CSE 542 – Advanced Data Structures and Algorithms *Jon Turner*

Lab 4 Solution

Part A**. **LfheapSet and rrobinF

1. (30 points) Paste a copy of your changes to *LfheapSet* below. Highlight your changes by making them bold. You may omit methods you did not change.

```
/** @file FheapSet.h
 *
 *  @author Jon Turner
 *  @date 2011
 *  This is open source software licensed under the Apache 2.0 license.
 *  See http://www.apache.org/licenses/LICENSE-2.0 for details.
 */

#ifndef LFHEAPS_H
#define LFHEAPS_H

#include "stdinc.h"
#include "FheapSet.h"

namespace grafalgo {

typedef int fheap;
typedef bool (*delftyp)(index);

/** The FheapSet class represents a collection of Fibonacci heaps.
 *  The heaps are defined over nodes numbered 1..n where n is specified
 *  when the object is constructed. Each node is in one heap at a time.
 */
class LfheapSet : public FheapSet {
public:     LfheapSet(int=26, delftyp=NULL);
            ~LfheapSet();

   fheap    findmin(fheap);
   fheap    deletemin(fheap);

   string& heap2string(fheap,string&) const;
private:
        delftyp delf;           ///< pointer to "deleted function"
                                ///< used in lazy deletion

   string& heap2string(fheap,bool,string&) const;
   void    makeSpace(int);
   void    freeSpace();

   void    purge(fheap);
};


/** Find the item with smallest key in a heap.
```

```
 *  @param h is the canonical element of some heap
 *  @return the the index of item in h that has the smallest key
 */
inline fheap LfheapSet::findmin(fheap h) {
   purge(h); h = makeheap(*tmpq);
   tmpq->clear();
   if (h != 0) h = mergeRoots(h);
   return h;
}

/** @file LfheapSet.cpp
 *
 *  @author Jon Turner
 *  @date 2011
 *  This is open source software licensed under the Apache 2.0 license.
 *  See http://www.apache.org/licenses/LICENSE-2.0 for details.
 */
#include "LfheapSet.h"

namespace grafalgo {

#define sib(x) sibs->suc(x)
#define kee(x) node[x].kee
#define rank(x) node[x].rank
#define mark(x) node[x].mark
#define p(x) node[x].p
#define c(x) node[x].c

#define deleted(x)((delf != 0 && (*delf)(x)))

/** Constructor for LfheapSet class.
 *  @param size is the number of items in the constructed object
 *  @param delf1 is a pointer to the deleted function
 */
LfheapSet::LfheapSet(int size, delftyp delf1) : FheapSet(size) {
   delf = delf1;
}

/** Purge deleted nodes from the top of a heap or sub-heap.
 *  Traverse the top of a heap or sub-heap, removing all nodes visited
 *  from their sibling lists and adding non-deleted nodes to *tmpq
 *  @param h refers to some node in a heap
 */
void LfheapSet::purge(fheap h) {
   if (h == 0) return;
   while (true) {
       if (!deleted(h)) {
           tmpq->addLast(h);
       } else {
           fheap ch = c(h);
           if (ch != 0) purge(ch);
           c(h) = 0;
       }
       p(h) = 0; mark(h) = false;
       fheap sh = sib(h);
       if (sh == h) return;
       sibs->remove(h);
       h = sh;
   }
}

/** Remove the item with smallest key from a heap.
 *  @param h is the canonical element of some heap
 *  @return the canonical element of the heap that results from
 *  removing the item with the smallest key
 */
fheap LfheapSet::deletemin(fheap h) {
   assert(1 <= h && h <= n());

   purge(h);
   if (tmpq->empty()) return 0;
   h = makeheap(*tmpq);
   tmpq->clear();

   // now h is a node that is not deleted,
   // of course some of its children may be deleted
   fheap ch = c(h);
   if (ch != 0) purge(ch);
   if (!tmpq->empty()) ch = makeheap(*tmpq);
   tmpq->clear();
   c(h) = 0;

   // now, h in one siblist, its purged children in ch's sublist
   sibs->join(h,ch);
   fheap sh = sib(h);
   if (sh == h) return 0;
   sibs->remove(h);
   return mergeRoots(sh);
}
```

2. (10 points) Compile and run the program *testLfheapSet* that you will find in the *unit* subdirectory of the *heaps* directory. Paste a copy of the output below.

