基于hash算法的DNA序列的k-mer index 问题

摘 要

序列比对是生物信息学中的经典问题，为了在数量日渐庞大的基因序列中查询k-mer序列的位置，文中基于哈希算法建立了合适的快速查询索引方法。首先将基因序列中的A、T、G、C分别用四进制数表示，然后利用四进制转换成十进制的方式取得关键码值建立哈希表进行索引查询，最后在16G内存的Windows系统下，采用C语言建立哈希表，并加入链表以达到快速索引的目的并使K的取值范围为[1~16]。

根据建模过程，将对建立索引的算法进行叙述和冲突分析；对建立索引算法的计算复杂度和空间复杂度进行分析；对索引查询进行叙述及性能分析；分析整套算法程序在不同k值下内存占用及极限分析。总结分析整套索引算法性能，对算法进行缺陷分析及改进方案。

**关键词：**索引算法；Hash表；链表；K-mer快速索引

1. 问题重述
2. **问题背景**

给定一个DNA序列，这个系列只含有4个字母ATCG，如 S =“CTGTACTGTAT”。给定一个整数值k，从S的第一个位置开始，取一连续k个字母的短串，称之为k-mer（如k= 5，则此短串为CTGTA），然后从S的第二个位置，取另一k-mer（如k= 5，则此短串为TGTAC），这样直至S的末端，就得一个集合，包含全部k-mer 。如对序列S来说，所有5-mer为｛CTGTA，TGTAC，GTACT，TACTG，ACTGT，TGTAT｝。

通常这些k-mer需一种数据索引方法，可被后面的操作快速访问。例如，对5-mer来说，当查询CTGTA，通过这种数据索引方法，可返回其在DNA序列S中的位置为｛1，6｝。

1. **问题要求**

现在以文件形式给定100万个DNA序列，序列编号为1-1000000，每个因序列长度为100。

1. 要求对给定k， 给出并实现一种数据索引方法，可返回任意一个 k-mer所在的DNA序列编号和相应序列中出现的位置。每次建立索引，只需支持一个k值即可，不需要支持全部k值。
2. 要求索引一旦建立，查询速度尽量快，所用内存尽量小。
3. 给出建立索引所用的计算复杂度，和空间复杂度分析。
4. 给出使用索引查询的计算复杂度，和空间复杂度分析。
5. 假设内存限制为8G，分析所设计索引方法所能支持的最大k值和相应数据查询效率。
6. 按重要性由高到低排列，将依据以下几点，来评价索引方法性能
7. 索引查询速度
8. 索引内存使用
9. 8G内存下，所能支持的k值范围 ? 建立索引时间
10. 问题分析

人类基因组计划[1]就是测定组成人类[染色体](http://baike.baidu.com/view/6633.htm" \t "_blank)(指[单倍体](http://baike.baidu.com/view/15182.htm" \t "_blank))中所包含的30亿个[碱基](http://baike.baidu.com/view/24576.htm" \t "_blank)对组成的DNA序列，从而绘制[人类基因组](http://baike.baidu.com/view/34582.htm" \t "_blank)图谱，并且辨识其载有的[基因](http://baike.baidu.com/view/8563.htm" \t "_blank)及其序列，达到破译人类遗传信息的最终目的。然而想要在数量庞大的基因序列中查询k-mer序列的位置，需要利用计算机来分析处理[2]，利用数据结构原理，建立模型构造算法，从而利用计算机有限的资源解决上述复杂问题。

