A Pragmatic Implemention of Monotone Priority Queues

Arne Andersson Mikkel Thorup

1 Introduction

Recently there have been several theoretical improvements in the area of sorting, priority queues, and searching [1, 2, 8, 9]. All these improvements use indirect addressing to surpass the comparison-based lower bounds. Inspired by these advances, and by the fact that algorithms based on indirect addressing have proven to be efficient in many practical applications, we have implemented a trie-based priority queue as part of the DIMACS implementation challenge. Having Dijkstra's single source shortest path algorithm in mind, we decided to restrict our attention to *monotone* priority queues, as defined in [9]. A monotone priority queue, is a priority queue where the minimum is non-decreasing – the minimum of an empty monotone priority queue is defined to be 0. The monotonicity condition is not a problem for greedy algorithms such as Dijkstra's single source shortest paths algorithm. Also, monotonicity is satisfied in event-simulations.

According to the specification of the DIMACS challenge, the data type is double precision floating point numbers. Our implementation was not tuned towards any of the problem instances provided by the challenge, but it turned out to run substantially faster than the provided binary heap in most cases. Since the heap runs fast in practice compared to other comparison-based data structures, we beleive that our implementation will compare favourably against all known comparison-based methods. It seems that indirect addressing is of practical relevance in connection with monotone priority queus.

Our priority queue implementation has some similarities with a recent implementation by Cherkassky, Goldberg, and Silverstein [5]. Their work is done independently of ours and we will return to the specific differences in the technical presentation of our data structure. According to experiment reported by Cherkassky et.al., their implementation runs generally slower than a standard heap. On the other hand, our implementation is substantially faster on many of the same problem instances (Cherkassky et.al. provided the problem instance generator used in this DIMACS challenge).

Below, we give a simplified presentation of our implementation. Afterwards comes the results of running the DIMACS challenge tests. Finally, the full C-code is included in an appendix.

2 Data structure

Our keys are possitive double floats, occuping 64 bits and stored in two consecutive 32-bit words. We view each key x as a string of 4 16-bit characters (short unsigned integers), referred to as $x[0]$, $x[1]$, $x[2]$, and $x[3]$.

Our data structure has three main components.

Multi-level bucket structure. The main data structure is a "lazy" trie of degree 2^{16} where only the leftmost path is fully (or partially) expanded. In this way, we only explicitly represent at most four trie nodes. This representation gives a multi-level bucket structure similar to the one described in [6]. This structure, denoted $\mathcal{B}$, has 4 levels and is implemented as a (4×2^{16}) array B of buckets.

Finger search within each bucket level. In order to make fast neighbour queries within the trie nodes, each bucket level – corresponding to an expanded trie node – is associated with a finger search structure. The finger search structure is in turn represented as a bit-mapped trie of degree 32 (= word length) over the interval $[0 \dots 2^{16} - 1]$; the root has degree 2 followed by three levels of degree 32.

A sorted list. In order to deal efficiently with small sub-problems, we combine $\mathcal{B}$ with a single sorted list L, which is a quite efficient priority queue for a small number of keys. The idea of switching to linked lists for small sub-problems is taken from experiences in implementing sorting algorithms [4, 7]. The main point with the sorted list is to avoid expanding trie nodes that only contains a few elements.

An important feature of our particular implementation is that once an element has been stored in L, it will remain there until it is deleted from the priority queue. As a consequence, L may become very long. However, only a small part of the list, the *head* of L, will be used to avoid expanding trie nodes, the rest of L will be merged with the contents in the bucket structure as the priority queue operations proceed.

We store both the buckets and the sorted lists as doubly linked lists. In this way, it is easy to pick out a key from the priority queue and reinsert it; this will be used during decrease-key operations. (The doubly linked list is a clear disadvantage if no decrease-key operations are made. If we do not need to pick out keys, singly linked lists would have sufficed, and then we would only have half as many pointers to manipulate.)

Below, we describe the data structure in more detail.

2.1 Combining the bucket structure and the list

Each key in our priority queue is either in the bucket structure $\mathcal{B}$ or in the sorted list L, but not in both. The current smallest key, denoted μ, is always in the list L.

For any key x, the *split-level of* x is the smallest i such that $x[i] \neq \mu[i]$. If x has spit-level i, logically, it belongs in bucket $B[i, x[i]]$. It may, however, alternatively be in L.

