**Stable-Duplicate Priority Queue**

Aliza Levinger, Ashley Haiger, Rahul Sood, Rosanna Corvino

Computer Science Department, Pace University

1 Pace Plaza, New York, NY 10038 USA

{AL11679N, AH67477P, RS76135N, RC14127N}@pace.edu

**Abstract**. This paper addresses the issue of instability in a priority queue. Implemented as a binary heap, it provides a useful tool for extracting either the minimum or maximum value in a bag of elements. However, though it can hold duplicate values, it does not guarantee the preservation of order regarding duplicates. Two suggested implementations are proposed, one static and one dynamic, which allow the priority queue to be stabilized. The static version applies the concepts of memoization and direct-addressing hash tables to array-based max heaps to provide stabilization, while the dynamic version utilizes linked lists within a binary search tree to achieve this result.

**1 Introduction**

A priority queue is an abstract data type that maintains a set of elements, each with an associated key, a value ascribing an element with a certain level of importance [1]. If a key value is duplicated in the data set and order is important to assigning priority, the priority queue might not be able to take this into account. Below are real-life scenarios when maintaining the order of duplicates would be useful:

1. **Boarding an airplane** – The primary basis for assigning priority is by class – First class, Business class, and Economy class. Within each class, priority should be assigned on a first come first served basis. So the particular class can be considered duplicate key values and in this case, order is indeed important.

2. **VIP Pass to Amusement Park** or other attraction – All VIP Pass holders automatically have greater priority, but within a group of Pass holders, order is important, they should be served on a first come first served basis.

3. **Store** – give priority to biggest orders - whether that’s biggest quantity or highest profit - in first come first served order, or similar to the previous example, give priority to preferred customers, (ex: Amazon Prime)

4. **Rental Car** – A customer request for a particular model car should be fulfilled with the least used cars first, in order to distribute use and thus wear and tear. A rental car company will have “duplicate values” – i.e. multiple cars of a specific make and model – and they should be assigned on a basis of minimal use first.

5. **Distributed Computer Processing** – given a specific amount of time or space, choose projects fitting those available constraints according the order in which they came in.

"When scheduling programs for execution in a multitasking operating system, at any given moment there might be several programs (usually called jobs) ready to run. The next job selected is the one with the highest priority. Priority is indicated by a particular value associated with the job. When a collection of objects is organized by importance or priority, we call this a priority queue. " [3]

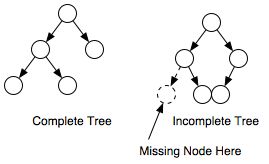
6. **Bandwidth Management** – in the event of insufficient bandwidth on a network, traffic should be assigned some kind of priority based on importance, but assuming level of importance is the same, order should be preserved

Two suggested implementations will be discussed, one static and one dynamic, which allow the priority queue to be stabilized. The static version applies the concepts of memoization and direct-addressing hash tables to array-based max heaps to provide stabilization, while the dynamic version utilizes linked lists within a binary search tree to achieve this result.

**2 Heap Data Structure**

A heap is an implementation of a priority queue.. A heap is physically implemented using either an array or a linked-list. Logically, we view and discuss the heap as a binary tree. Although a heap is not stable, it conforms to a sorting principle: every node has a value greater than (assuming a max heap) either of its children [2]. Additionally, a heap is a complete tree, meaning that the tree is filled from left to right, by level [1].

The upheap and downheap process of swapping values are used when adding or removing nodes in order to preserve the heap’s partial sorting characteristic, which is what can potentially enable the “wrong” duplicate value to be deleted.



**Input:** an array of n elements, with at least 2 values of n being the same

**Output**: a max heap where elements are deleted in a stable manner

**3 Prior Research**

**Prior Research Overview**

A plethora of research has been done on heap data structures; however, there appears to be little to no research done on using a memoization table to stabilize heaps. This section of the papers will, however, discuss how 8 other research papers that worked on stabilizing their heaps. All of the research found in this section concentrates on attempts to stabilize the heap data structure.

