

又向遍历 set(). remove() 需要更多逻辑 (详见13章)

Idea: 在列表的双向运行  $\Rightarrow$  需要实现 `instance methods`, / list 方法 & 另一个迭代器.

/ 迭代器允许访问基础列表 (`remove()`); 但在一个迭代中用另一个访问修改基础列表

将发生异常 (why): 莫名惊底层在怎次为迭代器).

Iterable interface.    public interface Iterable<T>

{

    Iterator<T> iterator();    实现 Iterable.

}

$\rightarrow$  任何有迭代器的类都满足

## Lecture 23

在集合中 查找 一个元素

[集合  $C$ . key value  $K$ , 查找与  $K$  匹配的的对象]

已知的集合与搜索

Array	未排序数组: sequential	$O(N)$
	已排序数组: binary	$O(\lg N)$
Linked List	未排序链表: sequential	$O(N)$
	已排序链表: sequential	$O(N)$

目前技术为直接比较

(寻找 key 是否等于对象中的 key).

Symbol table / dictionary: 将值与 key 相关联的抽象结构  $\Rightarrow$  使用 key 在数据结构中搜索 值

将其定义为 接口: dictionary specification 不需要特殊实现

import java.util.Iterator;

public interface DictionaryInterface<K, V>

{

    public V add(K key, V value); // 加入 key 及对应 value

    key  
    ↓  
    value  
    ↓

Dictionary 是一个抽象类, 用以储存键值对  
 $\Rightarrow$  给出键和值, 将其存储在 Dictionary 对象中  
(像一个键/值对列表)

    public V remove(K key); // 从 dictionary 中移除 key(及对应 value).

    public V getValue(K key); // 返回该键的值.

    public boolean contains(K key); // 是否包含

    public Iterator<K> getKeyIterator(); //

    public Iterator<V> getValueIterator(); //

    public boolean isEmpty(); // 测试该 dictionary 是否不存在从键到值的映射(是否为主),

    public int getSize(); // 返回条目数即键的数量

    public void clear(); //

}

Dictionary 接口用现成的知识 (sorted array, linked list) 实现  
- direct comparisons of keys  $\rightarrow$  为找到 target key  $k$ , 与更多的键进行比较

方法: { 假定一个数组  $T$ , 大小为  $M$

假定一个函数  $h(x)$ : 从键值空间映射到索引集合  $\{0, 1, \dots, M-1\}$ .  $h(x)$  runtime 与 key 值长成正比

如何插入找到某个 key  $x$ ?

0  
1  
2  
3  
...  
i      x  
...  
M-1

插入:  $i = h(x); T[i] = x;$   
 $//$  把值  $x$  (属于 key) 给到  
数组  $T$  的 index  $i$

查找:  $i = h(x);$   
 $if (T[i] == x)$   
    return true;  
else  
    return false;

} hashing 哈希

发生冲突:  $h(x_1) = h(x_2)$ , where  $x_1 \neq x_2$

[两个不同的 key 指向同一个位置].

避免冲突. (precon: key space  $K \leq$  table size  $M$ )

$|k| \leq M$ , 存在完美哈希

$|k| > M$ , 根据 鸽子原理, 冲突无法避免 (易多, 繁少).  
usual case.

## Collision Resolution

1) Open addressing 开放寻址法: 若表中索引处发生冲突, 尝试其他索引直到冲突解决

- 连续检查直至找到空槽插入

2) Closed addressing 封闭寻址: 每个索引表示一个键的集合

- index  $i$  的碰撞意味着在位置  $i$  的集合搜索多个 key

- 一个表的 key 存储数取决于集合允许的最大尺寸

? 我们是否可以查探碰撞次数? good hash function

- 使碰撞成为伪随机事件

- 使用整体 key, 探究 key 间的差异

- 使用 hash table 的完整地址空间

哈希函数  $addr = H(key)$

例: 基于电话号码的哈希函数,  $M=1000$

? 给前3位.  $\Rightarrow H(412-XXX-XXXX) = 412$

并不好, 前三位是区号.

最后三位更好, 更有代表性! 伪随机

例如：将单词放入大小为M的表的hash function

尝试：添加ASCII值

$$H("STOP") \rightarrow 83 + 84 + 79 + 80 = 326$$

问题1：一未充分考虑字母值差异

(不同的字母值相同的单词 or POTS 和 STOP 同一个哈希)

一未确定字母位置

问题2：一未使用完整地址空间

即大，单词的  $H(x)$  值不会相差非常多  $\Rightarrow$  对  $M=1000$  而言，中间容易碰撞，而两头较少。

优化！ 使用所有字符位置表格

$$1234 \neq 4321$$

?