Note that the split-level of x and hence its place in B dependends on the smallest key μ. Also note that the keys in $B[i_1, j_1]$ are smaller than those in $B[i_2, j_2]$ if and only if (i_1, j_1) is lexicographically smaller than (i_2, j_2), i.e. $i_1 < i_2$ or $i_1 = i_2$ and $j_1 < j_2$. In this case, for brevity, we will say that *bucket* $B[i_1, j_1]$ *is smaller than bucket* $B[i_2, j_2]$. Thus, the smallest element in $\mathcal{B}$ is the smallest element in the smallest non-empty bucket. If $B[i, j]$ is the smallest non-empty bucket, then we call i the *depth* of $\mathcal{B}$; if there are no keys in the $\mathcal{B}$, the depth is -1. In contrast to Cherkassky et.al. [5] we do *not* require that the the split-level of keys in L is stricly bigger than the depth of $\mathcal{B}$. This means that we can decrease the depth of $\mathcal{B}$ without having to move any keys from L to $\mathcal{B}$. Indeed, once a key occurs in L it will never be moved to $\mathcal{B}$.

2.2 Finger search among 16-bit keys

When the smallest bucket on a level is emptied, we must find the next non-empty bucket on the same level.

We treat this problem by a finger search structure. The reason for doing finger search is that we wish to take advantage of the cases when the next minimum 16-bit number is relatively close to the deleted minimum. The finger search structure is an oblivious trie of depth 4 over the interval $[0 \ldots 2^{16} - 1]$. The root of this trie has $2^1 = 2$ children, while all descendent nodes have $2^5 = 32$ children, giving the desired total of 2^{16} leaves. For a node v with 32 children, we have a one-word bit-map m_v, that for each child tells if it is non-empty. Thus, v has a non-empty child if and only if $m_v > 0$.

Adding and removing edges to this trie is simply done by bit flipping. When the smallest key in the set is deleted, we follow the leftmost path in the trie bottom-up until we find a node with an outgoing edge. The bottom-up traversal corresponds to a finger search. When a node v with an outgoing edge is found, the position of the edge is located by finding the leftmost 1-bit in m_v. To find this bit, we just need to convert m_v to a float and then extract the exponent; on some computers there is even a special instruction for finding the leftmost 1-bit in a word.

3 Implementing priority queue operations

Having described the data structure, we are ready to describe how the different operations are performed. Let the *head* of L be the set of elements in L whoose keys match the minkey down to the current depth of $\mathcal{B}$ and let the **tail** be the rest of L.

find-min. Returns μ.

insert(x). First, we find the split level i of x. If i is smaller than or equal to the depth of $\mathcal{B}$, we insert x into bucket $B[i, x[i]]$. Otherwise, we try to insert x into the head of the sorted list L. However, if the latter is to expensive in the sense that L has ≥ 35 elements smaller than x, the head of L has become very large. We then increase the depth of $\mathcal{B}$ to i and insert x in $B[i, x[i]]$. After this, the head of L will contain at most 35 elements, in many cases it will contain cosiderably less elements. Here, 35 is a constant identified experimentally. Any number between 20 and 50 seemed about equally good.

delete-min. Our main problem is to maintain the invariant that the sorted list L should always contain the minimum element μ (the head of L should be non-empty). Let μ' be the second element in the sorted list, and let i be the split level of μ' relative to μ. Suppose $B[i, x[i]]$ is strictly smaller than any non-empty bucket in $\mathcal{B}$. Then μ' is strictly smaller than all keys in $\mathcal{B}$, so we can just delete μ from L, turning μ' into the new smallest element. Note that this change in minimum does not change the split level of any of the keys in $\mathcal{B}$.

If $B[i, x[i]]$ is not strictly smaller than any non-empty bucket in $\mathcal{B}$, we first find the minimum key μ'' in the minimum bucket in $\mathcal{B}$. If $\mu'' < \mu'$, we remove μ'' from $\mathcal{B}$ and insert it in L. Now L contains the second smallest key, and we can remove μ from L. The problem now is that (except if μ'' was a singleton $< \mu'$) the keys in the smallest non-empty bucket are misplaced in the sense that their split level is increased by one relatively to the new minimum. They are then all reinserted one by the insertion procedure; we say that the bucket is *expanded*.

