**江西师范大学计算机信息工程学院学生实验报告（10）**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **专业：** | **数据科学与大数据技术2班** | **姓名：** | **赖丽婷** | **学号：** |  | **日期：** | **2021.12.26** |

|  |  |  |  |
| --- | --- | --- | --- |
| 课程名称 | 数据结构 | 实验室名称 | 计算机综合实验室 |
| 实验名称 | 排序 | | |
| 指导教师 | 李云清 | 成绩 |  |

**1.实验目的**（结出本次实验所涉及并要求掌握的知识点）

1. 理解并掌握内部排序的各种算法性能和适用场合
2. 理解并掌握Shell排序的基本思想及其算法
3. 理解并掌握堆排序的基本思想及其算法
4. 理解并掌握快速排序的基本思想及其算法
5. 理解并掌握归并排序的基本思想以及算法
6. 理解并掌握基于链表的基数排序的基本思想及其算法
7. 能根据具体问题的要求，选择合适的算法

**2.实验内容**（结出实验内容具体描述）

1. 设计直接插入排序算法函数进行升序排序，测试不同数据规模下的排序效率
2. 设计二分插入排序算法函数进行升序排序，测试不同数据规模下的排序效率
3. 设计Shell插入排序算法函数进行升序排序，测试不同数据规模下的排序效率
4. 设计简单排序算法函数进行升序排序，测试不同数据规模下的排序效率
5. 设计堆排序算法函数进行升序排序，测试不同数据规模下的排序效率
6. 设计冒泡排序算法函数进行升序排序，测试不同数据规模下的排序效率
7. 设计快速排序算法函数进行升序排序，测试不同数据规模下的排序效率
8. 设计归并排序算法函数进行升序排序，测试不同数据规模下的排序效率
9. 设计基数排序算法函数进行升序排序，测试不同数据规模下的排序效率

**3.算法描述及实验步骤**（用适当的形式表达算法设计思想与算法实现步骤）

1. 初始可以认为文件中第1个记录以及排好序，然后依次将第2个到第n个记录依次插入已排序的记录组成文件中，在对第i个记录就行插入时，第1到i-1都已经排好序，将第i个记录放到a[0]中，然后将a[0]与已经排好序的排序码就行从右向左依次比较，比较成功的往后移，比较不成功则找到了i应该插入的位置，进行插入
2. 在查找i的位置时，前i-1个记录已经排好了序，将第i位置的key与前i-1的中间位置的编码进行比较，如果key小于中间位置则可以在前半部分进行进行二分查找，否则在后半部分进行查找，直到查找的范围为空，即可确定插入key的位置，然后将此位置到i-1的值进行后移
3. 对有n个记录进行排序，首先取1个整数d<n,将n个记录分为d组，所有位置相差d的倍数的记录分在一组。在每组中使用直接插入排序进行组内排序。然后缩小d的值，重复进行分组和组内排序，一直到d=1结束
4. 首先从所有n个待排序记录中选出排序码最小的记录然后将该记录与第一个记录进行交换，再从剩下的n-1个记录中选出最小的与第2个记录进行交换。重复这样的操作，直到剩下两个记录时，再从中选出排序码最小的记录和第n-1个记录进行交换剩下的就是最大的排序码
5. 可以把堆看成一个完全二叉树，但是堆是一个一维数组，具有特征可表示为完全二叉树中任意分支结点都小于或者等于它的左右儿子结点的值。首先我们要对有左或者右儿子的位置进行建堆，建堆过程中一旦发生交换，需要在交换处进行调整，直到调整为根的子树为堆位置，这就是筛选算法，通过筛选算法，可以得到堆的第一个元素为最小的排序码，然后与堆的最后一个元素进行交换，同时让堆的个数减1因为此时根结点的左右子树都还满足堆的条件，所以可以从根节点处继续调整建堆
6. 对所有记录进行从左到右的比较，如果不符合排序要求进行交换，直到排到最后的位置，然后最后位置-1，继续从前面开始进行冒泡，如此重复，直到没进行冒泡交换为止
7. 从n个待排序的记录中取第1个记录，设法将该记录放置于排序后它最终应该放的位置，使它前面的记录排序编码不大于它的编码，它后面的排序编码不小于它的编码，然后对前后两部分排序记录重复上述操作，可将排序完成
8. 一个待排序记录构成的文件，可以看成由多个有序文件组成，对有序子文件进行若干次使用归并，得到一个有序文件，归并是指将两个有序子表合并成一个表的过程，对有n个记录的文件，从长度为1的次序进行归并，直到排到文件末尾然后以原来长度的两倍进行归并操作，还要注意最后能否进行归并操作进行判断处理

