← A[p+1...n]
    Merge-Sort(B, P)
    Merge-Sort(C, n-p)
    A[1...n] ← Merge(B, p, C, n-p)
```

Merge (L, n_L, R, n_R)

```
n ← n_L + n_R
Let A[1...n] be a new Array
i ← 1   j ← 1
for k ← 1 to n
    if i ≤ n_L and (j > n_R or L(i) ≤ R(j))
        A[k] ← L[i] ; i ← i+1
    else
        A[k] ← R[j] ; j ← j+1
return A
```

Quick_Sort (A[1...n], lo=1, hi=m)

```
p ← partition (A, lo, p-1)
Quick-Sort (A, lo, p-1)
Quick-Sort (A, p+1, hi)
```

O(nlogn) 最坏 O(n²)

partition (A, lo, hi)

```
p ← Random (lo, hi)
pivot ← A[p]
L ← lo , R ← hi
for int i ← lo to hi
    if (A[i] < pivot)  A'[L++] ← A[i]
    else  A'[R--] ← A[i]
A[lo..hi] ← A'
return L ;
```

linked list

traverse (A):

```
if (A = Null)
    return
else
    print A.value
    traverse (A.next)
```

recursion 递归

Interation 迭代

traverseIteration(A)

```
node trav ← A
while (trav != null)
    print trav.value
    trav = trav.next
```

InsertNode (A, node q, i);

```
a ← 0   node p ← A,
while (i-1 > a)        O(n)
    p ← p.next
    a ← a+1
temp ← p.next
p.next ← q
q.next ← temp
return A.
```

deleteNode (A, i)

```
a ← 0    node p ← A
while (i-1 > a)
    p ← p.next
    a ← a+1
p.next ← p.nextnext
return A
```

O(n)

Find (A, x)

```
a ← 0    node p ← A
while (p != null)
    if (x = p.value)
        return p
    p ← p.next
return -1
```

O(n)

UpdateItem (A, x)

```
a ← 0 , node p ← A
while (p != null)
    if (x = p.value)
        p.value ← y
    p ← p.next
return A
```

Stack & Queue

Stack: FILO

push, pop, peak

pop( )

```
if (top == -1)
    stack is EMpty
else
    top --
```

push (item):

```
if (top == MAXsize -1)
    Stack is Full
else
    top++
    S[top] = item
```

★ 中缀 转 后缀 表达式

```
InfixToPostfix (String Infix)
n ← len[s]
String Postfix ← empty String
operator stack ops ← empty stack
for i ← 0 to n-1
    token ← Infix[i]
    if (token is operand) then
        Postfix += token
    else
        if (ops is empty) then
            ops.push(token)
        else if (token is '(')
            ops.push(token)
        else if (token is ')')
            tops ← tops.top()
            repeat
                add to tops to Postfix String
                ops.pop()
                tops ← ops.top()
            until (tops is '(')
        else
            tops ← ops.top()
            while (tops precedence over token)
                add to tops to Postfix String
                ops.pop()
                tops ← ops.top()
            ops.push(token)
while (ops is not empty)
    tops ← ops.top()
    add to tops to postfix String
    ops.pop()
return Postfix
```

Queue FIFO   deQueue, enQueue

enQueue (item):

```
if (rear < MAXSize)
    S[rear] = item
    rear ++
else
    Queue is Full
```

de Queue ( )

```
if (front < rear)
    front ++
else
    Queue is empty
```

Ring Queue

enQueue (item)

```
if (!isFull())
    array[rear] = item
    rear = (rear + 1)
        % (MaxSize + 1)
```

deQueue ( )

```
if (!isEmpty())
    front = (front + 1)
        % (MaxSize + 1)
```

isFull( )

```
return (rear+1) % (Max+1) == front
```

isEmpty( )

```
return rear == front
```

String Matching

Rabin-Karp (T, P, d, q)

```
n ← len(T), m ← len(P)
h ← d^(m-1) (mod q), p ← 0   t0 ← 0
for j ← 0 to m-1
    p ← (dp + P[j]) mod q
    t0 ← (dt0 + T[j]) mod q
for i ← 0 to n-m
    if p != t_i then
        t_{i+1} ← (d(t_i - T[i]h) + T[m+i]) mod q
    else
        if P[0...m-1] = T[i, i+m-1]
            pattern occurs with shift i
```

d 进制   q 质数

· 构建第一个

```
for j ← 0 to m-1
    hash ← (d * hash + t[j]) % q
    target ← (... target + P[j]) ...
if i != m-1, then h ← (h * d) % q
```

· 记得匹配时 (★最后)

```
hash ← (d * (hash - h * t[i]) + t[i+m]) % q
```

FSA:

Σ : 字集   X : P[1...] 的状态.