It should be noted that there is a difference between inserting a new key and reinserting a key during a bucket expansion. In the second case, we know that the first i characters of the reinserted keys match the minkey. By letting i be a parameter to the insertion procedure, we can take advantage of this when finding the split-level of the inserted keys. (When inserting new keys, this parameter is set to -1.)

decrease-key(x, key). x is either stored in L or $\mathcal{B}$; in both cases it is stored in some doubly-linked list. We delete x from its list. Then, we simply insert x again into the priority queue with its new value *key*.

4 Performance considerations

It turns out (from profiling) that the main cost of our priority queue is from taking keys in and out of buckets. If we ignore decrease-key operations, the number of times that a key in and out of buckets is bounded by the depth of $\mathcal{B}$ which is at most 4. However, if our data is uniformly distributed, and if we are dealing with $< 2^{32}$ keys, we expect the depth of $\mathcal{B}$ to be at most 2. Thus, our bucket structure is immediately taking advantage of any randomness in the distribution. The bucket structure is also taking advantage of cases where inserted keys are clustered around the current minimum. Then, the split level of the inserted element is high, meaning that this element will not be involved in much rebucketing.

In connection with decrease-key, it should be noted that a decrease-key cannot decrease the split level of a key. Hence a decrease-key gives rise to at most one extra rebucketing. In fact, a decrease-key may save us some rebucketting if it brings the key so close to the current minimum that the split-level of the decreased key increases by more than one.

While we take advantage of large decrease-keys that bring the key close to the current minkey, the heap has an opposite behaviour. If a decrease-key does not change the value of a key by much, the number of levels the key is bubbled up towards the root will be few; typically zero. More precisely, if the rank of a key is less than halved, on the average, it will not decrease below the value of its parent, and hence no structural changes are needed. This contrast between the two data structures is evident in the experimental results.

It should be noted that our priority queue could easily be extended to perform well on small decreases: divide the input randomly into atomic sets of, say, size 10, letting only the smallest key participate in the above described priority queue. Then it would only be when a key became the smaller of the 10 that we would need to affect our priority queue. For each of these atomic sets, we maintain the minimum. Insert and decrease-key just leads to a comparison with the minimum, and possibly an update, while if the smallest key is deleted, we need to scan all 10 keys in the set.

5 The results

The experiment were run on an Ultra-SPARC under SunOS. An exception is the "sp_bad" files, these were very small and in order to get measurable times, they were run on a Sun SPARCstation 5.

The given DIMACS test files were used as input. In order to avoid counting the cost of the DIMACS interface, we implemented a dummy priority queue with empty implementations of the different operations. The times quoted below are the total running times subtracted with the running time on the same data with the dummy priority queue.

Each of the running times are the average over 3 runs. Below "heap" stands for the priority queue provided by the DIMACS challenge while "us" stands for our priority queue implementation. For the description of our test data, we refer to the DIMACS challenge.

input file	heap	us	us/heap (%)
dimacs_rand_4.1048576.1	2985	1081	36
dimacs_rand_4.1048576.2	3006	984	33
dimacs_rand_4.1048576.3	2983	1185	40
dimacs_rand_4.1048576.4	2749	1362	50
dimacs_rand_4.1048576.5	3034	1374	45
dimacs_rand_4.262144.1	582	226	39
dimacs_rand_4.262144.2	584	212	36
dimacs_rand_4.262144.3	579	214	37
dimacs_rand_4.262144.4	696	233	33
dimacs_rand_4.262144.5	613	191	31
dimacs_rand_4.524288.1	1381	446	32
dimacs_rand_4.524288.2	1266	528	42
dimacs_rand_4.524288.3	1294	430	33
dimacs_rand_4.524288.4	1254	422	34
dimacs_rand_4.524288.5	1205	376	31