**Similar Research Papers**

Herman, Ted and Pirwani, Imran wrote a research paper that describes a heap construction that supports insert and delete operations in arbitrary (possibly illegitimate) states. After any sequence of at most O(m) heap operations, the heap state is guaranteed to be legitimate, where m is the initial number of items in the heap. The response from each operation is consistent with its effect on the data structure, even for illegitimate states. The time complexity of each operation is O(lgK) where K is the capacity of the data structure; when the heap's state is legitimate the time complexity is O(lgn) for n equal to the number items in the heap. The main topic of their study was to describe a heap construction that supports insert and delete operations in arbitrary states. Stabilizing heap construction; availability and stabilization of heap construction [4].

Ted Herman also wrote another paper with Toshimitsu on stabilizing data structure using heaps. Their paper stated while there are numerous studies of fault-tolerant data structures, the fault model for these studies does not consider recovery from unlimited transient faults in the data structure. Self-stabilization is required to deal with unlimited transient faults, yet very few papers in the area of self-stabilization treat data structures in the model of operations applied to the structures. The notion of an available and stabilizing data structure is new in, which presents an available, stabilize binary heap [5].

A paper titled, ‘*A Self-Stabilizing Max-Heap Protocol In Tree Networks*,’ focused on stabilizing a heap. It discussions about how a fault-tolerant protocol is important in a distributed system consisting of processes that are interconnected in a network. One prominent technique used for implementing fault tolerance is a self-stabilizing protocol. A self-stabilizing protocol stabilizes in obtaining a solution regardless of the network status from which execution begins. This property enables a self-stabilizing protocol to tolerate arbitrary transient faults. This paper uses a self-stabilizing protocol for implementing interprocess synchronization in a tree network to propose a self-stabilizing protocol for configuring a heap-ordered tree. The proposed protocol for configuring the heap-ordered tree, which has stabilizing time O(h) and space complexity of each process O(K), has improved both the stabilizing time and space complexity compared with other known results (where h is the height of the tree and K is the input size) [6].

The 4th paper, entitled, ‘*Heaps And Unpointed Stable Homotopy Theory*,’ focused on showing how certain “stability phenomena" in unpointed model categories provide the sets of homotopy classes with a canonical structure of an abelian heap, i.e. an abelian group without a choice of a zero. In contrast with the classical situation of stable (pointed) model categories, these sets can be empty [7].

The 5th research paper is titled, ‘*Computing The Abelian Heap Of Unpointed Stable Homotopy Classes Of Maps,* is a continuation of paper 4. It is an algorithmic computation of the set of unpointed stable homotopy classes of equivariant fibrewise maps that was described in the recent paper. In this paper, it describes a simplification of the prior paper that uses an abelian heap structure. The heap is essentially a group without a choice of its neutral element; in addition, the heap is allowed to be empty [8].

The 6th paper is titled, *The Stability Of The Heap Branch,* and it discusses the stability of the heap by setting the factors that contribute to instability, after which the steady state analysis of talus using several methods of calculation, and in the end, were designed some steps to stabilize tripping or slipping. The stability analysis was performed using the dedicated geotechnical software Slide. After the stability analyses it can be concluded that the heap Coroieşti Branch II is stable when the sterile material is deposited under normal conditions of humidity and may become unstable in terms of increasing the humidity up to saturation, what it is, however, unlikely, considering the large capacity of dispose of water by the stockpiled rocks. [9].

The final 2 research papers are titled, ‘*Linked List Problems,’* and ‘*Linked List Basics.’* The papers goes into great detail on linked, for example it demonstrated basic linked list code techniques and then works through 18 linked list problems covering a wide range of difficulty. It talks to how a linked list has a list of integer elements in order to be built. Linked lists and arrays are similar since they both store collections of data. The terminology is similar as an array and linked lists store elements on behalf of client code. The specific type of element is not important since essentially the same structure works to store elements of any type. Lastly, the papers explain the basic operations for a link while showing the proper way to perform deletion and insertion on that link list [12].

**4 Static Stable Duplicate Max Heap Building**

A binary heap in the form of a two dimensional array, together with a hash table used for memoization, are utilized in tandem to stabilize a heap in a static manner. Our static stable heap build adds one step to the traditional one-by-one insertion method. Before we insert the value into the array and do any sort of percolation, we check our memoization/hash table for value. If, for example, we are inserting the value '8' into our array, we check record 8 in the hash table and column 1 in our hash table. Column 1 contains the number of times that that value appears in our heap. Column 2 contains the next sequence number to be deleted, allowing us to delete as first-in first-out. When inserting a value into the heap, we increment column one and set column 2 equal to 1. Then we insert into the 2-dimensional heap array, both the value and its sequence number.

