CS5633 — Analysis of Algorithms
Term Project — Priority Queues

Humberto Ortiz Zuazaga
584–87–6255

December 15, 1995

Contents

Abstract

The implicit binary heap is a simple and natural representation on which to base a priority queue. Extremely efficient implementations of the abstract operations `Enqueue()` and `Dequeue()` should be possible, and most textbooks have several tips for implementing queues in this way. I present an efficient implementation of priority queues based on this data

structure, and compare my performance with previous work. My queues are significantly faster than previously reported data. Since my code is freely distributable, I hope it can be used as a benchmark in future priority queue studies.

1 Introduction

In the handout for the Term Project for CS 5633, Dr. Das discusses implicit binary heaps and rejects them as an alternative thusly: "... but we still cannot take the CLR solution, because [of] the requirement that the heap must be maintained as a complete binary tree so that it can be mapped to an array. This precludes us from using unknown heap sizes that is unknown at compile time." I had already developed a dynamically allocated priority queue for a research project at Cincinnati [1], and decided to develop an implementation that overcame Dr. Das's objection.

Cormen *et al.* show in their textbook [3], show how to insert into and delete elements from dynamically allocated tables (*i.e.* arrays) where no more than a constant fraction of the space is unused, with an amortized cost of $O(1)$. These dynamic tables form the basis of my implicit heap.

A more serious objection to implicit heap implementations of priority queues is that the worst-case performance is $O(n)$ for both the `Enqueue()` and `Dequeue()` operations. It can be claimed that the expected time for an `Enqueue()` is $O(1)$ when the distribution of priorities in the queues does not have a very low bias (*c.f.* [4]). Nevertheless, the `Dequeue()` is inescapably $O(n)$.

It is my belief that simple data structures and algorithms may be implemented efficiently with less effort than that needed for optimizing implementations of queues with more desirable big-Oh access times. In the case of implicit binary heaps, very efficient machine level code can be used to implement the equivalent of "tree rebalancing" operations which have no direct machine-level code.

2 Methods

The code was developed and tested on `aleph.uthscsa.edu`, a PCI-bus i486DX/2-66 computer with 16Mb of RAM. This machine runs linux 1.2.13. All the programs were compiled with gcc v2.7.0, using the flags `-O6 -fomit-frame-pointer -DNDEBUG`. Some tests were also performed on `wayside.cs.utsa.edu`, a 4 processor Sparc running Solaris 2.4. Programs on the sparc were compiled with gcc v2.6.4.

I measured the time it took several implementations of queues to perform a fixed number (usually 1,000,000) of *holds*, where I dequeue an element, modify it's priority according to a certain function, and then enqueue the same element again.

The test driver is included in the source code in the appendix. It uses a dummy timing loop to find the overhead of random number generation and bookkeeping, and subtracts that time from the time to perform the holds. The code is portable to any UNIX system that supports the `times()` interface (*e.g.* SVID, AT&T, POSIX, X/OPEN, BSD 4.3).

My first implementation was derived from the pseudocode routines in [3]. Below is the actual code fragment for the `Heapify()` routine. The heap implementation described here is based on a structure that stores the size of the allocated space, the index of the last node in the heap, and a pointer to a dynamically allocated array of HeapNodes.

$\langle$*Heap structure*$\rangle\equiv$

```
typedef struct {
  size_t size;
  size_t last;
  struct heapnode *heap_ptr;
} Heap;
```

Heap nodes could be arbitrarily complex, but for our simulations we only used a structure with two fields: the priority, and a bogus data field.

$\langle$*Heap node structure*$\rangle\equiv$

```
typedef struct heapnode {
  double priority;
  long int data;
} HeapNode;
```

Given a heap `heap` and an offset `i` into the heap, and a condition that the subtrees rooted at `Left(i)` and `Right(i)` are heaps, the `Heapify()` routine lets the node `heap[i]` trickle down to the correct position in the heap by exchanging it with the larger of it's two children if necessary, and then calling itself on the subtree it just altered.

