02: ordered containers #1

Pointers | Singly Linked Lists | Doubly Linked Lists | Vector Data Structure
Amortized Complexity | List vs. Vector Performance | Binary Search | HW01

Pointers

```
1:  // pointers.cpp - download here
2:
3:  #include <iostream>
4:
5:  struct SingleNode {
6:    SingleNode * m_next;
7:    long long m_value;
8:  };
9:
10: void printAddrValues(SingleNode * node, long long addr, const char * title){
11:   std::cout << "================================================" << std::endl;
12:   std::cout << "= " << title << " addrs/values: " << std::endl;
13:   std::cout << "================================================" << std::endl;
14:   std::cout << "addr(" << title << "): " << std::hex << addr << std::endl;
15:   std::cout << "value(" << title << "): " << std::hex << node << std::endl;
16:   std::cout << "addr(" << title << ".m_next): " << std::hex << &(node->m_next) << std::endl;
17:   std::cout << "value(" << title << ".m_next): " << std::hex << node->m_next << std::endl;
18:   std::cout << "addr(" << title << ".m_value): " << std::hex << &(node->m_value) << std::endl;
19:   std::cout << "value(" << title << ".m_value): " << std::hex << node->m_value << std::endl;
20:   std::cout << std::endl;
21: }
22:
23: int main(int argc, char *argv[]){
24:
25:   std::cout << "================================================" << std::endl;
26:   std::cout << "= sizes: " << std::endl;
27:   std::cout << "================================================" << std::endl;
28:   std::cout << "sizeof(SingleNode): " << sizeof(SingleNode) << std::endl;
29:   std::cout << "sizeof(SingleNode *): " << sizeof(SingleNode *) << std::endl;
30:   std::cout << "sizeof(long long): " << sizeof(long long) << std::endl;
31:   std::cout << std::endl;
32:
33:   SingleNode * head = new SingleNode();
34:   printAddrValues(head, (long long) &head, "head");
35:
36:   SingleNode * one = new SingleNode();
37:   one->m_value = 1;
38:
39:   SingleNode * two = new SingleNode();
40:   two->m_value = 2;
41:
42:   SingleNode * three = new SingleNode();
43:   three->m_value = 3;
44:
45:   head->m_next = one;
46:   one->m_next = two;
47:   two->m_next = three;
48:   printAddrValues(head, (long long) &head, "head");
49:   printAddrValues(one, (long long) &one, "one");
50:   printAddrValues(two, (long long) &two, "two");
51:   printAddrValues(three, (long long) &three, "three");
52:
53:   return 0;
54: }
```

```
================================================
= sizes:
================================================
sizeof(SingleNode): 16
sizeof(SingleNode *): 8
sizeof(long long): 8

================================================
= head addrs/values:
================================================
addr(head): 7fff16456ff0
value(head): 0x1c20010
addr(head.m_next): 0x1c20010
```

```
value(head.m_next): 0
addr(head.m_value): 0x1c20018
value(head.m_value): 0

================================================
= head addrs/values:
================================================
addr(head): 7fff16456ff0
value(head): 0x1c20010
addr(head.m_next): 0x1c20010
value(head.m_next): 0x1c20030
addr(head.m_value): 0x1c20018
value(head.m_value): 0

================================================
= one addrs/values:
================================================
addr(one): 7fff16456ff8
value(one): 0x1c20030
addr(one.m_next): 0x1c20030
value(one.m_next): 0x1c20050
addr(one.m_value): 0x1c20038
value(one.m_value): 1

================================================
= two addrs/values:
================================================
addr(two): 7fff16457000
value(two): 0x1c20050
addr(two.m_next): 0x1c20050
value(two.m_next): 0x1c20070
addr(two.m_value): 0x1c20058
value(two.m_value): 2

================================================
= three addrs/values:
================================================
addr(three): 7fff16457008
value(three): 0x1c20070
addr(three.m_next): 0x1c20070
value(three.m_next): 0
addr(three.m_value): 0x1c20078
value(three.m_value): 3
```

stack

addr	value	label
0x7fff16456ff0:	0x1c20010	head
0x7fff16456ff8:	0x1c20030	one
0x7fff16457000:	0x1c20050	two
0x7fff16457008:	0x1c20070	three

heap

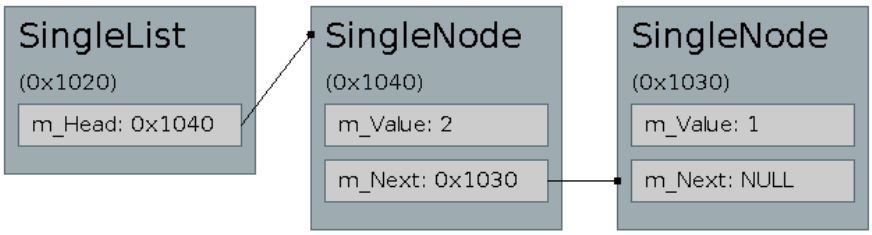
addr	value	label
0x1c20010:	0x1c20030	head.m_next
0x1c20018:	0	head.m_value
0x1c20020:		
0x1c20028:		
0x1c20030:	0x1c20050	one.m_next
0x1c20038:	1	one.m_value
0x1c20040:		
0x1c20048:		
0x1c20050:	0x1c20070	two.m_next
0x1c20058:	2	two.m_value
0x1c20060:		
0x1c20068:		
0x1c20070:	0x0000000	three.m_next
0x1c20078:	3	three.m_value