Once the insertion takes place, percolation might be the next step. Every time we percolate we update our memoization table to maintain the array element of each duplicate. The element number plus 2 of our hash table will correspond with the duplicate number. Meaning that, if I swap my third 8 as I am percolating then I go to record 8, element 5 (3+2) in my memo/hash table and update the value in it to represent the current position in the array. Every swap will require two trips to the hash table.

Figure 4.1 shows an array of integers that have yet to be heapified into a max heap. Figure 4.2 depicts the array once it has been heapified using the aforementioned algorithm; the additional dimension captures the sequence number of each duplicate. Figure 3.3 shows the hash table upon completion of the building of the max heap. Figure 3.4 provides a visual of the build heap process and maintains the duplicate order visually.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| A[i] | 3 | 8 | 1 | 8 | 9 | 8 | 2 |

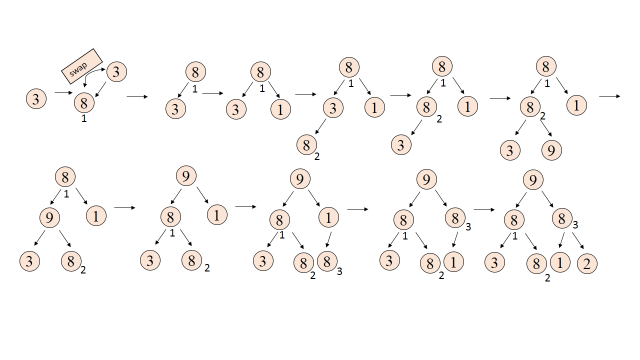
Figure 4.1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| A[i] | 9 | 8 | 8 | 3 | 8 | 1 | 2 |
| # Seq | 1 | 1 | 3 | 1 | 2 | 1 | 1 |

Figure 4.2

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Index of hash |  | Next Item To go | Array position of element 1 | Array position of duplicate 2 | Array position of duplicate 3 | Array position of duplicate 4 |
| 0 |  |  |  |  |  |  |
| 1 | 1 | 1 | 3 |  |  |  |
| 2 | 1 | 1 | 7 |  |  |  |
| 3 | 1 | 1 | 1 |  |  |  |
| 4 |  |  |  |  |  |  |
| 5 |  |  |  |  |  |  |
| 6 |  |  |  |  |  |  |
| 7 |  |  |  |  |  |  |
| 8 | 3 | 1 | 2 | 5 | 3 |  |
| 9 | 1 | 1 | 1 |  |  |  |

Figure 4.3

Figure 4.4

Below is part of the code for a max heap [2]:

public class MaxHeap{

private int[] Heap;

private int size;

private int maxsize;

private static final int FRONT = 1;

public MaxHeap(int maxsize){

this.maxsize = maxsize;

this.size = 0;

Heap = new int[this.maxsize + 1];

Heap[0] = Integer.MAX\_VALUE;

}

private void swap(int fpos,int spos){

int tmp;

tmp = Heap[fpos];

Heap[fpos] = Heap[spos];

Heap[spos] = tmp;

UpdateHash(Heap[fpos], fpos);

UpdateHash(Heap[spos], spos);

}

public void insert(int element){

Heap[++size] = element;

int current = size;

while(Heap[current] > Heap[parent(current)]){

swap(current,parent(current));

current = parent(current);

}

public void updateHash(int val, int idx){

hash[val][1] ++;

int dup = hash[val][1];

hash[val][dup] = idx;

}

}

**5 Static Stable Delete Max**

Every constant time delete will require a check to the memo table, and at most one swap. The check is to determine if the value’s sequence number matches the 'next sequence to go' column in our memo table. If it matches, the value is popped and the corresponding value in our memo table is decremented. If the value at the root (i.e. about to be deleted) is NOT the next to go, then the memo table is used to find the position number in the array of the next to go, and the values are swapped. The element in the memo table is then updated.

