Queues, Priority Queues and Heaps

The goals of this lab are manifold: to practice your C pointer skills, to work with different kinds of queues, and to experiment with a particularly efficient, both in time and space, implementation of a priority queue.

1 The Basics

A queue is an abstract data type that stores items until they are retrieved; the operations on a queue are given by the following pseudo-function prototypes (our signatures will be slightly different):

```
Queue create(void);
int add(Queue, Item [, other arguments]);
Item remove(Queue);
```

From these prototypes, you can see that the `create()` method is a constructor, and both `add()` and `remove()` are mutative transformers – i.e. they cause the Queue ADT to change as a side effect of the operation.

Note that the `remove()` operation returns the next item from the queue, removing that item from the queue as a side effect of the operation. Different kinds of queues are structured differently, so that the next item to remove differs.

2 Kinds of Queues

There are three basic kinds of queues:

- FIFO – this kind of queue causes the least recently added entry to be the next item to be returned – i.e., first-in-first-out.
- LIFO – this kind of queue causes the most recently added entry to be the next item to be returned – i.e., last-in-first-out.
- Priority – this kind of queue causes the highest priority entry in the queue to be returned; when there are many entries at the same priority, they are usually treated FIFO, although this is not required

The following header file will be used to define the symbolic constants and function prototypes for implementations for the different kinds of queues in this lab. **Note that lower values indicate higher priority!**

queue.h

```
#ifndef _QUEUE_H_
#define _QUEUE_H_

typedef struct queue Queue;
typedef void *Item;

#define MIN_PRIO 10
#define MAX_PRIO 1
```

```
/*
 * create a Queue that holds Items
 * returns NULL if the create call failed (malloc failure)
 */
Queue *q_create(void);
/*
 * add Item to the Queue; 3rd arg is priority in MIN_PRIO..MAX_PRIO;
 * return 1/0 if successful/not-successful
 */
int q_add(Queue *q, Item i, int prio);
/*
 * remove next Item from queue; returns NULL if queue is empty
 */
Item q_remove(Queue *q);

#endif /* _QUEUE_INCLUDED */
```

3 Exercises

You will use the following main program to test all of your queue implementations:

qtest.c

```
#include "queue.h"
#include <stdio.h>
#include <stdlib.h>

#define PRIODIF (MIN_PRIO - MAX_PRIO + 1)
#define MAXLINE 1024

/*
 * this program reads a list of integers from the standard input, one per
line
 * it adds each integer to a queue until it sees EOF
 * it then removes each item from the queue, printing out the integer on
 * a line by itself on standard output
 */
int main()
{
   Queue *q;
   char buf[MAXLINE];
   int *i, prio;

   if ((q = q_create()) == NULL) {
      fprintf(stderr, "Unable to create queue of integers\n");
      return -1;
   }
   while (fgets(buf, MAXLINE, stdin) != NULL) {
      i = (int *)malloc(sizeof(int));
      if (! i) {
         fprintf(stderr, "Unable to allocate int on heap\n");
         return -2;
      }
      *i = atoi(buf);
      prio = MAX_PRIO + ((*i - MAX_PRIO) % PRIODIF);
      if (! (q_add(q, i, prio))) {
         fprintf(stderr, "Unable to add %d to the Queue\n", i);
         return -2;
      }
   }
   while ((i = (int *)q_remove(q))) {
      printf("%d\n", *i);
      free(i);
   }
   return 0;
}
```

3.1 queueFIFO.c

The following source code implements a FIFO queue using a linked list. Study it carefully, as it shows how to allocate the elements of the linked list and storage for the data contained by each element.

queueFIFO.c

```
   #include "queue.h"
   #include <stdlib.h>
   #include <string.h>

   /*
    * implementation of a FIFO queue using a linked list
    * ignore priority argument in add()
    */

   struct q_element {
      struct q_element *next;
      void *value;
   };

   struct queue {
      struct q_element *head;
      struct q_element *tail;
   };

   /*
    * create a Queue that holds Items
    * returns NULL if the create call failed (malloc failure)
    */
   Queue *q_create(void) {
      struct queue *p;

      if ((p = (struct queue *)malloc(sizeof(struct queue))) != NULL) {
         p->head = NULL;
         p->tail = NULL;
      }
      return p;
   }

   /*
    * add Item to the Queue; 3rd arg is priority in MIN_PRIO..MAX_PRIO;
    * return 1/0 if successful/not-successful
    */
   int q_add(Queue *q, Item i, int prio) {
      struct q_element *p;

      p = (struct q_element *)malloc(sizeof(struct q_element));
      if (p != NULL) {
         p->value = i;
         p->next = NULL;
         if (q->head == NULL)
            q->head = p;
         else
            q->tail->next = p;
         q->tail = p;
         return 1;
      }
      return 0;
   }

   /*
    * remove next Item from queue; returns NULL if queue is empty
    */
   Item q_remove(Queue *q) {
      struct q_element *p;
      Item i;

      if (q->head == NULL)
         return NULL;
      p = q->head;
      q->head = p->next;
      if (q->head == NULL)
         q->tail = NULL;
      i = p->value;
      free(p);
      return i;
   }
```

