CP2410 - Assignment 1

Student Name: Udaya Bhaskar Reddy Malkannagari

Student ID: 13368171

Section 1 - Pseudocode

Pseudocode for the Sparse Array Class:

Sparse Array Class

Node Class

def \_\_init\_\_ (self, e = None, next = None) Initialize node’s fields with default values set to None

self.e = e Set user element as e

self.next = next Set next node as next

def \_\_str\_\_ (self) Print the element in the node

return str(self.e) Return element as a printable string

def \_init\_\_(self, n, head=None): Initialize the Sparse Array with size set to n and head set to None

self.size = n Set the Sparse Array size to n

self.head = head Refers to the head of the linked list

self.usage = 0 Set initial count of non-empty cells to 0

def \_str\_\_(self):

Assign a variable current\_node to self.head

Initialize an empty list named ‘linked\_list’

while current\_node

append the index, element and reference to the list

set current\_node to current\_node.next

return list as a printable string

def \_len\_\_(self):

return the number of elements in the Sparse Array

def fill (self, seq):

if seq is not a list

show type error message

set count to 0

if sequence length is greater than self.size:

show Index Error message

for every node in sequence:

if every node is None:

pass the operation

else:

set count and node in \_\_setitem\_\_  
increment count by 1

increment usage by 1 to count the non-empty nodes

def \_\_getitem\_\_(self, j):

if j is less than 0

increment j by self.size

elif j greater than equals self.size

show Index Error message

elif not self.head or len(self) less than j

show Index Error message

assign current\_node to self.head

while current\_node:

if current\_node.e[0] equals to j

return current\_node.e[1]

set current\_node to current\_note.next

def \_setitem\_\_ (self, j, e):

if j is less than 0

increment j by self.size

elif j is greater than equals self.size

show Index Error message

Link a node by setting self.head to self.Node([j, e], self.head)

Pseudocode for the testing algorithms

Import the time module to calculate the time taken

Import the random module to generate random integers

Import the SparseArray class

n = User input for the sequence size

m = User input for the non-empty values in the sequence

L = Sparse Array of size n

**Function create\_sequence()**

Create first list of n ‘None’ values

Create second list of m ‘random’ values’

Set count to 0

While count is not equal to m

For index, z in enumerate (first list)

Produce 0 or 1 at random, if it is 1

First list[index] = random choice (second list)

Increment count by 1

If count is equal to m

Break

Return the first list

**Function test\_get():**

Initialize an empty list to store the running time

Generate a random number

For i in range (5)

Calculate the start time

run the get item function to access an element

Calculate the elapsed time

Display the elapsed time in micro seconds

Append the list with the elapsed time

Display the average running time

**Function test\_set():**

Initialize an empty list to store the running time

Generate a random number j as index

Generate a random number e as element

For i in range (5)

Calculate the start time

run the set item function to access an element at a certain index

Calculate the elapsed time

Display the elapsed time in micro seconds

Append the list with the elapsed time

Display the average running time

**Function test\_get\_python\_list()**

Initialize an empty list to store the running time

Create a python list with random numbers from 1 to 100 of size n

Generate a random number

For i in range (5)

Calculate the start time

Access the element from the python list

Calculate the elapsed time

Display the elapsed time in micro seconds

Append the list with the elapsed time

Display the average running time

**Function test\_set\_python\_list()**

Initialize an empty list to store the running time

Create a python list with random numbers from 1 to 100 of size n

Generate a random number between 0 and n as index

Generate a random number as element

For i in range (5)

Calculate the start time

Insert the element from the python list

Calculate the elapsed time

Display the elapsed time in micro seconds

Append the list with the elapsed time

Display the average running time

Section two - Efficiency Analysis

Time Complexity analysis

For understanding the efficiency of the algorithms, we look at how the program behaves for various inputs and calculate the time taken by the algorithm as it grows with different input values. In simpler words, we calculate the rate of growth of the time taken with respect to the input.