⟨*Heapify sample code*⟩≡

```
void Heapify (Heap *heap, int i)
{
  int l, r, largest;

  if (0 >= heap->last)          /* empty heap! */
    return;

  l = Left (i);
  r = Right (i);

  if ((l <= heap->last) &&
      (heap->heap_ptr[l].priority > heap->heap_ptr[i].priority))
    largest = l;
  else
    largest = i;

  if ((r <= heap->last) &&
      (heap->heap_ptr[r].priority > heap->heap_ptr[largest].priority))
    largest = r;

  if (largest != i)             /* node is smaller than it´s child */
  {
    swap (heap->heap_ptr + i, heap->heap_ptr + largest);
    Heapify (heap, largest);
  }
}
```

I ran some preliminary tests on these queues, and found that 26.13% of the time is spent in `Heapify()`. This suggested the first improvement to the algorithms. This implementation of `Heapify()`, although correct, is recursive. One of the exercises in the text is to rewrite this function iteratively. The textbook by Brassard and Bratley [2] has such an algorithm, and I implemented it also.

⟨*sift down*⟩≡
```
void sift_down (Heap *heap, int i) {
  int k, j, n;

  k = i;
  n = heap->last;
  do {
    j = k;
    if ((Left(j) <= n) && (heap->heap_ptr[Left(j)].priority >
                           heap->heap_ptr[k].priority)) {
      k = Left(j);
    }
    if ((Right(j) <= n) && (heap->heap_ptr[Right(j)].priority >
                            heap->heap_ptr[k].priority)) {
      k = Right(j);
    }
    swap (heap->heap_ptr + j, heap->heap_ptr + k);
  } while (j != k);
}
```

Both the `Heapify()` and `sift_down()` routines repeatedly call `swap()` to exchange the positions of two heap nodes. This copying can consume large amounts of time. By adding an extra level of indirection to the heap, the amount of copying can be reduced to a single pointer versus an entire record. In applications where each heap node is large, this can be a significant savings. On the other hand, the extra level of indirection can slow down the heap operations.

In this implementation, the declaration of a heap becomes:

⟨*Heap of pointers structure*⟩≡
```
typedef struct {
  size_t size;
  size_t last;
  struct heapnode **heap_ptr;
} Heap;
```

The actual changes to the code are minor, as an example, here's a line from `sift_down()`:

⟨*Sample heap of pointers code*⟩≡
```
if ((Right(j) <= n) && (heap->heap_ptr[Right(j)]->priority >
                        heap->heap_ptr[k]->priority)) {
```

(*i.e.* replace the structure references "." with pointer references "->".)

3 Results

My results are summarized in Figure 1. Even for queues with 100,000 elements, I can perform a hold in approximately 55 microseconds on my home computer. The same operation can be done in approximately 17 microseconds on `wayside`. Converting the recursive calls to `Heapify()` to calls to the iterative `sift_down()` function resulted in a 26% speedup on queues with 100,000 elements (curve not shown).

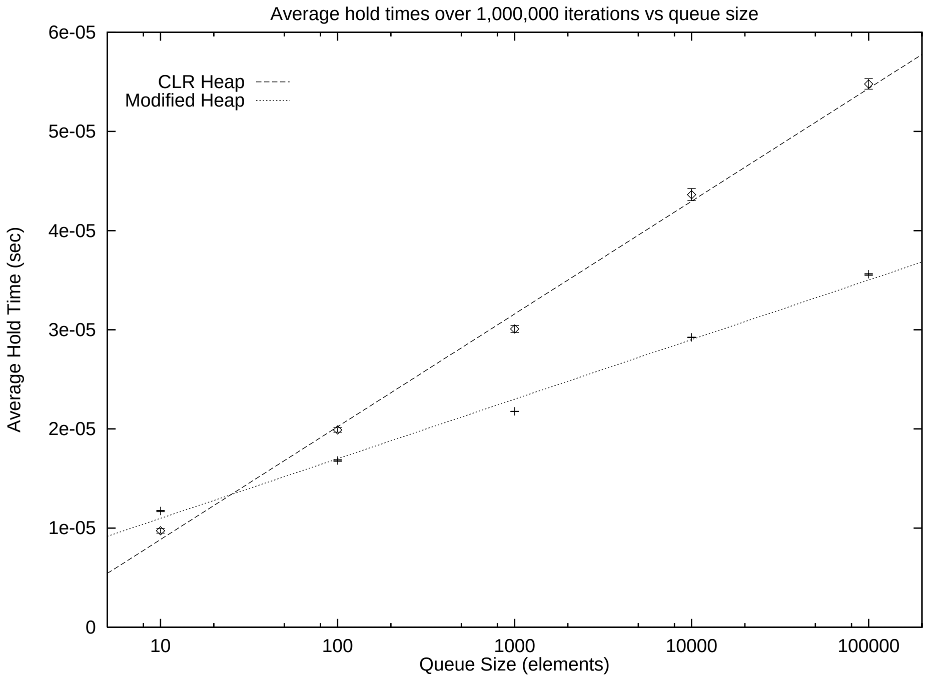

Figure 1: Average hold times over 1,000,000 holds for queues of various sizes. Each point represents the average of 10 trials run for 1,000,000 iterations each.

