Research Project 3: Heap Merging

Kuan Lu

Ru-zhang Zheng

Jia-cheng Pan

Date: 2014-12-15

Chapter 1: Introduction

We are assigned to implement Merge operation on ordinary heaps, leftist heaps, and skew heaps. The size of input can be taken from 1000 to 10000. The run times must be plotted with respect to the sizes to illustrate the difference,

In the report we demonstrate the algorithm to merge these heap and give a clear analysis on the tests, we have also done a lot of comparison to find out whether the time complexity we figured accord with the testing result.

Chapter 2: Algorithm Specification

I. Algorithm I, merge two ordinary heaps

Data structure

Heap

struct HeapStruct{

    int Size;

    int *Elements;

};

Algorithm Description:

```
PriorityQueue OrdinaryMerge(PriorityQueue h1, PriorityQueue h2){
    int i;
    if(h1==NULL)
        return h2;
    if(h2==NULL)
        return h1;
    //Compare the size of two heap to decide that choose which one to be
inserted
    if(h1->Size > h2->Size){
        //Insert the element from h2 to h1 one by one
        for(i=1;i<=h2->Size;i++)
            h1 = O_Insert(h2->Elements[i],h1);
        return h1;
    }
    else{
        //Insert the element from h2 to h1 one by one
        for(i=1;i<=h1->Size;i++)
            h2 = O_Insert(h1->Elements[i],h2);
        return h2;
    }
```

```
}
```

II. Algorithm II, merge two leftist heaps

Data structure

Leftist Heap

struct heap{

int num;

heapnode lchild;

heapnode rchild;

int NPL;

};

```
heapnode Merge(heapnode h1, heapnode h2){
    if(h1==NULL)
        return h2;
    if(h2==NULL)
        return h1;
    //To decide that which heap's root is smaller
    if(h1->num < h2->num)
        return Merge1(h1,h2);
    else
        return Merge1(h2,h1);
}

//The actual routine to merge leftist heaps
heapnode Merge1(heapnode h1, heapnode h2){
    heapnode tmp;
    if(h1->lchild==NULL)
        h1->lchild = h2;
    else{
        //Using recursive way to do the same operation
        h1->rchild = Merge(h1->rchild, h2);
        //Judge if we need to swap the two child of it
        if(h1->lchild->NPL < h1->rchild->NPL){
            tmp = h1->lchild;
            h1->lchild = h1->rchild;
            h1->rchild = tmp;
        }
```

```
        //Adjust the Npl of the heap
        h1->NPL = h1->rchild->NPL + 1;
    }
    return h1;
}
```

III. Algorithm III, merge two skew heaps

Data structure

struct heap{

int num;

heapnode lchild;

heapnode rchild;

};

```
heapnode Merge(heapnode h1, heapnode h2){
    if(h1==NULL)
        return h2;
    if(h2==NULL)
        return h1;
    //To decide that which heap's root is smaller
    if(h1->num < h2->num)
        return Merge1(h1,h2);
    else
        return Merge1(h2,h1);
}

//The actual routine to merge skew heaps
heapnode Merge1(heapnode h1, heapnode h2){
    heapnode tmp;
    if(h1->lchild==NULL)
        h1->lchild = h2;
    else{
        //Using recursive way to do the same operation
        h1->rchild = Merge(h1->rchild, h2);
        //Do the swapping directly
        tmp = h1->lchild;
        h1->lchild = h1->rchild;
```

```
        h1->rchild = tmp;
    }
    return h1;
}
```

Chapter 3: Testing Results(Based on algorithm I)

I. Testing Environment

Intel Core i5-3230M CPU @2.60GHz

RAM 3.85GB

Microsoft Visual Studio 2010 on Windows 8.0 (64bits)

II. Some Modification

Add a free function the free the memory.

```
void freeheap(heapnode h){
        if(h->lchild!=NULL)
                freeheap(h->lchild);
        if(h->rchild!=NULL)
                freeheap(h->rchild);
        free(h);
}
```

III. Testing Results

Ps: all the following running time is base the on my CPU's frequency, because I don't know the exact frequency, so I can't get the exact running time, using FREQUENCY to represent my CPU's frequency, the

