Algorithm in Practice

Author: Zhong-Liang Xiang
Date: Aug. 1st 2017

Prerequisite

生成随机整数数组，打印数组, 元素交换．

```
#include <stdlib.h>
#include <iostream>
#include <time.h>
using namespace std;

#define MAX 10 // 数组最大长度

void initList(int *, int len, int m);
void printList(int *, int len);
void swap(int *, int *);

int main() {
    int L[MAX];
    initList(L, MAX, 100);
    printList(L, MAX);

    return 0;
}

// 初始化 a list with len 个(0, m)之间的随机数
void initList(int *L, int len, int m) {
    srand((unsigned) time(NULL)); //初始化随机数种子
    for (int i = 0; i < len; i++) //产生len个随机数
        *(L + i) = (rand() % m);    // 0 - m之间随机数
}

//打印 1 维数组；
void printList(int *L, int len) {
    for (int i = 0; i < len; i++)
        cout << *(L + i) << ' ';
    cout << endl;
}

void swap(int *l, int *r) {
    int temp = *l; *l = *r; *r = temp;
}
```

0. Searching

0.1. Sequential-Search

```
// 无序数组顺序查找
int search_Seq(int *L, int len, int key) {
    int result = -1;
    for (int i = 0; i < len; i++) {
        if (*(L + i) == key) {
            result = i;
            break;
        }
    }
    if (result < 0)
        result = -1;
    return result;
}
```

0.2. Binary-Search

针对下面数组，查找7：

0 1 2 3 4 5 6 7 8 9

lo hi

Binary Search 算法描述：

```
while(lo <= hi):
        mid = (lo + hi)/2 = 4
        若 key == mid, 输出mid;
        否则  若key < mid, 调整 hi = mid - 1;
        否则  若key > mid, 调整 lo = mid + 1;
返回 -1;
```

```c
// 有序数组二分查找
int bin_Seq(int *L, int len, int key) {
    int lo = 0, hi = len - 1, mid;

    while (lo <= hi) {
        mid = (lo + hi) / 2;
        if (key == L[mid])
            return mid;
        if (key < L[mid])
            hi = mid - 1;
        else if (key > L[mid])
            lo = mid + 1;
    }
    // 循环退出，说明没找到key，返回-1
    return -1;
}
```

1. Sorting

默认升序．

- 插入排序
  - 直接插入排序
  - 二分插入排序
  - Shell-Sort
- 交换排序
  - Bubble-Sort
  - **Quick-Sort**
- 选择排序
  - 直选排序
  - 树型排序
  - **堆排序**
- Merge-Sort
  - 二路归并排序
  - 多路归并排序
- 分配排序
  - 关键字排序
  - 基数排序

1.1 Insertion-Sort

1.1.1 直接插入排序

- Stable

- Time complexity $O(n^2)$
- Space complexity $O(1)$

Idea: 左手 sorted 牌，右手拿新牌往左手牌里插．

```
// 代码片段
#define MAX 20 // 数组最大长度

void insertion_sort(int *L, int len){
    int key, i;
    for (int j = 1; j < len; j++){
        key = L[j];
        i = j - 1;
        // while循环目的：  移动 sorted 元素，为 key 找位置
        while(i >= 0 && L[i] > key){
            L[i + 1] = L[i];
            i -= 1;
        }
        // 循环退出，说明已为key找到插入位置，为 L[i+1] 处
        L[i + 1] = key;
    }
}

//测试
int main(){
    //数据准备
    int L[MAX];

    //排序前
    initList(L, MAX, 100); //0-100 random int number
    printList(L, MAX);

    //排序后s
    insertion_sort(L, MAX);
    printList(L, MAX);
}
```

1.1.2 二分插入排序

1.1.3 Shell 排序

1.2 交换排序

1.2.1 Bubble-Sort

- Stable
- Time complexity $O(n^2)$
- Space complexity $O(1)$

Idea: 小泡泡往上升．
算法描述：

```
do while 终止条件: 本趟若无交换，表示数组已排好序，无需再排序
i < len - 1; 最后一个元素不扫
L[0]为leftElem, L[0+1]为rightElem
大元素升到索引高的位置
```

```
void bubble_sort(int *L, int len) {
    bool swapped;
    do {
        swapped = false;
        for (int i = 0; i < len - 1; i++) {
            if (*(L + i) > *(L + i + 1)) {
                swap(L + i, L + i + 1);
                // 本趟若无交换，表示数组已排好序，无需再排序
                swapped = true;
            }
        }
    } while (swapped);
}
```

1.2.2 Quick-Sort

- Divide and conquer method
- Unstable
- Average time complexity $O(nlogn)$， worst case time complexity $O(n^2)$
- Space complexity $O(1)$