每个数字有不同的10次方

$$1234 = 1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$$

$$4321 = 4 \times 10^3 + 3 \times 10^2 + 2 \times 10^1 + 1 \times 10^0$$

有不同 digit, 不同位置次方不同

$\Rightarrow$  用 ASCII (power of 256)

$x^* 256^n$  此时使用了 character & position 信息，如何使用 table?

原数值 % M

实践中，数字迅速变大，值的计算应当高效。

hashing 的另一个途径： $h(x) = f(x) \bmod M \rightarrow$  质数

↓ 某个将  $x$  转化为大的随机整数的函数。

idea：将 key 转化成较大整数，信号素原理的频率将降低。

开放寻址的方案：Linear Probing 线性探测

Idea：碰到碰撞发生；按顺序寻址 ( $i+1, i+2, \dots \bmod M$ ) 直至冲突解决。

Insert 的解决方案：找到 empty location

Find 的解决方案：{ 找到元素

未找到，但找到空位或完成循环返回 }

Index	Value	Probes $\rightarrow$ 找空位
0		$h(x) = x \bmod 11$ (求余数)
1	34	1 ✓ 14
2		✓ 17
3	14	1 ✓ 25
4	25	2 37
5	37	2 (下移1位) ✓ 34
6	17	
7	16	3 (下移1位) 16
8	26	5 (下移4位) 26
9		4

正常使用是  $O(1)$

? 表满了怎么办  $\Rightarrow$  定义荷载系数

$$\alpha = \frac{N}{M} \rightarrow \text{key} \rightarrow \text{index}$$

↓ 越小越好。

3 (index 3 已被占,  $i+1=4$ )

4 (index 4 被占,  $i+1=5$ )

5 (index 5, 6 都被占,  $i+2=7$ )

4

给一个key  $x$ , 插入到像均匀散列后的概率是  $\frac{1}{m}$ .

filled  
filled  
filled  
filled  
filled  
→ 指示这, }  
占据了  $C$  个连续位置  
} C+1 个位置都满足  
由于 probing linear, 往往这样.

基坏处在于找到空位时, 必须解决冲突.  
必须遍历整个散列表  
↓ 又增加, cluster 变大.  
搜索时间变长很多.

## Lecture 24

撞上,  $\alpha \rightarrow 1$ , run time 从  $O(1)$  变到接近  $O(N)$ . 如何解决?

→ 碰撞后, 使 all locations available to a key

使自己填充位置的概率将被重新分配到剩余的位置, 而不是直接占座位

思想: 有  $C$  个已填的位置及  $(M-C)$  个空位

使得  $P(\text{insert at a filled location}) = 0$

$P(\text{insert at any empty location}) = \frac{1}{m} + \frac{C/M}{M-C}$   $\Rightarrow$  有  $\frac{C}{M}$  的可能哈希值到已填位置

想把该概率给  $M-C$  个空位

即使 i 处发生冲突, 其他 open 位置仍能插入新建

## Double Hashing

思想: 碰撞时如 LP 增加索引, 但增量不自动为 1.

使用  $h_2(x)$  来决定增量

→ 散列到相同位置的键有不同的增量

当  $x_1 = x_2, h_1(x_1) = h_2(x_2)$ ;

$h_2(x_1) = h_2(x_2)$  的可能性很低

此方法允许碰撞的键建在表里

任意移动 (取决于  $h_2(x)$  的值)

但当冲突后, 由于  $h_2(x)$  不变

? 确保  $h_2(x)$  工作

$h_2(x) > 0$  [ $\because$  有  $i \rightarrow i+1$ ]

↓ increment.

