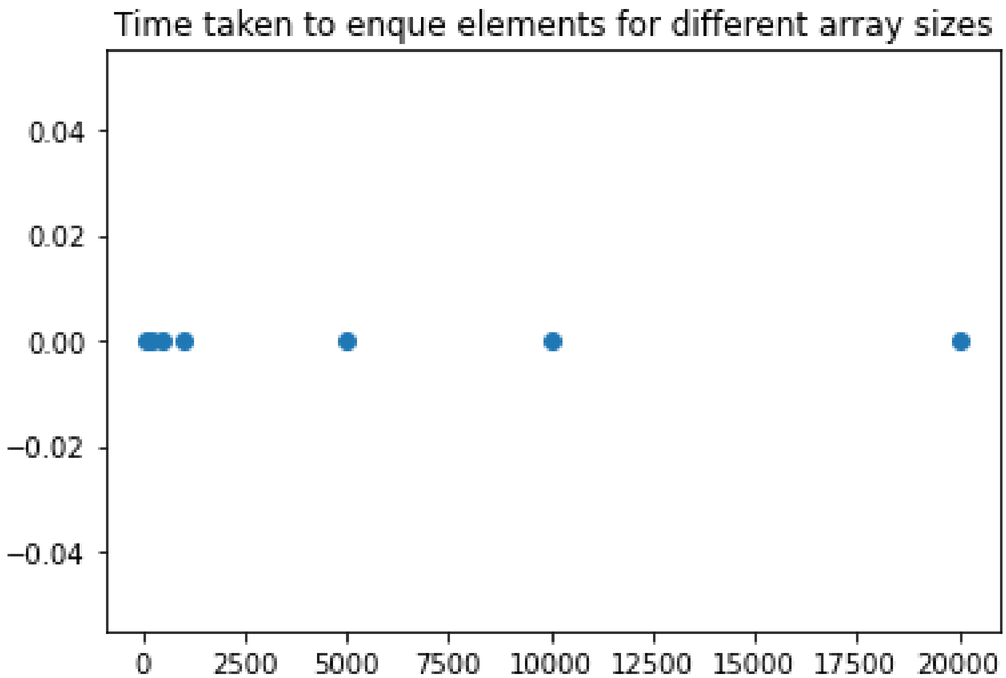
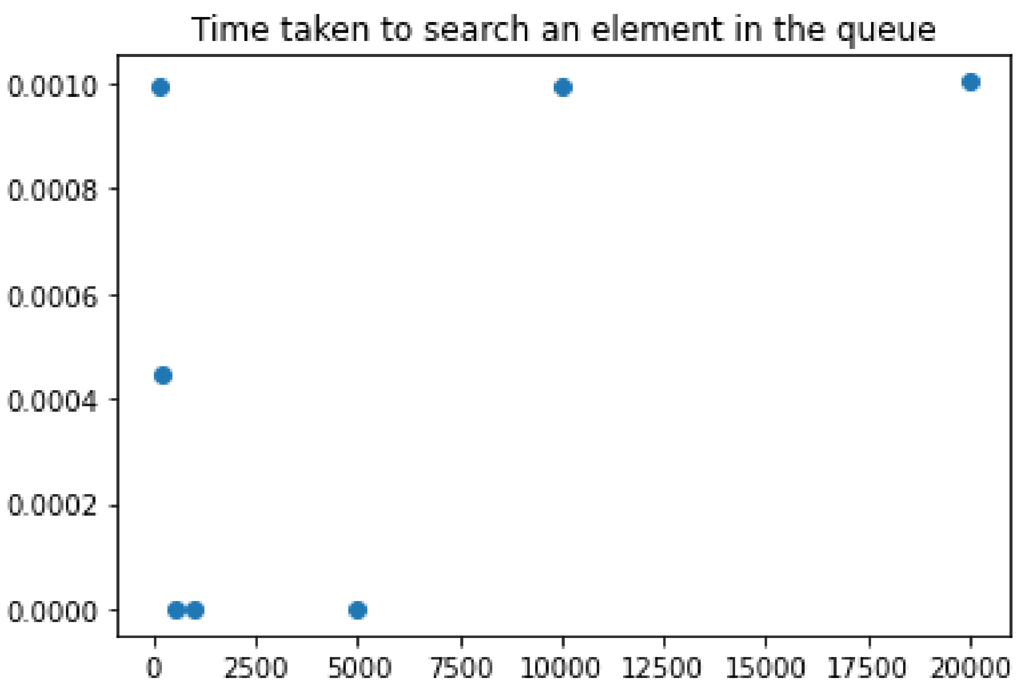
Queues

```
import matplotlib.pyplot as plt

import random as rn
import time

def enque (arr,x):
    arr.append(x)
    #return arr

def deque (arr):

    if (len(arr)==0):
        print("Queue is empty")
        return

    else:
        for i in range (len(arr)-1):
            arr[i]=arr[i+1]
        arr.pop()
        #return arr



size=[100,200,500,1000,5000,10000,20000]
timeEnque=[]
timeDeque=[]

for j in range (len(size)):
        queue=[]
        for i in range (size[j]):
            queue.append(rn.randint(1,20))
        start=time.time()
        enque(queue,25)
        stop=time.time()
        timeEnque.append(stop-start)

        start=time.time()
        deque(queue)
        timeDeque.append(stop-start)




plt.scatter(size,timeEnque)
plt.title('Time taken to enque elements for different array sizes')
plt.show()
```




```
def searchQueue(arr,x):
    k=-1
    for i in range (len(arr)):
        if (arr[i]==x):
            k=i
            break

    if (k==-1):
        print("The element is not present in the queue")
        return

    else:
        print("The element is present at the index ",k)
        return

timeSearch=[]

for j in range (len(size)):
    queue=[]
    queue=[]
    for i in range (size[j]-1):
        queue.append(rn.randint(1,20))

    queue.append(25)

    start=time.time()
    searchQueue(queue,25)
    stop=time.time()
```

```
    timeSearch.append(stop-start)

print(timeSearch)

The element is present at the index  99
The element is present at the index  199
The element is present at the index  499
The element is present at the index  999
The element is present at the index  4999
The element is present at the index  9999
The element is present at the index  19999
[0.0009968280792236328, 0.0004448890686035156, 0.0, 0.0, 0.0, 
0.0009963512420654297, 0.001003265380859375]

plt.scatter(size,timeSearch)
plt.title('Time taken to search an element in the queue')
plt.show()
```



Time complexity

1) For insertion or enque the element is added directly to the end of the queue. Hence the time complexity for enque is O(1).

