|  |
| --- |
| **La Roche College** |
| **Final Project** |
| **FA13-CSCI2025-01 - Systems Programming** |

|  |
| --- |
| **Bolong Yan**  **2013/12/11** |

Contents

**Section 1 Purpose3**

**Section 2 Definitions4**

**a. Acronyms and Abbreviations4**

**b. Parameter Description4**

**Section 3 References5**

**Section 4 Overall Description5**

**a. Software Description and Rationale6**

**b. Software Features6**

**c. User Characteristics6-7**

**d. Constraints7**

**e. Assumptions and Dependencies7**

**Section 5 Design Specifics8**

**a. Files8**

**b. Functions9**

**Section 6 Testing10**

**Section 7 Developer’s Guide12-13**

**Section 1: Purpose**

This C program implements the priority queue abstract data structure.

The priority queue implementation must fulfill the following requirements:

• It must be backed by an array-based binary max heap data structure.

• It must be generic, i.e. able to work with elements of various data types.

• It must provide the following functions:

**priority\_queue\* pq\_create( int ( \*cmp ) ( const void\* element1, const void\* element2 ),**

**size\_t capacity )** – Creates and returns a pointer to a priority queue instance of a fixed size

(capacity) that uses a specific comparator to maintain the heap property.

**void pq\_destroy( priority\_queue\* pq )** – Frees the memory used by the specified priority

queue.

**int pq\_empty( priority\_queue\* pq )** – Returns 1 if the priority queue is empty; or 0

otherwise.

**void pq\_insert( priority\_queue\* pq, const void\* element )** – Inserts an element in the

priority queue.

**void\* pq\_max( priority\_queue\* pq )** – Returns the max (top) element of the priority queue.

**void\* pq\_remove\_max( priority\_queue\* pq )** – Returns and removes the max element of

the priority queue.

**Section 2: Definitions**

**Acronyms and Abbreviations:**

pq – priority\_queue

cmp – compare

res – restore

idx – index

q – queue

tmp – temporary

lrg – largest

**Parameter Description:**

pq\_create(parm1,parm2)

Create a queue for comparing the functions and receive these two parameters from parm1

Parm2 is the largest number of elements in the queue.

The function returns values:

\*returns 0 if d1 and d2 have the same priorities

\*returns [negative value] if d1 have a smaller priority than d2

\*returns [positive value] if d1 have a greater priority than d2

Then return the queue

pq\_insert(parm1,parm2)

Add the element into the queue

parm1: elements need to be added to the queue

parm2: added elements

pq\_max(parm1)

Returns the largest element in the queue

parm1: queue

Then return the largest element in the queue

pq\_remove\_max(parm1)

Returns the largest element in the queue and this element is removed from the queue parm1: queue

Then return the largest element in the queue

pq\_destroy(parm1)

Destruction queue, recycling resources

parm1: queue

**Section 3: References**

Baidu. "Priority Queue and Its Applications." Baidu Library. Baidu, n.d. Web. 11 Dec. 2013.

<http://wenku.baidu.com/link?url=Ey8JKiQv58J7NQ9XVI9Hi4xRMkTFK4PS2ZKQ7hQgjRE9L2dLcD88JyjBzkhCR6c-bwRwOSGlINlRMqFvRkZL5KXxfyUBXDiw-DCgAhVHKoO>

Djkpengjun. "Priority Queue-djkpengjun-ChinaUnix Blog." Priority Queue-djkpengjun-ChinaUnix Blog. N.p., 28 July 2010. Web. 11 Dec. 2013. <http://blog.chinaunix.net/uid-21712186-id-1818266.html>

Yuanlai. "Priority\_queue in C - CnBlog." Priority\_queue in C - CnBlog. Yuanlai, 31 July 2011. Web. 11 Dec. 2013.

<http://www.cnblogs.com/this-543273659/archive/2011/07/31/2122639.html>

**Section 4: Overall Description**

1. **Software Description and Rationale:**

This C program implements the priority queue abstract data structure.

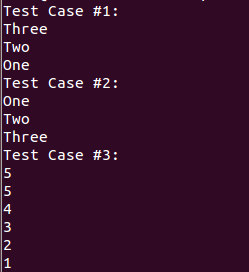
This program is backed by an array-based binary max heap data structure. And it is generic, able to work with elements of various data types

The new element in my program is swapped with its parent as long as its precedence is higher so that makes the queue in order.

Compile the program in Linux:

C:\Users\LuvCat\AppData\Roaming\Tencent\Users\85868501\QQ\WinTemp\RichOle\J%MF__PR[$_J_(F(@4J8I)1.jpg

The screen output:



1. **Software Features**
   1. After a queue has been created, the queue will be obtained proper maintenance when users use it**.**
   2. Queue processing is faster with less resource overhead**.**
2. **User Characteristics**

This program is offered for those staffs that need to filter a large number of data processing such as professors and teachers. They may create a data queue for all students

**Section 4: Overall Description**

and then sort the grades for them while they could also list the name and find the sort results.

1. **Constraints**
   1. Each element in the queue has the same attribute, and has been assigned.
   2. Comparison function correctly compares the certain attributes of a user-defined element and returns the correct value.

**e. Assumptions and Dependencies**

**a)** Queue data structure must be given the maximum number of elements in this queue when you create.

**b)** We must define comparison function first when use this queue data structure, comparison function relies on the data/elements structure.

**Section 5: Design Specifics**

**Files:**

**main.c**

The main part of the program, include the test function code.

