Binary Heap

Andy Mee (atm112@uakron.edu)

Abstract

Heaps are data structures organized in such a way as to where data of the highest or lowest value have priority over the rest of the data. As data is removed, the highest or lowest value is removed from the structure. One way of implementing a heap is the use of a complete binary tree. Implementing a heap in such a way is known as a binary heap. Within a binary heap, child nodes will always hold values of lesser or equal priority than the parent node (ie, they hold lesser values in a max heap).

Introduction

The purpose of our study is to determine whether the binary tree implementation of the heap is the ideal implementation or not. That is, in both runtime and memory efficiency, the goal is to determine if the binary tree reigns supreme, in that no matter the problem size, it will have the shortest runtime and take up the least space compared to other implementations of the heap.

Discussion

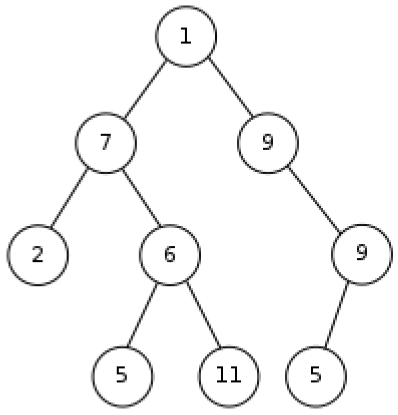
Hypothesis: It is expected that the binary heap will be the more time and space efficient heap than other implementations. Given the nature of a complete binary tree, the height will be big theta of logn. Since many operations on the binary tree will require traversing a path of nodes to the top or bottom of the tree, many worst cases for such operations will be O(logn). Since our binary tree will be a complete tree, this allows us to implement our tree with the underlying container being an array, with nodes stored breadth first, minimizing the amount of overhead in allocated space. Other implementations, such as using an underlying linked list for the container may provide faster operations in some aspects, but the consistency of logarithmic time will be in the favor of our binary heap.

In order to determine whether this is true or not, we will first thoroughly analyze what the binary heap is, what it's supporting operations are, and the complexity of such operations. We will then consider other possible implementations of a heap and compare against them. Measurement of time of like operations will be measured in a program for further comparison.

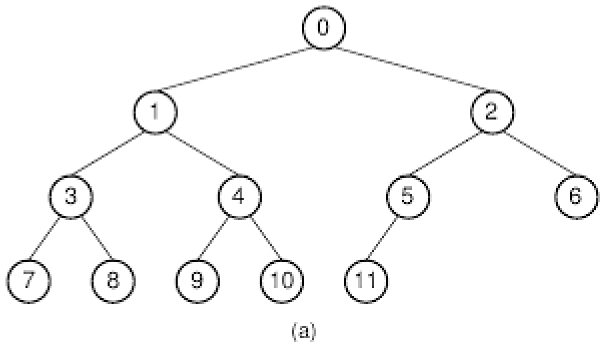
Overview of the binary heap

A heap is a sort of queue, organized in such a way where data with the highest priority, in our case, data with either the highest or lowest value, are always dequeued when it comes time to do so. This is why some refer to the heap as the priority queue. A heap that is organized to prioritize the lowest value is called a min heap. Likewise, a heap prioritizing the highest value is called a max heap. A tree is an abstracted way of organizing data where there is a parent-child relationship between nodes. That is, at the top of the tree, there is the root node, and branching from said root node are child nodes. Every node in a tree, except for the root node, has one parent node. A node can either be a leaf: a node with no children, or an internal node: a node with one or more children. Any node that shares a parent node is considered a sibling node.

In a binary tree, every node has two children. These children can either contain data or simply be null. Null children are not considered when analyzing the data of a binary tree. The open DSA module 05.05.01 defines a complete binary tree as having "a restricted shape obtained by starting at the root and filling the tree by levels from left to right." Every node in a complete binary tree, except for nodes at the bottom level, are completely full. What it means for a binary tree to be full is to have every node either be a leaf node, or be an internal node with exactly two children.