2) For deletion or dequeing the element at 0 index is removed and the other n-1 elements are all shifted to the left by 1 index. Hence the time taken for deletion is O(n).

3) For searching, the search technique is similar to that of linear search i.e. all the elements are checked one by one till the element desired is obtained. Hence the time complexity of searching is O(n)

Double ended queues

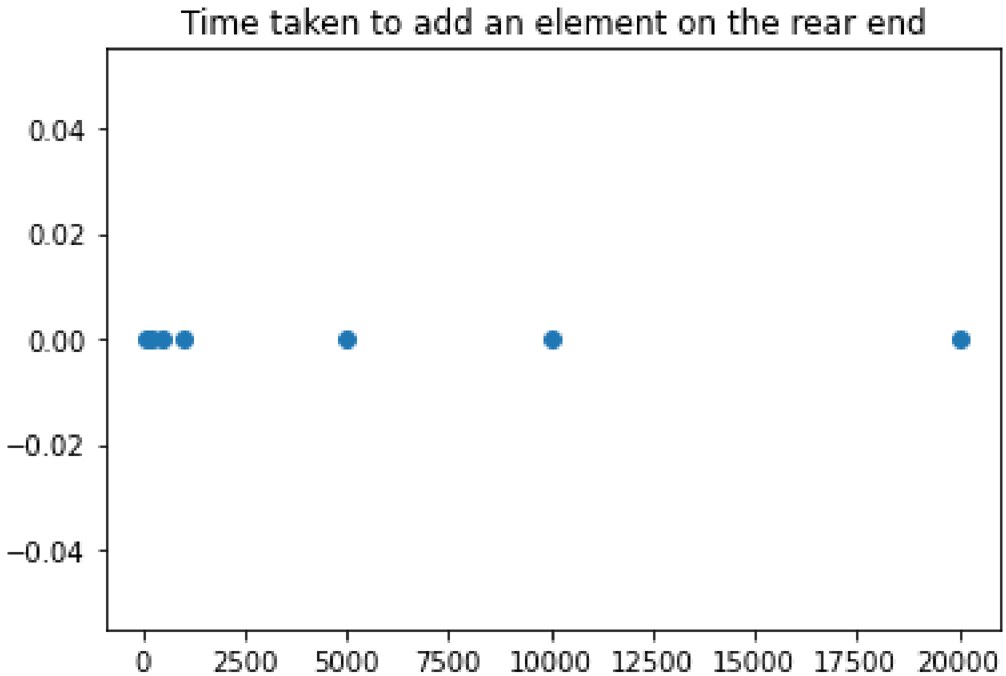
```
def pushBack (arr,x):
    arr.append(x)
    #return arr

def popBack(arr):
    if (len(arr)==0):
        print("queue is empty")
        return
    else:
        arr.pop()
        #return arr

def pushFront(arr,x):
    if (len(arr)==0):
        arr.append(x)
        return
    else:
        temp=arr[len(arr)-1]
        for i in range (len(arr)-1):
            arr[i+1]=arr[i]

        arr.append(temp)
        arr[0]=x
        #return arr

def popFront(arr):
    if (len(arr)==0):
        print("Queue is empty")
        return

    else:
        for i in range (len(arr)-1):
            arr[i]=arr[i+1]
        arr.pop()
        #return arr



def searchQueue(arr,x):
    k=-1
    for i in range (len(arr)):
        if (arr[i]==x):
            k=i
            break

    if (k==-1):
```

```
        print("The element is not present in the queue")
        return

    else:
        print("The element is present at the index ",k)
        return

timePushBack=[]
timePushFront=[]
timePopBack=[]
timePopFront=[]
timeSearch=[]

for j in range (len(size)):
    queue=[]
    for i in range (size[j]):
        queue.append(rn.randint(1,20))

    start=time.time()
    pushBack(queue,25)
    stop=time.time()
    timePushBack.append(stop-start)

    start=time.time()
    popBack(queue)
    stop=time.time()
    timePopBack.append(stop-start)

    start=time.time()
    pushFront(queue,25)
    stop=time.time()
    timePushFront.append(stop-start)

    start=time.time()
    popFront(queue)
    stop=time.time()
    timePopFront.append(stop-start)

    queue.append(25)

    start=time.time()
    searchQueue(queue,25)
    stop=time.time()
    timeSearch.append(stop-start)
```

