数据结构

链表

单链表

```
// head存储链表头， e[]存储节点的值，ne[]存储节点的 next指针，idx表示当前用到了哪个节点
int head, e[N], ne[N], idx;

// 初始化
void init() {
    head = -1;
    idx = 0;
}

// 在链表头插入一个数a
void insert(int a) {
    e[idx] = a;
    ne[idx] = head;
    head = idx;
    idx++;
}

// 将x插到下标是k的点后面
void insert(int k, int x) {
    e[idx] = x;
    ne[idx] = ne[k];
    ne[k] = idx;
    idx++;
}

// 将头结点删除。需要保证头结点存在
void remove() {
    head = ne[head];
}

// 将下标是k的点后面的点删掉
void remove(int k) {
    ne[k] = ne[ne[k]];
}
```

双向链表

```
// e[]表示节点的值，l[]表示节点的左指针，r[]表示节点的右指针，idx表示当前用到了哪个节点
int e[N], l[N], r[N], idx;

// 初始化
void init() {
    // 0是左端点，1是右端点
    r[0] = 1, l[1] = 0;
```

```
    idx = 2;
}

// 在节点a的右边插入一个数x
void insert(int a, int x) {
    e[idx] = x;
    l[idx] = a, r[idx] = r[a];
    l[r[a]] = idx, r[a] = idx;
    idx++;
}

// 删除节点a
void remove(int a) {
    l[r[a]] = l[a];
    r[l[a]] = r[a];
}
```

栈

```
// top 表示栈顶
int stk[N], top = -1;

// 向栈顶插入一个数
stk[++top] = x;

// 从栈顶弹出一个数
top--;

// 栈顶的值
stk[top];

// 判断栈是否为空
top >= 0;
```

队列

普通队列

```
// head表示队头，tail表示队尾
int q[N], head = 0, tail = -1;

// 向队尾插入一个数
q[++tail] = x;

// 从队头弹出一个数
head++;

// 队头的值
q[head];

// 判断队列是否为空
head <= tail
```

循环队列

```
// head表示队头，tail表示队尾的后一个位置
int q[N], head = 0, tail = 0;

// 向队尾插入一个数
q[tail++] = x;
if (tail == N) tail = 0;

// 从队头弹出一个数
head++;
if (head == N) head = 0;

// 队头的值
q[head];

// 判断队列是否为空
head != tail;
```

单调栈

思想：动态规划

常见题目：找出每个数左边离它最近的比它小的数

```
int tt = 0;
for (int i = 1; i <= n; i++) {
    while (tt && check(stk[tt], i)) tt--;
    stk[++tt] = i;
}
```

单调队列

常见题目： 找出滑动窗口中的最大值

```
int hh = 0, tt = -1;
for (int i = 0; i < n; i++) {
    while (hh <= tt && check_out(q[hh])) hh++;
    while (hh <= tt && check(q[tt], i)) tt--;
    q[++tt] = i;
}
```

KMP

```
// s[]是长文本，p[]是模式串，n是s的长度，m是p的长度

// 求模式串的next数组
for (int i = 2, j = 0; i <= m; i++) {
    while (j && p[i] != p[j + 1]) j = ne[j];
    if (p[i] == p[j + 1]) j++;
    ne[i] = j;
}
```

```
// 匹配
for (int i = 1, j = 0; i <= n; i++) {
    while (j && s[i] != p[j + 1]) j = ne[j];
    if (s[i] == p[j + 1]) j++;
    if (j == m) {
        j = ne[j];
        // 匹配成功后的逻辑
    }
}
```

Trie 树

- 高效地存储和查找字符串集合的数据结构

```
int son[N][26], cnt[N], idx;
// 0号点既是根节点，又是空节点
// son[][]存储树种每个节点的子节点
// cnt[]存储以每个节点结尾的单词数量

// 插入一个字符串
void insert(char *str) {
    int p = 0;
    for (int i = 0; str[i]; i++) {
        int u = str[i] - 'a';
        if (!son[p][u]) son[p][u] = ++idx;
        p = son[p][u];
    }
    cnt[p]++;
}

// 查询字符串出现的次数
int query(char *str) {
    int p = 0;
    for (int i = 0; str[i]; i++) {
        int u = str[i] - 'a';
        if (!son[p][u]) return 0;
        p = son[p][u];
    }
    return cnt[p];
}
```

并查集

作用：

- 将两个集合合并
- 询问两个元素是否在一个集合之中

基本原理： 每个集合用一棵树来表示，树根的编号就是整个集合的编号，每个节点存储它的父节点。

优化方法：

- 路径压缩（常用）
- 按秩合并

朴素并查集

```
int p[N]; // 存储每个点的祖宗节点

// 返回 x 的祖宗节点
int find(int x) {
    if (p[x] != x) p[x] = find(p[x]);
    return p[x];
}

// 初始化，假定节点编号是 1~n
for (int i = 1; i <= n; i++) {
    p[i] = [i];
}

// 合并 a 和 b 所在的两个集合
p[find(a)] = find(b);
```