Singly Linked Lists

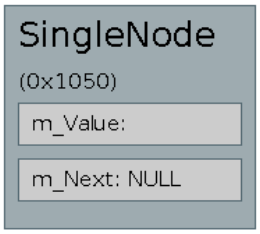
An item can be inserted in the middle of a singly linked list without moving all the remaining elements over. This is required in a vector.

insertion at the head

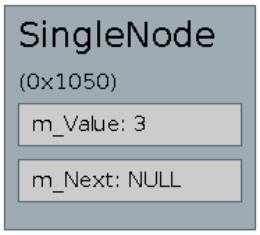
1. start:



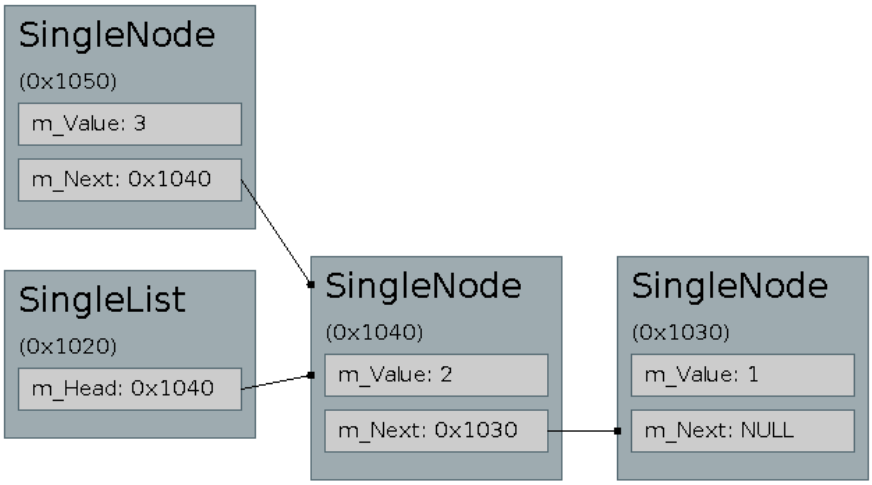
2. create a new_node:



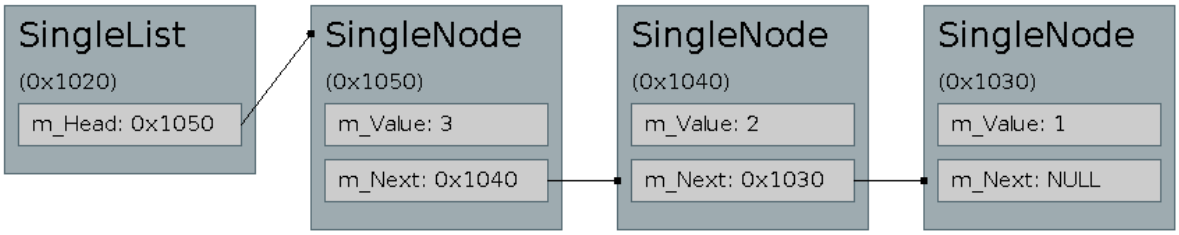
3. set the value of the new_node:



4. the new_node's next_nointer is set point to where the list's head_pointer currently points to