(Image 1: binary tree)



(Image 2: complete binary tree)

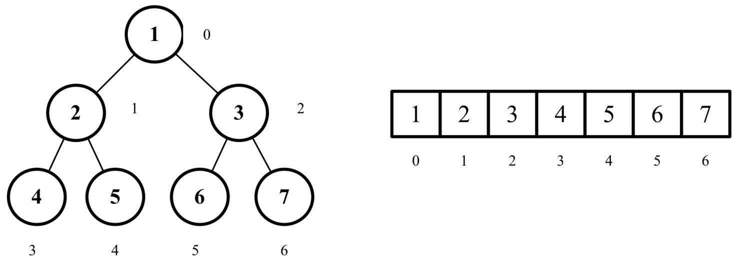
The binary heap is a complete binary tree that follows the simple rule that the children of each parent node must be of lower priority than their respective parent. This ensures that the highest priority element will always be the root of the entire tree. The only item to be removed from the tree at an instance is the highest priority item. After the root is removed, the next highest priority item is promoted to be the root. The heap structure must be maintained after the update. We will see later how this is maintained.

With the restricted shape of our tree, having breadth first insertion, and knowing how many children nodes will have, this allows us to be able to organize our tree with the use of an array. Some trees doing so place the root at index 1 and organize accordingly. In our case, we will have our root start at 0. The arithmetic for finding the index of the left child, right child, and parent nodes will be as follows.

Left child of node n = (n * 2) + 1 (plus 1 to account for root being at 0)

Right child of node n = (n * 2) + 2 (Notice that the right child is one index right of left)

Parent of node n = (n / 2) - 1 (same consideration as children for root being at 0)



(Image 3: array representation)

One question may arise in regards to finding the parent node: What happens if node n is on an odd index, that is, if n is on index 5? Would that not result in 1.5? Given how integer division works in C++, this is not the case. The result of the division will be the lower bound, so the parent of 5 would actually be 1. Having a solid understanding of how to access desired nodes is critical in the implementation of the heap and the operations required of the heap.

ADT and Operations

Here is our current ADT for our heap class: (note: this may not represent the final structure that will be submitted, as modifications may be made, ie new operations)

```
template <typename T>
class BinaryHeap{
        T *data; //underlying array used for heap
        int last; //last available position for heap
        int limit; //limit of allocated space within the heap

        void swapNodes(int, int);//swaps the elements of the two passed index within the underlying array
        public:
        ctrs//
        BinaryHeap()//default ctr
        BinaryHeap(int)//overloaded ctr with specified size for allocation
        BinaryHeap(T*, int, int)//overloaded ctr with an initial list, and size
        //dtr
        ~BinaryHeap();

        //getters
        int getLast();//returns position of last element
        int getLimit();//returns limit of heap

        //node access
        int getParent(int);//returns parent of current node
        int getLeftChild(int);//returns left child of current node
        int getRightChild(int);//returns right child of current node

        //other member functions
        bool isALeaf(int);//determines if the current node is a leaf
        void siftUp(int);//starts at the top of a tree, swaps elements with higher priority child if there is one
        void siftDown(int);//starts at the bottom of a tree, swaps element if current node's value is higher than
parent
        void insertNode(T);//inserts an element within the heap
        T remove();//removes the highest priority in the heap
        void build();//takes the current array in the heap and ensures it is heapified
        void reallocate(int);//takes double the size of the current limit and reallocates more space for the heap,
copying everything over
};
```

While not the same, portions of this implementation are loosely based on a mix of the OSDA(1) module for the heap, and the one seen in U1 ds05 in lecture(2). These structures were studied to get a good idea for how to implement each of the functions used. These modules were used to gain a better understanding of how the logic for some of the member functions work. The data members involved with this implementation are as follows. First, there is a dynamically allocated array used as the underlying container for our heap. Next, there are two integers declared. Last keeps track of the last available location for insert within our heap. Limit is the number of allocated space for our container. No further insertions can be made if last equals limit. Next, we have our member functions.

Being a queue of sorts, our binary heap will need two major operations: Insertion of new elements and removal of the highest priority value. Given that our binary heap needs to maintain the heap structure, supporting operations need to be implemented to simplify these jobs. The sift down function starts at the root of any non-leaf tree within the entire tree, typically the top of the tree, compares the two children of the node, and swaps values of the higher priority child if either of the children have a higher priority than the parent. If a swap was not made, then heap structure is maintained and the function will simply return. If a swap was made, the function moves to the position of the swapped node until either a leaf is reached or heap structure is maintained.