input file	heap	us	us/heap (%)
dimacs_grid_slong.16384x16.1	148	103	70
dimacs_grid_slong.16384x16.2	200	160	80
dimacs_grid_slong.16384x16.3	206	162	79
dimacs_grid_slong.16384x16.4	191	144	75
dimacs_grid_slong.16384x16.5	207	162	78
dimacs_grid_slong.32768x16.1	416	326	78
dimacs_grid_slong.32768x16.2	416	332	80
dimacs_grid_slong.32768x16.3	416	326	78
dimacs_grid_slong.32768x16.4	438	328	75
dimacs_grid_slong.32768x16.5	413	336	81
dimacs_grid_slong.8192x16.1	104	69	66
dimacs_grid_slong.8192x16.2	103	69	67
dimacs_grid_slong.8192x16.3	99	63	64
dimacs_grid_slong.8192x16.4	98	64	65
dimacs_grid_slong.8192x16.5	100	65	65

input file	heap	us	us/heap (%)
spbad.1024.1	4	24	600
spbad.512.1	3	16	533
spbad.512.2	3	16	533
spbad.512.3	3	17	567
spbad.512.4	3	17	567
spbad.512.5	3	16	533
spbad1024.2	5	20	400
spbad1024.3	4	20	500
spbad1024.4	6	21	350
spbad1024.5	8	21	263
spbad2048.1	8	34	425
spbad2048.2	9	35	389
spbad2048.3	9	34	378
spbad2048.4	8	33	413
spbad2048.5	9	36	400

input file	heap	us	us/heap (%)
evsimgen 99, 100, 1, 500000	193	133	69
evsimgen 99999, 100, 1, 500000	188	146	78
evsimgen 999999999, 100, 1, 500000	197	153	78
evsimgen 99, 50000, 1, 500000	296	157	53
evsimgen 99999, 50000, 1, 500000	295	163	55
evsimgen 999999999, 50000, 1, 500000	297	154	52
evsimgen 99, 100, 33, 500000	176	140	80
evsimgen 99999, 100, 33333, 500000	177	175	99
evsimgen 999999999, 100, 333333333, 500000	209	194	93
evsimgen 99, 50000, 33, 500000	334	174	52
evsimgen 99999, 50000, 33333, 500000	472	242	51
evsimgen 999999999, 50000, 333333333, 500000	467	317	68

input file	heap	us	us/heap (%)
mongen 99, 100, 1, 500000	152	134	88
mongen 99, 100, 5, 500000	90	128	142
mongen 99, 100, 10, 500000	76	136	179
mongen 999999999, 50000, 1, 500000	343	213	62
mongen 999999999, 500, 100, 500000	66	99	150
mongen 999999999, 5, 10000, 500000	61	110	180

6 Interpreting the results

The results seem easy to interpret. Our priority queue performs very well on all input except the "sp_bad" and "mongen" data. However, these input files contain mainly decrease-key operations where the values are only decreased very little. This is exactly the situation where we expect the heap to be very fast; an element whose key is decreased just slightly will often remain at the same position in the heap. (On some of the "mongen" data we perform well, this is because the number of decrease-key operations are relatively few for these files). Also, for "sp_bad", the number n of keys is small. While the heap takes direct advantage of a small n (operation cost $O(\log n)$), our priority queue has a large initialization costs which is independent of n.

We beleive that in many practical applications, the decrease-key operations will move the keys over large distances and that decreased keys will be concentrated close to the minimu key, in which case our implementation will be a good choice. Indeed, our implementation is very successful on all the standard Dijkstra tests (dimacs_rand and dimacs_grid_slong).

References

[1] A. Andersson, Sublogarithmic Searching Without Multiplications, In *Proc. 36th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 655–663, 1996.

[2] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? In *Proc. 27th ACM Symposium on Theory of Computing (STOC)*, pages 427–436, 1995.

[3] A. Andersson, P.B. Miltersen, and M. Thorup. Fusion trees can be implemented with AC^0 instructions only. BRICS-TR-96-30, Aarhus, 1996.

[4] J. L. Bentley and M. D. McIlroy. Engineering a sort function. *Software–Practice and Experience*, 23(11):1249–1265, 1993.

[5] B.V. Cherkassky, A.V. Goldberg, and C. Silverstein, Buckets, Heaps, Lists, and Monotone Priority Queues, In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms*, pages 83–92, 1997.

[6] D.V. Denado and B.L. Fox, Shortest-Route Methods: 1. Reaching, Pruning, and Buckets. *Oper. Res.*, 27:161–186, 1979.

[7] P. M. McIlroy, K. Bostic, and M. D. McIlroy. Engineering radix sort. *Computing Systems*, 6(1):5–27, 1993.