To do this, we define a model machine to execute our program. This machine performs a sequential operation and let us say, for our measurement, it takes

* 1 unit of time for simple arithmetic and logical operations
* 1 unit of time for assignment to a variable
* 1 unit of time to return from a function

We calculate the cost (in units) and the no of the times the operation will be executed for the different set of algorithms

Calculating the efficiency of the algorithms

**\_\_getitem\_\_(j)**

**def** \_\_getitem\_\_(self, j): Cost No of times  
 **if** j < 0: 1 1   
 j += self.\_size 2 1

**elif** j >= self.\_size: 1 1 **raise** IndexError 1 1

**elif not** self.\_head **or** len(self) < j: 2 1  
 **raise** IndexError 1 1  
  
 current\_node = self.\_head 1 1

**while** current\_node: 1 n   
 **if** current\_node.e[0] == j: 2 n  
 **return** current\_node.e[1] 2 n  
 current\_node = current\_node.next 1 n

f(n) = 4n + constant

f(n) = O(n)

If we add the total cost with the number of times the algorithm is executed, the resultant value will be a function of n. Something like, c1 n + c2, where c1 and c2 are constants. This will be a linear function and can be defined with O (n) notation.

For this function, the time complexity O(n) increases with the size of the array. This will be explained in the experiments section in detail.

**\_\_setitem\_\_(j,e)**

**def** \_\_setitem\_\_(self, j, e): Cost No of times

**if** j < 0: 1 1  
 j += self.\_size 2 1  
 **elif** j >= self.\_size: 1 1  
 **raise** IndexError 1 1  
  
 self.\_head = self.Node([j, e], self.\_head) 1 n

The \_\_setitem\_\_ function is executed n times as it is looped n times in the fill function.

f(n) = n + constant

f(n) = O(n)

As with the case of the previous algorithm, if we add the total cost with the number of times the algorithm is executed, the resultant value will be a function of n. Something like, c1n + c2, where c1 and c2 are constants. This will be a linear function and can be defined as O (n).

It can also be noted that in inserting an element at the beginning of the linked list, the time complexity will be constant and is calculated as O (1). However, setting an element at different locations in the list will be O(n).

**\_\_fill\_\_(seq)**

**def** fill(self, seq): Cost No of times  
 **if not** isinstance(seq, list):  1 1

**raise** TypeError 1 1

count = 0 1 1   
  
 **if** len(seq) > self.\_size: 1 1**raise** IndexError 1 1

**for** elem **in** seq: 1 n  
 **if** elem **is None**: 1 n **pass  
 else**: 1 n  
 self.\_\_setitem\_\_(count, elem) 1 n  
 count += 1 2 n  
 self.\_usage += 1 2 n

f(n) = 6n + constant

f(n) = O (n)

The efficiency of the fill function can be explained using the O (n) notation. For every element in the sequence, the loop will run ‘n’ times and the time complexity of this algorithm is a linear function, c1n + c2, where c1 and c2 are constants. We can calculate this result by adding up the total cost with the number of times the algorithm is executed.

This function will call the set.item algorithm n times as the loop is executed.

More detailed analysis of this mechanism is provided in the experiments section.

Section three - Experiments

All experiments are based on real test of the python code.

We use different values of m to look at levels of sparseness for several different values of n. In each of these cases we calculate the time taken to set and get items at random locations within the Sparse Array. For comparison purposes, we also consider the time taken to set and get items at random locations within a randomly generated python list. This will help us understand the relevance of a Sparse Array in comparison to an actual python list.