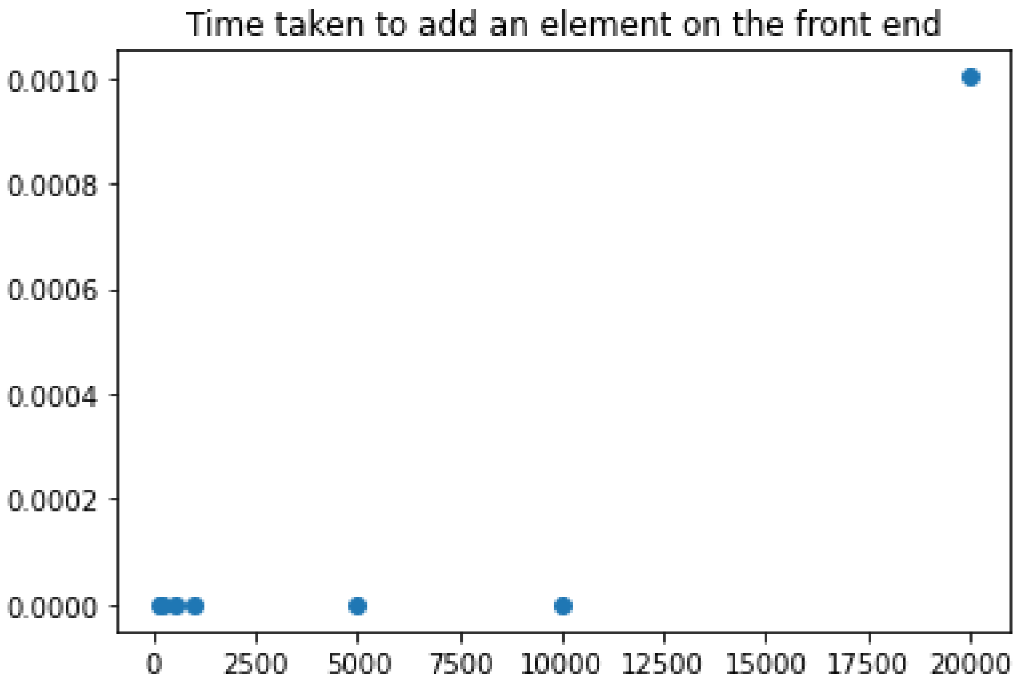
```
The element is present at the index  100
The element is present at the index  200
The element is present at the index  500
The element is present at the index  1000
The element is present at the index  5000
```

```
The element is present at the index  10000
The element is present at the index  20000
```

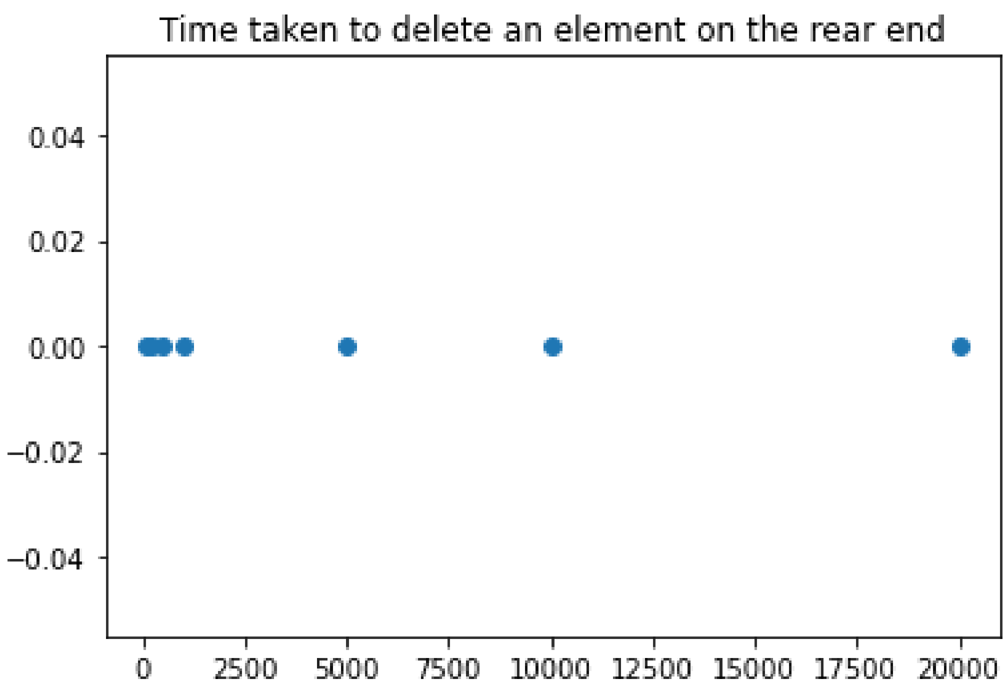
```python
plt.scatter(size,timePushBack)
plt.title('Time taken to add an element on the rear end')
plt.show()
```



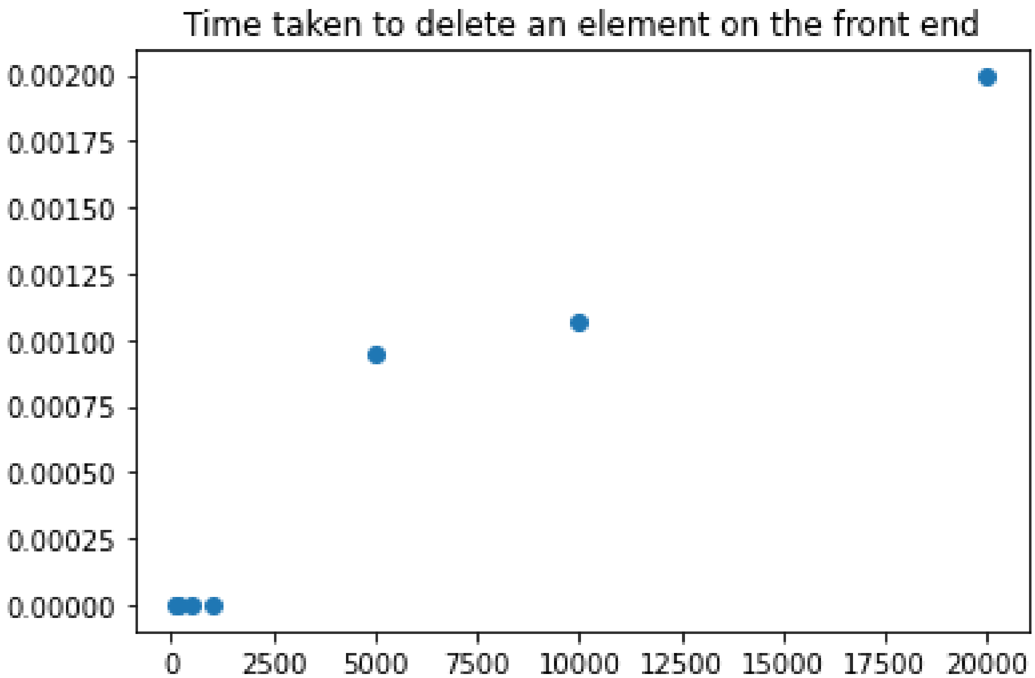

```python
plt.scatter(size,timePushFront)
plt.title('Time taken to add an element on the front end')
plt.show()
```