Combining `sift_down()` with converting to a heap of pointers results a 40% improvement over the naive CLR heaps for hold times on queues with 100,000 elements. I call this implementation the Modified Heap. With very small queue sizes, the overhead of managing the memory for the nodes and dereferencing the pointers to them makes this implementation slightly slower.

I could not perform experiments on queues with more than 200,000 elements or so, because the queues consume all free memory on my home machine. The hold times increase precipitously when swapping occurs while traversing the heap.

I also have some limited results on timing the individual `Enqueue()` and

`Dequeue()` operations. Profiling studies using gprof indicate that `Enqueue()` takes approximately 10 microseconds (0.01 msec) in both the CLR and Modified implementations. The `Dequeue()` operation takes roughly 40 microseconds in the CLR heap, and 20 microseconds in the Modified heap. Unfortunately, the resolution of the timer used by gprof on `aleph` seems to be a hundredth of a millisecond. It does, however confirm the times measured across the loops.

4 Discussion

I have demonstrated that efficient, dynamically resizable implicit binary heaps can be used as a foundation for priority queues. The average time per hold on stock microcomputer hardware is much better than that reported in [4]. My computer is in fact faster than the Sequent Symmetry S81 used by Ronngren and Ayani to perform their experiments, but they claim nearly 400 microseconds are required per hold on a queue with only 1,000 elements. My implementation can perform a hold in 20 microseconds on a queue of that size, a 20-fold difference.

The current implementation of priority queues could be improved upon. Memory management for the individual heap nodes is unnecessarily complex. The code for dynamic tables could be borrowed and used to allocate contiguous arrays of heap nodes, freeing the user from the bookkeeping details.

In addition, since the heap now consists entirely of word sized pointers, the routines to swap two elements could be coded as machine level `xor` instructions instead of generalized assignments. This could lead to a further increase in speed.

The complete source code for the queues is available via the World Wide Web from:

`http://momo.uthscsa.edu/~humberto/Queues/`

It is my hope that this implementation can serve as a benchmark for other queue implementations, and that it can be eventually extended to support more abstract queue operations.

References

[1] F. S. Annexstein, J. V. Franco, H. Ortiz-Zuazaga, and R. P. Swaminathan. Distributed implementations of algorithms for satisfiability problems. Ohio Aerospace Institute / Ohio Supercomputer Center / NASA, Proceedings of Symposium on Application of Parallel and Distributed Computing, April 18–19, 1994.

[2] G. Brassard and P. Bratley. *Algorithms: Theory & Practice.* Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, Massachusetts, 1990.

[4] R. Rönngren and R. Ayani. A Comparative Study of Parallel and Sequential List Algorithms. In *Performance Issues in Discrete Event Simulation.*

A Complete source code

This code was developed for the class project for CS 5633. It was derived from earlier code written by me for a CNF-SAT solver.

⟨*Copyright notice*⟩≡

```
Copyright (C) 1995 by Humberto Ortiz Zuazaga.
Copyright (C) 1993, 1994 by Fred Annexstein, John Franco, Humberto
Ortiz-Zuazaga, Manian Swaminathan

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Since this is a derived work, it also falls under the GPL.

This file was prepared with `noweb`, a literate programming system available from:

`ftp://bellcore.com/pub/norman/www/noweb/intro.html`

or

`http://www.cs.purdue.edu/homes/nr/noweb`

The .tex, .c, and .h files are all extracted from a single source document. Here are the top down specifications for the .c and .h files.

⟨*⟩≡
```
/* ⟨Copyright notice⟩ */
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include "heap.h"

⟨local definitions⟩
⟨local function prototypes⟩

⟨functions⟩
```

The header chunk exports externally visible data structures and functions. Any objects with global linkage should be collected and published here.

⟨header⟩≡
```
/* ⟨Copyright notice⟩ */
#ifndef HEAP_H
#define HEAP_H

typedef short boolean;
#define FALSE 0
#define TRUE !FALSE

⟨Data structures⟩
⟨external function prototypes⟩
#endif /* HEAP_H */
```

A.1 Data structures

The heap is a max-heap of heap nodes.

⟨Data structures⟩≡
```
⟨Heap node⟩
⟨Heap⟩
```

A.1.1 Heap nodes

The heap nodes hold only two values, the priority, and a bogus data field, used for simulating real work.

⟨*Heap node*⟩≡