针对上述问题：哈希表算法[3]以及链表数据结构[4]能够实现快速索引查询到k-mer序列的位置。

1. 根据题中所给的信息，将数据以一个二维数组的形式存储为 ，然后在每一行中，依所给的k值将基因序列分成组数据，求出每组数据的关键码值，并将关键码值与此组k-mer所在位置建立对应关系并存储到表中，从而建立哈希表。
2. 因为将k-mer序列与哈希表中的位置建立了一一对应的关系，将需要查找的k-mer序列的关键码值计算出，则可以直接将其在哈希表中的位置输出，进而达到快速索引查询。
3. 根据实验过程中得到的数据对建立索引的计算复杂度和空间复杂度进行分析，详细分析见V-1.2,1.3。
4. 对索引查询的计算复杂度和空间复杂度进行分析详见V-2.2,2.3。
5. 给定不同的k 值，算法可动态自动分配内存建立哈希表，可实现的k值 取值范围为[1~14]，因为是直接在哈希表中查询对应的地址进行输出，因此索引查询速度非常快。
6. 按照题中所给指标的重要性顺序，在建模过程中，首先考虑索引查询速 度，其次动态内存分配是尽量减少索引对内存的消耗，在8G内存限制下，使k值支持[1~14]，最后优化添加计数器记录已经存在地址的k-mer个数，倘若达到所有k-mer种类数，则停止建立索引，索引成功建立。
7. 符号说明

针对题中的问题，我们建立哈希表使所有的k-mer序列与表中的地址一一对应，然后当给定一个k-mer序列时，便可直接在哈希表中查找其相对应的地址输出。为了简化问题，将在基因序列中查询k-mer序列问题进行数学建模，利用计算机及数据结构进行分析处理，进而解决问题。其中所使用的相关符号表示如下表1所示：



1. ：文中所用符号说明表
2. 模型建立
3. **算法结构介绍**
4. **哈希表**

哈希表是根据设定的哈希函数H(key)和处理冲突方法将一组关键字映射到一个有限的地址区间上，并以关键字在地址区间中的象作为记录在表中的存储位置，这种表称为哈希表或散列，所得存储位置称为哈希地址或散列地址。作为线性数据结构与表格和队列等相比，哈希表无疑是查找速度比较快的一种。

散列表,它是基于高速存取的角度设计的，也是一种典型的“空间换时间”的做法。顾名思义，该数据结构能够理解为一个线性表，可是当中的元素不是紧密排列的，而是可能存在空隙。

散列表（Hash table，也叫哈希表），是依据关键码值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。散列函数的计算结果是一个存储单位地址，每一个存储单位称为“桶”。设一个散列表有m个桶，则散列函数的值域应为[0,m-1]。

1. **链表**

链表是一种物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。链表由一系列结点（链表中每一个元素称为结点）组成，结点可以在运行时动态生成。每个结点包括两个部分：一个是存储数据元素的数据域，另一个是存储下一个结点地址的指针域。 相比于线性表顺序结构，操作复杂。由于不必须按顺序存储，链表在插入的时候可以达到的复杂度，比另一种线性表顺序表快得多，但是查找一个节点或者访问特定编号的节点则需要的时间，而线性表和顺序表相应的时间复杂度分别是和。

使用链表结构可以克服数组链表需要预先知道数据大小的缺点，链表结构可以充分利用计算机内存空间，实现灵活的内存动态管理。但是链表失去了数组随机读取的优点，同时链表由于增加了结点的指针域，空间开销比较大。链表最明显的好处就是，常规数组排列关联项目的方式可能不同于这些数据项目在记忆体或磁盘上顺序，数据的存取往往要在不同的排列顺序中转换。链表允许插入和移除表上任意位置上的节点，但是不允许随机存取。链表有很多种不同的类型：单向链表，双向链表以及循环链表。

1. **哈希表设计**
2. **K-mer序列关键码值生成函数H(x)**

DNA序列是由四个字母A、T、C、G排列而成的，则可将其等效成四进制数，进而H(x)函数根据这个特征将四进制数转为十进制数作为哈希表关键码值。其对应关系为式(1)：

 (1)

若,则

 (2)

从式(1)中看出每个字母对应一个四进制数，对应的四进制数为，根据式(2)算出它的十进制数为。这个十进制数即为5-mer序列在哈希表中的关键码值。根据式(2)可以推出关键码值的一般计算式为：

 (3)

