Report

Binary Heap:

Root:0 parent:$\frac{(i-1)}{2}$ left_child:$2 \times i + 1$ right_child:$2 \times i + 2$

Min: Build the min binary heap, so the minimum of the heap is the root, return the value of root. The complexity is O(1).

Max: all parent nodes in a minimal binary heap are smaller than their children, the maximum value will be found at the node with no children, so look up from the last node in the binary heap. Set the last node to the maximum value and compare it with the previous node, swap it with the previous node if it is smaller, and repeat until the node has children. The number of node that has no children is (n+1)/2, So the complexity is O(n).

Search: iterate through the binary heap array and return the array index if the value to be found exists in the binary heap, or print that the number was not found if it does not exist. The complexity is O(n).

Insert: Firstly the inserted node is placed at the end of the array, secondly the length of the array is recalculated and the index of the inserted node is obtained. The inserted node is compared with the parent node, if the inserted node is smaller than the parent node then it is continuously compared to the parent node up to the root node. If the inserted node is larger than the parent node, the insertion is complete. The height of the binary heap is logn. So the complexity of O(logn).

Extract: Firstly use the search function to find the index of the array of values to be extracted. The node to be extracted is swapped with the last node and then deleted using the pop function. In a minimal binary heap, each parent node is smaller than a child node. After extracting , recalculating the size of the binary heap and use the swapdown function on a binary heap makes the heap maintain the properties of a binary heap.

In the swapdown function, When a child node is smaller than its parent, its position is swapped with that of the parent node until it reach the last node. There are two cases of children of a node, one where a left child exists and one where both left and right children exist. First when only the left child node exists, only it will be compared with the parent node. When both left and right children are present, first compare the values of the left and right children and compare the smaller child node with the parent node. When a child node is smaller than its parent, the positions are swapped and the index of the current node is transferred to the position of its child node. The height of the binary heap is logn. So the complexity of O(logn).

Beap:

Root: 0; the height of the node i is $h = \sqrt{2i + \frac{3}{4}} - \frac{1}{2}$ ↑; left_parent: $i - h$; right_parent: $i - h + 1$; left_child: $i + h$; right_child: $i + h + 1$; left range is $\frac{h \times (h-1)}{2}$;right range is $\frac{h \times (h+1)}{2} - 1$

Min: Build the beap, so the minimum of the beap is the root, return the value of root. The complexity is O(1).

Max: Since all parent nodes in a beap are smaller than their children, the maximum value will be found at the node with no children, so look up from the last node in the beap. Set the last node to the maximum value and compare it with the previous node, swap it with the previous node if it is smaller, and repeat until the node has children. The number of nodes that has no child is $\approx \sqrt{2n}$, So the complexity is O($\sqrt{n}$).

Search: The search starts at the first node of the last level. This node is found using the getheight() and getrange().

When the value to be found is smaller than the current node, it goes to the right parent node of the current node. Since the current node is the left parent node of the right child node in the next layer, only the right parent node is entered for comparison during the search.

When the value to be found is larger than the current node, there are two cases. When the right child of the current node exists, the current node goes to the right child. When the right child node of the current node does not exist, the index range of the level above the current node is obtained and the current node enters its right parent node. At this point there is the possibility that the current node enters its right parent node out of the index range, meaning that the node is the last node searched, i.e. the value to be found does not exist in the beap.

When the search reaches the value to be found, return its index to end the loop. Or exit the loop when the current node is smaller than the root node or when the current node is the last node to be searched and the value is still not found. When the node is searched upwards to the root node, the index value is returned in the previous round if the root node matches the value to be searched. Since the right parent node of the root node is still denoted as 0, when the index of 0 appears for the second time, the root node has been searched and the value to be searched for is not found, then the loop is exited. The worst-case search complexity is$2\sqrt{2n}$, so the complexity is O($\sqrt{2n}$).

Insert: Firstly the inserted node is placed at the end of the array, secondly the length of the array is recalculated and the index of the inserted node is obtained. The inserted node need to compare with the parent node, If the inserted node is larger than the parent node, the insertion is complete. If not, there are three situation for the node. Firstly we obtain the height of the node by using the formula above. Secondly we calculate the index range of the node on the previous level by using the formula above.