0		
1	34	1
2		
3	14	1
4	37	1
5	16	1
6	17	1
7		
8	25	2
9		
10	26	2

$h_1(x) = x \bmod 11$   
 $h_2(x) = (x \bmod 7) + i$

— 确保所有 index 都被试过一次前，没有一个 index 被试过两次  
(否则容易碰碰撞)

方法：令 M 为素数

$\alpha = \frac{N}{M}$  越大，double hashing 提升越高

[但性能仍会下跌，趋于  $\Theta(N)$ ]

↓ why？ 一表内只有 M 个位置，空位只会越来越少

— 即便均匀，多次碰撞也会发生。

↓ 尤其对于 Insert & Unsuccessful Find.  $\Rightarrow$  { 这两种操作下都要找到空位。  
到后面要接近 M 个单元才能找到 }

## 开放寻址问题

↓ N 接近 M 时，性能降低

我们希望 table 内总有一部分是空的 { Linear Probing,  $\alpha = \frac{1}{2} \checkmark$   
Double Hashing,  $\alpha = \frac{3}{4} \checkmark$  }

? 如何实现

监视逻辑大小 N (number of entries) vs 物理大小 M (array length) 来计算

resize array & rehash all values, 当  $\alpha$  超过阈值 (工作量似乎很大)

Index Value (=如何删除？)

0

这是个 LP 表，假设  $H(z) = 2$ ，放在 index 4 是因为碰撞

1 W

? 搜索历程：2、3、4，在 4 处到

2 X

现在删除 Y 再找 Z：搜索停留在 3，找不到 Z 即便它存在  
(删除 Y 破坏了链)

3 Y

⇒ 如何处理？

4 Z

对于 DH，将数据标记为 “deleted”  $\Rightarrow$  { 插入可以重用  
查找可以经过 }

方法：Y 删除后，Z 移上去

↓ 但该法不适用 double hashing

DH 可以做啥，考虑每个位置的 3 种状态。

同样适用于 linear probing (=)

{ Empty  $\begin{cases} \text{插入 stop.} & (\text{把值放那}) \\ \text{查询 stop} & (\text{return not found}) \end{cases}$   
Full  $\begin{cases} \text{插入 not stop} & (\text{持续查看}) \\ \text{查询 not stop} & (\text{inside a cluster}) \end{cases}$   
Deleted  $\begin{cases} \text{插入 stop} & (\text{重复利用 position}) \\ \text{查询 not stop} & \end{cases}$

需要周期性刷表  
少使用过的位数不过“空”。

## 封闭寻址问题

每个位置代表一个数据集合 (冲突在集合内解决，而不改变位置)

常见形式：separate chaining (分离链连接法)

Index	Value	$h(x) = x \bmod 11$
0		以链表存储
1	→ [34]	
2		
3	→ [4] → [25]	
4	→ [27] → [26]	
5		
6	→ [4]	
7		

性能取决于 chain length

a) not found search 将遍历整个 chain

其平均 chain length =  $\alpha$

{ ① 最坏  $N > M$ ,  $O(N)$

② 但是  $N$  过大时,  $\alpha$  变大, 需 resize array, 得  $O(N)$

③ Worst, poor hash f 也会降至  $O(N)$

public class HashedDictionary<K, V> implements DictionaryInterface<K, V>

(key, value) 对

Linear Probing 的 Java 实现

[内部类的定义]

```
protected final class Entry<K, V>
{
    private K key;
    private V value;
    private Entry(K searchKey, V dataValue)
    {
        key = searchKey;
        value = dataValue;
    } // 构造器
    private K getKey()
    {
        return key;
    } // 获取 key
    private V getValue()
    {
        return value;
    } // 获取 value
    private void setValue(V newValue)
    {
        value = newValue;
    } // 设置 new value
} // 哈希表(K, V)的设置与输入 .
```

哈希表是条目  $(K, V)$  对象的集合, 而不是  $K, V$  对的基础

将条目  $(K, V)$  存进数组 .

private Entry<K, V>[] hashTable;
private int tableSize;
private int numberOfEntries;

少暴力构造器:

public HashedDictionary(int initialCapacity) // 初始化容量 .

```
{
    initialCapacity = checkCapacity(initialCapacity);
    numberOfEntries = 0;
    int tableSize = getNextPrime(initialCapacity); // 获得 hash 表物理尺寸,
    checkSize(tableSize); // check .
    Entry<K, V>[] temp = (Entry<K, V>[]) new Entry[tableSize];
    hashTable = temp;
    initialized = true;
}
```

数组深度 .