After the deletion occurs, we put the last value of the array into the top node position and percolate accordingly, as we c=would with a regular heap.

public int remove(){

int popped = Heap[FRONT];

Heap[FRONT] = Heap[size--];

maxHeapify(FRONT);

return popped;

}

private void maxHeapify(int pos){

if (!isLeaf(pos)){

if ( Heap[pos] < Heap[leftChild(pos)] || Heap[pos] < Heap[rightChild(pos)]){

if (Heap[leftChild(pos)] > Heap[rightChild(pos)]){

swap(pos, leftChild(pos));

maxHeapify(leftChild(pos));

}else

{

swap(pos, rightChild(pos));

maxHeapify(rightChild(pos));

}

}

}

}

private void swap(int fpos,int spos){

int tmp;

tmp = Heap[fpos];

Heap[fpos] = Heap[spos];

Heap[spos] = tmp;

UpdateHash(Heap[fpos], fpos);

UpdateHash(Heap[spos], spos);

}

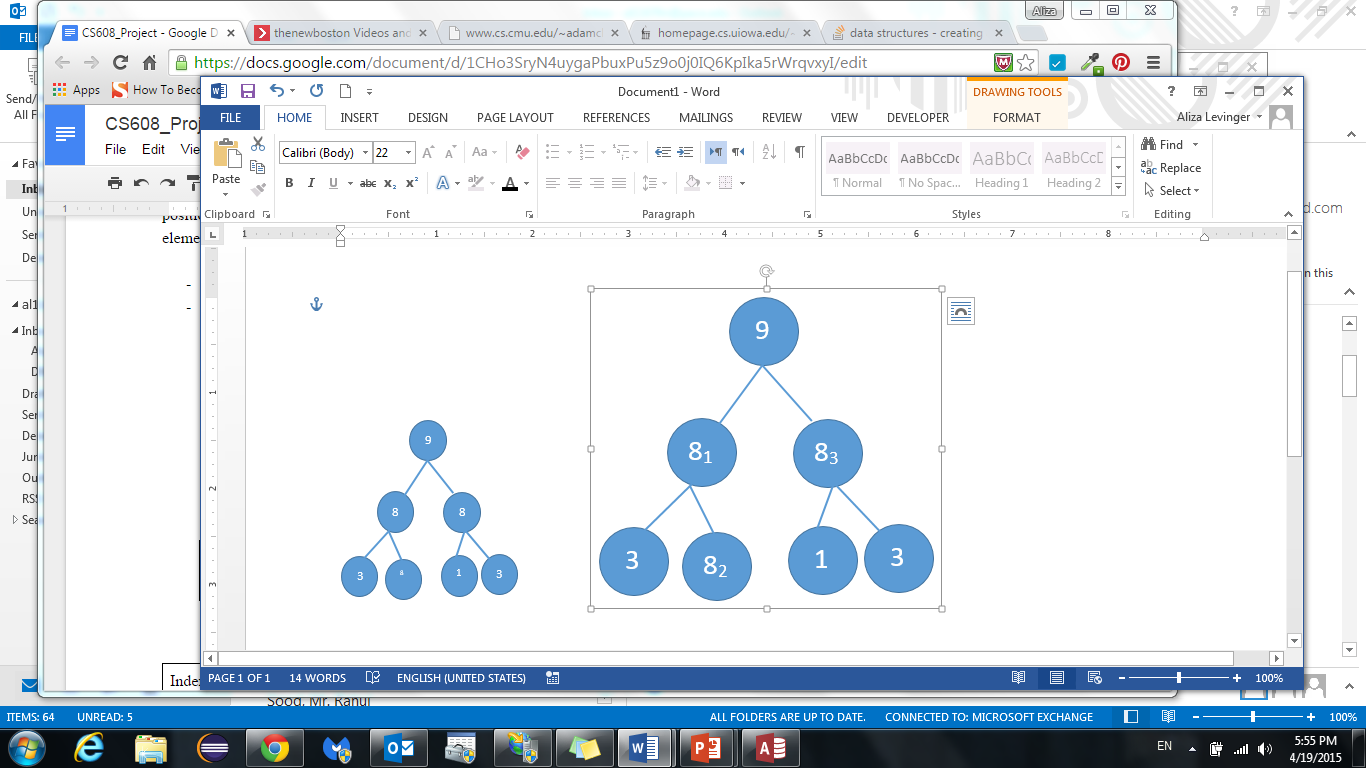
public void updateHash(int val, int idx){

hash[val][1] ++;

int dup = hash[val][1];

hash[val][dup] = idx;