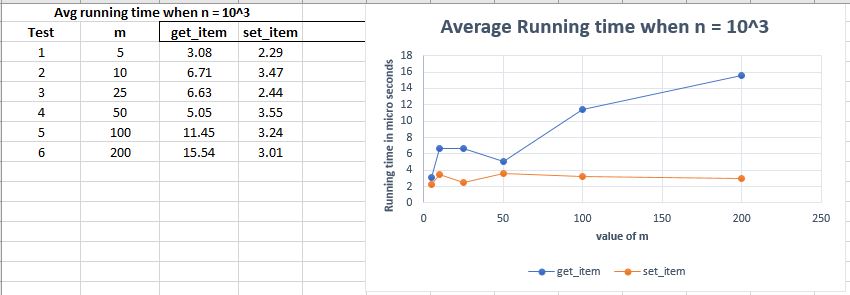
Test Case 1:

**Sparse Array:**

Size of the Sparse Array, n = 1000

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **n** | **m** | **get\_item (Sparse Array)** | **set\_item (Sparse Array)** | **get\_item (Python list)** | **set\_item (Python list)** |
| 10000 | 5 | 3.081 | 2.291 | 0.395 | 1.343 |
| 10000 | 10 | 6.716 | 3.477 | 0.474 | 1.027 |
| 10000 | 25 | 6.637 | 2.449 | 0.395 | 1.106 |
| 10000 | 50 | 5.057 | 3.556 | 0.237 | 0.869 |
| 10000 | 100 | 11.457 | 3.240 | 0.237 | 1.027 |
| 10000 | 200 | 15.644 | 3.002 | 0.237 | 0.948 |

*Table 1: Showing the get and set item running times in micro seconds for n = 10^3*

****

*Figure 1 – Showing average running times (in micro seconds) to get and set an item, n = 10^3*

In this experiment, we analyze the time taken to set and get an item from a Sparse Array of size 1000 with different levels of sparseness (m values). The number of non-empty cells, m, is set to different values and the times taken is plotted as a graph which is illustrated in figure 1.

As we can see, with an increase in the denseness of the Sparse Array, we notice a gradual increase in the time taken to access an item. The higher the number of non-empty cells, the longer is the time taken to get an item.

The time taken to access an element in the Sparse Array which is implemented in the form of a linked list is O(m) in an average case. In a linked list, the only information we have is the address of the head node. To access an element at a particular position in the list, we start at the head node and traverse through each node of the list and reach the target node. In the worst case, we will be traversing through the whole list. This is a time-consuming process. Hence, we can observe that for higher values of m, the time taken increases gradually increases.

However, we note that the time taken to set an item does not show a similar trend. The average times to set an item is almost steady. This will be further analyzed in the subsequent experiments.

**Python list running time Comparison**

We now analyze the time take to set and get an item from an actual python list of size n and compare the results.

**Python List – (size = 1000)**

Average running time to get an item from a python list of size 10^3 **= 0.395 micro seconds**

Average running time to insert an item within a python list of size 10^3 = **1.106 micro seconds**

We can clearly notice the difference in time while accessing an element. As compared to the Sparse Array of the same size, an actual python list is much faster. In terms of time complexity, accessing an element is denoted as O (1) which is a constant time.

The time taken to insert an element within an actual python list is in comparison less than the Sparse array for this size.