[8] R. Raman. Priority queues: small monotone, and trans-dichotomous. *Proc. ESA'96, LNCS 1136*, 1996, 121–137.

[9] M. Thorup. On RAM priority queues. In *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 59–67, 1996.

A The program

A.1 Some external type specifications

```
#define NULL_ITEM 0
/* null value for item locator*/
#include "pq_instantiate.h"
/* contains pr_type and in_type declarations */

typedef long  *it_type;
/* PUBLIC: type locates item via lookup  */
typedef long  *pq_ptr;
/* PUBLIC: type points to prioroty queue */
```

A.2 The monotone priority queue implementation

```
#include "pq_types.h"

/* Included for profiling
#include <prof.h>                    */

/* Standard things                           */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>

/* To test or not to test the user */
#define usercheck 0

/* To test or not to test ourselves */
#define safety 0

/* Often changed parameters                 */
#define bottommax       35

/*hardware assumptions*/
#define w               32
#define realmax (1.78e308)

typedef unsigned short int ourchar;
#define keymax           3
#define bucketmax      65535


/* some standard defs                              */
#define true            1
#define false           0
#define NULL            0
#define boolean       int

#define ERROR(msg) {printf("%s","ERROR: " #msg); \
printf("\n"); exit(1);}

/* Insert a list into another list. The list to be
inserted starts at pointer fi and ends at pointer
la. The list is inserted before the node pointed
to by p. Can of course also be used to insert
just one node. Then fi=la. */
#define INS_BEFORE(fi, la, p) { la->nxt = p;\
 fi->pre = p->pre;  \
 p->pre = la; fi->pre->nxt = fi; }

/* Remove a node from a list.   */
#define OUT(p)  p->nxt->pre = p->pre; \
p->pre->nxt = p->nxt;

#define START_BUCKET(bucket, thenew) { \
bucket->pre = thenew; \
bucket->nxt = thenew; \
thenew->nxt = bucket; \
thenew->pre = bucket;}

/* Macros for the bit map telling which
buckets are used */
#define CLR(i,j) map_unset_bit(used[i],j);
#define MRK(i,j) map_set_bit(used[i],j);
#define USED(i,j) map_bit(used[i],j)

/* comparisons                   */
#define GREATER(x,y) x.real>y.real
#define GEQ(x,y) x.real>=y.real
#define LESS(x,y) x.real<y.real
#define LEQ(x,y) x.real<=y.real


/* Type defs. The "external" type of pointer
(our list pointer)  is called "usrpt" */
typedef union {
   ourchar chars[keymax + 1];
   double real;
   } keytype;
typedef struct list {
  keytype key;
  struct list *pre, *nxt;
  in_type data;
} list;
typedef list *usrpt;

typedef struct {
  unsigned int signbit : 1;
  unsigned int exponentbias1023 : 11;
/*  unsigned int mantissa : 52; */
} doublemask;

typedef union {
   double as_double;
   doublemask as_mask;
   } doubleandmask;

int expdouble(double d)
{return ((((doublemask*) &d)->exponentbias1023)-1023);}


/* Takes a 32 bit word and finds the leftmost bit */
long unsigned int word_left_bit(long unsigned int x)
{ doubleandmask fm;
  fm.as_double=(double) x;
  return (1054-fm.as_mask.exponentbias1023);
}
```

```
static long unsigned int bit[32] = {
 0x1 <<31, 0x1 <<30, 0x1 <<29, 0x1 <<28, 0x1 <<27,
 0x1 <<26, 0x1 <<25, 0x1 <<24, 0x1 <<23, 0x1 <<22,
 0x1 <<21, 0x1 <<20, 0x1 <<19, 0x1 <<18, 0x1 <<17,
 0x1 <<16, 0x1 <<15, 0x1 <<14, 0x1 <<13, 0x1 <<12,
 0x1 <<11, 0x1 <<10, 0x1 << 9, 0x1 << 8, 0x1 << 7,
 0x1 << 6, 0x1 << 5, 0x1 << 4, 0x1 << 3, 0x1 << 2,
 0x1 << 1, 0x1 << 0 };

#define word_set_bit(x,i) x|=bit[i]
#define word_unset_bit(x,i) x&=~bit[i]