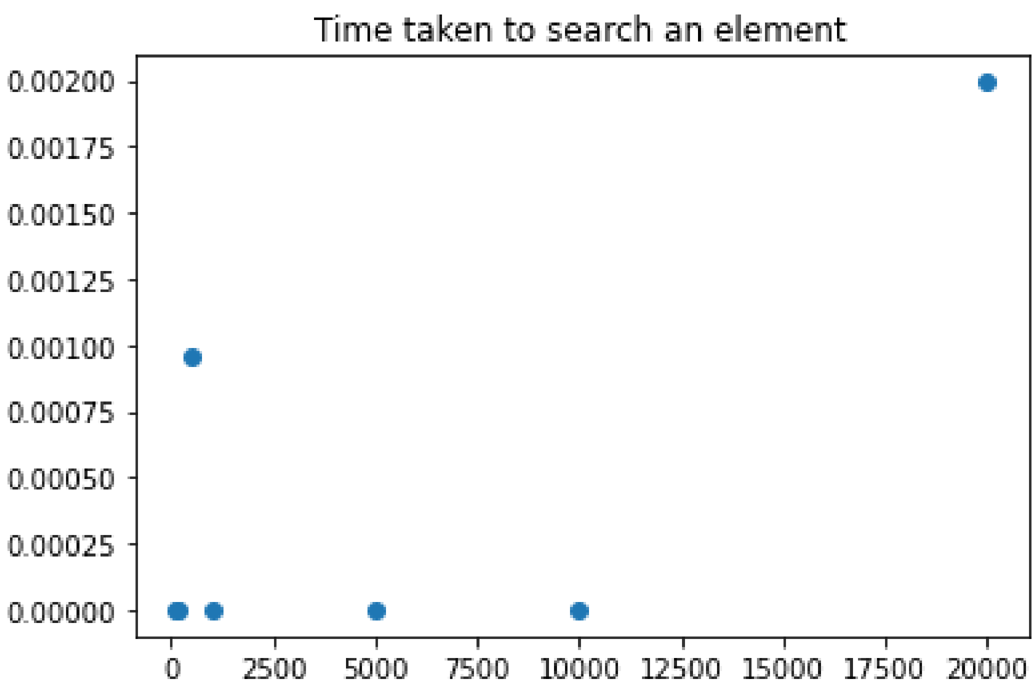

```
plt.scatter(size,timePopBack)
plt.title('Time taken to delete an element on the rear end')
plt.show()
```




```
plt.scatter(size,timePopFront)
plt.title('Time taken to delete an element on the front end')
plt.show()
```




```
plt.scatter(size,timeSearch)
plt.title('Time taken to search an element')
plt.show()
```



Time Complexity

1) For push back we add the elements from the rear side and hence the complexity is O(1)

2) For pop back we delete the elements from the back and hence the complexity is O(1)

3) For push front we shift the n elements to the right by 1 index and add the element in the front. Hence the complexity is O(n)

4) For pop front we delete the first element and shift the n elements to the left by 1 index . Hence the time complexity is O(n)

5) For searching, the search technique is similar to that of linear search i.e. all the elements are checked one by one till the element desired is obtained. Hence the time complexity of searching is O(n)

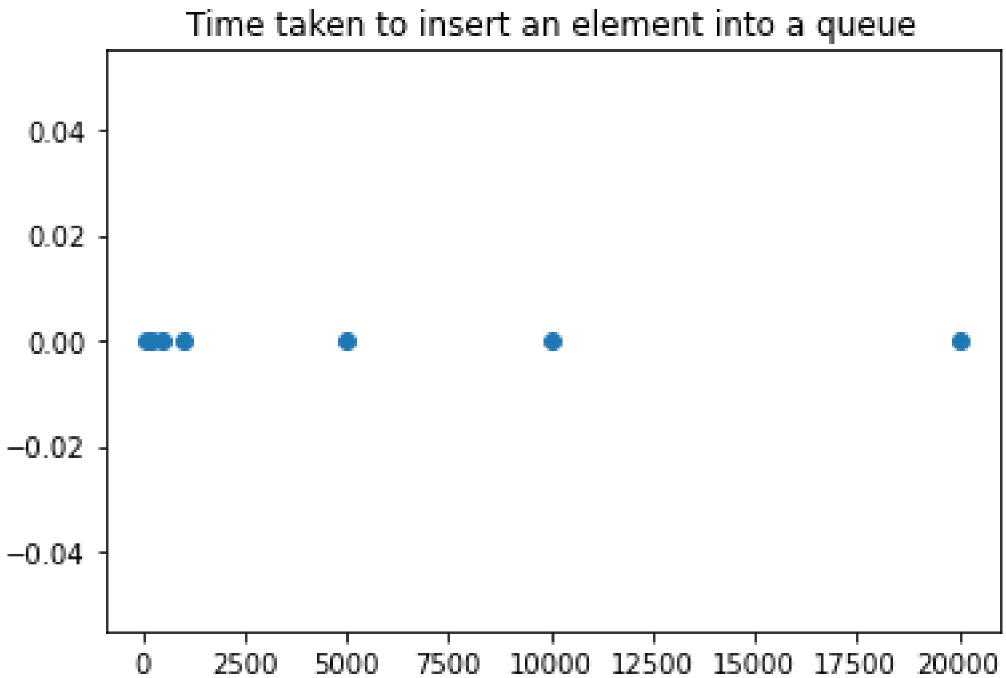
Priority queues

```
def enque (arr,x):
    arr.append(x)
    #return arr

def deque (arr):

    if (len(arr)==0):
        print("Queue is empty")
        return

    else:
        minimum=arr[0]
        minIndex=0

        for i in range (len(arr)):
            if (arr[i]<minimum):
                minimum=arr[i]
                minIndex=i

        for i in range (minIndex,len(arr)-1):
            arr[i]=arr[i+1]

        arr.pop()
    #return arr

def searchQueue(arr,x):
    k=-1
    for i in range (len(arr)):
        if (arr[i]==x):
            k=i
            break

    if (k==-1):
        print("The element is not present in the queue")
```