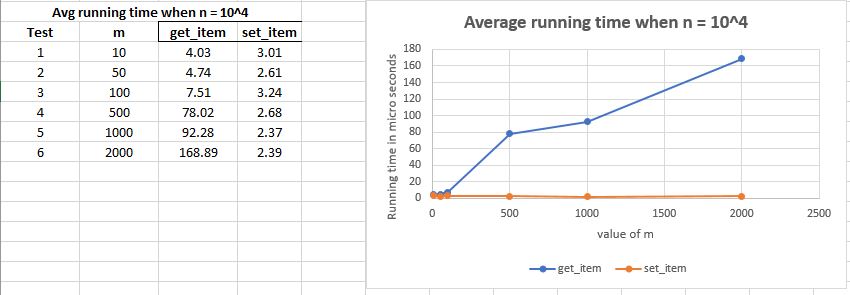
Test Case 2

**Sparse Array:**

Size of the Sparse Array, n = 10000

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **n** | **m** | **get\_item (Sparse Array)** | **set\_item (Sparse Array)** | **get\_item (Python list)** | **Set\_item (Python list)** |
| 10000 | 10 | 4.03 | 3.01 | 0.316 | 1.027 |
| 10000 | 50 | 4.74 | 2.60 | 0.237 | 2.291 |
| 10000 | 100 | 7.51 | 3.24 | 0.237 | 4.741 |
| 10000 | 500 | 78.02 | 2.68 | 0.395 | 3.951 |
| 10000 | 1000 | 92.28 | 2.37 | 0.316 | 4.435 |
| 10000 | 2000 | 168.89 | 2.39 | 0.395 | 6.435 |

*Table 2: Showing the get and set item running times in micro seconds for n = 10^4*



*Figure 2 – Showing average running times (in micro seconds) to get and set an item, n = 10^4*

***Discussion:***

In figure 2, we see a similar trend as observed with the previous Sparse Array of lesser size. The graph showing the get\_item indicates an upward trend. We can notice a jump in the time taken when the value of m is changed from 100 to 500. At m = 500, it requires approx. 78.02 micro seconds to access an element from the Sparse Array. As the denseness of the array increases, the time taken increases at a higher rate.

However, the time taken to set an item within the Sparse Array does not change much. And it is quite low even for higher values of m. (For m = 1000, time taken for set\_item is 2.37 micro seconds). The two experiments conducted so far provide similar results. This clearly illustrates that there is a general tendency evolving when it comes to the time taken to set an item.

The time taken is consistently lower despite an increase in the size of the Sparse Array from 10^3 to 10^4.

**Python list running time Comparison**

We now analyze the time taken to set and get an item from an actual python list of size 10^4 and compare the results.

**Python List – (size = 10^4)**

Average running time to get an item from a python list of size 10^4 **= 0.378 micro seconds**

Average running time to set an item within a python list of size 10^4 = **2.291 micro seconds**

These results also mirror the results from the previous experiment. There is a marginal increase in the set\_time as compared to the previous list of size 10^3. However, this difference is negligible (less than 1 micro second).

The time taken to set an item in the python list is almost the same as the time taken to set an item within the Sparse Array.

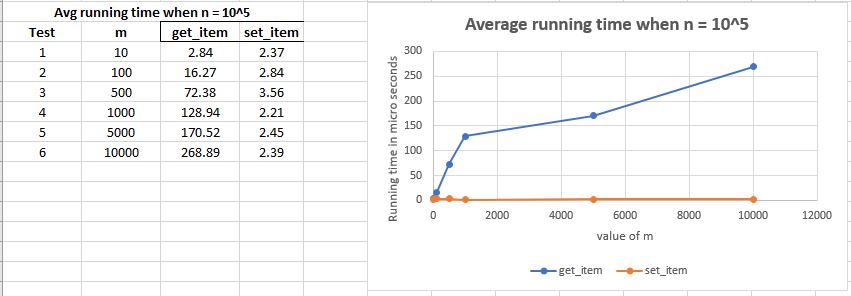
Test Case 3

**Sparse Array:**

Size of the Sparse Array, n = 10^5

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **n** | **m** | **get\_item (Sparse Array)** | **set\_item (Sparse Array)** | **get\_item (Python list)** | **set\_item (Python list)** |
| 100000 | 10 | 2.84 | 2.37 | 0.711 | 31.526 |
| 100000 | 100 | 16.27 | 2.84 | 0.395 | 20.148 |
| 100000 | 500 | 72.375 | 3.56 | 0.553 | 32.869 |
| 100000 | 1000 | 128.94 | 2.21 | 0.395 | 36.109 |
| 100000 | 5000 | 170.52 | 2.45 | 0.553 | 27.338 |
| 100000 | 10000 | 268.89 | 2.39 | 0.395 | 19.437 |

*Table 3: Showing the get and set item running times in micro seconds for n = 10^5*



*Figure 3 – Showing average running times (in micro seconds) to get and set an item, n = 10^5*

Figure 3 illustrates the continuing trend of the running times with regards to accessing the items from the Sparse array. We have observed that this trend is independent of the size of the array so far.