   ```
   all tests passed
   ```

   Note that *testLfheapSet* may report errors if your heaps do not have exactly the same form as the "expected" output specified in the unit test. This may not indicate a real error, as there are many equivalent ways to represent a given heap. When you see a reported difference, check it carefully to make sure your version is consistent with the expected output.

3. (10 points) Paste a copy of your code for *rrobinF* below. Highlight your changes by making them bold.

   ```
   #include "stdinc.h"
   #include "Dlist.h"
   #include "Partition.h"
   #include "LfheapSet.h"
   #include "Wgraph.h"

   #include "mstStats.h"

   using namespace grafalgo;

   Wgraph *gpf;
   Partition *ppf;

   // Return true if the endpoints of e are in same tree.
   bool delff(edge e) {
   ```

```
        return (*ppf).find((*gpf).left((e+1)/2)) ==
               (*ppf).find((*gpf).right((e+1)/2));
    }

    /** Find a minimum spanning tree of wg using the round robin algorithm.
     *  This version uses Fibonacci heaps instead of leftist heaps.
     *  @param wg is a weighted graph
     *  @param mstree is a list in which the edges of the mst are returned;
     *  it is assumed to be empty, initially
     */
    void rrobinF(Wgraph& wg, list<edge>& mstree, mstStats& stats) {
        int t0 = Util::getTime();
        edge e; vertex u,v,cu,cv;
        Dlist q(wg.n()); List elist(2*wg.m());
        fheap *h = new fheap[wg.n()+1];
        Partition prtn(wg.n());
        LfheapSet heapSet(2*wg.m(),delff);
        gpf = &wg; ppf = &prtn;
        for (e = 1; e <= wg.m(); e++) {
            heapSet.setKey(2*e,wg.weight(e));
            heapSet.setKey(2*e-1,wg.weight(e));
        }
        for (u = 1; u <= wg.n(); u++) {
            elist.clear();
            for (e = wg.firstAt(u); e != 0; e = wg.nextAt(u,e)) {
                elist.addLast(2*e - (u == wg.left(e)));
            }
            if (!elist.empty()) {
                h[u] = heapSet.makeheap(elist);
                q.addLast(u);
            }
        }
        int t1 = Util::getTime();
        stats.init = t1 - t0;
        while (q.get(2) != 0) {
            vertex q1 = q.first();
            h[q1] = heapSet.findmin(h[q1]);
            if (h[q1] == 0) { q.removeFirst(); continue; }
            e = (h[q1]+1)/2; mstree.push_back(e);
            u = wg.left(e); v = wg.right(e);
            cu = prtn.find(u); cv = prtn.find(v);
            q.remove(cu); q.remove(cv);
            h[prtn.link(cu,cv)] = heapSet.meld(h[cu],h[cv]);
            q.addLast(prtn.find(u));
        }
        stats.total = Util::getTime() - t0;
        stats.steps1 = heapSet.mrCount;
        stats.steps2 = prtn.findCount;
        delete [] h;
    }
```