Fetch a header element from the queue loop, and then compare with the previous value.

If the value is less than the previous, it will return true.

**pqueue.c**

This file includes the priority queue function and comparison function

Allocates memory for a new Priority Queue structure

Comparison function:

Returns 0 if d1 and d2 have the same priorities

Returns 1 if the priority queue is empty; or 0 otherwise

De-allocates memory for a given Priority Queue structure

Adds a new element to the Priority Queue

Returns the element with the biggest priority from the queue

Removes and Returns the element with the biggest priority from the queue

Turn an "almost-heap" into a heap

**pqueue.h**

This head file includes stdio.h and stdlib.h.

Also, it checks for a NULL pointer, and prints the error message, source file and line via 'stderr'. If the check fails the program exits with error code (-1) .

**Section 5: Design Specifics**

**Functions:**

**priority\_queue\* pq\_create( int ( \*cmp ) ( const void\* element1, const void\* element2 ), size\_t capacity )** :

Creates and returns a pointer to a priority queue instance of a fixed size

(capacity) that uses a specific comparator to maintain the heap property.

**int pq\_empty( priority\_queue\* pq ):**

Returns 1 if the priority queue is empty; or 0 otherwise

**void pq\_insert( priority\_queue\* pq, const void\* element ):**

Add an element inside the Priority Queue

**void\* pq\_max( priority\_queue\* pq ):**

Returns the max (top) element of the priority queue

**void\* pq\_remove\_max( priority\_queue\* pq ):**

Returns and removes the max element of the priority queue.

**void pq\_destroy( priority\_queue\* pq )**:

Frees the memory used by the specified priority queue. De-allocates memory for a given Priority Queue

**Section 6: Testing**

I have done several tests, including numeric type elements, char type elements, word types and other elements, the queue contains thousands of elements, by a simple code (fetch a header element from the queue loop, compare with the previous value, if less than the previous, return true) at the end the test queue returns is correct.

Test code:

/\*\* Test Function \*/

int main( )

{

int i;

word word1, word2, word3;

word1.str = "One";

word1.count = 100;

word2.str = "Two";

word2.count = 50;

word3.str = "Three";

word3.count = 20;

// Create a priority queue that uses a word comparator

priority\_queue\* pq1 = pq\_create( word\_cmp, 3 );

pq\_insert( pq1, &word1 );

pq\_insert( pq1, &word2 );

pq\_insert( pq1, &word3 );

printf( "Test Case #1:\n" );

while( ! pq\_empty( pq1 ) )

{

printf( "%s\n", ( ( word\* ) pq\_remove\_max( pq1 ) )->str );

}

// Create a priority queue that uses a count comparator

priority\_queue\* pq2 = pq\_create( count\_cmp, 3 );

pq\_insert( pq2, &word1 );

pq\_insert( pq2, &word2 );

pq\_insert( pq2, &word3 );

**Section 6: Testing**

printf( "Test Case #2:\n" );

while( ! pq\_empty( pq2 ) )

{

printf( "%s\n", ( ( word\* ) pq\_remove\_max( pq2 ) )->str );

}

// Create a priority queue that uses a number comparator

priority\_queue\* pq3 = pq\_create( number\_cmp, 5 );

for( i = 1; i <= 5; i++ )

{

int\* number = ( int\* ) malloc( sizeof( int ) );

\*number = i;

pq\_insert( pq3, number );

}

printf( "Test Case #3:\n" );

printf( "%d\n", \*( ( int\* ) pq\_max( pq3 ) ) );

while( ! pq\_empty( pq3 ) )

{

printf( "%d\n", \*( ( int\* ) pq\_remove\_max( pq3 ) ) );

}

pq\_destroy( pq1 );

pq\_destroy( pq2 );

pq\_destroy( pq3 );

return 0;

}

**Section 7: Developer’s Guide**

1. Include the “pqueue.h” at the header.
2. User defines an element type, and determines an attribute of the element as to compare.
3. Based on step b, user defines a comparison function to compare the properties of the elements and returns the results.
4. Use pq\_create to create the queue, given comparison function and the largest number of elements in this queue.
5. Use pq\_insert to insert elements into the queue.
6. While using the queue:
   1. Use pq\_max to get the largest element in the queue.
   2. Use pq\_remove\_max to remove the head of the queue element (max element).
7. When finished using the queue and no longer use, call pq\_destroy to delete the queue and recovery of resources.

Examples:

/\*example code\*/

#include "pqueue.h"

int main(int argc, char \*argv[]) {

priority\_queue \*q = NULL;

int i; //number of elements

//define comparison function

int cmp\_int(const void \*element1, const void \*element2) {

return \*(int \*)element1 - \*(int \*)element2;

}

printf("Test priority queue with int value!\n");

/\* Create a priority Queue containing a maximum of 10 elements \*/

q = pq\_create(cmp\_int, 10);

//insert the element into queue

for(i = 0; i < 10; ++i) {

/\* Adding elements to priority Queue \*/

pq\_insert(q, &data\_int[i]);

}

printf("priority queue created!\n");

//output the maximum element in the queue

printf("max element is %d\n", \*(int\*) pq\_max(q));

//go over the queue and list all elements from max to min

while(!pq\_empty(q)) {

printf("%d removed\n", \*(int \*) pq\_remove\_max(q));

}

//after using, delete the queue and get the resource back.

pq\_destroy(q);

printf("priority queue destroyed!\n");

return (0);

}