**Python list running time Comparison**

We now look at the python list running times to further compare the results of our experiment.

**Python** **list: (size = 10^5)**

Average running time to get an item from a python list of size 10^4 **= 0.553 micro seconds**

Average running time to set an item within a python list of size 10^4 = **32.869 micro seconds**

Note that accessing the item takes approximately the same time irrespective of the size, however, we notice a significant change from the previous experiment with regards to setting an element within the python list. The time taken now is approx. **32 micro seconds** which is significantly higher than the time taken to set an item within the Sparse Array of the same size (which is approx. **2.45 micro seconds**)

This case allows us to understand the necessity of having a linked list. With the use of a linked list, we can insert an item within the Sparse Array much faster than an actual python list of the same size.

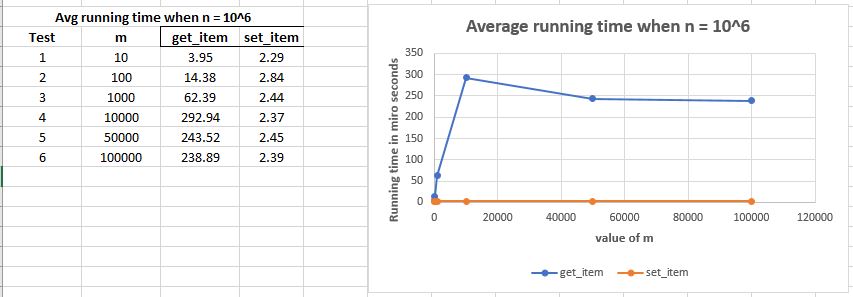
Test Case 4:

**Sparse Array:**

Size of the Sparse Array, n = 10^6

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **n** | **m** | **get\_item**  **(Sparse Array)** | **set\_item (Sparse Array)** | **get\_item (Python list)** | **set\_item (Python list)** |
| 1000000 | 10 | 3.95 | 2.29 | 0.489 | 157.55 |
| 1000000 | 100 | 14.38 | 2.84 | 0.395 | 103.66 |
| 1000000 | 1000 | 62.39 | 2.44 | 0.553 | 234.86 |
| 1000000 | 10000 | 292.94 | 2.37 | 0.395 | 254.18 |
| 1000000 | 50000 | 243.52 | 2.45 | 0.790 | 321.68 |
| 1000000 | 100000 | 238.89 | 2.39 | 0.395 | 373.43 |

*Table 4: Showing the get and set item running times in micro seconds for n = 10^6*



*Figure 4 – Showing average running times (in micro seconds) to get and set an item, n = 10^6*

***Discussion:***

Setting the size of the Sparse Array to 10^6 gives us similar patterns with regards to the running time of the functions. Higher values of m increase the time taken to access an item.

**Python list running time Comparison**

**Python list: (size = 10^6)**

Average running time to get an item from a python list of size 10^6 **= 0.485 micro seconds**

Average running time to set an item within a python list of size 10^6 = **278.89 micro seconds**

At this size, the python list is taking more time than the Sparse Array for insertion of an element to the list.