**调试过程及运行结果**（详细记录在调试过程中出现的问题及解决方法。记录实验执行的结果）

1. **总结**（对实验结果进行分析，问题回答，实验心得体会及改进意见）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 方法 | 1万 | 5万 | 10万 | 30万 | 50万 |
| 直接插入排序 | 0.074 | 2.432 | 8.403 | 35.255 | 68.180 |
| 二分插入 | 0.059 | 1.930 | 6.679 | 27.825 | 52.116 |
| Shell排序 | 0.003 | 0.012 | 0.024 | 0.088 | 0.371 |
| 简单选择 | 0.198 | 3.530 | 13.452 | 120.838 | 336.222 |
| 堆排序 | 0.001 | 0.011 | 0.018 | 0.078 | 0.138 |
| 冒泡 | 0.267 | 7.387 | 28.187 | 189.768 | 472.19 |
| 快速排序 | 0.001 | 0.008 | 0.016 | 0.071 | 0.134 |

快速排序适合大部分排序数据，归并排序虽然快，但是如果数据量大还要申请一个很大的数据空间，空间复杂度大，直接插入排序等插入排序方法时间复杂度很大,二分插入虽然可以加快查找效率，但是更换次数还是一样多，虽然Shell插入排序效率还可以，但是不稳定。堆排序既可以保存中间的比较结果又不会占用大量的附加储存空间，使排序效率提高，简单选择排序以及冒泡排序效率也很低

**6.附录**（程序源代码等）

void insertSort(int a[], int n)

{ /\*直接插入排序\*/

int i, j;

for(i = 2; i <= n ; i++){

a[0] = a[i];

j = i-1;

while(a[0] < a[j]){

a[j+1] = a[j];

j--;

}

a[j+1] = a[0];

}

}

void binInsertSort(int a[],int n)

{

int mid, left, right, i, j;

for(i = 2; i <= n; i++){

left = 1;

right = i - 1;

a[0] = a[i];

while(left <= right){

mid = (left + right)/2;

if(a[0] > a[mid]){

left = mid + 1;

}else{

right = mid - 1;

}

}

for(j = i -1; j >= left; j--){

a[j+1] = a[j];

}

a[left] = a[0];

}

}

void shellSort(int a[],int n)

{

int i, j, d;

d = n/2;

while(d>=1)

{

for (i = d + 1; i <= n; i++)

{

j = i - d;

a[0] = a[i];

while (j > 0 && a[j] > a[0])

{

a[j + d] = a[j];

j = j - d;

}

a[j + d] = a[0];

}

d = d/2;

}

}

void selectSort(int a[],int n)

{

int i, j, k;

for(i = 1; i <= n-1; i++){

k = i;

for(j = i+1; j <= n; j++){

if(a[j] < a[k]){

k = j;

}

}

if(k != i){

a[0] = a[i];

a[i] = a[k];

a[k] = a[0];

}

}

}

void sift(int a[],int k,int n)

{

int i,j,finished;

i=k;j=2\*i;

a[0]=a[k];

finished=0;

while((j<=n)&&(!finished))

{

if((j<n)&&(a[j+1]>a[j]))

j++;

if(a[0]>=a[j])

finished=1;

else

{

a[i]=a[j];

i=j;j=2\*j;

}

}

a[i]=a[0];

}

void heapSort(int a[],int n)

{

int i;

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

sift(a,i,n);

for (i=n;i>1;i--)

{

a[0]=a[i];

a[i]=a[1];

a[1]=a[0];

sift(a,1,i-1);

}

}

void bubbleSort(int a[],int n)