对于给定的所有基因序列，文中利用图1所示的方法将每行的100个基因序列分成(100-k+1)组k-mer序列。图1中给出的是k=5时的5-mer序列的形成。



1. ：k-mer序列的形成
2. **哈希表结构**



1. ：哈希表存储结构图

将k-mer通过公式(3)转换为十进制的关键码值存入哈希表中的关键码值一列，并将关键码值与此k-mer所在位置建立对应关系，从而便于索引寻址。这里采用的方法是：桶定址法

桶：一片足够大的存储空间。桶定址：为表中的每个地址关联一个桶。如果桶已经满了，可以使用开放定址法来处理。

文中建立的哈希表并利用链表存储全部k-mer基因序列的结构如上图2所示。

1. **冲突处理方法**
2. 线性探查法：冲突后，线性向前试探，找到近期的一个空位置。缺点是会出现堆积现象。存取时，可能不是同义词的词也位于探查序列，影响效率。
3. 双散列函数法：在位置冲突后，再次使用还有一个散列函数产生一个与散列表桶容量互质的数，依次试探，使探查序列跳跃式分布
4. 模型算法及性能分析
5. **建立索引的算法、计算复杂度及空间复杂度分析**
6. **算法分析**

文中利用进制转换及哈希表对k-mer序列实现快速索引查询，整个算法流程图如图3所示。哈希表初始化时，根据用户输入的k值， 计算出存储哈希表需要的空间，利用内存动态分配函数动态分配内存，k-mer所在行数为100 0000以内所以我们采用LNode结构体来描述一个索引（占用20字节）存储。

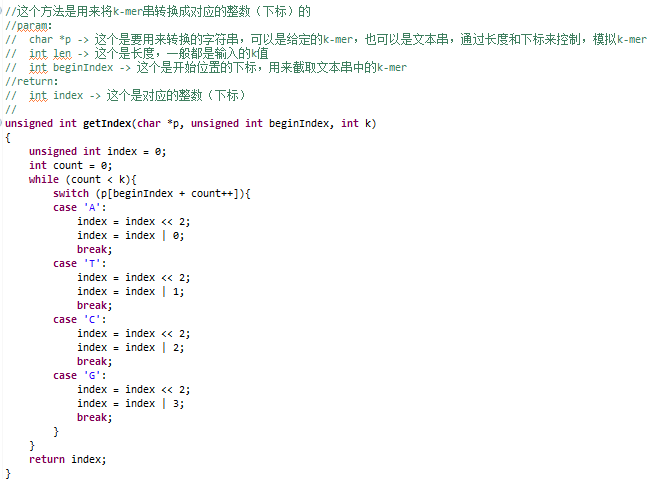


1. ：算法流程图

索引结构代码如下所示：

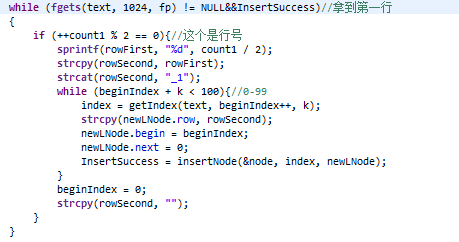
求关键码值如上叙述采用四进制转十进制数作为关键码值存入哈希表用于位置索引。

求关键码值代码：



建立哈希表过程中先将传入的长度为100的字符串分成每k个字符一段的串，每分好一个就求一个关键码值存储哈希表，为了提高建立索引的速度，每个关键码值只对应一个位置信息，也就是说索引查询时只返回k-mer的一个位置信息。倘若每个关键码值都有其对应的位置信息了，就退出索引建立，索引建立完成。

建立哈希表代码：



1. **计算复杂度分析**

计算机科学中，算法的时间复杂度[5]是一个函数，它定量描述了该算法的运行时间。其计算方法如下：

1. 一般情况下，算法的基本操作重复执行的次数是模块的某一个函数，因此，算法的时间复杂度记做：。
2. 在计算时间复杂度的时候，先找出算法的基本操作，然后根据相应的各语句确定它的执行次数，再找出 的同数量级(它的同数量级有以下：1，，，，的平方，的三次方，2的次方，!)，找出后， = 该数量级，若求极限可得到一常数，则时间复杂度。