```
typedef struct heapnode {
  double priority;
  long int data;
} HeapNode;
```

A.1.2 Heap

The heap stores the size of the allocated space, the index of the last node in the heap, and a pointer to a dynamically allocated array of pointers to HeapNodes.

⟨*Heap*⟩≡

```
typedef struct heap {
  size_t size;
  size_t last;
  struct heapnode **heap_ptr;
} Heap;
```

A.1.3 `CreateHeap()`

allocates and initializes to an empty state a heap of size `size`, in fact pointers to `size + 1` nodes are allocated but `heap_ptr[0]` is unused. The heap is empty after creation, the user must fill in the weights and heapify the array by hand, or use `Enqueue()`.

⟨*external function prototypes*⟩≡

```
Heap *CreateHeap (size_t size);
```

⟨*functions*⟩≡
```
Heap *CreateHeap (size_t size) {
  Heap *heap;

  assert (0 < size);

  heap = (Heap *) malloc (sizeof (Heap));
  if (NULL != heap) {
    heap->heap_ptr = (HeapNode **) calloc (size + 1, sizeof (HeapNode *));
    if (NULL == heap->heap_ptr) {
      free (heap);
      heap = NULL;
    }
    else {
      heap->size = size;
      heap->last = 0;
    }
  }
  return heap;
}
```

A.1.4 `DestroyHeap ()`

Deallocate the space reserved for the weight heap. The user must deallocate the memory for the HeapNodes manually.

⟨*external function prototypes*⟩+≡
```
void DestroyHeap (Heap * heap);
```

⟨*functions*⟩+≡
```
void DestroyHeap (Heap *heap) {
  if (NULL != heap) {
    if (NULL != heap->heap_ptr) {
      while (0 < heap->last) {
        if (NULL != heap->heap_ptr[heap->last]) {
          free(heap->heap_ptr[heap->last]);
        }
        heap->last--;
      }
      free (heap->heap_ptr);
    }
    free (heap);
    heap = NULL;
  }
}
```

A.2 Heap manipulation

Routines to actually maintain the heap property. These are C implementations of the algorithms given in "An Introduction to Algorithms" by Cormen et al.. In the implicit heap, the children of node i are stored at $2i$ and $2i + 1$. These macros ensure the more efficient bitwise shift operations are used.

$\langle$*local definitions*$\rangle\equiv$

```
#define Left(i) ((i) << 1)
#define Right(i) (((i) << 1) + 1)
#define Parent(i) ((i) >> 1)
```

A.2.1 `Enqueue()`

Insert a new node into a heap. The caller is responsible for providing space for the new node.

$\langle$*external function prototypes*$\rangle+\equiv$

```
void Enqueue (Heap *heap, HeapNode *newnode);
```

⟨*functions*⟩+≡
```
void Enqueue (Heap *heap, HeapNode *newnode) {
  int i;

  assert (NULL != heap);
  assert (NULL != newnode);

  heap->last++;
  /* Is the heap full? */
  if (heap->last > heap->size) {
    /* resize heap */
    heap->heap_ptr = realloc (heap->heap_ptr, sizeof (HeapNode *) *
                              (heap->size * 2 + 1));
    if (NULL == heap->heap_ptr) {
      fprintf (stderr, "heap: error resizing heap, cannot insert.\n");
      exit (1);
    }
    heap->size = heap->size * 2;
  }
  i = heap->last;
  while ((i > 1) && (heap->heap_ptr[Parent(i)]->priority <
                     newnode->priority)) {
    heap->heap_ptr[i] = heap->heap_ptr[Parent(i)];
    i = Parent(i);
  }
  heap->heap_ptr[i]  = newnode;
  assert (VerifyHeap(heap, 1));
}
```

A.2.2 `sift_down()`

Brassard and Bradley's version of the function to restore the heap property after deleting a node.

⟨*local function prototypes*⟩≡
```
void sift_down (Heap *heap, int i);
```

⟨*functions*⟩+≡