4. (10 points) Compile the provided code in your *lab4* directory using the *makefile*. Verify your changes by typing

```
checkMst <bg10
(a,f,55) (e,f,23) (e,h,43) (e,i,55) (g,h,57) (b,g,24) (b,d,26) (c,d,45)
(b,j,49)
2 6 22 20
(a,f,55) (b,g,24) (c,d,45) (e,f,23) (e,h,43) (e,i,55) (b,j,49) (b,d,26)
(g,h,57)
```

```
7 11 49 123
(a,f,55) (b,g,24) (c,d,45) (e,f,23) (e,h,43) (e,i,55) (b,j,49) (b,d,26)
(g,h,57)
3 9 54 163

checkMst <bg20
(a,l,36) (c,l,12) (d,l,19) (c,k,20) (k,t,26) (c,j,27) (b,j,29) (b,q,11)
(c,s,31) (i,l,39) (h,i,21) (e,h,25) (h,p,38) (n,p,36) (g,n,30) (g,m,28)
(e,r,62) (f,q,69) (o,t,77)
3 12 44 86
(a,l,36) (b,q,11) (c,l,12) (d,l,19) (e,h,25) (f,q,69) (g,m,28) (h,i,21)
(c,j,27) (c,k,20) (g,n,30) (o,t,77) (n,p,36) (e,r,62) (q,s,31) (k,t,26)
(h,p,38) (b,j,29) (i,l,39)
22 34 191 382
(a,l,36) (b,q,11) (c,l,12) (d,l,19) (e,h,25) (f,q,69) (g,m,28) (h,i,21)
(c,j,27) (c,k,20) (g,n,30) (e,o,77) (n,p,36) (e,r,62) (c,s,31) (k,t,26)
(b,j,29) (h,p,38) (i,l,39)
8 28 160 467

evalMst 100 200
100 200 0.5us 22.9us 3135 9046 38.6us 75us 10669 19954 18.6us 84us 9611
27349
```

Part B. Evaluating performance for increasing n, fixed density.

1. (10 points) Run the provided *script1* and use the resulting data to complete the table below. Show the units. For each performance counter, compute the ratios of the values from one row to the next.

		init time		*total time*		*steps1*		*steps2*	
n	*m*	*time*	*ratio*	*time*	*ratio*	*count*	*ratio*	*count*	*ratio*
prim									
1K	2K	1.2us		262us		4.2K		16K	
4K	8K	4.7us	3.92	1.19ms	4.54	20K	4.67	78K	5.02
16K	32K	14.5us	3.09	6.08ms	5.11	91K	4.53	377K	4.82
64K	128K	52.6us	3.63	46ms	7.57	405K	4.47	1.8M	4.68
256K	512K	233us	4.43	270ms	5.87	1.8M	4.42	8.1M	4.58
1M	2M	907us	3.89	1.43sec	5.30	7.9M	4.39	36M	4.51
rrobin									
1K	2K	350us		740us		12K		22K	
4K	8K	1.48ms	4.23	3.2ms	4.32	48K	4.07	90K	4.02
16K	32K	7.4ms	5.00	16.5ms	5.16	200K	4.02	359K	4.00
64K	128K	59.9ms	8.09	128ms	7.76	790K	4.00	1.4M	4.00
256K	512K	284ms	4.74	654ms	5.11	3.1M	4.00	5.7M	4.00
1M	2M	1.2sec	4.37	3.0sec	4.56	12.5M	4.00	23M	4.00
rrobinF									
1K	2K	176us		888us		11K		31K	
4K	8K	842us	4.78	3.9ms	4.39	45K	4.08	127K	4.06
16K	32K	4.54ms	5.39	19.6ms	5.03	180K	4.03	510K	4.01
64K	128K	47.6ms	10.48	153ms	7.81	720K	4.00	2.0M	4.00
256K	512K	243ms	5.11	812ms	5.31	2.9M	4.00	8.1M	3.99
1M	2M	1.1sec	4.44	3.7s	4.54	11M	4.00	32M	3.99

2. (10 points) The round-robin algorithm is asymptotically faster than Prim's algorithm for sparse graphs. Does the data in the table demonstrate this? Explain how. Be specific. Consider both the runtime data and the data for the number of key steps.