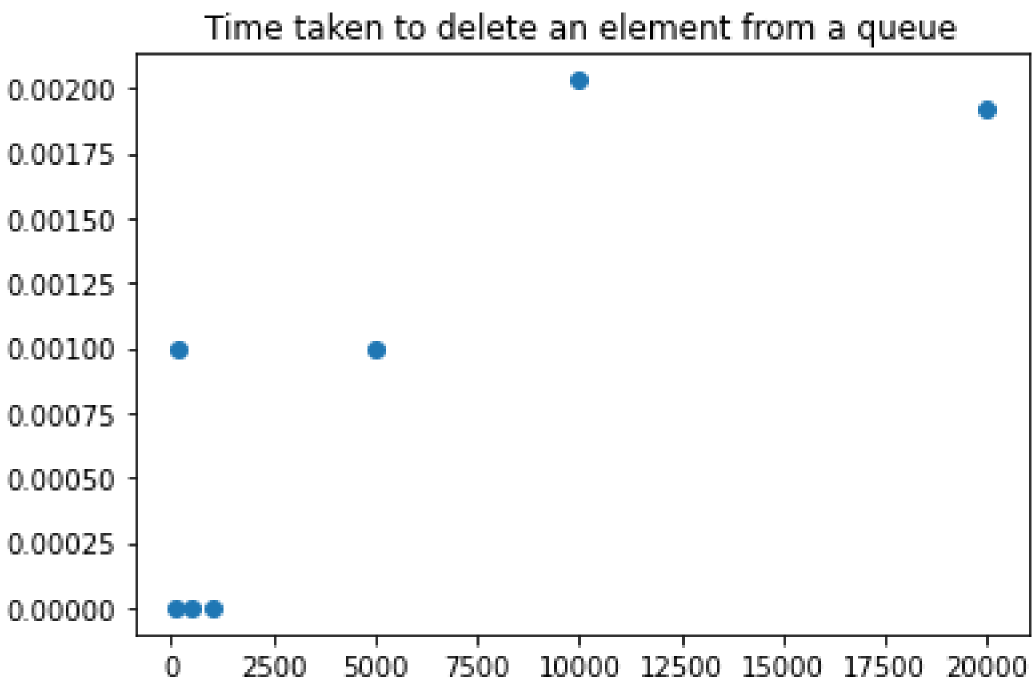
```
        return

    else:
        print("The element is present at the index ",k)
        return

timeEnquep=[]
timeDequep=[]
timeSearchp=[]

for j in range (len(size)):
    queue=[]
    for i in range (size[j]):
        queue.append(rn.randint(1,20))
    queue.append(25)

    start=time.time()
    enque(queue,rn.randint(1,10))
    stop=time.time()
    timeEnquep.append(stop-start)

    start=time.time()
    deque(queue)
    stop=time.time()
    timeDequep.append(stop-start)

    start=time.time()
    searchQueue(queue,25)
    stop=time.time()
    timeSearchp.append(stop-start)
```

```
The element is present at the index  99
The element is present at the index  199
The element is present at the index  499
The element is present at the index  999
The element is present at the index  4999
The element is present at the index  9999
The element is present at the index  19999
```

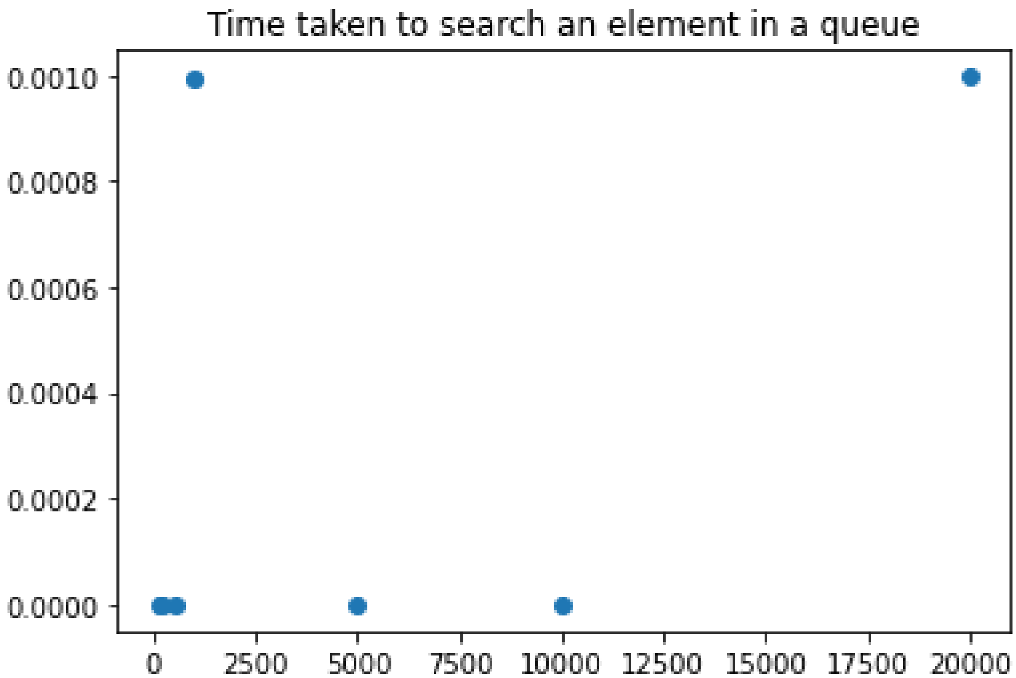
```
plt.scatter(size,timeEnquep)
plt.title('Time taken to insert an element into a queue')
plt.show()
```




```
plt.scatter(size,timeDequep)
plt.title('Time taken to delete an element from a queue')
plt.show()
```




```
plt.scatter(size,timeSearchp)
plt.title('Time taken to search an element in a queue')
plt.show()
```



Time Complexity

1) For insertion or enque the element is added directly to the end of the queue. Hence the time complexity for enque is O(1).