```
void sift_down (Heap *heap, int i) {
  int k, j, n;

  assert (NULL != heap);
  assert (1 <= i);

  k = i;
  n = heap->last;
  do {
    j = k;
    if ((Left(j) <= n) && (heap->heap_ptr[Left(j)]->priority >
                           heap->heap_ptr[k]->priority)) {
      k = Left(j);
    }
    if ((Right(j) <= n) && (heap->heap_ptr[Right(j)]->priority >
                            heap->heap_ptr[k]->priority)) {
      k = Right(j);
    }
    heap->heap_ptr[0] = heap->heap_ptr[j];
    heap->heap_ptr[j] = heap->heap_ptr[k];
    heap->heap_ptr[k] = heap->heap_ptr[0];
  } while (j != k);
  assert (VerifyHeap (heap, i));
}
```

A.2.3 Dequeue()

returns the node with the greatest weight, removes it from the heap, and restores the heap property. The caller must dispose of the memory for the node.

⟨*external function prototypes*⟩+≡

```
HeapNode * Dequeue (Heap *heap);
```

⟨*functions*⟩+≡
```
HeapNode * Dequeue (Heap *A) {
  HeapNode *max;

  assert (NULL != A);

  if (0 >= A->last) {
    fprintf (stderr, "Extracting from empty heap!\n");
    return NULL;                        /* empty heap! */
  }

  max = A->heap_ptr[1];
  A->heap_ptr[1] = A->heap_ptr[A->last];
  A->last--;
  sift_down (A, 1);             /* B+B */

  /* Check if the heap is less than a quarter full. */
  if ((A->last * 4) < A->size) {
    /* resize heap to a half */
    A->heap_ptr = realloc (A->heap_ptr, (1 + A->size / 2) *
                           sizeof(HeapNode *));
    if (A->heap_ptr == NULL) {
      fprintf (stderr, "error resizing heap.\n");
      return NULL;
    }
  }

  return max;
}
```

A.3 Testing

A.3.1 ShowHeap()

⟨*external function prototypes*⟩+≡
```
void ShowHeap (Heap *heap);
```

⟨*functions*⟩+≡

```
void ShowHeap (Heap *heap) {
  int i;

  assert (NULL != heap);

  puts("Heap status.");
  for  (i=1; i<= heap->last; i++)
  {
    printf ("Heap [%d] = pri %f, val = %ld\n", i,
                  heap->heap_ptr[i]->priority,
                  heap->heap_ptr[i]->data);
  }
}
```

A.3.2 `VerifyHeap()`

Verify that a heap indeed maintains the heap property, namely, all nodes have a key greater than or equal to their children's keys.

⟨*external function prototypes*⟩+≡

```
int VerifyHeap (Heap *heap, int i);
```

⟨*functions*⟩+≡

```
int VerifyHeap (Heap *heap, int i) {
  boolean l_stat, r_stat;

  assert (NULL != heap);
  assert (0 < i);

  if (0 >= heap->last)
    return TRUE;                /* empty heap trivially correct */

  if (heap->last >= Left(i))
    l_stat = (heap->heap_ptr[i]->priority >=
               heap->heap_ptr[Left(i)]->priority) &&
              VerifyHeap (heap, Left(i));
  else
    l_stat = TRUE;              /* no left children */

  if (heap->last >= Right(i))
    r_stat = (heap->heap_ptr[i]->priority >=
               heap->heap_ptr[Right(i)]->priority) &&
              VerifyHeap (heap, Right(i));
  else
    r_stat = TRUE;              /* no right children */

  return (l_stat && r_stat);
}
```

A.3.3 Test driver

⟨*test*⟩≡

```
#include <assert.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/times.h>
#include "heap.h"
#include "r250.h"

int main (int argc, char *argv[]) {
  int n, m, i;
  Heap *heap_ptr;
  HeapNode *newnode;
  struct tms start_tms, end_tms;
  clock_t start0, start1, end0, end1;
  double inc;

  if (3 != argc) {
    fprintf (stderr, "Usage: %s n m.\n", argv[0]);
    fprintf (stderr, "Perform hold test m times on queue of size n.\n");
    exit (1);
  }

  n = atoi (argv[1]);           /* size of queue */
  if (0 >= n) {
    fprintf (stderr, "queue size is negative!");
    exit(1);
  }
  if (1000000 <= n) {
    fprintf (stderr, "%d is too large for the queue.", n);
    exit(1);
  }

  m = atoi (argv[2]);           /* number of iterations */
  if (0 >= m) {
    fprintf (stderr, "too few heap operations");
    exit(1);
  }
  if (10000000 <= m) {
    fprintf (stderr, "too many heap operations");
    exit(1);
  }