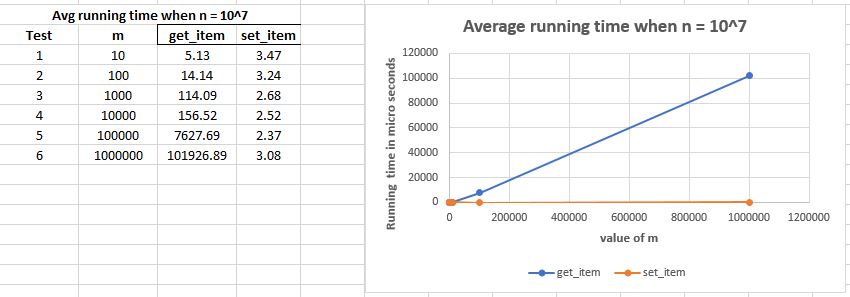
Test case 5:

**Sparse Array:**

Size of the Sparse Array, n = 10^7

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **n** | **m** | **get\_item**  **(Sparse Array)** | **set\_item**  **(Sparse Array)** | **get\_item (Python list)** | **set\_item**  **(Python list)** |
| 10000000 | 10 | 5.13 | 3.47 | 0.474 | 4891.88 |
| 10000000 | 100 | 14.14 | 3.24 | 1.106 | 3377.03 |
| 10000000 | 1000 | 114.09 | 2.68 | 1.027 | 6163.86 |
| 10000000 | 10000 | 156.52 | 2.52 | 0.632 | 7487.18 |
| 10000000 | 100000 | 7627.69 | 2.37 | 0.790 | 5494.68 |
| 10000000 | 1000000 | 101926.89 | 3.08 | 0.474 | 4029.43 |

*Table 5: Showing the get and set item running times in micro seconds for n = 10^7*



*Figure 5 – Showing average running times (in micro seconds) to get and set an item, n = 10^7*

***Discussion:***

At this size, there is a very high growth in the running times while accessing an item from the Sparse Array. The graph indicates a straight line signifying exponential growth.

The running time for setting an element is still very minimal.

**Python** **list: (size = 10^7)**

Average running time to get an item from a python list of size 10^7 **= 0.632 micro seconds**

Average running time to set an item within a python list of size 10^7 = **5494.68 micro seconds**

At this size, the python list is taking a significantly longer time than the Sparse Array for the set\_item functionality.

***Further Experiments….***

***(Higher values of n is causing a MemoryError on the windows laptop used for running this python code)***

Summarize the findings:

Here, we try to summarize the results obtained upon experimenting with the code for different values of m and n. To get a better analysis, we compare the ‘get item’ and ‘set item’ running times for the Sparse array and the Python list. We consider a value of 10% Sparseness while calculating the time for the Sparse Array.

|  |  |  |
| --- | --- | --- |
| **n** | **get\_item (Sparse Array)** | **get\_item (Python list)** |
| 1000 | 11.45 | 3.24 |
| 10000 | 92.28 | 2.37 |
| 100000 | 268.89 | 2.39 |
| 1000000 | 238.89 | 2.39 |
| 10000000 | 101926.89 | 3.08 |

*Table 6: Showing the running times for both the Sparse Array and the python list (in micro seconds)*

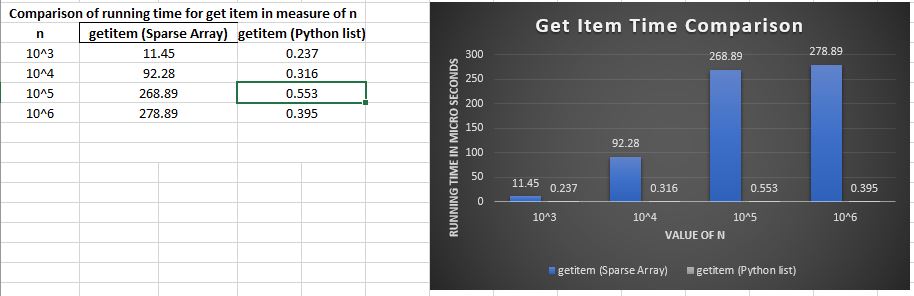
***Sparse Array: Python List:***

Size = n Size = n

Non-empty elements, m = 10 percent of n

**Get item Time comparison**

*Sparse Array vs Python List*



*Figure 6 – Showing average running times (in micro seconds) to get an item*

We notice two things,

* Time taken to access an element from the python list is small and consistent irrespective of the size of the list.
* Time taken to access an element from the Sparse Array increases with an increase in the size of the array.