2) For dequeueing or deletion of an element in a priority queue , we first need to search the element with the highest priority(i.e. the minimum value in the queue), and then delete it. After that we have to shift all the elements after it 1 index to the left. Searching has a time complexity of O(n) and shifting also has a time complexity of O(n). Hence the net time complexity is O(n)+O(n)= O(n).

3) For searching, the search technique is similar to that of linear search i.e. all the elements are checked one by one till the element desired is obtained. Hence the time complexity of searching is O(n)

Stack

```
def push(arr,x):
    arr.append(x)
    return

def pop(arr):
    if (len(arr)==0):
        print("The stack is empty")
        return
    else:
```

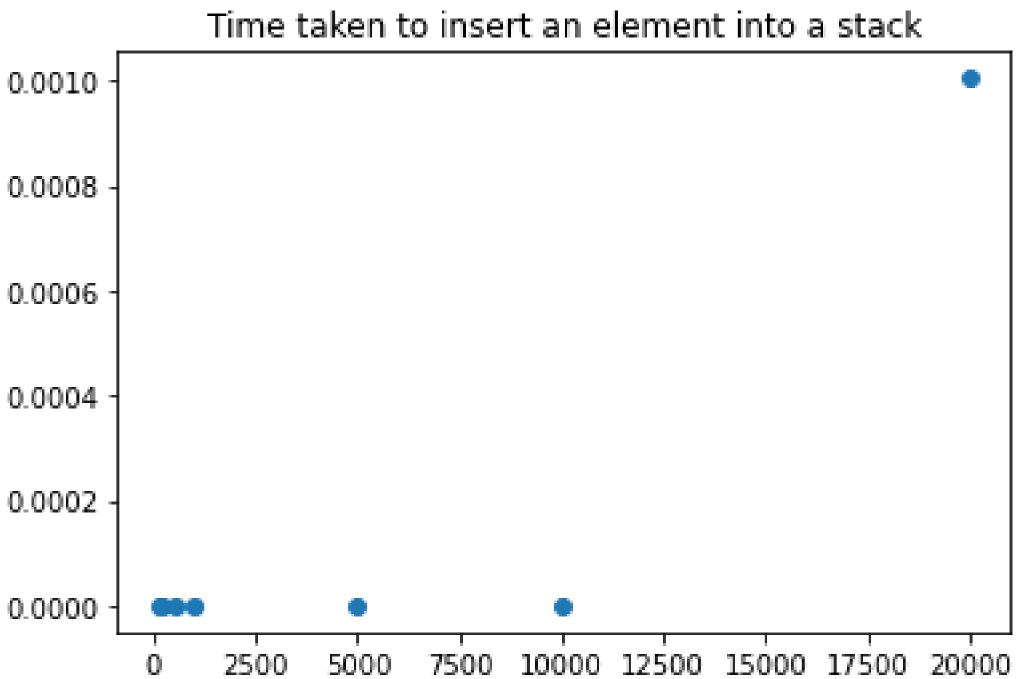
```python
        arr.pop()
        return

def search(arr,x):
    k=-1
    for i in range (len(arr)):
        if (arr[i]==x):
            k=i
            break

    if (k==-1):
        print("The element is not present in the stack")
        return

    else:
        print("The element is present at the index ",k)
        return

timePush=[]
timePop=[]
timeSearch=[]

for j in range (len(size)):
    stack=[]
    for i in range (size[j]):
        stack.append(rn.randint(1,20))

    start=time.time()
    pop(stack)
    stop=time.time()
    timePop.append(stop-start)

    start=time.time()
    push(stack,25)
    stop=time.time()
    timePush.append(stop-start)

    start=time.time()
    search(stack,25)
    stop=time.time()
    timeSearch.append(stop-start)
```

```
The element is present at the index  99
The element is present at the index  199
The element is present at the index  499
The element is present at the index  999
The element is present at the index  4999
The element is present at the index  9999
The element is present at the index  19999
```

