2413 Assignment 9

Due Date
4-18-16 11:59pm

For this assignment you will be implementing an Adaptable Priority Heap using an array to model a complete binary tree.

The Priority Heap Tree Properties:

1. Every node's priority is greater (or smaller) than the priority of their children.
2. The tree is complete:
   (a) Insert: new nodes are added at the lowest level with open spaces, filling from left to right.
   (b) Remove: The last node in the lowest level is removed first.

Auxiliary Functions For Maintenance of Properties.

1. upheap: if child node's content is of greater priority than parent, swap child and parent content.
2. downheap: if parent's node content is less than child's content, swap content with child of greatest priority.
3. resize: expand the array with enough space to model a new empty level in the tree.

Adaptable Priority Queue Functions

1. enqueue : queue , element , priority $\rightarrow$ handle *
2. isEmpty : queue $\rightarrow$ boolean
3. dequeue : queue $\rightarrow$ element
4. update : queue , handle * , priority $\rightarrow$ void

In order to test the adaptable priority heap, you will need to keep an array of handle pointers in the main function. Each enqueue operation will be counted (starting from 0) and the handle pointer returned from the queue will be saved in the proper index of the array. This will allow the console interface to refer to elements enqueued based on the order in which they were enqueued. A size of 100 handle pointers will be more than sufficient for this assignment.

Console Commands:

```
enqueue <first name> <last name> <priority>
dequeue
update <integer> <new priority>
```

Priorities are integers

We will use a custom compare function to allow us to easily change the direction of ”greater priority”

```
int compare(priority left, priority right){
if(left < right) return -1;
if(left > right) return 1;
return 0;
}
```

1 Data Structures

Handle

1. The current array index of the element the handle is tracking in the priority queue.
    - This value will change whenever the position of the element changes. It will need to be updated whenever a swap occurs within upheap and downheap operations.

Priority Queue Entry

1. element
2. priority
3. handle *

Priority Queue

1. pointer to array of Priority Queue Entries
2. array size
3. number of entries
4. next level size

2 Auxiliary Functions

swap(index parent , index child)

1. swap the entry content in the provided indexes.
2. update the handles of the entries with the new index information.
3. `printf("Swapping index %d and %d\n", parent, child);`

upheap (index of array)

1. If index is 1, root position, no upheap can be performed
2. If index > 1, compare priority of child and parent (index/2) and swap if necessary
3. If swapped, call upheap on parent index.

downheap(index of array)

1. If index is on empty position (compare index to number of entries) do nothing
2. If index is of a leaf node (compare left child index to number of entries) do nothing
3. If index is of inner node, compare priority of parent to children (right child may not exist) and swap with greatest child if the child priority is greater.
4. If a swap occurred, call downheap on the child index that was swapped.

resize(queue)

1. allocate new array of size (array size + next level size)
2. copy content from old array to new array preserving position.
3. update array size with new size
4. double the next level size

5. `printf("Resized Array To %d\n", array size);`

createQueue()

1. allocate memory and initialize
2. start with an array of size of 2 (space for position 1 in the queue)
3. next level size = 2

3 Interface Functions

enqueue(queue, element, priority)

1. position of new entry in queue is (number of entries +1)
2. set element and priority of new position
3. allocate space for a new handle
4. set handle pointer of new position.
5. upheap on new position
6. return handle pointer

update(queue, handle *, priority)

1. get the index from the handle
2. update the priority of the entry at the index
3. upheap or downheap on the index as needed.

dequeue(queue)→element

1. We assume that this function is not called when the queue is empty
2. Copy the element from index 1 to return at end of function
3. Copy the content from the last occupied index (element, priority, handle*) into index 1
4. Since we are implementing the heap with an array there is no node to free, decrement the number of elements in the array to indicate the last position has been emptied.
5. downheap on index 1

isEmpty(queue)→boolean

1. return 0 if number of elements is larger than 0, otherwise return 1.

4 Program CONSOLE IO

Console Command: `enqueue <first name> <last name> <priority>`

1. `Enqueue of <first name> <last name> with priority <priority>`
2. Followed by a sequence of swap statements generated by upheap

Console Command: `update <integer> <new priority>`

1. `Updating <integer> with priority <new priority>`
2. Followed by a sequence of swap statements generated by upheap/downheap

Console Command: `dequeu`

1. If the queue is not empty, print:
   (a) `Dequeue`
   (b) Followed by a sequence of swap statements generated by upheap/downheap
   (c) `Dequeue of <first name> <last name>`
2. If the queue is empty:
   (a) `Queue Is Empty`

5 Sample IO

INPUT:

```
dequeue
enqueue john doe 50
enqueue jane doe 60
enqueue janet doe 23
enqueue jim doe 12
update 1 6
dequeue
dequeue
```

OUTPUT:

```
Queue Is Empty
Enqueue of john doe with priority 50
Enqueue of jane doe with priority 60
Swapping index 1 and 2
Enqueue of janet doe with priority 23
Enqueue of jim doe with priority 12
Updating 1 with priority 6
Swapping index 2 and 4
Dequeue
Swapping index 1 and 3
Dequeue of jane doe
Dequeue
Dequeue of janet doe
```