The sift up function is relatively the same idea as sift down, except you are working your way up towards the root. Starting at a node (most likely the last element in the tree), compare the current node with its parent. If it has a higher priority than the parent, they are swapped. If not, then stop. In the event of a swap, the new parent node is then compared with its parent and the same process continues until either the root is reached or heap structure is maintained. Sifting up is made simpler since there is no need to consider siblings of any of the nodes. Just simply the current node and its parent.

Other miscellaneous functions are implemented to simplify other parts of the functions. Swap takes two indexes within the heap and swaps their contents in the underlying array. This function is made private to prevent outside tampering of the heap to maintain heap structure. A boolean function to check if a node is a leaf was also implemented. You can determine if a node is a leaf within the array by determining if the node is on the last level of the heap. All bottom layer nodes will have greater indexes than the parent of the last element on the heap. This node must also be within the range of data within the heap. We have already determined the parent of a node is the node / 2 - 1. Any node that either is or exceeds the last element in the heap is not within our current tree. So in order to be a leaf, both conditions must be true.

The inclusion of these two functions simplifies the process of both insertion and removal into our heap. Starting off with our insert operation, named insertNode in our code, we start off by checking if there is room left in our array to insert a node. If there is not, space must be allocated for any further node to be inserted. If or once there is space, the element being inserted is placed in the first available position. Remember, our data is inserted breadth first, so this will be the current last level, as far left as we can insert it. From here, we sift upwards to ensure our new element finds its proper position. In code, the algorithm will look like this

```
void insertNode(T value){
if (last >= limit){
   reallocate(limit * 2);
   std::cout << "Reallocating space... new limit is: " << limit << '\n';
}

int current = last; set current to the current last item in the heap
++last; //increment the last index of the heap
data[current] = item; // set the last element of the heap to the passed value
          siftUp(current); //sift the last element up to its rightful place
          return;
}
```

Our removal operation is a little more complicated. We cannot simply remove the first element and shift everything over once like we would in a sequential container. This would change the entire structure of our tree, and would need to re heapify the whole thing. Instead, first, we check to see if our heap is empty. If it is, there is nothing else left to do, so we return

```
}
```

NULL. If there are elements in our heap, we first take account of what value is in our root. This will be the value returned from our function. We then decrement the heap size and swap the last element of the heap with the first. This will “remove” the item we are returning from the heap. Realistically, this just sets that value out of the range in our current heap. By the time we reach that slot in the array again, that value will be replaced by the new value. After the swap, the value in the root of the heap will be sifted down to its rightful place. The removal algorithm will look something like this:

```
T remove(){
        if (last <= 0){ //If heap is empty (or somehow below zero)
                std::cout << "Heap is empty, nothing could be removed. Returning null";
                return NULL;
        }
        T root = data[0];
        –last; decrements the position of the last node on the heap
        swapNodes(0, last);//Swaps the root and last node in the heap
        siftDown(0); //Sift down from the top
        return root; //returns the (now former) biggest element of the heap
}
```

Lastly, we have our build function. This takes the current set of items in the underlying array and sifts them around to assure they maintain the heap structure. This is done by starting at the parent of the last node and visiting every node from right to left, sifting down along the way, until the root is reached. We do not need to sift down at root nodes because there is nothing else to sift down from.

The algorithm is as follows:

```
void build(){
        for (int i = (getParent(last)); i >= 0; --i){
                siftDown(i);
        }
}
```

So after discussing what these operations are, what is their time complexity? We'll start by looking at the insert.We'll find that most instructions within the insertNode function are constant, and do not seem to grow with respect to problem size. All of insert's complexity appears to come from the sift up function. You will find, in fact, comes from both of the provided sift functions. So now taking a look at sift up, the main loop starts from the bottom of the tree and traverses its way to the root. Given that it goes nowhere else within the entire tree, at worst, this function traverses the height of the binary tree. According to the OSDA module on heaps, this takes O(logn) time, as the height of the tree is measured as logn, with n being the number of elements within the heap(1).