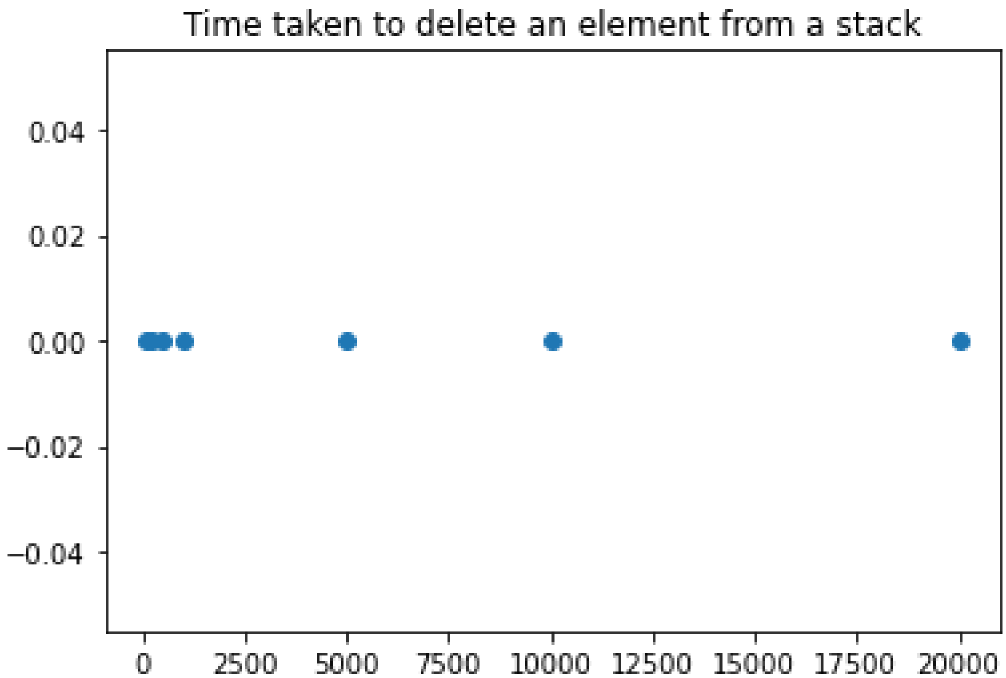
```
print(timePush)
```

```
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.001004934310913086]
```

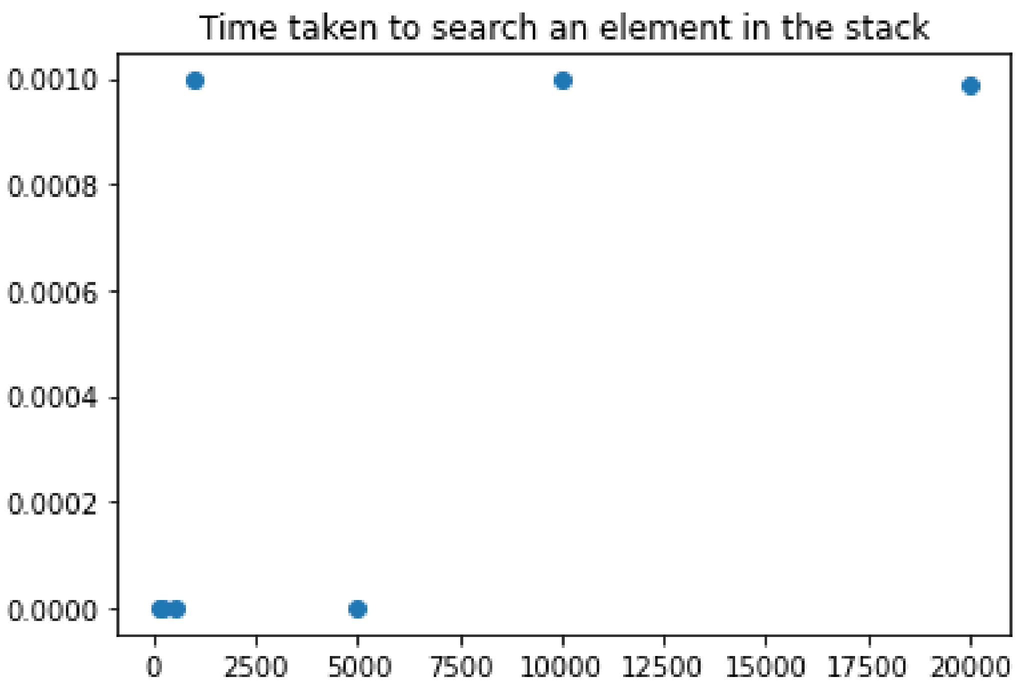
```
plt.scatter(size,timePush)
plt.title('Time taken to insert an element into a stack')
plt.show()
```




```
plt.scatter(size,timePop)
plt.title('Time taken to delete an element from a stack')
plt.show()
```




```
plt.scatter(size,timeSearch)
plt.title('Time taken to search an element in the stack')
plt.show()
```



Time Complexity

1) The elements are added at the end. Hence the time complexity for insertion is O(1).

2) The elements are deleted from the end and hence the time complexity for deletion is O(1)

3) For searching, the search technique is similar to that of linear search i.e. all the elements are checked one by one till the element desired is obtained. Hence the time complexity of searching is O(n)