If its left parent node is smaller than the left border, it is compared only with its right parent node, and if its right parent node is larger than the right border it is compared only with its left parent node. If both left and right parent nodes are exist, the comparison is made with the larger of the nodes. If the inserted node is smaller than the parent node then it is continuously compared to the parent node up to the root node. The height of the beap is $\sqrt{2n}$. So the complexity of O($\sqrt{n}$).

Extract: Firstly use the search function to find the index of the array of values to be extracted. The node to be extracted is swapped with the last node and then deleted using the pop function. In a beap, each parent node is smaller than a child node. After extracting , recalculating the size of the beap and use the swapdown function on a beap makes the beap maintain the properties.

In the swapdown function, When a child node is smaller than its parent, its position is swapped with that of the parent node until it reach the last node. There are two cases of children of a node, one where a left child exists and one where both left and right children exist. First when only the left child node exists, only it will be compared with the parent node. When both left and right children are present, first compare the values of the left and right children and compare the smaller child node with the parent node. When a

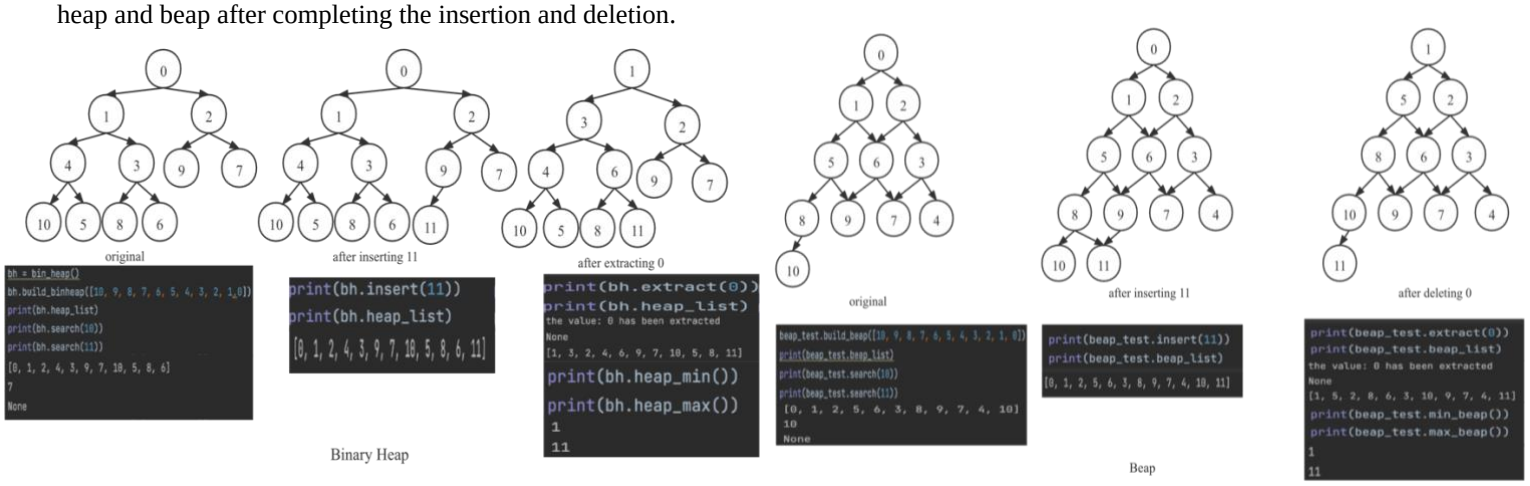
child node is smaller than its parent, the positions are swapped and the index of the current node is transferred to the position of its child node. The height of the beap is $\sqrt{2n}$. So the complexity of O($\sqrt{n}$).

TEST:

The build function is constructed separately in the binary heap and beap, i.e. the heap is obtained by inserting one by one.