维护size的并查集

```
int p[N], size[N];
//p[]存储每个点的祖宗节点, size[]只有祖宗节点的有意义，表示祖宗节点所在集合中的点的数量

// 返回x的祖宗节点
int find(int x) {
    if (p[x] != x) p[x] = find(p[x]);
    return p[x];
}

// 初始化，假定节点编号是1~n
for (int i = 1; i <= n; i ++ ) {
    p[i] = i;
    size[i] = 1;
}

// 合并a和b所在的两个集合：
if (find(b) != find(a)) {
    size[find(b)] += size[find(a)];
}
p[find(a)] = find(b);
```

维护到祖宗节点距离的并查集

```
int p[N], d[N];
//p[]存储每个点的祖宗节点, d[x]存储x到p[x]的距离

// 返回x的祖宗节点
int find(int x) {
    if (p[x] != x) {
        // 注意下面三个语句的顺序
        int u = find(p[x]);
        d[x] += d[p[x]]; // x到祖宗节点的距离是x到父节点的距离+父节点到祖宗节点的距离
        p[x] = u;
```

```
    }
    return p[x];
}

// 初始化，假定节点编号是1~n
for (int i = 1; i <= n; i ++ ) {
    p[i] = i;
    d[i] = 0;
}

// 合并a和b所在的两个集合：
p[find(a)] = find(b);
d[find(a)] = distance; // 根据具体问题，初始化find(a)的偏移量
```

堆

功能

- 插入一个数 $O(logn)$

  ```
  heap[++size] = x; up(size);
  ```

- 求集合当中的最小值 $O(1)$

  ```
  heap[1];
  ```

- 删除最小值 $O(logn)$

  ```
  heap[1] = heap[size]; size--; down(1);
  ```

- **删除任意一个元素** $O(logn)$

  ```
  heap[k] = heap[size]; size--; down(k); up(k);
  ```

- **修改任意一个元素** $O(logn)$

  ```
  heap[k] = x; down(k); up(k);
  ```

- **建堆** $O(n)$

  ```
  for (int i = n / 2; i >= 1; i--) down(i);
  ```

代码

```
 // h[N]存储堆中的值, h[1]是堆顶，x的左儿子是2x, 右儿子是2x + 1
// ph[k]存储第k个插入的点在堆中的位置
// hp[k]存储堆中下标是k的点是第几个插入的
int h[N], ph[N], hp[N], size;

// 交换两个点，及其映射关系
void heap_swap(int a, int b) {
    swap(ph[hp[a]], ph[hp[b]]);
```

```
    swap(hp[a], hp[b]);
    swap(h[a], h[b]);
}

void down(int u) {
    int t = u;
    if (u * 2 <= size && h[u * 2] < h[t]) t = u * 2;
    if (u * 2 + 1 <= size && h[u * 2 + 1] < h[t]) t = u * 2 + 1;
    if (u != t) {
        heap_swap(u, t);
        down(t);
    }
}

void up(int u) {
    while (u / 2 && h[u] < h[u / 2]) {
        heap_swap(u, u / 2);
        u >>= 1;
    }
}
```

Hash 表

存储结构

拉链法

```
int h[N], e[N], ne[N], idx;

// 向哈希表中插入一个数
void insert(int x) {
    int k = (x % N + N) % N;
    e[idx] = x;
    ne[idx] = h[k];
    h[k] = idx;
    idx++;
}

// 在哈希表中查询某个数是否存在
bool find(int x) {
    int k = (x % N + N) % N;
    for (int i = h[k]; i != -1; i = ne[i]) {
        if (e[i] == x) return true;
    }
    return false;
}
```

开放寻址法

- 数组大小一般开 2~3 倍。

```
#define null 0x3f3f3f3f

int h[N];
```

```
memset(h, 0x3f, sizeof(h));

// 如果x在哈希表中，返回x的下标；
// 如果x不在哈希表中，返回x应该插入的位置。
int find(int x) {
    int t = (x % N + N) % N;
    while(h[t] != null && h[t] != x) {
        t++;
        if (t == N) t = 0;
    }
    return t;
}
```

字符串哈希

- 将字符串看成P进制数，P的经验值是131或13331，取这两个值的冲突概率低。
- 取模的数用2^{64}，这样直接用$unsigned\ long\ long$存储，溢出的结果就是取模的结果。
- 假设不会发生冲突。

```
typedef unsigned long long ULL;

const int P = 131;
ULL h[N]; // h[k]存储字符串前k个字母的哈希值
ULL p[N]; // p[k]存储P^k mod 2^64

// 初始化
p[0] = 1;
for (int i = 1; i <= n; i++) {
    h[i] = h[i - 1] * P + str[i];
    p[i] = p[i - 1] * P;
}

// 计算子串 str[l ~ r] 的哈希值
ULL get(int l, int r) {
    return h[r] - h[l - 1] * p[r - l + 1];
}
```