Linked lists

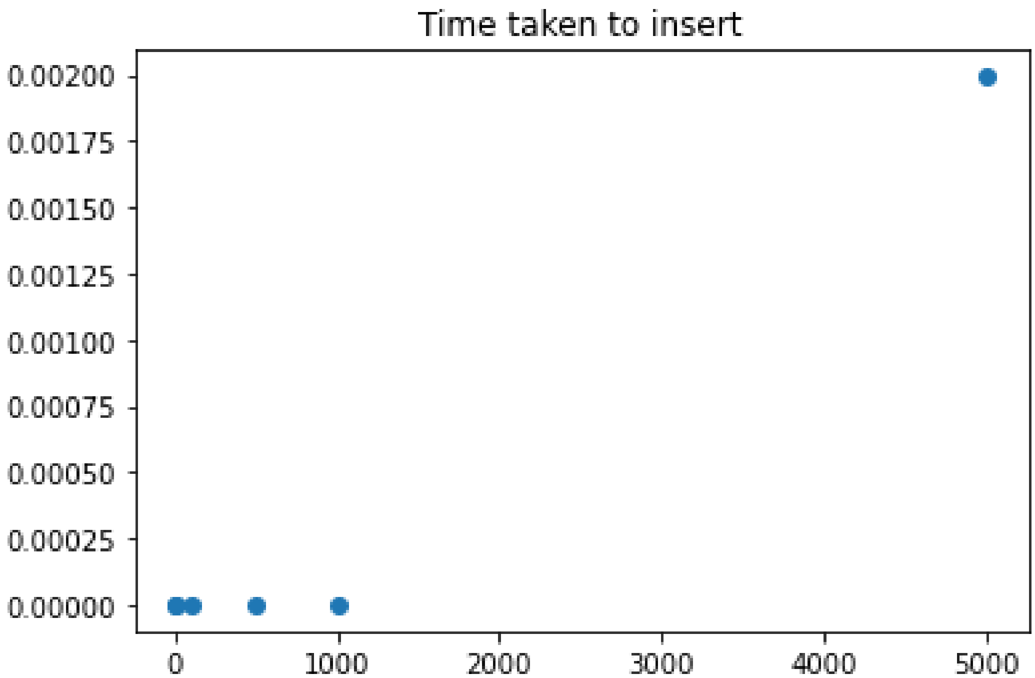
```
class Node:
    def __init__(self, data=None, next=None):
        self.data = data
        self.next = next

class linkedList:
    def __init__(self):
        self.head = None

    def print(self):
        if self.head is None:
            print("Linked list is empty")
            return

        itr = self.head
        llstr = ''
        while itr:
            llstr += str(itr.data)+' , '
            if itr.next else str(itr.data)
            itr = itr.next
        print(llstr)

    def length(self):
        count = 0
        itr = self.head
        while itr:
            count+=1
            itr = itr.next

        return count

    def insertBeginning(self, data):
        node = Node(data, self.head)
        self.head = node

    def insertEnd(self, data):
        if self.head is None:
            self.head = Node(data, None)
            return
```

```
        itr = self.head

        while itr.next:
            itr = itr.next

        itr.next = Node(data, None)

    def insert(self, index, data):
        if index<0 or index>self.length():
            print("invalid input")
            return


        if index==0:
            self.insertBeginning(data)
            return

        count = 0
        itr = self.head
        while itr:
            if count == index - 1:
                node = Node(data, itr.next)
                itr.next = node
                break

            itr = itr.next
            count += 1

    def delete(self, index):
        if index<0 or index>=self.length():
            print("Invalid Index")
            return


        if index==0:
            self.head = self.head.next
            return

        count = 0
        itr = self.head
        while itr:
            if count == index - 1:
                itr.next = itr.next.next
                break

            itr = itr.next
            count+=1
```

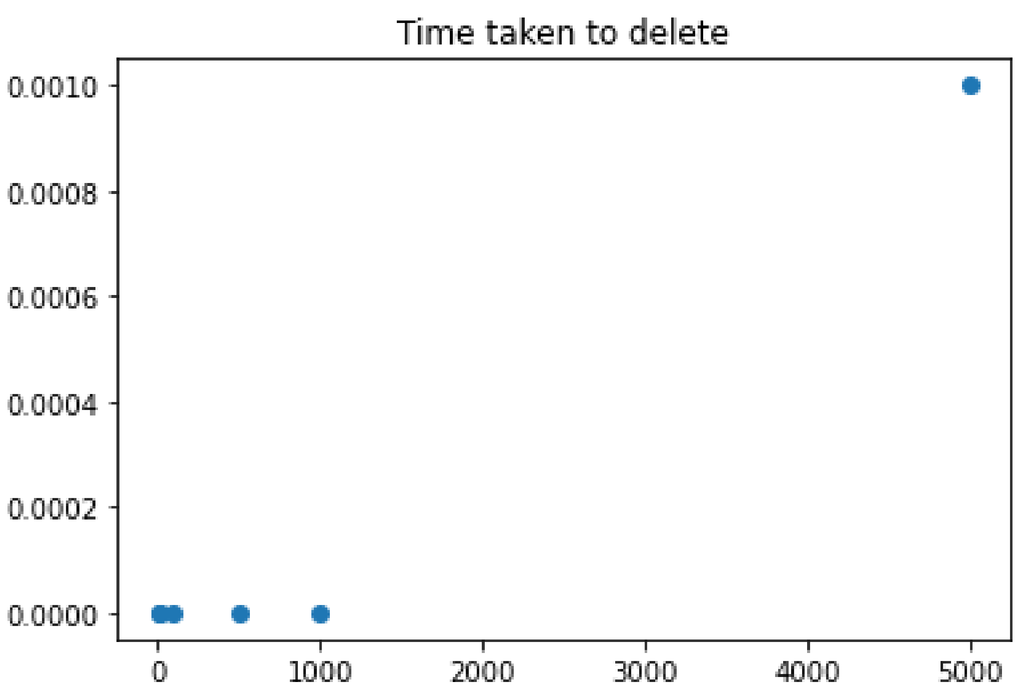
```python
    def search(self,value):
        if self.head is None:
            print("Linked list is empty")
            return

        itr = self.head
        llstr = ''
        while itr:
            llstr += str(itr.data)+' , '
            if itr.next else str(itr.data)
            itr = itr.next




timeInsert=[]
timeDelete=[]


size=[5,10,100,500,1000,5000]
for j in range (len(size)):
    ll=linkedList()

    for i in range (size[j]):
        ll.insertEnd(rn.randint(1,20))
        ll.insertEnd(25)

    start=time.time()
    ll.insert(size[j],23)
    stop=time.time()
    timeInsert.append(stop-start)

    start=time.time()
    ll.delete(size[j])
    stop=time.time()
    timeDelete.append(stop-start)

print(timeInsert,timeDelete)
```

```
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0019927024841308594] [0.0, 0.0, 0.0, 0.0,
0.0, 0.0010001659393310547]
```

```python
plt.scatter(size,timeInsert)
plt.title('Time taken to insert')
plt.show()
```




```
plt.scatter(size,timeDelete)
plt.title('Time taken to delete')
plt.show()
```



Time complexity

1) For inserting an element we need to move the pointer to the desired index and then add the node. To move the pointer to the desired location time Complexity is O(n) and time

taken to create and add the node is O(1). Hence the total time complexity for inserting is O(n).

2) For deleting an element we need to move the pointer to the desired index and then delete the node. To move the pointer to the desired location time Complexity is O(n) and time taken to delete the node is O(1). Hence the total time complexity for deleting O(n).