Build a version of qtest against queueFIFO.c and test it.

You can write your own Makefile if you want, or you can compile the two c files without a Makefile, e.g.:

```
clang -Wall -Werror qtest.c queueFIFO.c -o queueFIFO
```

You should test your implementation with the provided `test.dat`:

```
./queueFIFO < test.dat
```

3.2 queueLIFO.c

Starting from queueFIFO.c, now implement queueLIFO.c. The semantics here is that the most recently added element is the next item to be removed by q_remove (think of a stack). Test this to be sure that it works correctly.

3.3 queuePrioLL.c

Starting from queueFIFO.c, we are now going to generate a simple version of a priority queue; in this case, we still use a linked list to keep the entries. Since the semantics of a priority queue is that the next item to remove should be the highest priority in the list, this implies that the "head" member of the Queue listhead should point to the highest priority entry in the list. In turn, this means that the add() operation needs to determine the appropriate place in the linked list at which to insert the new entry. If there are several entries with the same priority as the new entry, the new entry should go after the last entry in the list with that priority.

Implement this version of the priority queue. Test this to make sure it works correctly.

What is the complexity of add() as a function of the number of entries already in the list? Of remove()?

3.4 queuePrioMultiLL.c

A more efficient way to structure the priority queue when there are a reasonably small number of priorities is to have a linked list for each priority. This can be achieved using an array of q_head structures, one for each priority. add() now simply appends a new item to the appropriate linked list for that priority.

remove() becomes a little more complex, in that it must start at the highest priority, checking to see of there are any items in that linked list. If not, it goes to the next lower priority, and continues this until it finds a non-empty list, or reaches a priority below MIN_PRIO.

Implement this version of the priority queue. Test this to make sure it works correctly.

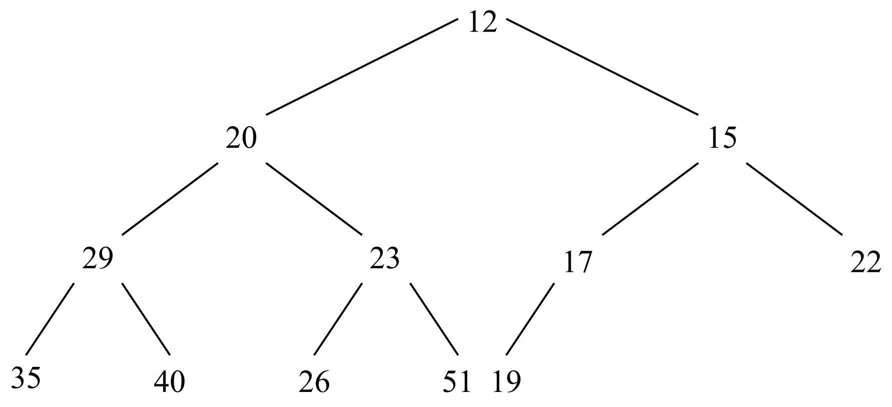
What is the complexity of add() for this implementation? Of remove()?

3.5 queuePrioHeap.c

If the number of items to be stored in the queue is bounded, we can use a heap structure to maintain a priority queue. I expect that you have been exposed to a heap structure in previous courses, but I will give you the essentials here just in case.

A heap is a data structure for representing a collection of items – just what we need for a queue. Since we are using a heap to represent priorities, we will be using a heap of numbers; elements of a heap can be any ordered type.