Idea: 应用分治法，选择某元素，如 L[lo]，放入k中，让小(乱序)的排在该元素左侧，大的在右侧(乱序)，k在中间，k所在的位置就是正确的位置．再对左右分别应用该方法．
Quick-Sort 算法描述：

```
1. 未排序数组找一元素作为标靶放入k中；
2. lo, hi 分别指向未排序数组首尾;
3. 从右向左扫描，若L[hi]>=key, hi--, 否则 L[lo] = L[hi];
4. 从左向右扫描，若L[lo]<=key, hi++, 否则 L[hi] = L[lo];
```

> Divide and conquer algorithm　简书
> 三步骤： Divide Conquer Combine
> 分治法与 Recursive method　被认为是孪生兄弟．
> 针对下面数组, 设定未排序数组最左侧的元素为标靶，如 L[0], 其值为：3

5	7	3	0	4	2	1	9	6	8
lo									hi

1 2 3 0 4 **5** 7 6 8 9
lo 标靶 hi

```
int divide(int *L, int lo, int hi) {
    int k = L[lo];
    do {
        // 正向思考，写全条件
        while (lo < hi && L[hi] >= k) hi--;
        // 若能执行 if 里的 code, 需满足 L[hi] < k, 值得学习
        if (lo < hi) { L[lo] = L[hi]; lo++; }
        while (lo < hi && L[lo] <= k) lo++;
        if (lo < hi) {L[hi] = L[lo]; hi--; }
    } while (lo != hi); //哨兵
    L[lo] = k;
    return lo;
}

void quick_sort(int *L, int lo, int hi) {
    int mid;

    if (lo >= hi) return; // 只有1元素或lo>hi, 递归终止条件
    // 分
    mid = divide(L, lo, hi);

    // 治
    quick_sort(L, lo, mid - 1); //递归
    quick_sort(L, mid + 1, hi);
}
```

1.3 选择排序

1.3.3 Heap-Sort

- Unstable
- Average time complexity $O(nlogn)$, worst case time complexity $O(nlogn)$, 而 Quick-Sort 是$O(n^2)$, 记录较少时不宜使用, 记录较多时表现很好.
- Space complexity $O(1)$

Idea: 完全二叉树，堆，性质为父节点大于等于子节点，根节点永远是最大(大顶堆)或最小节点(小顶堆)，依次弹出根节点，自然就排序了．

1.3.3.1 Heap

重点是堆概念，大顶堆，小顶堆，以及针对这两种堆的建堆和出堆操作．熟悉后，堆排序自然就会了. 下面代码表现了大顶堆的操作，包括:

- int Insert(MyHeap *pHeap, int nData) 插入新节点
  - int IncreaseKey(MyHeap *pHeap, int nPos) 比较调整新插入节点的位置，向上渗透
- int PopMaxHeap(MyHeap *pHeap) 返回堆中根节点的值，并删除根节点，删除后，还需调整堆

```cpp
// `ctrl+alt+f`: Format current file
#include <stdlib.h>
#include <iostream>
using namespace std;
// 大顶堆
typedef struct {
    int *pnData; //指向数据的指针
    int nSize; //当前堆中节点个数
} MyHeap;

int Insert(MyHeap *pHeap, int nData);
int IncreaseKey(MyHeap *pHeap, int nPos);
int PopMaxHeap(MyHeap *pHeap);

int main() {
    int i;
    MyHeap myHeap;
    //为结构成员分配存储空间
    //为什么不这么做？ & malloc(sizeof(int) * 11);
    myHeap.pnData = (int *) malloc(sizeof(int) * 11);
    myHeap.nSize = 0;

    //向空二叉树添加10个节点
    for (i = 1; i <= 10; i++)
        Insert(&myHeap, i); //边插入，边调整

    //查看插入的内容
    for (i = 1; i <= 10; i++)
        cout << myHeap.pnData[i] << ' ';
    cout << endl;

    //查看未被利用的 myHeap.pnData[0]
    cout << "未被利用的 myHeap.pnData[0]: " << myHeap.pnData[0] << endl;

    //弹出大顶，查看堆内容
    for (i = 1; i <= 10; i++) {
        cout << "PopMax: " << PopMaxHeap(&myHeap) << ' ' << endl;
        for (int j = 1; j <= 10 - i; j++) // check list after pop
            cout << myHeap.pnData[j] << ' ';
        cout << endl;
    }
    cout << endl;

    return 0;
}

int Insert(MyHeap *pHeap, int nData) { //插入新节点
    ++pHeap->nSize; //堆的节点数+1
    //索引号从 1 开始，方便操作；索引为 0 的地方浪费了？
    pHeap->pnData[pHeap->nSize] = nData;
    IncreaseKey(pHeap, pHeap->nSize);
    return 1;
}
```

```
//比较调整新插入节点的位置，向上渗透
int IncreaseKey(MyHeap *pHeap, int nPos) {
    while (nPos > 1) { //循环和其父节点比较
        int nMax = pHeap->pnData[nPos]; // temp
        int nParent = nPos / 2; //父节点号
        if (nMax > pHeap->pnData[nParent]) {
            pHeap->pnData[nPos] = pHeap->pnData[nParent];
            pHeap->pnData[nParent] = nMax;
            nPos = nParent;
        } else
            break;
    }
    return 1;
}

//返回堆中根节点的值，并删除根节点，删除后，还需调整堆
int PopMaxHeap(MyHeap *pHeap) {
    int nMax = pHeap->pnData[1];
    int nPos = 1; //当前节点
    int nChild = nPos * 2; //左孩子节点位置

    //最后一个节点赋值给根
    pHeap->pnData[nPos] = pHeap->pnData[pHeap->nSize];
    pHeap->nSize -= 1; //立即调整nSize

    //向下渗透
    int temp;
    //重点是循环条件设定  以及  下面的if条件
    //循环条件 nChild <= pHeap->nSize
    //if条件
    //nChild + 1 <= pHeap->nSize && temp < pHeap->pnData[nChild + 1]
    while (nChild <= pHeap->nSize) { //若左孩子存在
        temp = pHeap->pnData[nChild];

        //若右孩子存在，且比左孩子大
        if (nChild+1 <= pHeap->nSize && temp < pHeap->pnData[nChild+1]){
            nChild += 1;
            //temp里始终装左右孩子中的最大值，且nChild指针随之移动
            temp = pHeap->pnData[nChild];
        }

        //确定左右孩子大小后，大的那个若大于nPos元素，向下渗透
        if (pHeap->pnData[nPos] < temp) {
            pHeap->pnData[nChild] = pHeap->pnData[nPos];
            pHeap->pnData[nPos] = temp;
        }
        nPos = nChild; //重新调整 nPos
        nChild *= 2; //重新调整 nChild
    }
    return nMax;
}
```