不考虑8G空间限制，k在1~50的范围内每条DNA序列分段的循环不断增加，计算复杂度也随之增加，而50~100范围内循环则又开始减少。根据上述计算复杂度的定义，建立索引的计算复杂度为。

1. **空间复杂度分析**

一个算法的空间复杂度[6]定义为该算法所耗费的存储空间，它也是问题规模的函数。渐近空间复杂度也常常简称为空间复杂度。空间复杂度是对一个算法在运行过程中临时占用存储空间大小的量度。一个算法在计算机存储器上所占用的存储空间，包括存储算法本身所占用的存储空间，算法的输入输出数据所占用的存储空间和算法在运行过程中临时占用的存储空间这三个方面。

逻辑上哈希表空间占用计算公式：

 (4)

(每条索引信息逻辑空间占用为20Byte)

|  |  |
| --- | --- |
| k值 | 建立索引后空间占用（GB） |
| 1 | 0.00000007 |
| 2 | 0.00000030 |
| 3 | 0.00000119 |
| 4 | 0.00000477 |
| 5 | 0.00001907 |
| 6 | 0.00007629 |
| 7 | 0.00030518 |
| 8 | 0.00122070 |
| 9 | 0.00488281 |
| 10 | 0.01953125 |
| 11 | 0.07812500 |
| 12 | 0.31250000 |
| 13 | 1.25000000 |
| 14 | 5.00000000 |

1. ：不同K值对应的空间占用表

根据上表中的数据分析索引的空间占用问题并用matlab绘图表示为图4。



1. ：建立索引的空间占用分析(逻辑)

由以上图表可以清晰的看到，索引建立的内存消耗随着k的增长幂指数增长，与k-mer排列组合种类数()相对应，我们采用LNode结构体来描述一个索引（占用20字节）存储。电脑运行内存是16G，运行软件是Eclipse，理论是能支持到k=14,但是实际上最大只能支持k=13，原因是C程序中分配内存的malloc函数，最大分配的内存约2G（实测为1984M，程序见附录2）。考虑8G内存的限制，则能支持的k的取值范围为1~13。

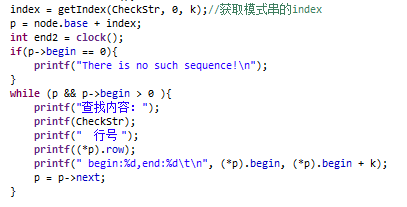
1. **索引查询的算法、计算复杂度及空间复杂度分析**
2. **算法分析**

索引查询需要将要查找的k-mer的关键码值求出来，然后直接从哈希表中输出对应关键码值的位置信息，求关键码值的函数同建立索引过程的函数，不再重复给出。索引查询算法的流程图如图5所示。



1. ：索引查询算法流程图

索引查询代码如下：



1. **计算复杂度分析**

索引查询的过程中只有求关键码值时需要进行计算，其计算复杂度为：。

1. **空间复杂度分析**

索引查询过程用到只一个存关键码值的中间变量，根据空间复杂度的定义可知：索引查询过程中的空间复杂度为。

1. **整套索引算法在不同k值下内存占用及极限分析**

在表3中统计了当用户取不同的k值时，k-mer序列的种类数量以及建立索引需要的的时间。

|  |  |  |  |
| --- | --- | --- | --- |
| k值 | k-mer种类数 | 建立索引耗时（秒） | 索引查询耗时（秒） |
| 1 | 4 | 9.721 | 0 |
| 2 | 16 | 9.857 | 0 |
| 3 | 64 | 10.037 | 0 |
| 4 | 256 | 10.764 | 0 |
| 5 | 1024 | 12.661 | 0 |
| 6 | 4096 | 13.245 | 0 |
| 7 | 16384 | 13.486 | 0 |
| 8 | 65536 | 14.201 | 0 |
| 9 | 262144 | 16.196 | 0 |
| 10 | 1048576 | 17.154 | 0 |
| 11 | 4194304 | 18.393 | 0 |
| 12 | 16777216 | 18.680 | 0 |
| 13 | 67108864 | 19.101 | 0 |

