**Priority Queue**

# Priority Queue

A priority queue is an ADT which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority. If two elements have the same priority, they are served according to their order in the queue.

often implemented with heaps, but they are conceptually distinct from heaps

can be implemented with a heap or a variety of other methods such as an unordered array

# Time Complexity

In the following time complexities O(f) is an asymptotic upper bound and Θ(f) is an asymptotically tight bound. Function names assume a min-heap.

|  |  |  |
| --- | --- | --- |
| Operation | Binary | Fibonacci |
| find-min | *Θ*(1) | *Θ*(1) |
| delete-min | *Θ*(log *n*) | *O*(log *n*) |
| insert | *O*(log *n*) | *Θ*(1) |
| decrease-key | *Θ*(log *n*) | *Θ*(1) |
| merge | *Θ*(*n*) | *Θ*(1) |

# Operations

## Primary Operations

### enqueue

add an item to the rear of the queue with an associated priority

### dequeue

remove an item from the front of the queue (assuming highest priority element is always on front)

## Additional Operations

### Peek

returns the highest-priority element but does not modify the queue

### isFull

check if queue is full

### isEmpty

check if queue is empty

# Applications

1. Bandwidth management
2. Discrete event simulation
3. CPU Scheduling
4. **Dijkstra’s Shortest Path Algorithm:** When the graph is stored in the form of adjacency list or matrix, priority queue can be used to extract minimum efficiently when implementing Dijkstra’s algorithm
5. **Prim’s algorithm**: It is used to implement Prim’s Algorithm to store keys of nodes and extract minimum key node at every step
6. **Data compression:** It is used in Huffman codes which is used to compresses data
7. **Operating systems:** It is also use in Operating System for load balancing (load balancing on server), interrupt handling
8. Best-first search algorithms
9. ROAM triangulation algorithm

# Priority Queue Implementations

## Naive implementations

Keep all the elements in an unsorted list. Whenever the highest-priority element is requested, search through all elements for the one with the highest priority.

* In big O notation: O(1) insertion time, O(n) pull time due to search
* Usually inefficient, ways to implement a priority queue

## Usual implementation

To improve performance, priority queues typically use a heap as their backbone, giving O(log n) performance for inserts and removals, and O(n) to build initially. Variants of the basic heap data structure such as pairing heaps or Fibonacci heaps can provide better bounds for some operations.

Alternatively, when a self-balancing binary search tree is used, insertion and removal also take O(log n) time, although building trees from existing sequences of elements takes O(n log n) time; this is typical where one might already have access to these data structures, such as with third-party or standard libraries.

From a computational-complexity standpoint, priority queues are congruent to sorting algorithms. Sorting algorithms can create efficient priority queues.

## Specialized heaps

There are several specialized heap data structures that either supply additional operations or outperform heap-based implementations for specific types of keys, specifically integer keys.

Example:

* Bucket queue
* Van Emde Boas tree
* Fusion tree algorithm by Fredman and Willard
* Monotone priority queues

# C Implementation Using Array

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX 6

int intArray[MAX];

int itemCount = 0;

int peek() {

return intArray[itemCount - 1];

}

bool isEmpty() {

return itemCount == 0;

}

bool isFull() {

return itemCount == MAX;

}

int size() {

return itemCount;

}

void insert(int data) {

int i = 0;

if(!isFull()) {

// if queue is empty, insert the data

if(itemCount == 0) {

intArray[itemCount++] = data;

}

else {

// start from the right end of the queue

for(i = itemCount - 1; i >= 0; i-- ){

// if data is larger, shift existing item to right end

if(data > intArray[i]){

intArray[i+1] = intArray[i];

}  
 else {

break;

}

} // end for

// insert the data

intArray[i+1] = data;

itemCount++;

} // end if(itemCount == 0)

} // end if(!isFull())

}

int removeData(){

return intArray[--itemCount];

}

int main() {

/\* insert 5 items \*/

insert(3); insert(5);

insert(9); insert(1);

insert(12);

// ------------------

// index : 0 1 2 3 4

// ------------------

// queue : 12 9 5 3 1

insert(15);

// ---------------------

// index : 0 1 2 3 4 5

// ---------------------

// queue : 15 12 9 5 3 1

if(isFull()){

printf("Queue is full!\n");

}

// remove one item

int num = removeData();

printf("Element removed: %d\n",num);

// ---------------------

// index : 0 1 2 3 4

// ---------------------

// queue : 15 12 9 5 3

// insert more items

insert(16);

// ----------------------

// index : 0 1 2 3 4 5

// ----------------------

// queue : 16 15 12 9 5 3

// As queue is full, elements will not be inserted.

insert(17);

insert(18);

// ----------------------

// index : 0 1 2 3 4 5

// ----------------------

// queue : 16 15 12 9 5 3

printf("Element at front: %d\n",peek());

printf("----------------------\n");

printf("index : 5 4 3 2 1 0\n");

printf("----------------------\n");

printf("Queue: ");

while(!isEmpty()){

int n = removeData();

printf("%d ",n);

}

return 0;

}

Output:

Queue is full!

Element removed: 1

Element at front: 3

----------------------

index : 5 4 3 2 1 0

----------------------

Queue: 3 5 9 12 15 16

# END