Conversely, we know that the remove function utilizes the sift down function. After acquiring the root of the tree and swapping the first with the last and decreasing the size, we start at the top of the tree and sift our way down. Worst case scenario, we sift our way all the way to the bottom of the tree. Much like sift up with our insert function, we know that traversing the height of the tree takes O(logn). This means that both major operations of our heap take, at worst, a nice, round, O(logn) time.

We have two ways of building a heap. We could either insert all of our elements with insert one at a time, or we could have an initial list and use our build function to heapify the list. The first method does our insert function n times. This means our first function will take O(nlogn) time to build our initial heap. With our second method, we already have all items we need within the list, it just needs to be heapified. According to the OSDA module, it is known that this building of the heap takes O(n) time(1). We should, however, consider that in our constructor, we copy all elements of the passed array, which also takes n time. While we know that constants do not get accounted for when considering the asymptotic analysis, it’s worth noting that this would be 2n.

Because we are working with an array as our container for this structure, it’s worth noting that our space complexity is linear to the number of elements within our structure. Assuming every allocated space within the array gets used, this also means that there is a potential for no spatial overhead. Looking at these time and space complexities, we can see that they are rather desirable values. However, how does this compare to say other implementations of a heap?

Other implementations of the heap: The linked list:

Another way we can implement our heap is with use of a linked list. We know we can use a linked list to implement a queue. So now all we need to do is put the “priority” in the priority queue. In our last research project, we took a look at the linked list and implemented our own(3). We will repurpose most of this structure and fit it to our needs of being heap. This will entail that we need both an insert function and a remove function. We will not need all of the functions used prior in the previous research project. We just need both the aforementioned insert and remove functions. They both need to be tweaked in order to be implemented as a max heap. The ADT for this heap is as follows:

```
template <class T>

        struct ListNode{

        T value; //Stored value

        ListNode *next; //Link to next node

        };
```

```
template <class T>

class LinkedHeap{

private:

ListNode <T> *head; //head node every list will have

public:

LinkedHeap<T>(){ //default ctr
        head = nullptr;
}

~LinkedHeap<T>(){ //destructor

ListNode<T> *nodePtr = head;

ListNode<T> *nextNode;

while(nodePtr != nullptr){

nextNode = nodePtr->next;

delete nodePtr;

nodePtr = nextNode;
}

}

public:

void insertNode(T); //Inserts node within the heap in descending order, assuring the highest priority value is at the
head at all times

T removeMax(T); //Removes the highest priority element of the heap and returns its value
};
```

As is any linked list, the main data member will be the head. This will point to the series of allocated nodes, which hold data elements of choice.

Our insert function will work as follows. A new node will be allocated. If this is the first element in our heap, this will be the head node. If not, we will start at the head and compare. If this element holds a greater value than the head, then this node will become the head. If not, we will traverse our heap until we either find a node with a value smaller than our new node or we reach the end of the list. This new node is inserted accordingly.

The remove function is simple. First, the list is checked to see if it is empty. If it is, nothing can be removed, so -1 is returned. Next, the list is checked to see if the head is the only element. If it is, the value of head is accounted for, the head is deleted and set to null, and the value is returned. If not, then the next node after the head becomes the head, the head is deleted, and the previous head's value is returned.

Being a linear data structure, there is nothing else that needs to be implemented. The highest priority item will always be at the head, so all you need is to insert items and remove the head. Many would find this implementation easier to understand and implement than the former binary heap. However, this simplicity might come with a few drawbacks.

In order to consider whether or not this implementation performs worse than the binary heap, we must analyze these operations, as well as space complexity. First, looking at our insert function, we know in order to find where to insert our new element, we must traverse the list. Given that this is a linear data structure, there is only one path to traverse, and that straight through it. We do not get the advantage of the heap being divided into two every time like with the binary heap, so at worst, we must traverse through every element in the heap before we find the slot to insert the element. This takes O(n) linear time(3), which is worse than the logarithmic, O(logn) time with our binary heap.