1.3.3.2 Heap-Sort

未排序数据一个个放入堆，再从堆中取出.下面代码与1.3.3.1相比，修改了`main`,添加了`void heap_sort(int * L, int len)`及其在程序首部的函数声明，其他未变．

```
void heap_sort(int * L, int len);

int main() {
    int L[] = { 3, 38, 5, 44, 15, 36, 26, 27, 2, 46, 4, 19, 47, 48, 50 };
    int len = sizeof(L) / sizeof(L[0]);
    heap_sort(L, len); //测试
    return 0;
}

//未排序数据一个个放入堆，再从堆中取出
void heap_sort(int * L, int len) {
    MyHeap myHeap;
    //为结构成员分配存储空间
    myHeap.pnData = (int *) malloc(sizeof(int) * (len + 1));
    myHeap.nSize = 0;

    // 打印未排序的数组
    cout << "Unsorted array is: " << endl;
    for (int i = 0; i < len; i++)
        cout << L[i] << ' ';
    cout << endl;

    // 建堆
    for (int i = 1; i <= len; i++)
        Insert(&myHeap, L[i - 1]);

    // 输出排序结果
    cout << "sorted array by Heap-Sort is: " << endl;
    while (myHeap.nSize > 0)
        cout << PopMaxHeap(&myHeap) << ' ';
    cout << endl;
}
```

1.4 Merge-Sort

1.4.1 二路归并排序

- Stable
- Average time complexity $O(nlogn)$
- Space complexity $O(n)$

Idea: 两个有序序列，取队首（较小）元素送入新队列，新队列就已经排序了．

```c
void merge(int *L, int lo, int mid, int hi) {
    if (lo >= hi) // 若1个元素或 lo > hi, 则无法合并
        return;
    int LL[hi - lo + 1]; //临时数组
    int i = lo, j = mid + 1, p = 0;

    // 正向思考，写全条件
    while (i <= mid && j <= hi)
        LL[p++] = L[i] < L[j] ? L[i++] : L[j++];

    while (i <= mid)
        LL[p++] = L[i++];

    while (j <= hi)
        LL[p++] = L[j++];

    // LL已经装满，需把数据依次送入 L[lo ... hi]
    for (i = lo, p = 0; i <= hi; i++, p++)
        L[i] = LL[p];
} //merge

void merge_sort(int *L, int lo, int hi) {
    if (lo >= hi)
        return; //递归终止条件
    int mid = (lo + hi) / 2;
    merge_sort(L, lo, mid);
    merge_sort(L, mid + 1, hi);
    merge(L, lo, mid, hi);
} //merge_sort
```

1.4.2 多路归并排序