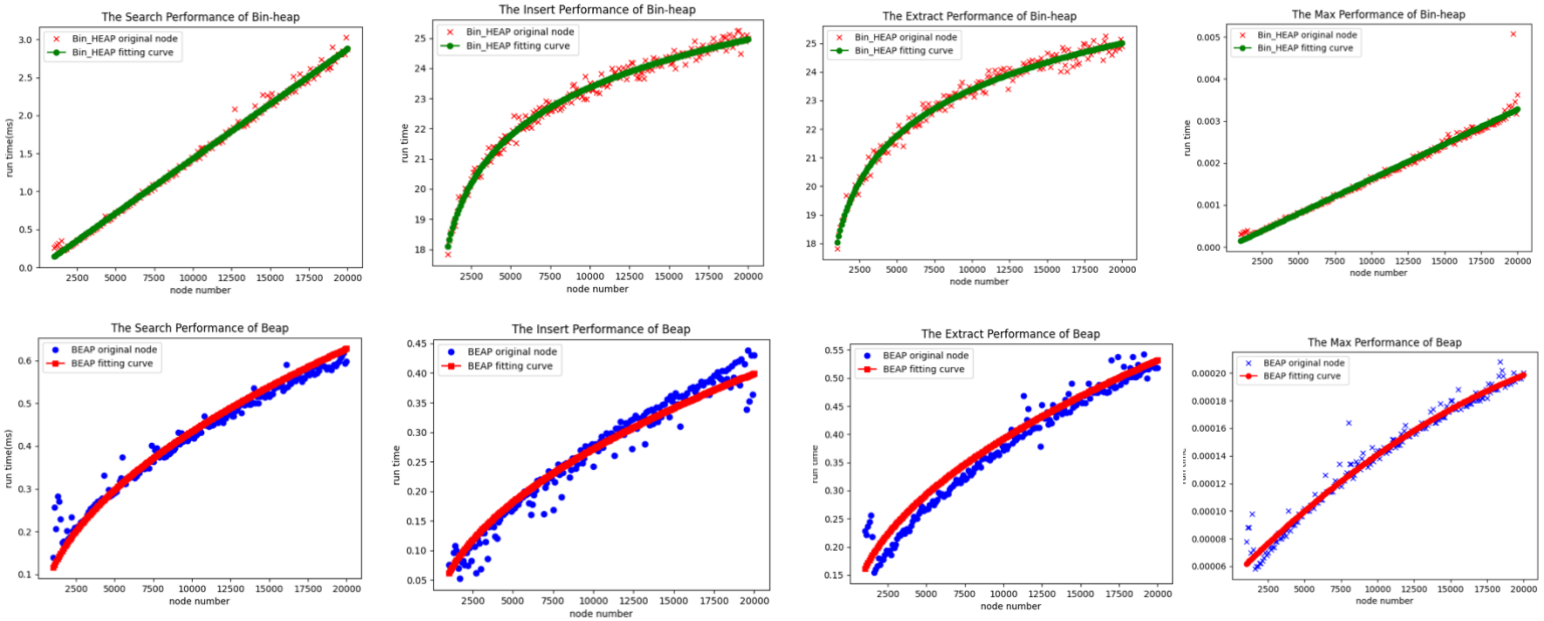
Functional implementation testing

Functional tests with fixed values:[10, 9, 8, 7, 6, 5, 4, 3, 2, 1,0], Use the checker function to check the correctness of the bianry heap and beap after completing the insertion and deletion.



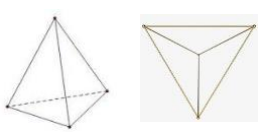
Extensive testing and complexity performance

In order to test the complexity of the different operations of the two algorithm, first set the number of nodes n in two heaps and use the random function to get the value of the n number to be inserted at random. Testing the two algorithms for finding the maximum value, searching, inserting and deleting at different numbers of nodes, Set the number of nodes to 1000-20,000 and the step size to 100 for each growth, and generate fitted curves through the Scattered Points. The insertion operation is setting $n/2$ as the insert value . In the delete operation, the root is selected for deletion. In the search process, for binary heap, the worst case scenario is to search for the last node. For beap, if the last layer is full, the worst case is to search for the last node, and if the last layer is not full, the worst case search node is the last node of the penultimate layer.



In summary, it can be seen that the time consumption trends of the individual operations of binary heap and beap satisfy the complexity of the operations of the two algorithms respectively

3-eaps:



I think of 3-eaps like a triangular pyramid, where a node with three child nodes is like a positive triangular pyramid and a node with three parent nodes is like an inverted triangular pyramid. The full number of each layer is 1,3,6,10,etc. so the operation and complexity is as follows:

Operation	Min	Max	Search	Insert	Delete
Complexity	1	O($\sqrt[3]{n}$)	O($\sqrt[3]{n}$)	O($\sqrt[3]{n}$)	O($\sqrt[3]{n}$)