{

int i, j, blob;

blob = 1;

i = 1;

while(i <= n && blob){

blob = 0;

for(j = 1; j <= n-i; j++){

if(a[j] > a[j+1]){

a[0] = a[j];

a[j] = a[j+1];

a[j+1] = a[0];

blob = 1;

}

}

i++;

}

}

/\*

请设计快速排序算法函数void quickSort(int a[],int low,int right)，对a[low]..a[right]进行升序排序。

并测试在不同数据规模下的排序效率。

\*/

#include "Arrayio.h"

#include "time.h"

#define N 50000 /\*N为数据量大小，因data1.txt中只有50万个数，所以自行设定N值时需让N<=500000\*/

/\*请将本函数补充完整，并进行测试\*/

void quickSort(int a[], int low, int right)

{

int i, j;

//为什么一定要这个判断条件

if (low < right)

{

a[0] = a[low];

i = low;

j = right;

while (i < j)

{

while (i < j && a[j] > a[0])

{

j--;

}

if (i < j)

{

a[i] = a[j];

i++;

}

while (i < j && a[i] < a[0])

{

i++;

}

if (i < j)

{

a[j] = a[i];

j--;

}

}

a[i] = a[0];

quickSort(a, low, i - 1);

quickSort(a, i + 1, right);

}

}

int main()

{

int a[N + 1], n; /\*数据存储在a[1]...a[N]中\*/

printf("数据初始化...\n");

n = readData(a, N, "../data1.txt"); /\*从data1.txt中读入N个整数存入数组a，n为实际读入的数据个数\*/

printf("%d个数据排序中...\n", n);

clock\_t start, finish;

double total\_time;

start = clock();

quickSort(a, 1, n);

finish = clock();

total\_time = (double) (finish - start) /CLOCKS\_PER\_SEC;

printf("%.3fs\n", total\_time);

saveData(a, n, "../out.txt"); /\*排序结果存放在out.txt文件中\*/

printf("排序结束，排序结果保存在out.txt文件中。\n");

return 0;

}

void merge(int a[],int b[], int u,int m,int v)

{ /\*将有序段a[u..m],a[m+1..v]归并到a[u..v]\*/

int i, j, k;

i = u;

j = m + 1;

k = i;

while(i <=m && j <= v){

if(a[i] < a[j]){

b[k++] = a[i];

i++;

}else{

b[k++] = a[j];

j++;

}

}

if(i <= m){

while(i<=m){

b[k++] = a[i++];

}

}

if(j <= v){

while(j <= v){

b[k++] = a[j++];

}

}

}

/\*----一趟归并------\*/

void mergepass(int a[], int b[],int n,int len)

{ /\*对a[1..n]进行长度为len的一趟并归\*/

int i, j, k;

i = 1;

while(i <= n - 2\*len + 1){

merge(a, b, i, i+len -1, i + 2\*len -1);

i = i + 2\*len;

}

//如果剩余长度为一段多一点的话

if(i + len - 1 < n){

merge(a, b, i, i+len-1, n);

}else{

for(j = i; j <=n; j++){

b[j] = a[j];

}

}

}

/\*----归并排序------\*/

void mergeSort(int a[],int n)

{

int len;

int b[N];

len = 1;

while(len < n){

mergepass(a,b, n, len);

len = len\*2;

mergepass(b, a, n, len);

len = len\*2;

}

}

void radixSort(linklist head)

{

struct node2 q[10];/\*队列\*/

linklist p,r;

int i,j,k,x;

for (j=0;j<10;j++)

q[j].rear=q[j].front=NULL;

for (i=0;i<6;i++) //排序的最大数为6位数，共进行6趟分配收集过程

{

p=head->next; //分配

while (p)

{

head->next=p->next;

x=p->info;

for (j=0;j<i;j++)

x=x/10;

k=x%10; //取出本次按位分配的值

if (q[k].front==NULL) //队列为空

{

q[k].front=q[k].rear=p;

}

else //队列不为空

{

q[k].rear->next=p;

q[k].rear=p;

}

p=head->next;

}

//收集

r=head; //r为链尾指针

for (j=0;j<10;j++)

{

if (q[j].front!=NULL)

{ r->next=q[j].front;

r=q[j].rear;

q[j].front=q[j].rear=NULL;

}

}

r->next=NULL;

}

}