}

Figure 5.1 shows the way we logically view a heap as a binary tree and demonstrates the problem of an unstable heap. Below that, Figure 5.2 and 5.3 show the proposed stable implementation. The two dimensional array stores the values and their frequency (i.e. the number of times the value occurs in the data set) and the hash table enables the tracking of each element and it’s duplicates within the array.

|  |  |
| --- | --- |
|  |  |
|  |  |

Figure 5.1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| A[i] | 9 | 8 | 8 | 3 | 8 | 1 | 2 |
| # Seq | 1 | 1 | 3 | 1 | 2 | 1 | 1 |

Figure 5.2

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Index of hash | Max Seq # | Next Item To go (if using a queue) | 2 (array position of element 1) | 3 (array position of duplicate 2) | 4 (array position of duplicate 3) | 5 (array position of duplicate 4) |
| 0 |  |  |  |  |  |  |
| 1 | 1 |  | 6 |  |  |  |
| 2 | 1 |  | 7 |  |  |  |
| 3 | 1 |  | 4 |  |  |  |
| 4 |  |  |  |  |  |  |
| 5 |  |  |  |  |  |  |
| 6 |  |  |  |  |  |  |
| 7 |  |  |  |  |  |  |
| 8 | 3 |  | 2 | 5 | 3 |  |
| 9 | 1 |  | 1 |  |  |  |

Figure 5.3

**6 Dynamic Stable Duplicate Max Heap Building**

A Binary Search Tree and queue, implemented by a linked list, are utilized in order to dynamically stabilize a max heap. A BST, as depicted in figure 6.1, is a binary tree that maintains a condition known as the Binary Search Tree Property, which states that “all nodes stored in the left subtree of a node whose key value is K have key values less than K. All nodes stored in the right of subtree of a node whose key value is K have key values greater than or equal to K” [1]. The second concept utilized in this dynamic approach is a queue, which we implement using a linked list.

Each node of the aforementioned BST will now be a queue. Each head of the queue will point to the head of its children or childrens’ queue. Each queue will contain any possible duplicates of that value. Inserting into the BST follows the standard approach; the new element is compared with the root to determine the next step, either left or right. Now, if the element is equal to the root in which it is being compared to then it is added to the queue of that node.

A pointer to the maximum value in the BST will be maintained throughout the creation of the BST. The maximum value pointer will follow the value to the most right of the BST, by first being assigned to the root node. If an element is placed only right, then it is assigned the max pointer. Thus we maintain contain access to the maximum value, while also having a means to store duplicates in a stable fashion.

Figure 6.1 depicts a Binary Search tree after being built. Figure 6.2 depicts the same Binary Search Tree with: a pointer to the maximum value of the BST, and a queue for the number 13 that shows all of the duplicates in the order that they were inserted (from left to right).

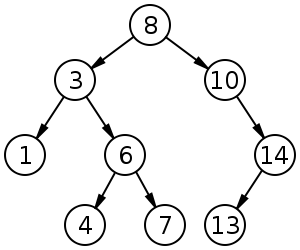


Figure 6.1

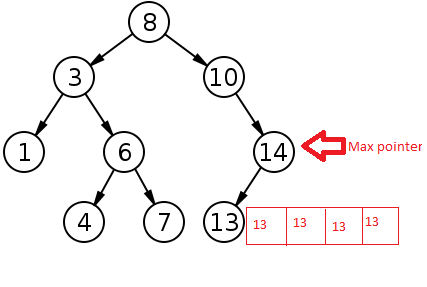


Figure 6.2

public void insert(T data){

root = insert(root, data);

}

private Node<T> insert(Node<T> p, T toInsert){

if (p == null)

return new Node<T>(toInsert);

if (compare(toInsert, p.data) == 0)

return p;

if (compare(toInsert, p.data) < 0)

p.left = insert(p.left, toInsert);

else if (compare(toInsert, p.data) > 0)

p.right = insert(p.right, toInsert);

// Value is equal to node, so add to the queue

else

enqueue(T data);

return p;

}

**7 Dynamic Stable Delete Max**

Maintaining a pointer to the max value in the BST makes finding the maximum value a trivial process. Deleting the maximum value is a three step process. First, we have a pointer to the maximum value, so our first step is to dequeue from the queue of the max value. The second step is reconnecting the tree. We reconnect the tree by pointing the parent of the popped value to the node of the maximum child of the just popped value. Lastly, the max pointer is moved to the popped value’s child; if there are no children then is pointed to the parent of the popped value.