1. ：不同k值对应的建立索引时间

测量上表耗时数据设备参数：

CPU：Intel(R) Xeon(R) CPU E3-1230 v3 @3.3GHz

内存（RAM）：DDR3 1600MHz 16G

硬盘：1T 7200转



1. ：k-mer种类数随k值变化曲线

根据实验结果，利用数学工具matlab进行绘图分析k-mer种类数随k值的变化趋势如图6所示。

依据表3中的数据对建立索引耗时和k取值之间的关系进行分析如下图7所示。



1. ：建立索引耗时随k值的变化曲线

对比图6和图7两折线图，不难发现建立索引耗时与k-mer种类数目变化趋势并不是一致的，原因有两点，第一点建立索引用时在9秒前几乎为零因为此算法在确认所有种类k-mer都有对应位置后索引就建立完成不管是否将所有数据都遍历过，9秒前k-mer种类较少，所以没有检索所有数据就已经完成了索引建立；第二点建立索引耗时并没有跟着k-mer种类指数上升而是放缓，原因是k-mer种类太多，将所有数据都遍历结束后，有的k-mer也没有找到与其匹配的位置，所以耗时基本就是遍历所有数据的时间。

1. **算法缺陷分析及改进方案**
2. **缺陷分析**

建立索引过程中会有一个k-mer与多个位置对应，由于hash表存储限制发生了冲突，只能返回k-mer的一个位置信息，不能将k-mer的所有位置信息都返回。

1. **改进方案**

针对这个缺陷，可以采取hash算法冲突处理方法——开放定址法，或采用hash的改进算法tri-树（字典树）算法，在这里我们采用冲突处理的方案对此算法进行改进，在内存有限制的情况下，放弃一点k值的范围，腾出部分空间来存储更多k-mer位置信息，从而弥补之前的缺陷，代码见附录。

1. 参考文献
2. 陈竺，李伟俞，曼熊慧，陈赛娟，[人类基因组计划的机遇和挑战，从结构基因组学到功能基因组学](http://www.cnki.net/kcms/detail/detail.aspx?filename=SMHX805.001&dbcode=CJFQ&dbname=CJFD1998&v=" \t "_blank)[J]. 生命的化学. 1998.05。
3. 刘红梅，刘国庆，[基于k-mer组分信息的系统发生树构建方法](http://www.cnki.net/kcms/detail/detail.aspx?filename=XXSW201302006&dbcode=CJFQ&dbname=CJFDTEMP&v=" \t "_blank)[J]. 生物信息学. 2013.02
4. 严蔚敏，数据结构与算法分析[M]，北京：清华大学出版社，2011．
5. Stephen Prata（著），张海龙（译），袁国忠（译）．C++ Primer Plus（第六版）中文版：人民邮电出版社，2012.07．
6. 谢楚屏，陈慧南编，[数据结构](http://ss.zhizhen.com/detail_38502727e7500f261865aea9e9d965d2798af533cebf1df41921b0a3ea25510134114c969f2eae5cb626bb6e70e4bc99b08fcc1cc94d4dd14db3e95a0d0d60a864a13fd6c09a8cf49f962bbd414566b5" \t "_blank)[M]，北京：人民邮电出版社，1994.
7. 许卓群(等)，[数据结构与算法](http://ss.zhizhen.com/detail_38502727e7500f2607d2d644ad54a10893acfab265e789081921b0a3ea25510134114c969f2eae5cf2ebaac0e876c978e182abf17a16c7970f30c9eed4a1817a47bdd0e6f70d839ec8f03a484e3d98c5" \t "_blank)[M]，北京：高等教育出版社.
8. 附录

**附录1：**

/\*

============================================================================

Name : NewMathModel.c

Author :

Version :