While this may initially seem bad, our other operation, remove, actually fairs better. Since the highest priority element is always at the head of the linked list, no traversal is needed. Simply obtain the value, update the head, erase the old head, and return. This yields O(1) constant time(3), which appears better than the binary heap's O(logn) for removal.

In terms of space, this linked heap appears to shine even less. For every element within our heap, there is also a pointer for the next node in the list. With heaps of the same size, assuming the binary heap utilizes all of its allocated space, the linked heap will have overhead proportionate to the size of the both of them. A pointer in this instance is 4 bytes, so there will be 4n bytes of overhead in the heap.

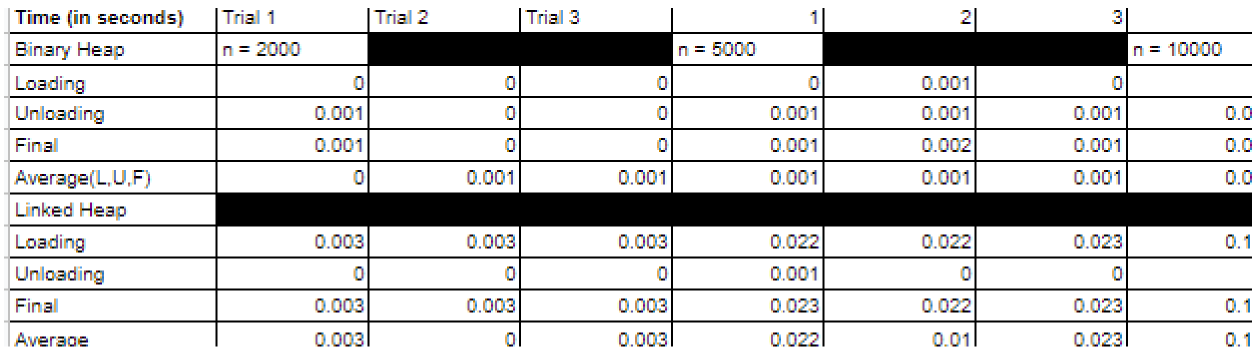
In theory, it appears that the binary heap has a more rounded way of inserting and removing elements, having both be done at O(logn) time. This would appear that this would end up more efficient overall when compared to the linked heap, because despite removal being done at constant time, when inserting all elements within the heap, the linked heap will take O(n^2) vs the binary heap's O(nlogn) (using the binary heaps inferior initialization method). These are because there are n number of insertions, and multiplying the time complexity of both structure's insert by n, we yield these growth rates. However, in order to be entirely sure, we must sufficiently test the two together to assure that case.

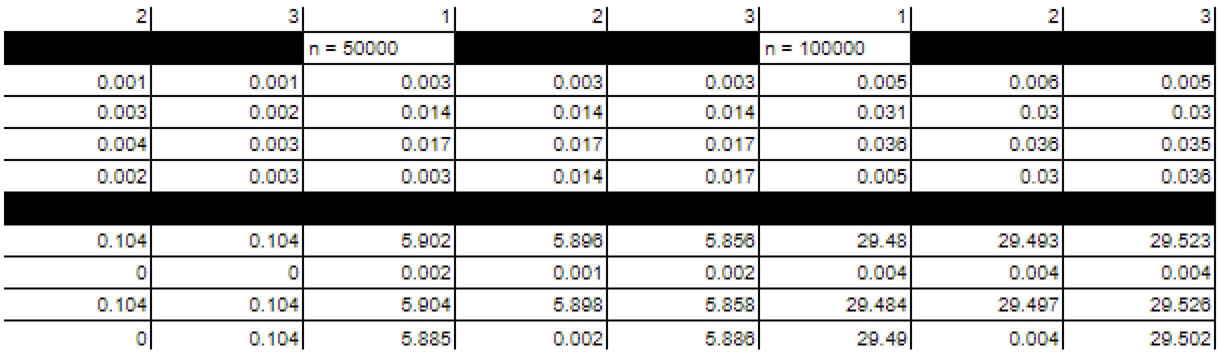
Testing: Binary Heap vs Linked Heap

The test will go as follows: measuring varying heap sizes, both heaps will be loaded with random values and then unloaded. The time it takes to load and unload, and the total time to do both will be recorded for both. This will be done 3 times for each problem size and the average of the 3 will be calculated and recorded, since random values will be inserted in each. These random values will be within the range between zero and the size of the problem, in an attempt to minimize the number of duplicates. Tests will be done with the following sizes: 2000, 5000, 10000, 50000, and 100000.