Figure 7.1 shows ‘14,’ the maximum value, being deleted from the Binary Search Tree. ‘10’ is reassigned to point to ‘13,’ the maximum child of ‘14.’ The maximum pointer is then assigned to the next highest value, which, in this case, is ‘13.’ Figure 7.2 shows the psuedo code for this.

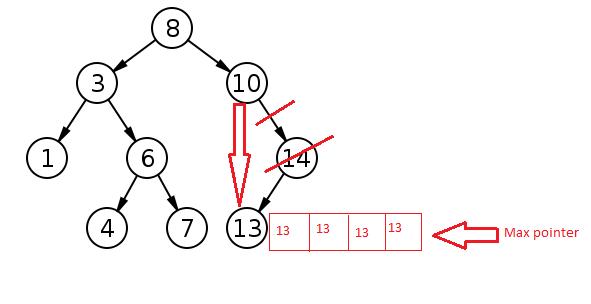


Figure 7.1

public void delete(T toDelete){

root = delete(maxPointer, toDelete);

}

private Node<T> delete(Node<T> p, T toDelete){

// Dequeue one value

Dequeue(toDelete);

if ( p.left == null ) // If true, p has a right subtree

{

/\* ========================================

p is the root

========================================= \*/

if ( p == root ){

root = root.right; // After root is deleted, BST = right tree of root

return;

}

parent = myParent; // myParent was set by findNode(x)....

**// Max Pointer moves to parent, or max child**

if (p.right == NULL && p.left == NULL) // No children

MaxPointer = p.parent;

if (p.right != NULL)

MaxPointer = p.right;

else

MaxPointer = p.left;

/\* ----------------------------------------------

Link p's RIGHT subtree to the parent node

---------------------------------------------- \*/

if ( parent.left == p ){ // Hang p's RIGHT tree to parent.left

parent.left = p.right; // Link p's right subtree to parent left

}

else{ // Hang p's RIGHT tree to parent.right

parent.right = p.right; // Link p's right subtree to parent right

}

return;

}

/\* ==================================================

We must do the same when p has a LEFT subtree

================================================== \*/

if ( p.right == null ) { // If true, p has a left child

if ( p == root ){

root = root.left;

return;

}

parent = myParent; // myParent was set by findNode(x)....

/\* ----------------------------------------------

Link p's left child as p's parent child

---------------------------------------------- \*/

if ( parent.left == p )

parent.left = p.left;

else

parent.right = p.left;

return p;

}

Figure 7.2

**8 Space and Time Complexity Analysis**

**Time complexity for building Static V. Dynamic**

The complexity of building a stable static heap starts with heapify. The maximum number of times a value can be shifted down the tree is equal to the height of the tree, which is log n. Since we are inserting n values, the time complexity for building a heap is O(nlogn). For the static stable heap build, we have to factor in that each swap requires 2 constant-time trips to the memoization hash table. We can potentially have every value in the height of the heap swapped; therefore, the time complexity would be O(nlogn \* 2).

In regards to the dynamic building of the heap, it takes the same amount of time as a regular heap build and as a binary search tree. Values are inserted one a time; at worst, values can be sifted down the entire height of the tree, which is log n. The time complexity for building our dynamic heap is O(logn \*n), which represents sifting down the height of the tree, for n number of times.

.**Time Complexity for deleting Static V. Dynamic**

When deleting from a heap, we always delete from the top, hence the time to access the element with highest priority is constant. However, after deletion the, the root node needs to be replaced; it is replaced with the last node in the heap. Then must percolate down to reheapfiy the heap. In the worst case scenario, the deletion will cause percolation down to the height of the tree. Since the height of the tree is O(logN), the worst running time is O(logN) [11]. For the static build, each potential swap down the tree will require two trips to the memoization hash table twice; therefore, the time complexity is O(logn \* 2), which is the same as O(logn).

