Lesson1（数组模拟数据结构）

链表与邻接表（数组模拟）

单链表&邻接表（n个链表）：邻接表主要用于存储图与树

- e[N]:值　ne[N]:next
- e与ne数组通过下标关联，空结点下标用-1表示，链表最后一个有效结点ne为-1

```java
static int N = 100010;

static int n;
static int head, idx;   //head表示头节点的下标，idx表示当前已经使用到了哪个点
static int[] e = new int[N], ne = new int[N];   // e[i]表示结点i的值，ne[i]表示节点i的next指针

static void init() {
    head = -1; idx = 0;
}

static void addToHead(int x) {
    e[idx] = x; ne[idx] = head; head = idx++;
}

static void add(int k, int x) {
    e[idx] = x; ne[idx] = ne[k]; ne[k] = idx++;
}

static void remove(int k) {
    ne[k] = ne[ne[k]];
}

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;

    init(); //记得初始化

    while (n-- > 0) {
        ins.nextToken(); String op = (String)ins.sval;

        if (op.equals("H")) {
            ins.nextToken(); int x = (int)ins.nval;
            addToHead(x);
        }
        else if (op.equals("D")) {
            ins.nextToken(); int k = (int)ins.nval;
            if (k == 0) head = ne[head];
            else remove(k-1);
        }
        else if (op.equals("I")) {
            ins.nextToken(); int k = (int)ins.nval;
            ins.nextToken(); int x = (int)ins.nval;
            add(k-1, x);
        }
    }

    for (int i=head; i!=-1; i=ne[i]) out.print(e[i]+" ");

    out.flush();
}
}
```

双链表：优化某些特定问题

- 在单链表基础上增加指向前一个结点的指针
  - l[N], r[N]
- 不定义头结点与尾结点，0：head　1：tail

```java
static int N = 100010;

static int n;
static int idx;
static int[] l = new int[N], r = new int[N], e = new int[N];

static void init() {    //0表示头节点，1表示尾结点，头尾结点只有指示作用实，idx从2开始
    r[0] = 1; l[1] = 0;
```

```
    idx = 2;
}

static void add(int k, int x) {      //在第k个插入的数右边插入一个结点
    e[idx] = x;
    l[idx] = k; r[idx] = r[k];
    l[r[k]] = idx; r[k] = idx++;
}

static void remove(int k) {
    r[l[k]] = r[k];
    l[r[k]] = l[k];
}

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;

    init();

    while (n-- > 0) {
        ins.nextToken(); String op = (String)ins.sval;

        if (op.equals("L")) {
            ins.nextToken(); int x = (int)ins.nval;
            add(0, x);
        }
        else if (op.equals("R")) {
            ins.nextToken(); int x = (int)ins.nval;
            add(l[1], x);
        }
        else if (op.equals("D")) {
            ins.nextToken(); int k = (int)ins.nval;
            remove(k+1);
        }
        else if (op.equals("IL")) {
            ins.nextToken(); int k = (int)ins.nval;
            ins.nextToken(); int x = (int)ins.nval;
            add(l[k+1], x);
        }
        else if (op.equals("IR")) {
            ins.nextToken(); int k = (int)ins.nval;
            ins.nextToken(); int x = (int)ins.nval;
            add(k+1, x);
        }
    }

    for (int i=r[0]; i!=1; i=r[i]) out.print(e[i]+" ");

    out.flush();
}
```

邻接表（多个单链表）

- head[1]->...
- head[2]->...
- ...
- head[3]->...

栈（先进后出）

```
//stk表示栈，tt表示栈顶
int stk[N], tt;

//插入x
stk[++tt] = x;

//弹出栈顶
tt--;

//判断是否为空
if (tt > 0) not empty
else empty

//栈顶
stk[tt];
```

- e1:

```
static int N = 100010;

static int n;
static int tt;  //栈顶指针
static int[] stk = new int[N];

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;

    while (n-- > 0) {
        ins.nextToken(); String op = (String)ins.sval;

        if (op.equals("push")) {
            ins.nextToken(); int x = (int)ins.nval;
            stk[++tt] = x;
        }
        else if (op.equals("pop")) tt--;
        else if (op.equals("empty")) {
            if (tt > 0) out.println("NO");
            else out.println("YES");
        }
        else out.println(stk[tt]);
    }

    out.flush();
}
```