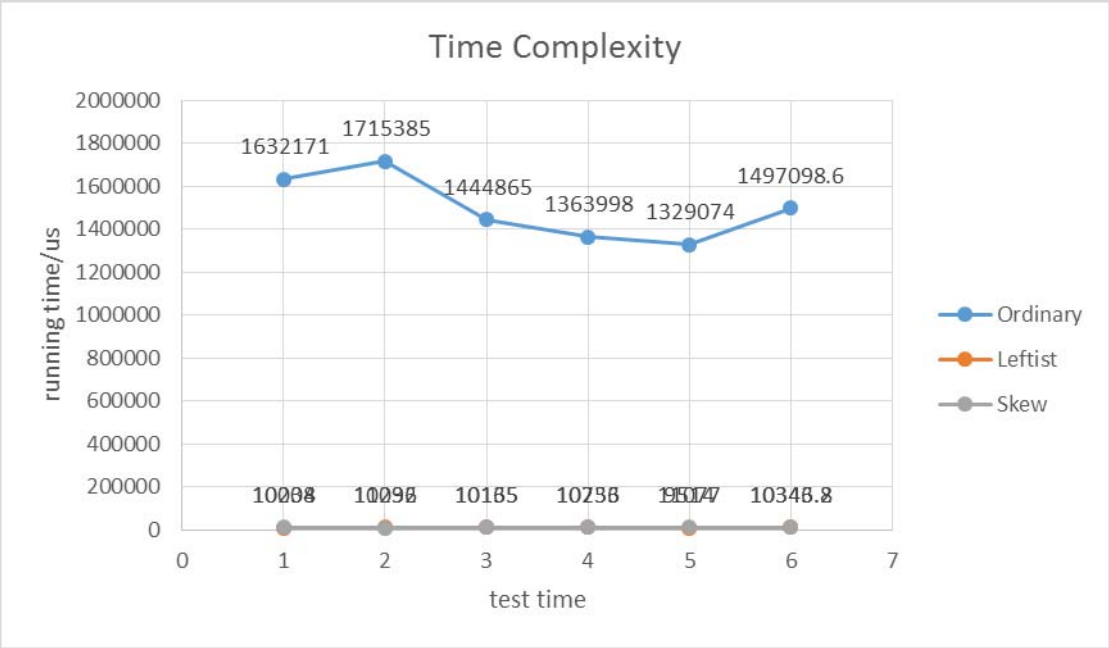
approximate running time is t/FREQUENCY*1000000 us.

Case1: All random input

Input: file//test1.in

Comments: it is a random input data, the size of heap1 is 4903 and the size of heap2 is 4109, time graph of the arithmetic is as follows.

Time Graph:



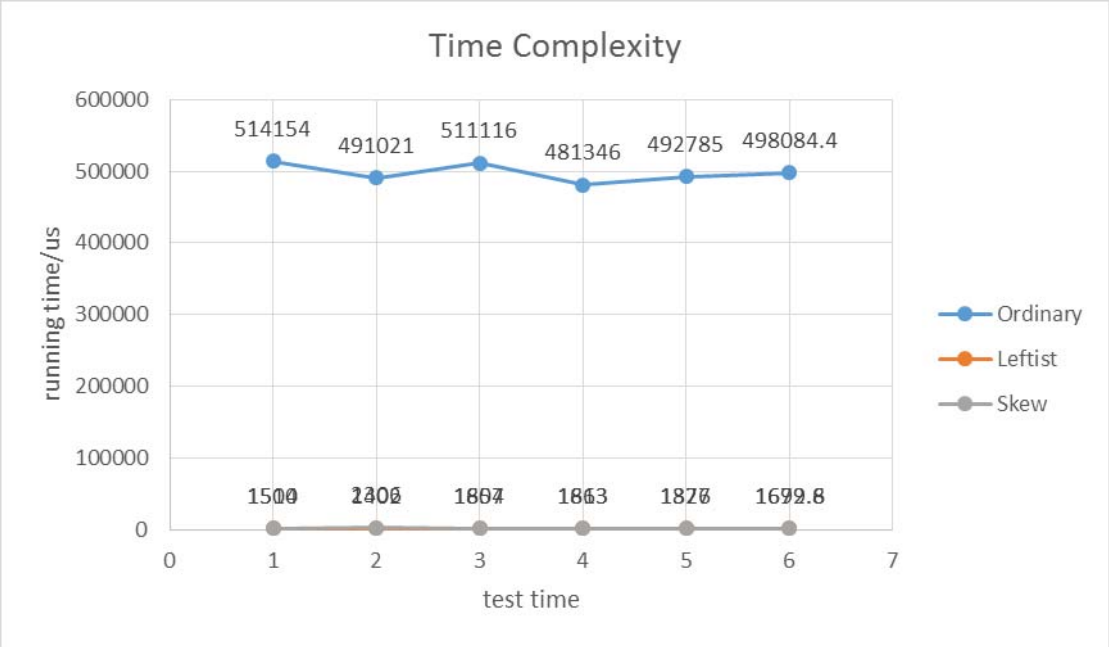
Comments: In case1, the average running time in Ordinary Heap Merge is 1497098.6, the average running time in Leftist Heap Merge is10346.2, and the average running time in Skew Heap Merge is 10343.8. Case1 is designed to test the program under the normal situation. The merge function works properly.