Copyright : Your copyright notice

Description : Hello World in C, Ansi-style

============================================================================

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <string.h>

#include <time.h>

//base:头指针

//index:索引下标

//length:长度

//listsize:最大长度

typedef struct

{

unsigned int listsize;

struct LNode \*base;

}Node;

//row:行号

//begin:起始位置

//next:下一个结果的指针，若没有则指针为0

typedef struct LNode

{

char row[9];

int begin;

struct LNode \*next;

}LNode;

//初始化线性表的表头

//param：

// \*N 线性表头指针

// k K-mer中的K

//初始化内存，并且创建一个结果链表的头指针。

void InitTable(Node \*N, int k){

(\*N).base = (LNode\*)malloc(((int)pow(4, k))\*sizeof(LNode));//分配4的k次方个空间用来存储结果的链表

if (!(\*N).base)

{

exit(0);

}

(\*N).listsize = (int)pow(4, k);

}

//初始化结果的链表

int InitLink(LNode \*p){

p = (LNode\*)malloc(sizeof(LNode));

if (!p){

return 0;

}

(\*p).begin = 0;

(\*p).next = 0;

return 1;

}

//这个方法是用来将k-mer串转换成对应的整数（下标）的

//param：

// char \*p -> 这个是要用来转换的字符串，可以是给定的k-mer，也可以是文本串，通过长度和下标来控制，模拟k-mer

// int len -> 这个是长度，一般都是输入的k值

// int beginIndex -> 这个是开始位置的下标，用来截取文本串中的k-mer

//return：

// int index -> 这个是对应的整数（下标）

//

unsigned int getIndex(char \*p, unsigned int beginIndex, int k)

{

unsigned int index = 0;

int count = 0;

while (count < k){

switch (p[beginIndex + count++]){

case 'A':

index = index << 2;

index = index | 0;

break;

case 'T':

index = index << 2;

index = index | 1;

break;

case 'C':

index = index << 2;

index = index | 2;

break;

case 'G':

index = index << 2;

index = index | 3;

break;

}

}

return index;

}

//

//增加新的结果结点

// node 结果表

// position 位置

// newP 新的结果节点

//return:

// 1 插入成功

unsigned int insertNode(Node\* node, unsigned int position, LNode newNode){

LNode \*p;

p = (LNode\*)malloc(sizeof(LNode));

if (!p){

return 0;

}

strcpy((\*p).row, newNode.row);

(\*p).begin = newNode.begin;

(\*p).next = newNode.next;

if ((\*((\*node).base + position)).begin > 0){//(\*((\*node).base + position)).begin > 0 表示不是第一个数据

p->next = ((\*node).base + position)->next;

((\*node).base + position)->next = p;

return 1;

}

else{//(\*((\*node).base + position)).begin = 0 表示是第一个数据，因为begin值在1-100，均大于0，0是初始值

(\*((\*node).base + position)).begin = newNode.begin;

(\*((\*node).base + position)).next = newNode.next;

strcpy((\*((\*node).base + position)).row, newNode.row);

free(p);

return 1;

}

}

int main(){

int InsertSuccess = 1;

char CheckStr[13]; //13

int k = 5; //这个K应该是用户输入的

int beginIndex = 0; //这个是起始的下标

unsigned int index = 0; //索引值

char text[1024]; //一行缓冲区

int count1 = 0;

int count2 = 0;

char rowFirst[7] = "";

char rowSecond[9] = "";

Node node; //索引表的定义

LNode newLNode;

LNode \*p; //新节点

FILE \*fp,\*fp1; //读取文件

//=====================================================================

InitTable(&node, k); //初始化索引表

fp = fopen("D://software//eclipse//workspace//mathModel//src//solexa\_100\_170\_1.fa", "r");

printf("建立索引表......\n");

int start = clock();

while (fgets(text, 1024, fp) != NULL&&InsertSuccess)//拿到第一行

{

if (++count1 % 2 == 0){//这个是行号

sprintf(rowFirst, "%d", count1 / 2);

strcpy(rowSecond, rowFirst);

strcat(rowSecond, "\_1");

while (beginIndex + k < 100){//0-99

index = getIndex(text, beginIndex++, k);

strcpy(newLNode.row, rowSecond);

newLNode.begin = beginIndex;

newLNode.next = 0;