   If we look at the growth rate for the key steps that are counted for the algorithms, we can see that for Prim's algorithm, the number of siftup and siftdown steps is growing a little faster than linearly with the number of verticies (we see ratios ranging from 4.39 to 5.02), while for both of the round robin algorithms, the ratios for the number of key steps in those algorithms are much closer to 4.0, indicating linear growth, since the number of vertices is going up by a factor of 4 at each step.

   For the running time, the cache effects make it harder to see, but if we take the ratio of the total runtimes for the largest and smallest graphs, we get 1430/.262=5458 for Prim's algorithm, 3000/.74=4054 for rrobin and 3700/.888=4166 for rrobinF, so it appears that the runtime for Prim is increasing faster than for the two round-robin algorithms. Still, the smaller constant factor for Prim *make it more than twice as fast when we have 1 million vertices and it seems likely to retain its advantage even for graphs with substantially more vertices.*

3. (5 points) For Prim's algorithm, the *steps2* counter is the value of *siftdownCount*. For a graph on *n* vertices, what you expect the final value of *siftdownCount* to be? Is the data consistent with this expectation? To answer this, you'll need to know the value of *d* and the depth of the heap.

   In this case, d=4, so the heap depth is $\log_4 n$ *and the final value of siftdownCount should be about* $4n \log_4 n$*. For n=1024, this is 20n=20K and for n=1M it is about 40n=40M. The values in the table are 16K and 36M, so they are entirely consistent with the expected value.*

4. (5 points) For the round robin algorithm (with leftist heaps), the *steps1* counter keeps track of the total number of recursive calls to the *meld* method. Based on the data in the table, about how many calls to meld are done per edge in the graph. Does this number make sense to you? Try to explain the observed value based on your understanding of how the algorithm works. Be as specific as you can.

   The number of meld steps is about 6 per edge. During the initialization, heaps are built on the edges incident to each vertex. We would expect about one meld step per heap item, so 2 meld steps per edge during the initialization. During the remainder of the algorithm, an edge is involved in a meld step if it appears on the right path of two heaps being melded. To get a good estimate of this, we would need to know how many melds are being done and the length of the right path. As an example, if there are 4n melds with an average findpath length of lg n, we would have 4n lg n = 2m lg n meld steps, or (lg n)/2 per edge. That would mean 5 when n=1K and 10 when n=1M. These are both a bit higher than the 4 that seems to be implied by the data, but given the limits of our analysis, the data appears reasonable.

5. (5 points) For *rrobin*, the s*teps2* counter keeps track of the total number of *find* steps done by the *partition* data structure. Based on the data in the table, about how many find steps are done per iteration of the main loop in *rrobin*. Does this number make sense to you? Try to explain the observed value based on your understanding of how the algorithm works. Be as specific as you can.

   The number of iterations is n–1, so the number of find steps is about 22 per iteration. Each iteration includes 2 calls to find(). Now, since findmin() calls the deleted() method, it accounts for 2 more calls to find() for every heap item visited. If we estimate an average of 5 calls to deleted(), each time findmin() is called, we get 10 more calls to find per iteration. If the find() calls have an average of 2 find steps, we get 24 find steps per iteration, which is close to the observed value. At first, these values seem surprisingly small, but note that during the first pass through the queue, each call to findmin() result in just one call to deleted() (because each heap is being touched for the first time during this first pass, and has no deleted nodes) and at least one of the two resulting find() calls is to a singleton set, so it involves just one find step. Since the first pass accounts for more than half of the iterations, this has a big effect on the average number of find() calls.

6. (5 points) For the *rrobinF*, the s*teps1* counter keeps track of the number of iterations done by the *mergeRoots*() method. Based on the data in the table, about how many *mergeRoot* steps are done per iteration of the main loop. Does this number make sense to you? Try to explain the observed value based on your understanding of how the algorithm works. Be as specific as you can.