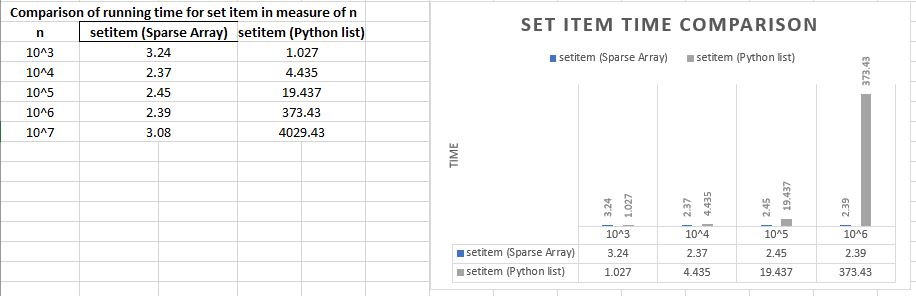
We can summarize as,

For array sizes greater than 10^4,

To access an element, **python list** is faster than a linked list implementation of a **Sparse Array**

**Set Item Time Comparison**

*Sparse Array vs Python List*

**

*Figure 7 – Showing average running times (in micro seconds) to set an item*

We notice two things,

* Time taken to insert an element within the Sparse Array is small and consistent irrespective of the size of the list.
* Time taken to insert an element within the python list increases with an increase in the size of the array.

We can summarize as,

For array sizes greater than 10^5,

To insert an element, a linked list implementation of the **Sparse Array** is faster than a **python list**

Discussion:

*Cost of accessing an element*

As summarized above, when we look at the cost of accessing an element, python list scores heavily over a linked list implementation of the Sparse array. This is due to the fact we access each element in a python list based on their set indices. In a python list, data is stored in a contiguous block of memory. Each element is allocated the same amount of memory. So, it is easy and quick to calculate the address of an element we want to access.

For example, if the memory address of the first element of the list is 100, the memory address of the nth element will be (100 + 4n). the number 4 corresponds to the bytes that are allocated to each element in the memory which is typically 4 bytes. Knowing an address for any element of the list is just a simple mathematical calculation which does not require much processing time. The time complexity for this operation as explained above will be O (1) which is a constant time.

When we look at the linked list implementation of a Sparse Array, data is not stored in a contiguous block of memory as in case of the python list. We have multiple blocks of memory at different addresses in the memory. Each block is called a node and includes two fields, one to store the data and one to store the address of the node. The only information we have in a linked list is the address of the first node, which is the head. To access an element in a linked list at a particular position, we first start at the head and traverse through all the individual nodes until we reach the target node. In the worst case, to access the last element in the linked list, we will be traversing through all the nodes of the list. Time taken will always be proportional to the number of elements in the linked list which is n. Time complexity will be O (n).

Hence, in terms of accessing an element, an **actual python list or a dynamic array** will heavily score over a linked list implementation of a **Sparse Array.**

*Cost of inserting an element*

The time complexity of inserting an element at various positions is given in the table below.

|  |  |  |
| --- | --- | --- |
| **Position** | **Python List/Dynamic Array** | **Sparse Array (Linked list)** |
| Beginning of the list | O (1) | O (1) |
| End of the list | O (1) if array is not full  O (n) if array is full | O (n) |
| At some i th position | O (n) | O (n) |

An array is a contiguous block of memory. To insert an element in a python list at a particular position, we have to shift each subsequent element by one position towards a higher index. If the size of the list is a million, we must perform approximately a million operations to insert a single element. If the array is full, a new block of memory will have to be allocated further increasing the time and the memory.

However, in a linked list we just insert an element at a particular node which requires only two operations. We traverse to the node and insert a new node at the target position. This is much more effective than a python list.

In terms of inserting an element, for bigger sizes, a linked list implementation of the **Sparse Array** will heavily score over an **actual python list or a dynamic array**.