The time complexity for deleting from the dynamic heap is O(1), constant time. Each delete requires 3 constant time actions: one dequeue, reassigning the max value, and reattaching the tree by pointing the parent to grandchild. Three constant time actions would make time complexity O(3), which simplifies to O(1).

**Space complexity Static V. Dynamic**

In a heap, the space used is equal to the number of values in the heap, In our static heap, we have to account for the size of the memoization hash table. The space of the memoization hash table is dependent on the maximum value of n (due to direct addressing), and the maximum value of duplicates; therefore the space complexity for our static stable heap is O(n) + MaxValue(n) \* MaxValue(k), with n being the values in the heap and k being the maximum number of duplicates a value has.

In our dynamic heap, we have to account for the size of the heap and the size of any potential linked list queues utilized. The of the heap, with no duplicates, is still O(n). In the worst case scenario, each node would have k number of duplicates; therefore, the space complexity is O(n) + n \* MaxValue(k).

**9 Correctness**

* **Proof of correctness - Loop invariant** must show three parts:

1. Initialization - the loop invariant holds to executing the loop, after initialization.
2. Maintenance - the loop invariant holds at the end of executing the loop.
3. Termination - the loop eventually terminates.

{ **Precondition**: n 1 }

fact(n)

i←1

fact←1

{ invariant P: fact = i! && i n }

while (i < n) do

i ←i + 1

fact←fact \* i

{ Postcondition: fact = n! }

**Initialization**: Prove assertion P true at start of loop:

i ← 1 n ------ by {Precondition: n 1}

fact ← 1 = 1! = i! ------ by definition of 1!

{ P: fact = i! && i n } is true before while.

Loop invariant for while loops:

**Maintenance**: Prove P is true at end of loop:

Assume:

{ P: fact = i! && i n } is true.

test condition **while** *(i < n*)is true.

New values inew and factnew of i and fact are:

inew ← i + 1

factnew ← fact \* (i+1) = (i + 1)! = inew!

since i < n also have:

inew ← i + 1 n

{ P: fact = i! && i n } is true at end of loop since fact = i! && i n.

**Termination:**  Prove that loop terminates.

At beginning:

i ← 1

Assuming the precondition n 1 holds, after n-1 loop traversals of:

i ← i + 1

the new value of i will be:

i = n

terminating the loop.

Loop terminates given that precondition holds:

n 1

{ P: fact = i! && i n } is then a loop variant.

**Postcondition** met:

Since P is loop invariant, by:

if the while terminates, { P: fact = i && i n } is then true

fact = i!

i n

and i < n is false, from Termination.

Hence at loop termination:

i = n

fact = i! = n!

* **Proof of correctness - By Induction**:

Loop invariant proofs can also be performed by induction.

Below the use of Pk and Pk+1 is not necessary but it is helpful for Induction process.

The loop invariant is:

Pk = k! where k n

Pre: n 1

Pn(n)

k := 1

Pk := 1

Pk = k! where kn

while k < n

Pk+1 := Pk \* k+1

k := k+1

Pk := Pk+1

Post: Pn = n!

Show Pk where k n is always true:

**Base step**: P1 = 1! where 1 n

given n 1

**Inductive step**: Pk Pk+1

At the end of the loop show,

Pk+1 = (k+1)! where (k+1) n

Assume Pk = k! where k < n

Pk = 1 \* 2 \* …… \* k

= k!

k < n for loop execution

Pk+1 = 1 \* 2 \* ….. \* k \* k+1

= Pk \* k+1

= k! \* (k+1)

= (k+1)!

k + 1 n k < n assumed true

k + 1 n

Pk+1 = (k+1)! where (k+1) n

After the loop completes

k = n

therefore, Pk = k! where k = n

So, Pn = Pk

**Correctness of Binary Search:**

Let P(n) be the assertion that binary search works correctly for inputs where right - left = n. If we can prove that P(n) is true for all n, then we know that binary search works on all possible arguments.

**Base Case**: In this case when n= 0, we know left=right=m. Since we assumed that function would only be called when x is found between left and right, it must be the case that x=a[m] and therefore the funcion will return m and index of x in array a.

**Inductive Case**: In this we have to prove that left-right = k+1. There are three cases where, x = a[m], where x < a[m] and where x > a[m].

Case x = a[m]. Clearly the function works correctly.