构建完成, 考虑 add() 方法 .

```
public V add(K key, V value)
{
    if ((key == null) || (value == null)) // precondition, 满足此就异常, 否则才执行 .
        throw new IllegalArgumentException("Cannot add null to a dictionary.");
    else
    {
        V oldValue;
        int index = getHashCode(key); // index 获得, 来自 key

        if ((hashTable[index] == null) || (hashTable[index] == AVAILABLE)) // 哈希表空引
        { // Key not found, so insert new entry
            hashTable[index] = new Entry<K, V>(key, value);
            numberOfEntries++;
            oldValue = null;
        }
    }
}
```

|/ 没有叫到 index, 插入  $K, V$  对  
并覆盖老值 .

这时可以插入条目

为空或 available

(之前是空位或完全空)

```
else // index location满了
{
    // Key found; get old value for return and then replace it
    oldValue = hashTable[index].getValue(); // 获得旧值
    hashTable[index].setValue(value); // 替代，更新新值
}

if (isHashTableTooFull())
    enlargeHashTable(); // 快满时扩大

return oldValue;
}
```

## ② enlargeHashTable() 的实现

```
private void enlargeHashTable()
{
    Entry<K, V>[] oldTable = hashTable;
    int oldSize = hashTable.length; // 获得老table的长度

    int newSize = getNextPrime(oldSize + oldSize); // newSize 是 oldSize 的两倍
    checkSize(newSize);
    Entry<K, V>[] tempTable = (Entry<K, V>[] )new Entry[newSize]; // 建一个和老hashTable一样大的临时数组
    hashTable = tempTable;
    numberOfEntries = 0; // 重置条目数，由 add() 递增

    for (int index = 0; index < oldSize; index++)
    {
        if ( (oldTable[index] != null) && (oldTable[index] != AVAILABLE) )
            add(oldTable[index].getKey(), oldTable[index].getValue());
    }
} // 将旧数组中的条目重新哈希进新的数组。
  // (剔除空位置和旧条目)
```

## Lecture 25

String Matching : 思想  
给一个长度为 M 的 pattern string, P  
给一个长度为 N 的 text string, A

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	H	E	T	H	E	Y	T	H	Y	T	H	E	M	E	M

P 中字符能否匹配到 A 上的字符

## Brute Force Algorithm 暴力算法

由两个串的开头开始，从左到右逐字比较

如果匹配不上，从 pattern 的开始再尝试 text 的下一位。

(直至完全匹配上)

用嵌套循环实现

```
public static int search1(String pat, String txt)
{
    int M = pat.length(); // My pat 长度
    int N = txt.length(); // My txt 长度
    for (int i = 0; i <= N - M; i++)
    {
        int j;
        for (j = 0; j < M; j++)
        {
            if (txt.charAt(i+j) != pat.charAt(j))
                break;
        }
        if (j == M) return i;
    }
    return N; // 没找到
}
```

## 暴力算法的性能

- Normal case  $O(N)$

- Worst case  $N > M$ ,  $O(NM)$

将该算法用于 `Index of()` 方法.

提升 worst case 性能

提高 normal case 性能

另法: hash 能用来匹配字符串吗?

hash function for string:

$$S[0]^* B^{n-1} + S[1]^* B^{n-2} + \dots + S[n-2]^* B^1 + S[n-1]$$

$B$  为 integer

若  $B =$  字符集中的字符数, 结果便唯一

意味只要整数值匹配, 字符串也匹配

→ 例. 若  $B = 32$ ,  $h("CAT") == 67 \times 32^2 + 65 \times 32^1$

$$+ 84 \times 32^0 = 70772$$

"hash" 来测试等值.

优化 1> 保持合理大小的整数值 (不大于 int & long)  
2> 增量更新哈希值, 查找匹配字符串

## ① Rabin Karp 算法

1) hash 值 mod 出一个大 integer, 确保不溢出.

2) 对于每个不匹配, 删左添右

注:  $h(X)$  对任何键唯一, 键空间 < 表大小  
(从而保证不碰撞) 但不切实际.

?发生碰撞 hash 值匹配但 string 不匹配

也测试 扩表  
逐字匹配.

1) check, 算法保证  $O(N)$  跑完

算法大根元率正向

2) Monte Carlo 版本.

2) check, 算法大根元率  $O(N)$  跑完.

worst case,  $T(MN)$ .