Results:

After testing every size 3 times, the following results were obtained:





Time (in seconds)	Trial 1	Trial 2	Trial 3	1	2	3		2	3	1	2	3	1	2	3
Binary Heap	n = 2000			n = 5000			n = 10000			n = 50000			n = 100000		
Loading	0	0	0	0	0.001	0		0.001	0.001	0.003	0.003	0.003	0.005	0.006	0.005
Unloading	0.001	0	0	0.001	0.001	0.001	0.0	0.003	0.002	0.014	0.014	0.014	0.031	0.03	0.03
Final	0.001	0	0	0.001	0.002	0.001	0.0	0.004	0.003	0.017	0.017	0.017	0.036	0.036	0.035
Average(L,U,F)	0	0.001	0.001	0.001	0.001	0.001	0.0	0.002	0.003	0.003	0.014	0.017	0.005	0.03	0.036
Linked Heap															
Loading	0.003	0.003	0.003	0.022	0.022	0.023	0.1	0.104	0.104	5.902	5.896	5.856	29.48	29.493	29.523
Unloading	0	0	0	0.001	0	0		0	0	0.002	0.001	0.002	0.004	0.004	0.004
Final	0.003	0.003	0.003	0.023	0.022	0.023	0.1	0.104	0.104	5.904	5.898	5.858	29.484	29.497	29.526
Average	0.003	0	0.003	0.022	0.01	0.023	0.1	0	0.104	5.885	0.002	5.886	29.49	0.004	29.502

(Image split in half to fit page formatting)

As we can see, despite having a relatively fast remove function, the linked queue is drastically held back by its insertion function. Any further testing with larger sizes could potentially result in times in the minutes. In contrast, for the binary heap, this yields a slow and steady growth in both insertion and removal. One thing that was not anticipated was the slightly faster growth when removing from the heap as opposed to insertion. Neither seem to have broken a second worth of time, however, at a certain point, removal very well would have. There is no question now that the use of a binary tree for the heap is superior to that of a linear structure, such as the linked list.

After testing against an inferior method of implementing a heap, how does our heap compare against that of someone else's implementation? Fortunately, another student was kind enough to supply us with their own implementation of a heap.

Comparing Two Binary Heaps

Another experimenter has provided their own implementation of a binary heap for us to test against and compare(4). Theirs is built to be a min heap, but it is expected that this will not make much of a difference compared to ours. This implementation is also not templated, and only accepts integers as the main data type. Their implementation shares many similarities with that of ours. To start, this implementation has 3 data members: the underlying array to store the data, an integer to store the maximum capacity of the entire container, and an integer to keep track of the last available position within the heap. This also has many supporting functions to simplify the implementation of the main operations, such as swap and node accessors. One of the biggest differences worth noting is that their sift down function, named min-heapify, works recursively rather than iteratively. The logic for this is just about the same, except instead of using a loop to traverse down the heap, they recursively call the function until either the bottom has been reached or a value not smaller than the one in the current node. This sifting function is used for their removal function, removeRoot. This recursive implementation may result in a different runtime when comparing these two structures.

(Algorithms for insertion and removal)

```
 void insertNode(int k) {
if(heapSize == capacity) {
   return;
}
else {
    ++heapSize;
    int i = heapSize - 1;
    ArrA[i] = k;

    while(i != 0 && ArrA[parent(i)] > ArrA[i]) {
        swap(ArrA[i], ArrA[parent(i)]);
        i = parent(i);
    }
}
}

int removeRoot() {
if (heapSize <= 0) {
    return 0;
}
if (heapSize == 1) {
    --heapSize;
    return ArrA[0];
}
else {
int root = ArrA[0];
ArrA[0] = ArrA[heapSize-1];
--heapSize;
minHeapify(0);
return root;
}
}
```

There are a couple unique functions within their data structure, such as a linear search and a display function. However, within our current implementation, we have nothing to compare against this, so we are not concerned with these functions.