```
m ← len(p) , x ← 0
for i ← 0 to m-2
    for each c in Σ
        if p[i+1] = c
            δ[i, c] = i+1
        else δ[i, c] ← δ[x, c]
    x ← δ[x, P[i+1]]
```

subString : P[a, b]

prefix : P[0, ...]

suffix : P[..., m]

KMP (String txt, String pattern)

```
int m = txt.length(), n = pattern.length();
int[] next = nextArray(pattern);
int p=0, i=0;
while (i < m) {
    if (txt.charAt(i) == pattern.charAt(p)) {
        i++; p++;
    else if (p!=0) {
        p = next.[p-1]
    else i++;
    if (p==n) return true;
return false
```

nextArray (String pattern)

```
int next[] = next int [pattern.length()];
next[0] = 0
int j = 0
for int i=1 to pattern.length
    while (j>0 && pattern.charAt(j) != pattern.charAt(j)) {
        j = next[j-1];
    }
    if (pattern.charAt(i) == pattern.charAt(j))
        j++
    next[i] = j;
return next[];
```

Tree

1. 性质: A tree with n nodes with n-1 edges | 含n个内部节点至少有2个子节点, 若m为叶节点数, 则内节点 (internal nodes) 最多 m-1个

2. 满足 { u=v / u是v的父/u是v的父的祖 } 中任意条则有 u是v的祖. ★ u!=v 时, u为 proper ancestor of v.

3. 叶节点 leaf nodes 无子节点

4. 每个 node 与 root 之间仅有一条确定的线路

5. 节点深度为 path 上的 edges 数且, 相同深度的节点称成一个级.

6. Binary tree

full level: 第l级有 2^l 个 nodes 叫为full级

完全二叉树: 深h的树, 0 ~ h-1 均为 Full level 第 h 级尽量左靠

性质: 一个有n个节点的完全二叉树高度为 O(logn)

traversing Binary Tree

```
preorderPrint(treeNode tree)
  print(root)
  if (root.left !=null)
      preorderprint(root.left)
  if (root.right !=null)
      preorderprint(root.right)
```

```
PostorderPrint(treeNode tree)
if (root.left !=null)
    PostorderPrint(root.left)
if (root.right !=null)
    PostorderPrint(root.right)
print(root)
```

```
InorderPrint(treeNode tree)
if (root.left !=null)
    InorderPrint(root.left)
print(root)
if (root.right !=null)
    InorderPrint(root.right)
```

层状 (Level order)

```
LevelOrderPrint(treeNode tree)
queue is a new Queue
queue.enqueue(tree)
while (!queue.isEmpty())
    treeNode c = queue.top();
    print(c)
    queue.dequeue;
    if (c != null && c.left !=null)
        queue.enqueue(c.left)
    if (c !=null && c.right !=null)
        queue.enqueue(c.right)
```

Huffman tree

1. 一个哈夫曼树至少有2个节点, 设节点 u,v have the top-2 lowest frequencies, then
   (1) u, v have same parents
   (2) depth(u), (v) ≥ depth(x) [x为任意叶节点]
2. 哈夫曼树编码是最佳前缀编码, 空间成本最低)

n nodes => n-1 edges

Proof: For each non-root node v, it has one and only edge point to it self. A tree with n nodes, thus the numbe of non-root nodes is n-1. Thus, this tree has n-1 edges

内部最多 m-1 节点.

Proof:

节点数 n≥2, 高为 O(logn)

proof:

h为高. Level 0 : $2^0=1$

Level 1 : $2^1=2$ Level 2 : $2^2=4$

⋮

Level h+1 = 2^{h-1} Level h : X (h≥1)

$\Rightarrow 2^0+2^1+\cdots+2^{h-1}+X=n$

$\Rightarrow (1-2^{h-1})/(1-2) = n-X$

$\Rightarrow 2^{h-1} < n$

$\Rightarrow h = O(\log n)$