  /* Bogus loop, for startup timings */
```

```
  newnode = (HeapNode*) malloc (sizeof (HeapNode));
  start0 = times (&start_tms);
  heap_ptr = CreateHeap (1);
  for (i = 1; i <= n; i++) {
    newnode->priority = dr250 ();
    newnode->data = i;
  }
  end0 = times (&end_tms);
  DestroyHeap (heap_ptr);

  /* Build the heap, should do some dequeues too. */
  start1 = times (&start_tms);
  heap_ptr = CreateHeap (1);
  for (i = 1; i <= n; i++) {
    newnode = (HeapNode*) malloc (sizeof(HeapNode));
    if (NULL == newnode) {
        fprintf (stderr, "no more memory for heap nodes");
        exit(1);
    }
    newnode->priority = dr250 ();
    newnode->data = i;
    Enqueue (heap_ptr, newnode);
  }
  assert (VerifyHeap (heap_ptr, 1));
  end1 = times (&end_tms);

#ifndef NDEBUG
  printf ("Start time = %ld, end time = %ld.\n", start1, end1);
  printf ("Bogus start = %ld, Bogus end = %ld.\n", start0, end0);
#endif

  printf ("Build heap elapsed time = %ld clock ticks at %d ticks/sec.\n",
          (end1 - start1) - (end0 - start0), CLOCKS_PER_SEC);

#ifndef NDEBUG
  ShowHeap (heap_ptr);
#endif

  /* bogus timing loop */
  start0 = times (&start_tms);
  for (i = 1; i <= m; i++) {
    /* Dequeue */
    do {
      inc = dr250 ();
    } while (inc < 0.001);
    log (inc);
```

```
    /* Enqueue */
  }
  end0 = times (&end_tms);

  start1 = times (&start_tms);
  for (i = 1; i <= m; i++) {
    newnode = Dequeue (heap_ptr);
    assert (NULL != newnode);
    assert (VerifyHeap (heap_ptr, 1));
#ifndef NDEBUG
    printf ("node.priority = %f, node.data = %ld.\n", newnode->priority,
            newnode->data);
#endif
    do {
      inc = dr250 ();
    } while (inc < 0.001);
    newnode->priority -= log (inc);
    Enqueue (heap_ptr, newnode);
    assert (VerifyHeap (heap_ptr, 1));
  }
  end1 = times (&end_tms);

#ifndef NDEBUG
  ShowHeap (heap_ptr);
#endif

  printf ("HOLD elapsed time = %ld clock ticks at %d ticks/sec.\n",
          (end1 - start1) - (end0 - start0), CLOCKS_PER_SEC);

  for (i = 1; i <= n; i++) {
    newnode = Dequeue (heap_ptr);
    free (newnode);
  }
  DestroyHeap(heap_ptr);

  return 0;
}
```

A.4 Makefile

Here's a sample makefile for testing the queue. It assumes you have noweb installed in your path.

⟨*Makefile*⟩≡

```
# Makefile for noweb.
# Humberto Ortiz Zuazaga 1995/3/1.

NOWEAVE = noweave
NOTANGLE = notangle
CC = gcc
CFLAGS = -O6 -fomit-frame-pointer -DNDEBUG

.SUFFIXES: .c .nw .tex .dvi .html .ps
.nw.html: ;     $(NOWEAVE) -filter l2h -index -html $*.nw > $*.html
.nw.tex: ;      $(NOWEAVE) -delay $*.nw > $*.tex
.nw.c:  ;       $(NOTANGLE) -L $*.nw > $*.c
.nw.h:  ;       $(NOTANGLE) -L $*.nw -Rheader > $*.h
.tex.dvi:
        latex '\scrollmode \input '"$*"; \
        while grep -s 'Rerun to get cross-references right' $*.log; \
        do latex '\scrollmode \input '"$*"; done
.dvi.ps: ;      dvips $*.dvi -o $*.ps

all: driver

report.ps: report.dvi
report.dvi: report.tex
report.tex: report.nw

driver: driver.o heap.o r250.o
        $(CC) $(CFLAGS) -o $@ driver.o heap.o r250.o -lm

heap.o: heap.c heap.h

driver.c: report.nw
        $(NOTANGLE) -L $< -Rtest > $@

heap.c: report.nw
        $(NOTANGLE) -L $< -R\* > $@

heap.h: report.nw
        $(NOTANGLE) -L $< -Rheader > $@

clean:
        rm -f *.o *~ core
```