Testing the two binary heaps

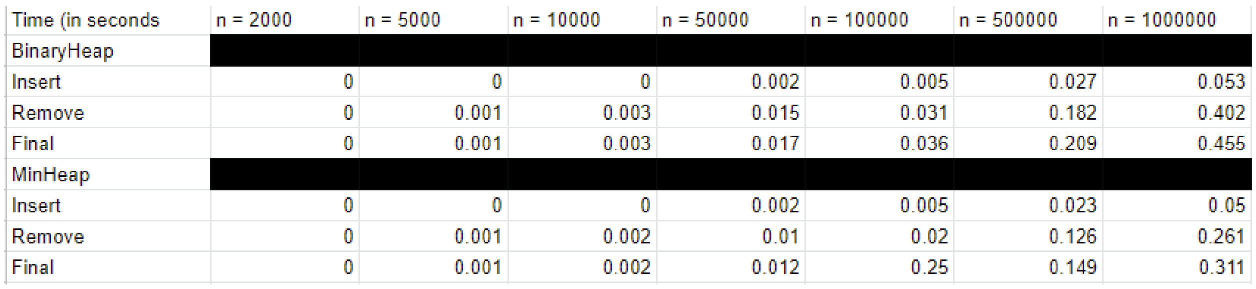
Much like how we tested our binary heap against that of the linked heap, we will be measuring the time it takes to both load and unload both heaps to compare how fast the two complete each task. Given that the other heap also potentially traverses the height of the tree with both the insert and remove functions, it can be reasonably explained that this heap inserts and removes at O(logn) time, and their runtimes will bear minimal differences. As mentioned before, the recursive implementation may be the only grounds for differing times, but this remains to be seen

Differing problem sizes will also be used to gauge if there is a difference in growth between these two structures. Since both of these structures are expected to operate in logarithmic time, a wider range of elements can be measured.

Like with the linked heap, a random selection of n items will be inserted into both heaps. These same items will all be removed. The time it takes to insert and remove said items will be recorded. The following increments of problem sizes will be used: 2000, 5000, 10000, 50000, 100000, 500000, and 1000000.

Results

After testing, we receive the following results:



Time (in seconds	n = 2000	n = 5000	n = 10000	n = 50000	n = 100000	n = 500000	n = 1000000
BinaryHeap							
Insert	0	0	0	0.002	0.005	0.027	0.053
Remove	0	0.001	0.003	0.015	0.031	0.182	0.402
Final	0	0.001	0.003	0.017	0.036	0.209	0.455
MinHeap							
Insert	0	0	0	0.002	0.005	0.023	0.05
Remove	0	0.001	0.002	0.01	0.02	0.126	0.261
Final	0	0.001	0.002	0.012	0.25	0.149	0.311

As anticipated, the differences in these results were minimal at best. The other experimenter, however, appears to have a slightly faster implementation of the heap. As also predicted, it would seem the remove function, the one that uses recursive logic to traverse, yielded a (marginally) faster result.

In terms of spatial complexity, there really should not be any difference. Since they both use an underlying array for their containers, the space complexity is linear with proportion to the problem size, so long as all elements within the array are utilized. There is no overhead in regards to pointers since these structures do use any to access other nodes.

We can safely conclude that our implementation of the binary heap yields nearly identical performance to that of our fellow experimenters.

Possible improvements to current implementations

After building and testing our binary heap, we can begin to consider possible improvements or extra functionality to be added to our heap. An idea that came to mind is reimplementation of our heap to allow it to either be implemented as a max heap or min heap, and be able to change implementations on the spot.

In order to go about this, we should rethink how our operations and member functions work. We could implement two versions of functions that rely on comparison of nodes. But how would our structure know which function to call?

We could add one more data member to our data structure. This will be a boolean value that determines whether our heap is a max heap or not. These min and max variants of these member functions will be made private, so they cannot be accessed outside of the structure itself. The user should still only be able to call insert or remove from the outside.What the user is actually calling is a helper function, which will check the object's "max" boolean, and call the appropriate version of the function. This will assure that depending on implementation, whether max or min, our heap will maintain the respective structure.