InsertSuccess = insertNode(&node, index, newLNode);

}

beginIndex = 0;

strcpy(rowSecond, "");

}

}

fp1 = fopen("D://software//eclipse//workspace//mathModel//src//solexa\_100\_170\_2.fa", "r");

InsertSuccess = 1;

while (fgets(text, 1024, fp1) != NULL&&InsertSuccess)//拿到第一行 why1024??

{

if (++count2 % 2 == 0){//这个是行号

sprintf(rowFirst, "%d", count2 / 2);

strcpy(rowSecond, rowFirst);

strcat(rowSecond, "\_2");

while (beginIndex + k < 100){//0-99

index = getIndex(text, beginIndex++, k);

strcpy(newLNode.row, rowSecond);

newLNode.begin = beginIndex;

newLNode.next = 0;

InsertSuccess = insertNode(&node, index, newLNode);

}

beginIndex = 0;

strcpy(rowSecond, "");

}

}

int end = clock();

fclose(fp);

printf("建立索引表完毕,耗时%dms\n", end - start);

//========================================================================

fflush(stdin);

printf("请输入一条%d-mer：\n", k);

gets(CheckStr);

while (strlen(CheckStr) != k){

printf("请输入%d-mer:", k);

fflush(stdin);

gets(CheckStr);

}

int start2 = clock();

index = getIndex(CheckStr, 0, k);//获取模式串的index

p = node.base + index;

int end2 = clock();

if(p->begin == 0){

printf("There is no such sequence!\n");

}

while (p && p->begin > 0 ){

printf("查找内容：");

printf(CheckStr);

printf(" 行号 ");

printf((\*p).row);

printf(" begin:%d,end:%d\t\n", (\*p).begin, (\*p).begin + k);

p = p->next;

}

printf("查找耗费时间:%dms\n", end2 - start2);

printf("按任意键继续……");

getchar();

//===================================================================

return 0;

}

**附录2：**

#include "stdio.h"

#include "stdlib.h"

#include "string.h"

struct mem\_link {

struct mem\_link \*prev, \*next;

void \*ptr;

};

long total\_mem = 0;

struct totalmem\_link{

struct mem\_link \*prev, \*next;

long total\_mem;

};

struct mem\_link \*alloc\_mem(struct totalmem\_link \* head) {

int sz = 0;

struct mem\_link \*ml = NULL;

for(;(sz < 2048) || (sz > 2048 \* 1024); ) {

sz = rand();

//printf("size = %d\n", sz);

}

ml = (struct mem\_link \*)malloc(sizeof(struct mem\_link));

if(!ml)

return NULL;

memset(ml, 0, sizeof(struct mem\_link));

ml->ptr = malloc(sz);

if(!ml->ptr) {

return NULL;

}

if(head->next)

{

head->next->prev = ml;

ml->next = head->next;

ml->prev = (struct mem\_link \*)head;

head->next = ml;

}

else

{

head->next = ml;

ml->prev = (struct mem\_link \*)head;

}

memset(ml->ptr, 0, sz);

head->total\_mem += sz;

//printf(" alloc %d bytes.\n", sz);

return ml;

}

int main() {

struct totalmem\_link head;

struct mem\_link \*ml;

long M = 0;

int failed = 0;

head.prev = NULL;

head.next = NULL;

head.total\_mem = 0;

while(1) {

ml = alloc\_mem(&head);

if(!ml){

printf("alloc failed\n");

failed ++;

if(failed > 10)

break;

}

// printf("total: %d M: %d \n", head.total\_mem, M);

if( (head.total\_mem >> 24) > M) {

M = (head.total\_mem >> 24);

printf("alloced memory %d M\r\n", (head.total\_mem >> 20));

}

}

return 0;

}