#define word_get_bit(x,i) ((x&bit[i])>0)

typedef struct {
  long unsigned int level1[2];
  long unsigned int level2[64];
  long unsigned int level3[2048];
  unsigned char level4[65537];
} Searchmap;

Searchmap *map_alloc()
{  Searchmap *m;
   m=calloc(1,sizeof(Searchmap));
   if (!m) {printf("not enough memory"); exit(1);}
   m->level4[65536]=1;
   return m;
 }


#define map_bit(m,i) (m->level4[i])

/* Find first value >=i (i<=65536) assume that
there are no smaller values */
unsigned int map_min(Searchmap *m, unsigned int i)
{
#if safety
   if (i>65536) {
     printf("%i is too  large to finger from\n", i);
     exit(1);}
#endif
   if (m->level4[i]>0) return i;
   i=i>>5;
   if (m->level3[i]==0) {
     i=i>>5;
     if (m->level2[i]==0) {
       i=i>>5;
       while (m->level1[i]==0) {
         i++;
         if (i>1) return 65536;
       }
       i=word_left_bit(m->level1[i])+(i<<5);
     }
     i=word_left_bit(m->level2[i])+(i<<5);
   }
   i=word_left_bit(m->level3[i])+(i<<5);
   return i;
}

void map_set_bit(Searchmap *m, unsigned int i)
 { unsigned int j;
#if safety
   if (i>=65536) {
     printf("%i is to large to set\n", i);
     exit(1);}
#endif
   if (m->level4[i]>0) return;
   m->level4[i]=1;
   j=i&0x1F; i=i>>5;
   if (m->level3[i] > 0) {
     word_set_bit(m->level3[i],j);
     return;}
   word_set_bit(m->level3[i],j);
   j=i&0x1F; i=i>>5;
   if (m->level2[i] > 0) {
     word_set_bit(m->level2[i],j);
     return;}
   word_set_bit(m->level2[i],j);
   j=i&0x1F; i=i>>5;
   word_set_bit(m->level1[i],j);
 }

void map_unset_bit(Searchmap *m, unsigned int i)
 { int j;

#if safety
   if (i>=65536) {
     printf("%i is to large to unset\n", i);
     exit(1);}
#endif
   if (m->level4[i]==0) return;
   m->level4[i]=0;
   j=i&0x1F; i=i>>5;
   word_unset_bit(m->level3[i],j);
   if (m->level3[i]>0) return;
   j=i&0x1F; i=i>>5;
   word_unset_bit(m->level2[i],j);
   if (m->level2[i]>0) return;
   j=i&0x1F; i=i>>5;
   word_unset_bit(m->level1[i],j);
 }

/* This should be put in one struct for each
   PQ instance */
Searchmap *used[keymax + 1];
list buckets[keymax + 1][bucketmax+1];
long PQsize, depth;
usrpt  nextfree, bottom, sentinel;
keytype triekey, minkey;
long n,r;
boolean mindeleted;


/************ Initialization ************/

usrpt usrpt_alloc()
{ usrpt freept;
  if (r) {
    nextfree++;
    r--;
    return nextfree;
  }
  freept=(usrpt) bottom->pre;
  if (freept!=sentinel) {
```

```
    bottom->pre=freept->pre;
    freept->pre->nxt=bottom;
    return freept;
  }
  printf("You are trying to allocate more than");
  printf(" requested %ld elements.",n);
  exit(1);
}

void Init()
{
  long i;

  mindeleted = true;
  nextfree=(usrpt) malloc((n+10)*sizeof(list));
  if (!nextfree) {
     printf("not enough memory for list elements");
     exit(1);}
  nextfree--;
  r=n+8;
  PQsize = 0;
  depth = -1;
  for (i = 0; i <= keymax; i++)
    used[i]=map_alloc();
  bottom = usrpt_alloc();
  minkey.real = 0.0;
  bottom->key = minkey;
  sentinel = usrpt_alloc();
  sentinel->key.real = realmax;
  bottom->nxt = sentinel;
  bottom->pre = sentinel;
  sentinel->nxt = bottom;
  sentinel->pre = bottom;
}


/************ Updating ***********************/

/* Search along leftmost path. It is assumed that
   this path is contained in minkey.
   We compare up to and including endpos.            */
long FirstDiff(x, startpos, endpos)
keytype x;
long startpos, endpos;
{ long i;
  i = startpos;
  while (x.chars[i] == minkey.chars[i] && i <= endpos)
    i++;
  return i;
}


/* Insert a new key, stored in a list node.
