4/17/2017

Zhenyu Yan

Project 3 report

Arraylist records	Prepend	Append	Get	Remove	search
10	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms
100	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms
1000	5 ms	4 ms	<1 ms	<1 ms	<1 ms
10000	255 ms	287 ms	<1 ms	<1 ms	<1 ms
100000	16122 ms	16781 ms	<1 ms	2 ms	1 ms
linkedList records	Prepend	Append	Get	Remove	Search
10	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms
100	<1 ms	<1 ms	<1 ms	<1 ms	1 ms
1000	<1 ms	3 ms	<1 ms	<1 ms	2 ms
10000	1 ms	234 ms	<1 ms	<1 ms	127 ms
100000	4 ms	31748 ms	1 ms	1 ms	23561 ms
DoubleLinkedList records	Prepend	Append	Get	Remove	Search
10	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms
100	<1 ms	<1 ms	<1 ms	<1 ms	<1 ms
1000	<1 ms	<1 ms	<1 ms	<1 ms	1 ms
10000	1 ms	1 ms	<1 ms	<1 ms	59 ms
100000	3 ms	6 ms	<1 ms	<1 ms	7477 ms

(the reason I use 10,100,1000,10000,100000 instead of 1000,10000,100000,100000,1000000 is because it takes more than 30 mins to create an array list, and for linkedlist and double link list, it takes more than 1 hour get the specific index. For append and prepend, I add 10 times, then empty it, then add 100 times, then empty it, and add 1000 times …… For get, remove, and search method, I already create a list of 10, 100, 1000, 10000, 100000, then I get, remove or search once.)

ArrayList:

Append:

Growth function: f(x) = 1 + n + 1 + 1

Big O notation: f(x) = O(n);

(as the runtime increase, every time it need to create a new array, appending 100000 times need to create 100000 new array, that is why it takes a long time)

Prepend:

Growth function: f(x) = 1 + n + 1 + 1

Big O notation: f(x) = O(n);

worst-case time complexity: depend on how big is the list.

(as the runtime increase, every time it need to create a new array, prepending 100000 times need to create 100000 new array, that is why it takes a long time)

get:

Growth function: f(x) = 1;

Big O notation: f(x) = O(1);

worst-case time complexity: none.

(get is very fast because we do not need to create a new array, it just go to the specific index)

remove:

Growth function: f(x) = 1+ n + 1;

Big O notation: f(x) = O(n);

worst-case time complexity: depend on how big is the list.

(by removing an element in the array, we need to create a new array which lengh-1, then put

all the element in, if the array is very big, it takes a long time to copy the element)

search:

Growth function: f(x) = n;

Big O notation: f(x) = O(n);

worst-case time complexity: depend on how big is the list and the search range.

(search is based on get, from lower bound and upper bound, we use a for loop to get each index form the range, then compare with the search element)

LinkedList:

Append:

Growth function: f(x) = 1 + n + 1 + 1;

Big O notation: f(x) = O(n);

worst-case time complexity: depend on how big is the list.

(first create a new node, then link to the last node, but in order to link to the last node, we need to go from the first node, as the link list is larger, it takes more time)

Prepend:

Growth function: f(x) = 1 + 1 + 1;

Big O notation: f(x) = O(1);

worst-case time complexity: none;

(it just create a new node, then link to the head node, then set it to the head node, it is only there steps, so it is very fast)

get:

Growth function: f(x) = 1 + n;

Big O notation: f(x) = O(n);

worst-case time complexity: depend on how big is the list.

(get is basic on the index, it goes in a for loop and get the next node until the index node, if the index is big, it takes more time)

remove:

Growth function: f(x) = 1 + 1 + n + n + 1;

Big O notation: f(x) = O(n);

worst-case time complexity: depend on how big is the list.

(by removing an element in the link list, we need to use a for loop to get to get the index node and the index-1 node, then remove the index node, then the index-1 node will point the the next node of index node, it takes a little longer than the get method);

search:

Growth function: f(x) = n;

Big O notation: f(x) = O(1);

(search is just like get, and just compare the element of the specific index to the element you want to search in the range, it takes similar time as get method).

DoubleLinkedList:

Append:

Growth function: f(x) = 1 +1 + 1 + 1

Big O notation: f(x) = O(1);

worst-case time complexity: none.

(first is to create a new node, then get the tail node, let the previous node of the new node point to the tail node, let the next node of tail node point to the new node, then let the new node become the tail node. Because it does not involve loop, so it is very fast)

Prepend:

Growth function: f(x) = 1 + 1 + 1 + 1

Big O notation: f(x) = O(1);

worst-case time complexity: none.

(first is to create a new node, then get the head node, let the next node of the new node point to the head node, let the previous node of head node point to the new node, then let the new node become the head node. Because it does not involve loo, so it is very fast)

get:

Growth function: f(x) = n;

Big O notation: f(x) = O(n);

worst-case time complexity: depend on how big is the list, and depend on the position if it is in the middle, it takes more time.

(get is to use a for loop to get the next or previous node to get to the specific node, the reason which is faster than the linked list is because I set a condition. The condition is when

index<(size/2), I count from the head and get the next node, when index > (size/2), I count from the tail and get the previous node, the time is twice faster than link list unless it is in the middle.)

remove:

Growth function: f(x) = n + 1 +1 + 1 + 1 + 1;

Big O notation: f(x) = O(n);

worst-case time complexity: depend on how big is the list, and depend on the position if it is in the middle, it takes more time.

(for remove, I use the get method to get index, then I use the next node and previous node in the specific index to get the next node and previous node, then link them together, because I use the get method, so it is twice faster than the remove method in linked list)

search:

Growth function: f(x) = n;

Big O notation: f(x) = O(1);

worst-case time complexity: depend on how big is the list, and depend on the position if it is in the middle, it takes more time.

(search is based on the get method, and compare to element of the node to the searching element, also it is twice as faster than linked list)

Summary:

Over all, array list is fastest when you try to get or search element. The reason is because the memory address is fixed, for example like int[5], the address is 1000-1005, so you want to get the 3rd one, so you just get the element in the address 1003, but for double Linked list and linked list, the address is not next to each other, you need to get the address base on the previous one. The advantage of linked list and double linked list, they both faster than array list when you want to remove, append, prepend. The reason is because for array list, you need to create a new array every time, but linked list and double linked list do not need. Compare to linked list and double Linked list, double link has two directions, so the time is at least twice faster than the linked list.