For example, here's a heap of twelve integers:



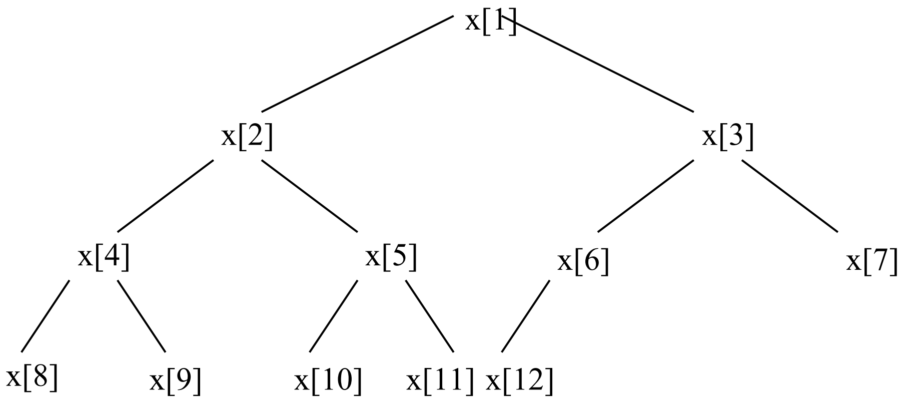
This binary tree is a heap by virtue of two properties:

- *order* – the value at any node is less than or equal to the values of the node's children; this implies that the least element of the set is at the root of the tree (12 in this case), but it does not say anything about the relative order of left and right children.

- *shape* – a heap has its terminal nodes on at most two levels, with those on the bottom level as far left as possible; if it contains N nodes, no node is at a distance of more than $\log_2 N$ from the root.

These two properties are restrictive enough to enable us to find the minimum element of a bag of numbers, but lax enough so that we can efficiently reorganize the structure after inserting or deleting an element.

Despite the fact that a heap is a binary tree, there is a very efficient way to represent the heap without using pointers since the heap possesses the *shape* property. A 12-element tree with the shape property is represented in the12-element array x[1]…x[12] as:



Notice that heaps use a 1-based array; the easiest approach in C is to declare x[N+1] and not use element x[0]. In this implicit representation of a binary tree with the *shape* property, the root is x[1], its two children are in x[2] and x[3], and so on. The typical functions on the tree are defined as follows:

- root = 1
- value(i) = x[i]
- leftchild(i) = 2*i
- rightchild(i) = 2*i + 1
- parent(i) = i / 2
- null(i) = (i < 1) or (i > N)

A C declaration for the 12-element tree shown above would be

```
int x[12+1] = {0, 12, 20, 15, 29, 23, 17, 22, 35, 40, 26, 51, 19};
```

Because the *shape* property is guaranteed by the representation, the name *heap* will mean that the value in any node is greater than or equal to the value in its parent. Phrased precisely, the array x[1]…x[N] has the heap property if

$$\forall_{2 \le i \le N}\ x[i/2] \le x[i]$$

This works correctly since integer division rounds down, so 4/2 and 5/2 both yield a parent of 2. We will need to be able to talk about a subarray x[l]…x[u] having the *heap* property, so we will mathematically define *heap(l, u)* as

$$\forall_{2l \le i \le u}\ x[i/2] \le x[i]$$

3.5.1 Two Critical Functions

When we add another element to a heap, assuming that the heap currently occupies x[1]…x[N-1], the easiest thing to do is to place the new element in x[N]. Unfortunately, *heap(1, N)* will most likely not be true. Therefore, we need an efficient algorithm to modify x[1]…x[N] such that *heap(1, N)* is true. The function that reestablishes the *heap* property is called *siftup*.

The code for siftup is as follows:

```
void siftup(int x[], int n) {
/* preconditions: n > 0 && heap(1, n-1)
 * postcondition: heap(1, n)
 */
   int p, i = n;

   while (i > 1) {
      p = i / 2;
      if (x[p] <= x[i])
         break;
      swap(x+p, x+i);
      i = p;
   }
}
```

If *heap(1, N)* is true, and we remove x[1], we would like to move the N-1 remaining elements into x[1]…x[N-1], but we need to reestablish *heap(1, N-1)*. The easiest way to do this is to place x[N] into x[1], and then reestablish *heap(1, N-1)*. This is achieved using a function *siftdown*.

```
void siftdown(int x[], int n) {
/* preconditions: heap(2, n) && n >= 0
 * postcondition: heap(1, n)
 */
   int c, i;

   i = 1;
   while (1) {
      c = 2 * i;
      if (c > n)
         break;
      if (c+1) <= n
         if (x[c+1] < x[c])
            c++;
      if (x[i] <= x[c])
         break;
      swap(x+c, x+i);
      i = c;
   }
}
```

3.5.2 Priority Queue Using a Heap

You should now design and implement a priority queue using a heap. You should statically allocate an array to hold the queue elements; this means that `q_add()` may fail if the array is already full. Compare it against your other priority queue implementations.