- e2: 表达式求值

```
static int N = 100010;

//使用两个栈模拟表达式树的计算方式
//表达式树：叶结点为数，其余结点为计算符，且符号优先级随深度增加保持严格单调递增
static int tt1, tt2;
static int[] num = new int[N];  //数字栈
static char[] op = new char[N]; //表达式栈

static void eval() {
    int b = num[tt1--];
    int a = num[tt1--];
    char c = op[tt2--];

    int x = 0;
    if (c == '+') x = a+b;
    else if (c == '-') x = a-b;
    else if (c == '*') x = a*b;
    else if (c == '/') x = a/b;

    num[++tt1] = x;
}

public static void main(String[] args) throws Exception {
    String str = inb.readLine();

    Map<Character, Integer> pr = new HashMap<Character, Integer>();
    pr.put('+', 1); pr.put('-', 1); pr.put('*', 2); pr.put('/', 2);

    for (int i=0; i<str.length(); i++) {
        char c = str.charAt(i);

        if (Character.isDigit(c)) {
            int x = 0, j = i;
            while (j < str.length() && Character.isDigit(str.charAt(j)))
                x = x*10 + str.charAt(j++)-'0';
            i = j-1;
            num[++tt1] = x;
        }
        else if (c == '(') op[++tt2] = c;
        else if (c == ')') {
            while (op[tt2] != '(') eval();
            tt2--;
        }
        else {
            //注意是>=，而非>，从而保持符号优先级严格递增
            while (tt2 > 0 && op[tt2] != '(' && pr.get(op[tt2]) >= pr.get(c)) eval();
            op[++tt2] = c;
```

```
        }
    }

    while (tt2 > 0) eval();

    out.print(num[tt1]);

    out.flush();
}
```

队列（先进先出）

```
//在队尾插入元素，在队头弹出元素
int q[N], hh, tt = -1;

//插入
q[++tt] = x;

//弹出
hh++;

//判断是否为空
if (hh <= tt) not empty
else emtpy

//取队头
q[hh]
```

- e1

```
static int N = 100010;

static int n;
static int hh, tt = -1;
static int[] q = new int[N];

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;

    while (n-- > 0) {
        ins.nextToken(); String op = (String)ins.sval;

        if (op.equals("push")) {
            ins.nextToken(); int x = (int)ins.nval;
            q[++tt] = x;
        }
        else if (op.equals("pop")) hh++;
        else if (op.equals("empty")) {
            if (hh <= tt) out.println("NO");
            else out.println("YES");
        }
        else out.println(q[hh]);
    }

    out.flush();
}
```

单调栈

常见模型：在一个序列中，找出每一个数左边离它最近的且满足某种性质（如 最大最小）的数的位置

先想暴力算法，再利用单调性进行优化

```
static int N = 100010;

static int n;
static int tt;
static int[] stk = new int[N];

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;

    while (n-- > 0) {
```

```
        ins.nextToken(); int x = (int)ins.nval;

        while (tt > 0 && stk[tt] >= x) tt--;
        if (tt > 0) out.print(stk[tt]+" ");
        else out.print("-1 ");

        stk[++tt] = x;
    }

    out.flush();
}
```

单调队列

经典应用：求滑动窗口中最大值or最小值

```
static int N = 1000010;

static int n, k;
static int[] a = new int[N];
static int hh, tt = -1;
static int[] q = new int[N];    //存储下标的单调队列(与目标值一一对应，所以合理)

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;
    ins.nextToken(); k = (int)ins.nval;

    for (int i=0; i<n; i++) { ins.nextToken(); a[i] = (int)ins.nval; }

    for (int i=0; i<n; i++) {
        if (hh <= tt && i-k+1 > q[hh]) hh++;

        while (hh <= tt && a[q[tt]] >= a[i]) tt--;
        q[++tt] = i;

        if (i-k+1 >= 0) out.print(a[q[hh]]+" ");
    }

    out.println();

    hh = 0; tt = -1;
    for (int i=0; i<n; i++) {
        if (hh <= tt && i-k+1 > q[hh]) hh++;

        while (hh <= tt && a[q[tt]] <= a[i]) tt--;
        q[++tt] = i;

        if (i-k+1 >= 0) out.print(a[q[hh]]+" ");
    }

    out.flush();
}
```

KMP

习惯下标从1开始

1. 暴力算法如何做

```
S[N], p[M];

//朴素做法
for (int i=1; i<=n; i++) {
    bool flag = true;
    for (int j=1; j<=m; j++) {
        if (s[i+j-1] != p[j]) {
            flag = false;
            break;
        }
    }

    if (flag) 匹配成功
}
```

2. 如何去优化

- **next[i] 表示以i为终点，且后缀和前缀相等的子串的最大长度。**

  next[i] = j　　p[1, j] = p[i-j+1, i]

```
static int N = 100010, M = 1000010;

static int n, m;
static int[] ne = new int[N];
static char[] p = new char[N], s = new char[M];


public static void main(String[] args) throws Exception {
    n = Integer.parseInt(inb.readLine());   //下标从1开始
    String tmp = inb.readLine(); tmp = " "+tmp; p = tmp.toCharArray();
    m = Integer.parseInt(inb.readLine());
    tmp = inb.readLine(); tmp = " "+tmp; s = tmp.toCharArray();

    for (int i=2, j=0; i<=n; i++) {
        while (j > 0 && p[i] != p[j+1]) j = ne[j];
        if (p[i] == p[j+1]) j++;
        ne[i] = j;
    }

    for (int i=1, j=0; i<=m; i++) {
        while (j > 0 && s[i] != p[j+1]) j = ne[j];
        if (s[i] == p[j+1]) j++;
        if (j == n) {
            out.print(i-n+" ");
            j = ne[j];
        }
    }

    out.flush();
}
}
```

作用

- 可用于求循环节（字符串哈希无法使用）