Case2: All the elements of the heap are same.

Input: file//test2.in

Comments: The elements of both the heaps are same, but the size of heap1 is different to the size of heap2, and the size of heap1 is 5291, the size of heap2 is 2272.

Time Graph:



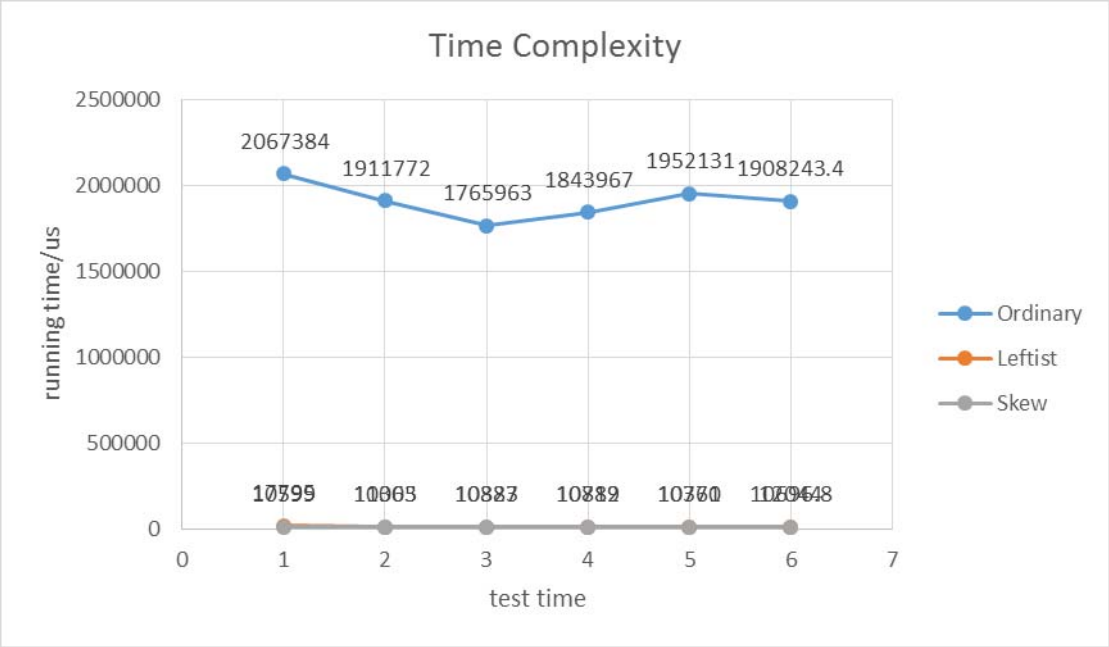
Comments: In case2, the average running time in Ordinary Heap Merge is 498084.4, the average running time in Leftist Heap Merge is 1679.8, and the average running time in Skew Heap Merge is 1692.6. Case2 is designed to test the case that the elements of both two heaps are same. The merge function works properly.

Case3: heap1 is same to heap2

Input: file//test3.in

Comments: case3 is the case that two same heaps merge.

Time Graph:



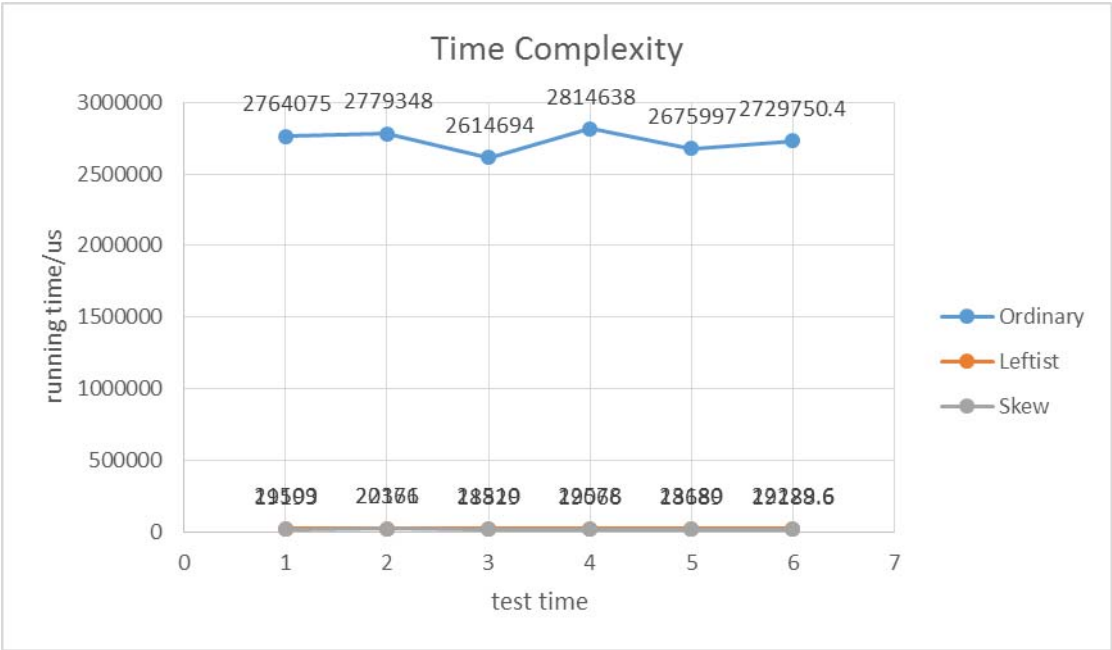
Comments: In case2, the average running time in Ordinary Heap Merge is1908243.4, the average running time in Leftist Heap Merge is 12044, and the average running time in Skew Heap Merge is 10696.8. Case3 is designed to test the case that merging two same heaps. The merge function works properly.

Case4:

Input: file//test4.in

Comments: The elements and size of both the heaps are same, the size is 10000, and the elements are continually increase.

Time Graph:



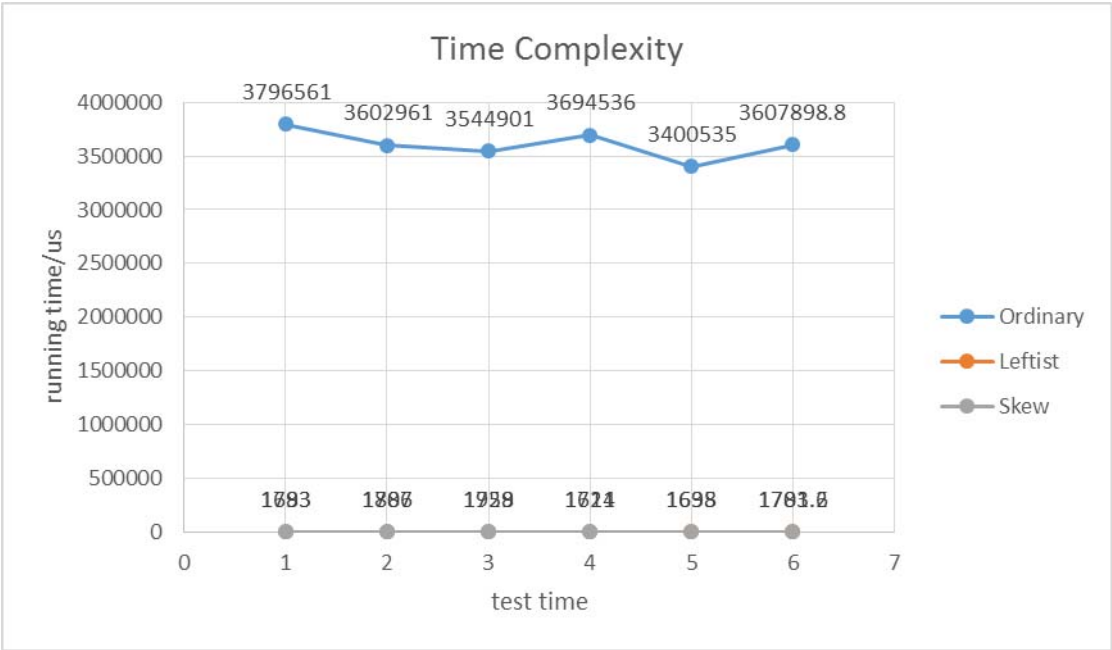
Comments: In case2, the average running time in Ordinary Heap Merge is 2729750.4, the average running time in Leftist Heap Merge is 22229.6, and the average running time in Skew Heap Merge is 19188.6. Case4 is designed to test whether the program will rich a worst case under this case. The merge function works properly.

Case5: opposite to case4

Input: file//test5.in

Comments: The elements and size of both the heaps are same, the size is 10000, and the elements are continually decrease.