This can be observed in the results demonstrated in the various experiments that we performed

Memory Requirements

We now look at the memory requirements for both the Sparse array and the python list. In case of a python list or a dynamic array, as explained in the previous sections, they are a contiguous block of memory of fixed size. The memory allocation keeps increasing with an increased array size. Even though there are null values in the list, the memory is allocated based on the index size.

In case of a linked list there is no unused memory. We require memory for one node at a time. No reserved space is kept for additional elements. We also use extra memory for the pointer variables, and this cannot be ignored when calculating the memory requirement for a linked list. However, linked list will fetch a greater advantage if the data part is large in size. For example, instead of an integer, if the data part is holding a data type of higher memory, the linked list can be a better option.

The primary factor that needs to be considered however is the number of used and unused cells in the arrays. As arrays are of fixed size, if there is no space available, we have no option but to copy all the elements of the array into a bigger memory block. And for larger sizes of an array, memory may not be available as a large block. But for the linked list, memory can be fragmented into multiple smaller blocks.

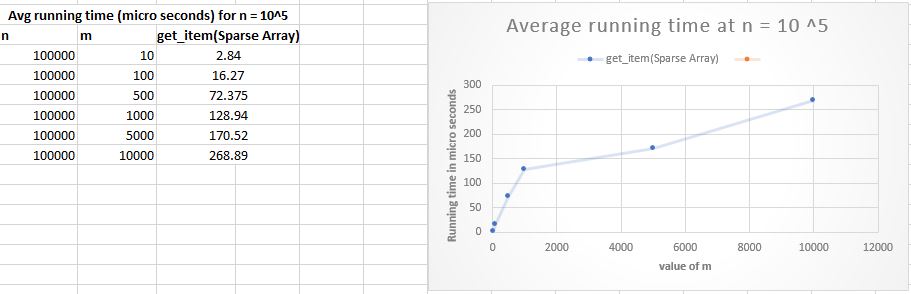
We need to keep these constraints into consideration while considering the choice of an Sparse array or a python list.

Sparse Array Performance Analysis:

*We now look at the results to analyze the performance of the Sparse Array for different sizes of m, and a fixed value for n*

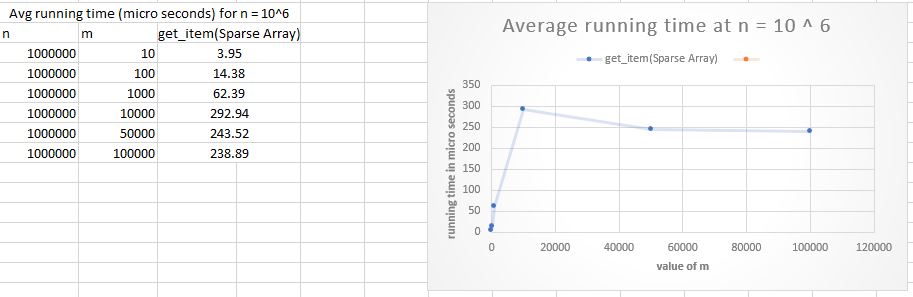
From the tables below, we notice that different values of sparseness also have a significant impact on the running time of the algorithm.

When n = 10^5, running time to get an item (in micro seconds)



*Figure 8 – Showing the average running for different values of m with n = 10^5*

When n = 10^6, running time to get an item (in micro seconds)



*Figure 9 – Showing the average running for different values of m with n = 10^6*

From the above figures, we can clearly deduce that,

By keeping the array size constant and setting the sparseness to different values can have a significant impact on the performance of the algorithm.

Conclusion:

The purpose of a Sparse Array is to save memory if we have a list that has lots of gaps in it. In a random case, if we only have 10 items, and the numbers that index them range from 0 to 1000, then a python list will have lots of null entries in it, and that will be quite wasteful. A Sparse array will use data structures internally to avoid that problem.