   The number of mergeRoots() steps is 11 per loop iteration. Since m=2n, the average vertex degree is 4, so during the first pass a typical heap selected for a findmin() will have four nodes, each of which is a single node tree. For such a heap, we do 7 mergeRoot steps. So, we expect this number for each *iteration in the first pass through the queue. In later passes, the number of mergeRoot steps will increase, as the number of deleted nodes within the heaps grows. To produce 11 per loop iteration, the average purge() operation must return a list of at most 11 subtrees of the heap. Since we would expect at least a few collisions when doing the merging, it seems likely that the average length of the list returned by purge() is in the range of 5–10.*

7. (5 points) For the *rrobinF*, the s*teps2* counter keeps track of the total number of *find* steps done by the *partition* data structure. Based on the data in the table, about how many find steps are done per iteration of the main loop. Does this number make sense to you? Try to explain the observed value based on your understanding of how the algorithm works. Be as specific as you can. Compare the value in this case to the value for the original *rrobin* algorithm. Can you account for the differences?

   The number of find steps is about 32 per iteration in this case. This is a little bit larger than we saw for rrobin, but the difference makes sense. With Fibonacci heaps, each call to findmin() must call deleted() for every tree root in the heap. Also, whenever it encounters a deleted node, it must call deleted() for all the children of that node. Since Fheaps can have several children per node, this will generally lead to more calls to deleted() than we would have with leftist heaps. If we reconsider the example in the answer to part 5, we see that the data is consistent with about 7 calls to deleted() per loop iteration which gives 14 calls to find(), or 16 altogether. If the average number of find steps per find() is 2, we get 32 find steps per iteration.

8. (5 points) Examine the time spent by each of the three algorithms on initialization. Compare it to the total time spent by the algorithm and discuss the differences among the algorithms. How much of a bearing does the initialization time have on the overall running time?

   Prim's algorithm spends very little time on initialization, since all it has to do is insert the edges incident to a single vertex into its heap. For the round robin algorithms, initialization is a substantial fraction of the total (about 40% for rrobin and 30% for rrobinF), since in this case every vertex has a heap of edges incident to it, and these must all be initialized. It's interesting to see that rrobinF spends less time in absolute terms than rrobin, presumably because the makeheap in Fheaps is less expensive than the heapify operation used in leftist heaps.

Part C**. **Evaluating performance for increasing n, m=$n^{3/2}$.

1. (10 points) Run the provided *script2* and use the resulting data to complete the table below. Show the units. For each performance counter, compute the ratios of the values from one row to the next.

		init time		*total time*		*steps1*		*steps2*	
n	*m*	*time*	*ratio*	*time*	*ratio*	*count*	*ratio*	*count*	*ratio*
prim									
64	512	1us		29us		256		932	
256	4K	2us	2.29	218us	7.52	1.3K	4.91	8K	8.54
1K	32K	4us	2.69	2.2ms	10.23	5.7K	4.55	64K	8.06
4K	256K	23us	5.40	44ms	19.60	26K	4.48	516K	8.05
16K	2M	87us	3.76	717ms	16.41	113K	4.43	4.1M	8.04
rrobin									
64	512	98us		132us		3K		1.6K	
256	4K	701us	7.15	860us	6.52	24K	8.02	6.7K	4.30
1K	32K	6.4ms	9.12	7.2ms	8.37	184K	7.68	272K	4.04
4K	256K	146ms	22.85	155ms	21.53	1.4M	7.77	107K	3.94
16K	2M	1.5sec	10.34	1.6sec	10.13	11M	7.85	428K	3.99
rrobinF									
64	512	39us		164us		2.4K		4.6K	
256	4K	350us	8.97	1.3ms	7.68	19K	7.96	32K	7.05
1K	32K	3.5ms	10.06	11ms	8.57	147K	7.62	223K	6.91
4K	256K	142ms	40.34	250ms	23.15	1.1M	7.65	1.6M	7.12
16K	2M	1.4sec	9.65	2.6sec	10.44	8.8M	7.77	11M	7.47

2. (10 points) Prim's algorithm is asymptotically faster than the round-robin algorithm for dense graphs. Does the data in the table demonstrate this? Be specific. Consider both the runtime data and the data for the number of key steps.