Time Graph:



Comments: In case2, the average running time in Ordinary Heap Merge is 3607898.8, the average running time in Leftist Heap Merge is 1781.2, and the average running time in Skew Heap Merge is 1703.6. The ordinary heap merge running time of the function in this case is similar to case4. But the leftist heap merge and the skew heap merge running time is greatly decrease. The merge function works properly.

Chapter 4：Analysis and Comments

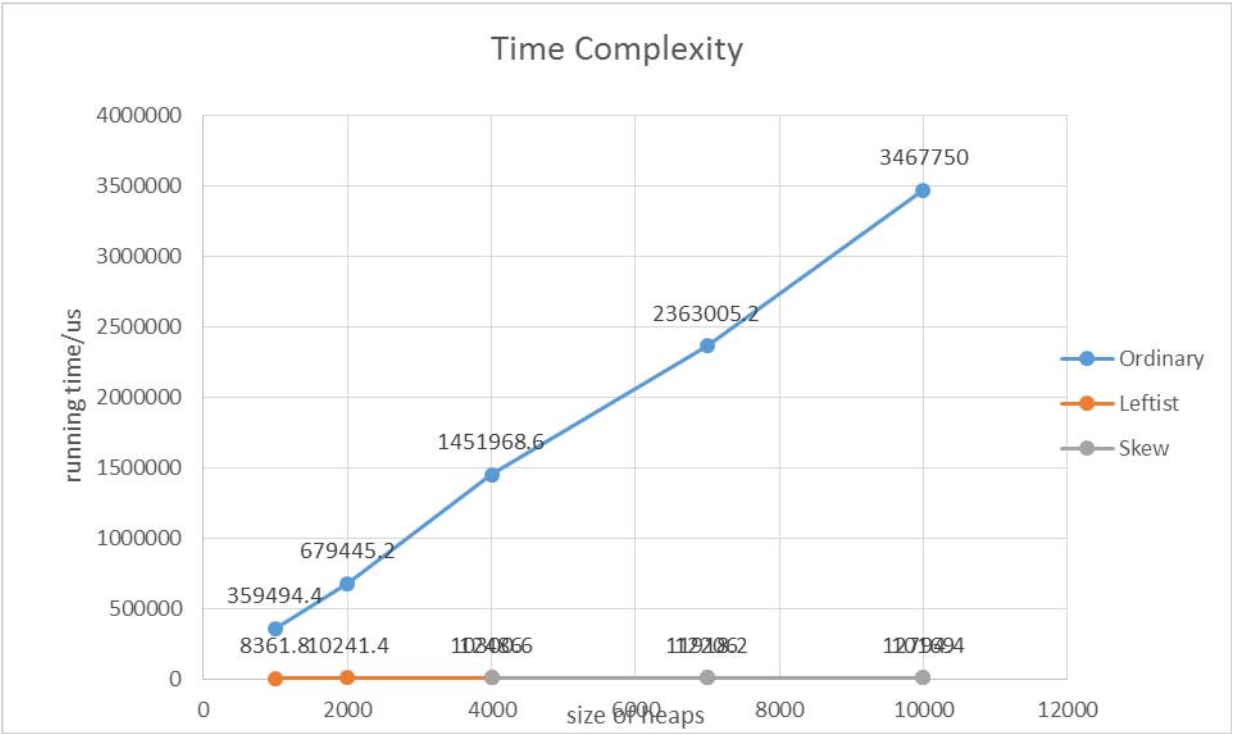
In our codes, the time complexity of ordinary heap merge is O(MlogN), the average time complexity of leftist heap merge is O(log(M+N)), and the average time complexity of skew heap merge is same to the leftist heap merge: O(log(M+N)), M represents the size of the small heap, and N represents the size of the large heap.

I use a function called “GetcycleCount” to get the Time Stamp of my CPU, and run 500 times, get the average running time.

It is like this:

```
inline unsigned __int64 GetCycleCount()
{
    __asm
    {
        _emit 0x0F;
        _emit 0x31;
    }
}
```

```
t1 = (unsigned long)GetCycleCount();
h1 = Merge(h1,h2);
t2 = (unsigned long)GetCycleCount();
```



From above, we can see that in the same input data, the running time of skew heap merge is similar to leftist heap merge, and the running time of ordinary heap merge is linearly increasing.

Declaration

We hereby declare that all the work done in this project titled "World's Richest" is of our independent effort as a group.

Author List:

Programmer: Ru-zhang Zhen
Tester: Jia-chen Pan
Report Writer: Kuan Lu