算法保证是正确的

## ② Las Vegas 版本

若 text 为  $N$ , pattern 为  $M$ , 时间为  $\frac{N}{M}$

大改进, sub-linear time

这个技术为 mismatched character heuristic

MC 不匹配字符后提示

A = 

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	W	A	B	C	X	A	B	C	Y	A	B	C	Z

P = 

0	1	2	3
A	B	C	D

先 HCD 与 W

不匹配直接跳到后 4 个

综上，暴力算法 { normal  $O(N)$   
worst  $O(MN)$  可能性低

Rabin-Karp { normal  $O(N)$   
worst  $O(MN)$  (Las Vegas版, 可能性非降低)

Boyer-Moore {  $O(\frac{N}{M})$ . MLC启发  
 $O(N)$  worst case, 由第二向启发保证

## Lecture 26

Stack 栈：仅从顶部添加或移除数据的数据结构  $\Rightarrow$  基础操作  
逻辑上，只有顶部的元素能看得到（只一层盘子）

Push: 将一个元素推到栈顶

Pop: 从栈顶弹出一个元素

Peek: 在不干扰的情况下看栈顶

```
public interface StackInterface<T>
{
    public void push(T newEntry); // 把一个新条目加到栈的顶部
                                    // newEntry 为形参 → 要被加入栈的对象
    public T pop(); // 删除该栈的顶部条目，并返回；返回的是顶部条目
                    // 如果空栈，抛出异常
    public T peek(); // 检查顶部条目；返回的是顶部；空栈就抛出异常
    public boolean isEmpty(); // 检测该栈是否为空；若栈空则返回true
    public void clear(); // 删除栈中所有条目
}
```

Stack 的访问顺序是后进先出  $\Rightarrow$  LIFO  $\Rightarrow$  Last In First Out

其应用：Run-time Stack for method calls (方法调用的运行栈，尤其是递归)

{ 当调用一个方法，它的调用记录被推入运行栈顶  
结束时，调用记录从栈顶弹出。

### 关于括号的匹配

( ) () () - match  
((((( ))))) - match  
(( ))<sup>4</sup> - don't match 右括不足  
( ) ( - don't match 括号乱序  
[ () ] - don't match 错误括号类型

$\Rightarrow$  用堆栈来编译测试匹配括号的代码吗？

{ a character variable, 用以存储当前字符  
a stack  
一个能插入数据的方法

MC是Boyer-Moore算法的两种实现之一

↓也有问题，可能导致错过。

修正：对pattern预处理，创建right array

{ 数组按照字母表的字符索引  
不匹配时值意味着可以略过的步数。 }

X Y X Y X ) X Y X Y. X与Z匹配，  
X Y X Y Z 但匹配中间的Y

思想：所有字符不在pattern中，设置为-1

越左，引起步数。

正确数组的值越大，能跳的越多

```
for all i right[i] = -1;  
for (int j = 0; j < M; j++)  
    right[p.charAt(j)] = j;
```

pattern p = ABCDE  
0 1 2 3 4  
→ j = ...

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	

```
for (int j = 0; j < M; j++)  
    right[p.charAt(j)] = j;
```

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	

-1是供 max skip  
pattern里的字母值>0  
提供 smaller skip.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	

0 1 2 3 4  
XYXYZ

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	

0 1 2 3 4  
XYXYZ

MC.

```
public int search(String txt) {  
    int M = pat.length(); // 声明 pat 长度 M.  
    int N = txt.length(); // 声明 txt 长度 N.  
    int skip; // 声明 skip 变量  
    for (int i = 0; i <= N - M; i += skip) {  
        skip = 0;  
        for (int j = M-1; j >= 0; j--) {  
            if (pat.charAt(j) != txt.charAt(i+j)) {  
                skip = Math.max(1, j - right[txt.charAt(i+j)]);  
                break;  
            }  
        }  
        if (skip == 0) return i; // 找到了  
    }  
    return N; // 没找到  
}
```

pat = X ... X<sup>n</sup>  
txt = Y X X X X  
j = 0,  
right['X'] = 4.  
skip 但只能取 1.