   If we look at the growth rate for the key steps that are counted for the algorithms, we can see that for Prim's algorithm, the number of siftdown steps is growing just slightly faster than linearly with the number of edges (we see ratios close to 8), which is exactly what we expect of Prim for dense graphs. On the other hand, for both versions of the round robin algorithm, the number of steps is growing more slowly than the number of edges, where we expect it to grow faster than the number of edges, due to the lg lg n term. So, for random graphs, the round-robin algorithms appear to be performing better than the worst-case bound, while Prim's performance is consistent with the worst-case bound.

   For the running time, the cache effects distort the results, but if we take the ratio of the total runtimes for the largest and smallest graphs, we get 717/.029=24.7K for Prim's algorithm, 1600/.132=12.1K for rrobin and 2600/.164=15.8K for rrobinF, so it appears that the runtime for Prim is increasing faster than for the two round-robin algorithms. Still, the smaller constant factor for Prim makes it more than twice as fast when we have 2 million edges.

3. (5 points) For the round robin algorithm, the number of find steps grows more slowly than the worst-case analysis would suggest. What would you expect based on the worst-case analysis? How does this compare to the actual data? Explain the difference as best you can.

   The analysis suggests that the number of find steps grows in proportion to m. So, we would expect ratios of a factor of 8 in this case. In reality, we see ratios of about 4. This is consistent with the find steps growing in proportion to the number of iterations. The number of find steps per iteration is about 26, and this is consistent across the entire range. This is a little more than the 22 we saw for sparse graphs, but is fairly close. The worst-case analysis of the number of find steps is based on the assumption that most edges are eventually removed from heaps during findmins. At least for the random graphs used here, it appears that most edges never get removed from the heaps, and this seems like it should be the typical case, since we only need to remove the deleted nodes that appear near the top of the heap.

4. (10 points) Notice that the initialization time for *rrobin* (using leftist heaps) is a large fraction of the total. Try to explain why this is. Is it because the initialization time is surprisingly large or because the time for the main loop is surprising small? You may find it useful to run additional experiments to better separate the contributions of the two parts.

   The initialization takes $\Omega(m)$ time. While the main loop takes $\Omega(m \lg \lg n)$ time in the worst-case, it takes less time than that on random graphs. The data for the number of findsteps is one indication of this. Also, if we subtract the initialization time from the total time for the largest and smallest graphs we get 60 ms and 33 μs, respectively. The ratio of these two is about 2000, so the running time for the main loop has increased by a factor of 2000, even though the number of edges grew by a factor of 4000. And we would normally expect a larger growth factor in the runtime, due to deteriorating cache performance for large graphs.

   So what accounts for the faster runtime of the main loop? The other big factor in the running time is the number of meld steps, but this is trickier to evaluate, since some of the meld steps happen during the initialization. Indeed, if you count these separately, it turns out that for the largest graphs, about 95% of the meld steps happen during the initialization.

   There are two factors that might explain this. One is that during the first pass through the queue, no nodes are deleted from heaps, so the heapify times are really small. This first pass lasts at least n/2 steps but could last longer than this (maybe even for n–1 steps). The longer the first pass lasts, the less time is spent in findmin(). The other factor is that the analysis assumes that most edges are eventually removed from their heap, but for dense random graphs, the number of edges that get deleted from the selected heap in most iterations is quite small. We get the largest number of deletions in the very last iteration. In this case, roughly half of the edges connect vertices in the same tree, so when purge() considers a node in the heap, about half the time, that node will be one that should be deleted from the heap. The binary tree structure of the heap ensures that even in this case, the number of nodes that are deleted is relatively small and consequently, the time for the heapify is also small.