Case x < a[m]. Since the array is sorted that x must be found between a[left] and a[m - 1]. So if the recursive call works properly this call will too. The n for recursive call is n = m-1-left = ⌊(left+right)/2⌋ − 1 − left. If left + right is odd, then n = (left+right - 1)/2-1-left = (right - left)/ 2 - 1 which is definitely smaller than right - left. If left + right is even then n = (left+right)/2 - 1 - left = (right-left)/2, which is also smaller than k + 1 = right - left because right - left = k+1 > 0. So the recursive call must be between 0 and k cells, and must be correct by our induction hypothesis.

Case x>a[m]. We need to show that r − (m + 1) ≤ right − left.We have r − (m + 1) − l = right − ⌊(left + right)/2⌋ − 1. If right+left is even, this is (right−left)/2 − 1, which is less than right−left. If right+left is odd, this is right− (left + right − 1)/2 − 1 = (right−left)/2 − 1/2, which is also less than right−left. Therefore, the recursive call is to a smaller range of the array and can be assumed to work correctly by the induction hypothesis.

Since in all cases the inductive step works, we can conclude that binary search tree (and its iterative variant) are correct.

**8 Conclusion**

Stabilizing the priority queue has a multitude of applications. We examined two implementations of a stable priority queue, one static and one dynamic. The static version applies the concepts of memoization and direct-addressing hash tables to array-based max heaps to provide stabilization, while the dynamic version utilizes linked lists within a binary search tree to achieve this result. The dynamic version allows us flexibility in regards to flow in input; we do not need to know the number of items prior to inserting the values. Space and time complexity are also in favor of the dynamic approach.

**References**

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein. *Introduction to Algorithms*. 3rd ed., Cambridge, Massachusetts: The MIT Press, 2009. Print.[2] Shaffer, Clifford A. *Data Structures & Algorithm Analysis in Java*. 2rd ed. Mineola, New York: Dover Publications, Inc., 2011. Print.s

[3] Knuth, Donald. *The Art of Computer Programming*. 2nd ed. Volume 3. Boston, Massachusetts: Pearson Education, Inc., 1998. 4, 149. Print.

[4] Herman, Ted, Pirwani, Imran. *A Composite Stabilizing Data Structure.* (n.d.): n. pag. *Webdocs*. Department of Computer Science, University of Iowa, 2001. Web.

[5] Herman, Ted, Masuzawa, Toshimitsu. *Available stabilizing heaps*. Information Processing Letters.

77:115-121, 2001.

[6] Ukena, Satoshige, Manabu Hasegawa, Yoshiaki Katayama, Toshimitsu Masuzawa, and Hideo Fujiwara. *A Self-stabilizing Max-heap Protocol in Tree Networks*. Electronics and Communications in Japan. *Part III:* Fundamental Electronic Science*,* 86.9 (2003): 63-72. Web.

[7] Vokrinek, Lukas. *Heaps And Unpointed Stable Homotopy Theory*. Archivum Mathematicum, 50.5 (2014): 323-32.

[8] Vokrinek, Lukas. *Computing The Abelian Heap Of Unpointed Stable Homotopy Classes Of Maps*. Archivum Mathematicum, 49.5 (2013): 359-68.

[9] Arad, Victor Ionica, Cristina,Arad, Victor. *The Stability Of The Heap Branch Ii Coroiești - E.P.C.V.J. Vulcan*. Revista Minelor / Mining Revue, 18.3 (2012): 28-31.

[10] Hugue, Michelle University of Maryland (1998) Heapsort*. Analysis and Partitioning*. Available <http://www.cs.umd.edu/~meesh/351/mount/lectures/lect14-heapsort-analysis-part.pdf> (Accessed 17 April 2015).

[11] Based on Jean-Paul Tremblay and Grant A. Cheston. *Data Structures and Software Development in an Object-Oriented Environment.Java Edition.* Prentice Hall, 2003. Available http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250\_Tremblay/L11-BinHeap.htm. (Accessed 17 April 2015).

[12] Parlant, Nick. "Linked List Problems." Linked List Problems (n.d.): n. pag.Stanford.edu. Stanford, 1998. Web. Available http://cslibrary.stanford.edu/105/LinkedListProblems.pdf and http://cslibrary.stanford.edu/103/LinkedListBasics.pdf ( Accessed 30 April 2015)