   The list node thenew need not to be cleared,
   the next and pre  pointers can be anything.
   newkey contains the key in thenew, passed
   as paameter for optimization reasons. from
   tells the depth where we should start
   scanning the trie path. For a new key, from
   is 0,when a key is reinserted during bucket
   expansion or decrease-key, from may be
   larger.
   The variable locdepth is just a copy of the
   global variable depth, used for optimization
   reasons. */
void Insert(thenew,newkey,from)
usrpt thenew; keytype newkey; int short from;
{ usrpt tmp; long i,d;
  ourchar newchar;
  long locdepth;

      locdepth = depth;

      /* Search along leftmost trie path    */
      d=FirstDiff(newkey, from, locdepth);

      /* If not too deep, insert in a bucket   */
      if (d <= locdepth) {
        newchar=newkey.chars[d];
        tmp = &buckets[d][newchar];
        if (USED(d,newchar)) {
          INS_BEFORE(thenew, thenew, tmp);
          return;
        }
        START_BUCKET(tmp, thenew);
        MRK(d,newchar);
        /* Check if new key is smaller
           than current triekey   */
        if (d == locdepth
        && newchar < triekey.chars[d])
          triekey.chars[d]=newchar;
        return;
      }

      /* If too deep, try to insert in bottom list */
      i = 0;
      tmp = bottom->nxt;
      while (i < bottommax &&
        GREATER(newkey, tmp->key)) {
        tmp = tmp->nxt;
        i++;
      }
      if (i < bottommax) {
        INS_BEFORE (thenew, thenew, tmp);
        return;
      }

      /* We end up here if insertion in bottom
         list turned out to be too expensive.
         That is, the "head" part of the bottom
         list bcomes too large. In order to
         shorten the head part, triedepth is
          increased; triekey is updated */
      if (locdepth >= 0)
        triekey.chars[locdepth]
        = newkey.chars[locdepth];
      do {
        locdepth++;
        triekey.chars[locdepth]
        = newkey.chars[locdepth];
      } while (newkey.chars[locdepth]
        == minkey.chars[locdepth]);
      depth = locdepth;
      newchar=newkey.chars[locdepth];
```

```
      tmp = &buckets[locdepth][newchar];
      START_BUCKET(tmp, thenew);
      MRK(locdepth,newchar);
}

void ExpandBucket()
{ usrpt expand, tmp,next; long i;
  keytype newkey,bottomnextkey; long newchar;
  int short from;

  newchar = triekey.chars[depth];
  tmp = &buckets[depth][newchar];
  expand=tmp->nxt;
  CLR(depth,newchar); /* Mark unused bucket */
  tmp->pre->nxt=NULL;

  /* Put as much as possible directly
     into bottom list */
  for (;;) {
      newkey=expand->key;
      i = 0;
      tmp = bottom->nxt;
      while (i < bottommax &&
        GREATER(newkey, tmp->key)) {
        tmp = tmp->nxt;
        i++;
      }
      if (i >= bottommax) break;
      next=expand->nxt;
      INS_BEFORE (expand, expand, tmp);
      expand=next;
      if (!expand) {
        bottom=bottom->nxt;
        minkey=bottom->key;
        do {
          newchar=minkey.chars[depth]+1;
          newchar=map_min(used[depth],newchar);
          if (newchar <= bucketmax) {
            triekey.chars[depth]=newchar;
            break;
         }
          triekey.chars[depth]=0;
          depth--;
        } while (depth>=0);
        return;
      }
  }


  /* If something remain in bucket,
     scan rest of it and
     replace bottom->nxt whenever relevant. */
  bottomnextkey=bottom->nxt->key;

  /* Since the first key in bucket was constly
     to insert, it need not to beconsidered.
     We start with the next one.   */