Worst case: O(MN)

为余两个分支？（保证 worst case 为 linear）

遇到一个左括号, push onto stack

遇到一个右括号, check the stack { 空栈 → 报错(无匹配左括号)

读入所有输入, check the stack  
非空栈 → 检查字符 { 与左括号不匹配 → 报错(不匹配)  
继续  
不空, 报错(没有括号匹配左括号)

```
public static boolean checkBalance(String expression)
```

```
{  
    StackInterface<Character> openDelimiterStack = new ArrayStack<Character>(); // 声明栈  
    int characterCount = expression.length(); // 声明 int 变量 characterCount 为表达式长度  
    boolean isBalanced = true;  
    int index = 0;  
    char nextCharacter = ' ';  
    while (isBalanced && (index < characterCount)) {  
        nextCharacter = expression.charAt(index);  
        switch (nextCharacter) {  
            case '(': case '[': case '{':  
                openDelimiterStack.push(nextCharacter);  
                break;  
            case ')': case ']': case '}':  
                if (openDelimiterStack.isEmpty()) isBalanced = false;  
                else {  
                    char openDelimiter = openDelimiterStack.pop();  
                    isBalanced = isPaired(openDelimiter, nextCharacter);  
                }  
                break;  
            default: break;  
        }  
        index++;  
    }  
    if (!openDelimiterStack.isEmpty())  
        isBalanced = false;  
    return isBalanced;  
}
```

栈可以用来评估 post-fix expressions 后缀表达式(运算符在操作数之后)

操作符跟随着操作数(无需括号)

2. 例.

例.  $20 \ 10 \ 6 - \ 5 \ 4 * + \ 14 \ - \ / = 20 / (10-6) + (5*4) - 14$  不成立 ✓ .

思想 在于每个运算符用于最近两个操作数

将操作数压入栈, 遇到运算符时, 弹出最后两个操作数; 执行操作, 再将结果压回栈顶

详, 读取下一个字符 { 这是个操作数, 把它放入栈顶

这是个运算符, 弹出右操作数 - 弹出左操作数 - 应用运算符

所有字符读完后, 栈弹出最终答案

20 10 6 - 5 4 \* + 14 - / =  
20 [ (10-6) + (5\*4) ] - 14 STACK

(10-6) + (5\*4)

20 / [ (10-6) + (5\*4) ] - 14 = 2

## 数组 & 链表的实现方式的数据结构

Array {  
 Push - add to end of logical array  
 Pop - remove from end of logical array  
 }  
 Linked List {  
 Push - onto front of linked list  
 Pop - from front of linked list
}

Queue 队列: 数据加到末尾，从前端移除

- 遇事上讲，只能访问 front item

基础操作 {  
 enqueue: 把一个元素放到队列末尾  
 dequeue: 把一个元素从队列前端移除  
 front: 查看前端元素而不干扰它
}

? 队列实现: 前后都可访问的结构体；enqueue与dequeue只需O(1)

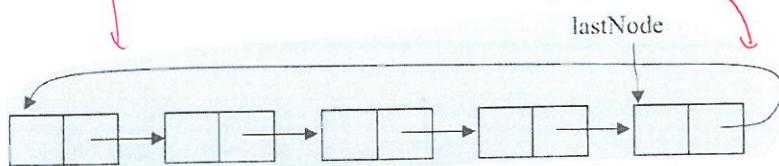
array-based queue: 循环方法访问数据

? 用链表实现队列: 有一个双链表可以访问前部或后部

- enqueue从后端加入对象, dequeue从前端移走对象 (简单)  
 (很多已在JDK预设)

Queue是一个接口, 而 LinkedList类实现 Queue

链式操作: 循环不链表 (最后的extra link有用)



关于enqueue操作: 加一个新节点到 lastNode 后.

```
newNode = new Node(newEntry, lastNode.next); // 声明新节点
lastNode.next = newNode; // 将 newNode 赋给 ...
lastNode = newNode; // 定义新的 lastNode.
```

关于dequeue操作: 把lastNode的后继节点移除.

```
frontNode = lastNode.next;
lastNode.next = frontNode.next;
return frontNode.data;
```

// frontNode 是 lastNode 的下一个,  
// 把 lastNode 的下一个变成 frontNode 的下一个  
// 即少掉 frontNode

## 一些数据结构的总结

- 1) Bag 包: ArrayList、LinkedList
- 2) List interface 列表接口: ArrayList、LinkedList .
- 3) Stack 栈: ArrayStack . LinkedStack .
- 4) Queue 队列: linked list ver. 、 circular array-based ver.

|  
| 以 Stack 和 Queue 为例, 等待 resize 时, 数组版本比链表版本快 .

|  
| push()、pop() → enqueue()、dequeue() . > 更快 .  
| 但不会自动 resize  
| (不足又浪费),

少但现实中多已预定义 . { 标通常数组  
队列通常链表 .