We can take this logic a step further with our initializing structure. In conjunction with the initial list passed to the constructor, a string can also be passed, determining which version of the heap the user wishes to implement. These strings will either be “max” or “min” respectively. If the user passes a string that is neither of the two required strings, the default implementation will be a max heap. The appropriate call to the build function will be called to heapify our list.

At any point, if the user wishes to change the type of heap on the spot, the user can simply call the reverse function, and the heap will be re-heapified as the opposite heap.

When all put together, the new ADT for our heap is as follows:

```
template <typename T>
class BinaryHeap{
        T *data; //underlying array used for heap
        int last; //last available position for heap
        int limit; //limit of allocated space within the heap
        bool max;//governing boolean which determines which versions of member functions are called

        void swapNodes(int, int);//swaps the elements of the two passed index within the underlying array

        //Variants
        void siftDownMax(int);
        voidsiftDownMin(int);
        void SiftUpMax(int);
        void SiftUpMin(int);
        void insertNodeMax(int);
        void inserNodeMin(int);
        T removeMax();
        T removeMin();
        void buildMax();
        void buildMin();

        public:
        ctrs//
        BinaryHeap()//default ctr
```

```
	BinaryHeap(int)//overloaded ctr with specified size for allocation
	BinaryHeap(T*, int, int, char*)//overloaded ctr with an initial list, and size. “min” for min heap, “max” for
max
	//dtr
	~BinaryHeap();

	//getters
	int getLast();//returns position of last element
	int getLimit();//returns limit of heap

	//node access
	int getParent(int);//returns parent of current node
	int getLeftChild(int);//returns left child of current node
	int getRightChild(int);//returns right child of current node

	//other member functions
	bool isALeaf(int);//determines if the current node is a leaf
	void siftUp(int);//starts at the top of a tree, swaps elements with higher priority child if there is one
	void siftDown(int);//starts at the bottom of a tree, swaps element if current node's value is higher priority
than parent
	void insertNode(T);//inserts an element within the heap
	T remove();//removes the highest priority in the heap
	void build();//takes the current array in the heap and ensures it is heapified to the proper version
	void reallocate(int);//takes double the size of the current limit and reallocates more space for the heap,
copying everything over
};
```

What could be the use for such an implementation? Such use would be the need to reverse the priority at any instance. Perhaps you are implementing a heap sort on some set of data, and depending on certain conditions, would either require ascending or descending order. Or perhaps some condition within code requires the priority of every member in the heap to be turned on its head. Quite simply, it would be nice to have a binary heap capable of being implemented as either. Whatever that need may be, this binary heap now possesses the possibility to be both within any given time.

Conclusion

Through our research, we have studied the behavior of the binary heap. We have gained an understanding of how an array based structure may be implemented. Through the use of empirical analysis, no matter how the binary tree heap may be implemented, through empirical analysis, we have determined that the complete tree structure for our heap is the favorable implementation of a priority queue in both time and space complexity. After our research, we have determined a way to implement such a heap to be either a max heap or a min heap. Despite having removal in potentially linear time, insertion onto a linear version of a priority queue proves to be taxing and an inferior method of insertion at O(n) time, as opposed to O(logn)time. While there is potential for other tree based implementations to yield faster results, it is safe to say that this implementation of a heap is one of the superior implementations.

Sources

1) OSDA module 05.05.05 Heaps and Priority Queues
    a) https://canvas.instructure.com/courses/4088313/assignments/27651691
2) Unit 1 DS05 Binary Tree Slides
    a) https://brightspace.uakron.edu/d2l/le/content/4523529/viewContent/7137204/View
3) Data Structures Project 1: Linked List - Andy Mee,. atm112@uakron.edu
4) Brandon Kulmala's Implementation of the MinHeap - Brandon Kulmala, bjk74@uakron.edu

External image sources

Image 1: https://en.wikipedia.org/wiki/Binary_tree

Image 2: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/CompleteTree.html

Image 3: https://www.oreilly.com/library/view/learning-javascript-data/9781788623872/a9269219-c494-4c63-a68b-e8db1ec12e45.xhtml