  tmp=expand->nxt;
  while (tmp) {
    if (GREATER(tmp->key, bottomnextkey)) {
       tmp=tmp->nxt;
    } else {
       next=tmp->nxt;
       if (next)
         next->pre=tmp->pre;
       tmp->pre->nxt=next;
       INS_BEFORE(tmp,tmp,bottom->nxt);
       bottomnextkey=tmp->key;
       tmp=next;
    }
  }
  bottom=bottom->nxt;
  minkey=bottom->key;

  /* Now bottom list is correct.
     The first element in expand list was
     hard to insert, so it can't be minimum.
     It will form a new bucket. */
  next=expand->nxt;
  depth++;
  from=depth;
  triekey.chars[depth]=newkey.chars[depth];
  while (newkey.chars[depth]
  == minkey.chars[depth]) {
    depth++;
    triekey.chars[depth]=newkey.chars[depth];
  }
  newchar=newkey.chars[depth];
  tmp = &buckets[depth][newchar];
  START_BUCKET(tmp, expand);
  MRK (depth,newchar); /* Mark used bucket */
  expand=next;

/* Now the remainder is inserted in
   the standard style */
  while (expand) {
      next=expand->nxt;
      Insert(expand,expand->key,from);
      expand=next;
  }
}

/* Remove the min key. If the first key in bottom
  list becomes too large, call Expandbucket
  The "real" Deletemin statement is:
  bottom = bottom->nxt;
  This is either done in ExpandBucket or here. */
void DeleteMin()
{
#if usercheck
  if (PQsize<0)
    ERROR(Trying to delete from empty queue);
#endif
  if (depth >= 0 && LESS(triekey,bottom->nxt->key))
    ExpandBucket();
  else {
    bottom = bottom->nxt;
    minkey = bottom->key;
  }
}

/* Called in decrease key */
void ClearBucket (p, k2)
usrpt p;
```

```
keytype k2;
{ usrpt tmp;keytype k1; long d;
  ourchar c1,c2; long ch;

  /*old key only element in some bucket */
  k1=p->key;
  p->key=k2;
  d=FirstDiff(k1,0,depth);
  c1=k1.chars[d];
  c2=k2.chars[d];
  if (c2==c1) return;
  CLR(d,c1);
  if (d==depth && c1==triekey.chars[d]) {
    if (c2>minkey.chars[d]) {
      tmp = &buckets[d][c2];
      START_BUCKET(tmp,p);
      MRK(d,c2);
      triekey.chars[d]=c2;
      return;
    }
    do {
      ch=map_min(used[depth],
                 triekey.chars[depth]+1);
      if (ch <= bucketmax) {
        triekey.chars[depth]=ch;
        break;
      }
      triekey.chars[depth]=0;
      depth--;
    } while (depth>=0);
  }
  Insert(p,k2,d);
}

/********* DIMACS interface ************/

pq_ptr construct_pq (int maxitems)
{
 n = maxitems+1;
 Init();
 return NULL;
}

it_type find_min (pq_ptr Q)
{
 if (mindeleted) {
   DeleteMin();
   mindeleted = false;
 }
 return (it_type)bottom;
}

pr_type del_min (pq_ptr Q)
{
  PQsize--;
  if (!mindeleted)
    mindeleted = true;
  else
    DeleteMin();
  return bottom->key.real;
}

it_type insert (pq_ptr Q,
        pr_type pr, in_type in)
{ usrpt p; keytype k;

  PQsize++;
  k.real = pr;
#if usercheck
  if (LESS(k,minkey))
    ERROR(Insertin gtoo small key);
#endif
  p = usrpt_alloc();
  PR_ASSIGN (p->key, k);
  IN_ASSIGN (p->data, in);
  Insert(p,k,0);
  return (it_type)p;
}


void decrease_p (pq_ptr Q,
     it_type item, pr_type pr)
{ keytype k; usrpt p;

  p = (usrpt)item;
  k.real = pr;
#if usercheck
  if (LESS(k,minkey))
    ERROR(Decrease key);
#endif
  if (p->nxt!=p->pre) {
      p->key.real = pr;
      OUT(p);
      Insert(p,k,0);
      return;
  }
  ClearBucket ((usrpt)item, k);
}

pr_type prioval (pq_ptr Q, it_type item)
{
  return ((usrpt)item)->key.real;
}

in_type *infoval (pq_ptr Q, it_type item)
{
  return &((usrpt)item)->data;
}

int size (pq_ptr Q)
{
  return PQsize ;
}

void clear (pq_ptr Q)
{
 ERROR(clear);
}
pq_ptr destruct_pq (pq_ptr Q)
{
  return NULL;
}
```