5. The list's head_pointer is set to point to the new_node

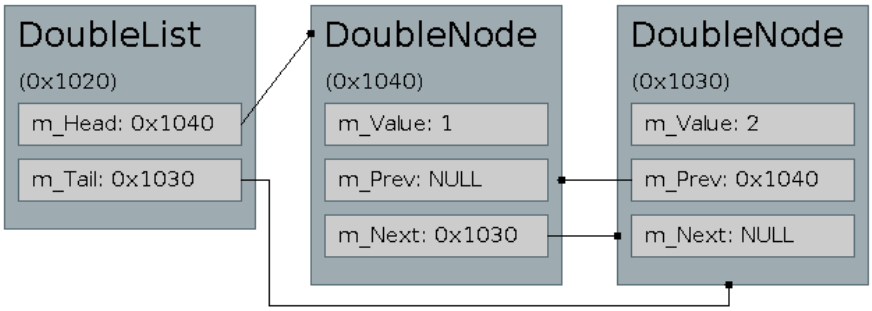


Doubly Linked Lists

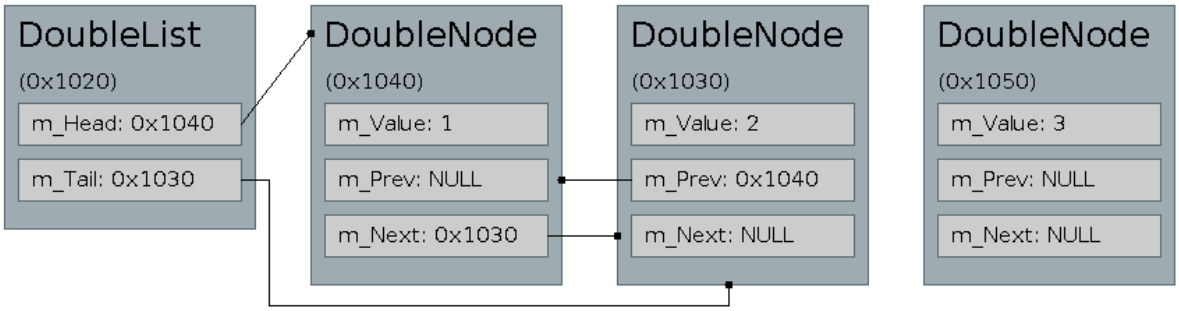
In a singly linked list you can only iterate in one direction. In a doubly linked list you can iterate going both forwards and backwards

adding a node at the end

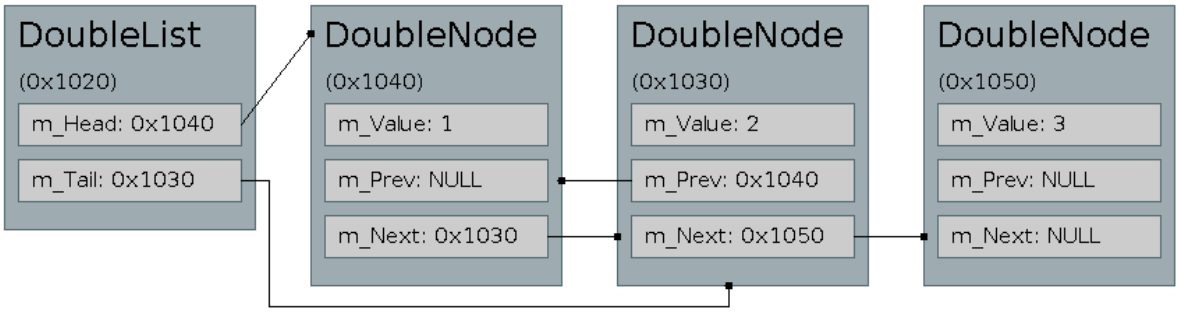
1. start:



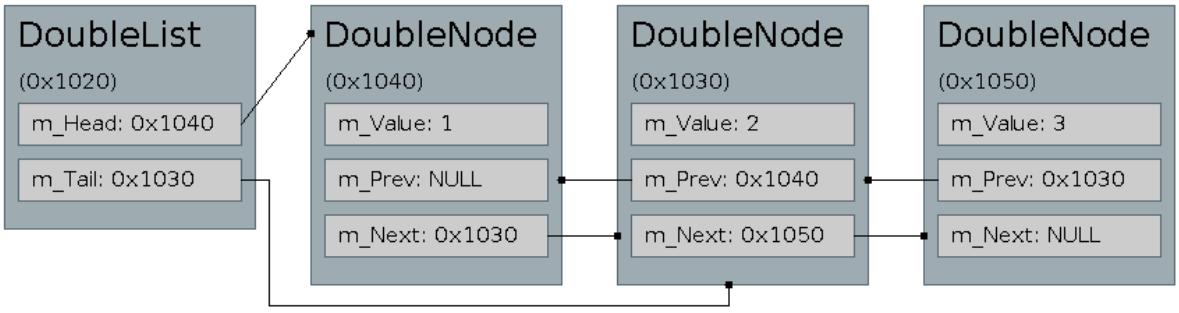
2. create a new node and initialize the value



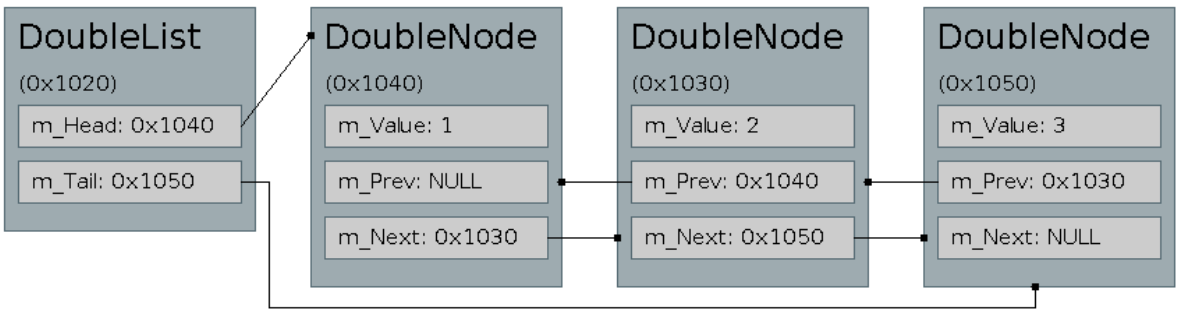
3. set the next of the tail to the new_node



4. set the prev of the new_node to the tail

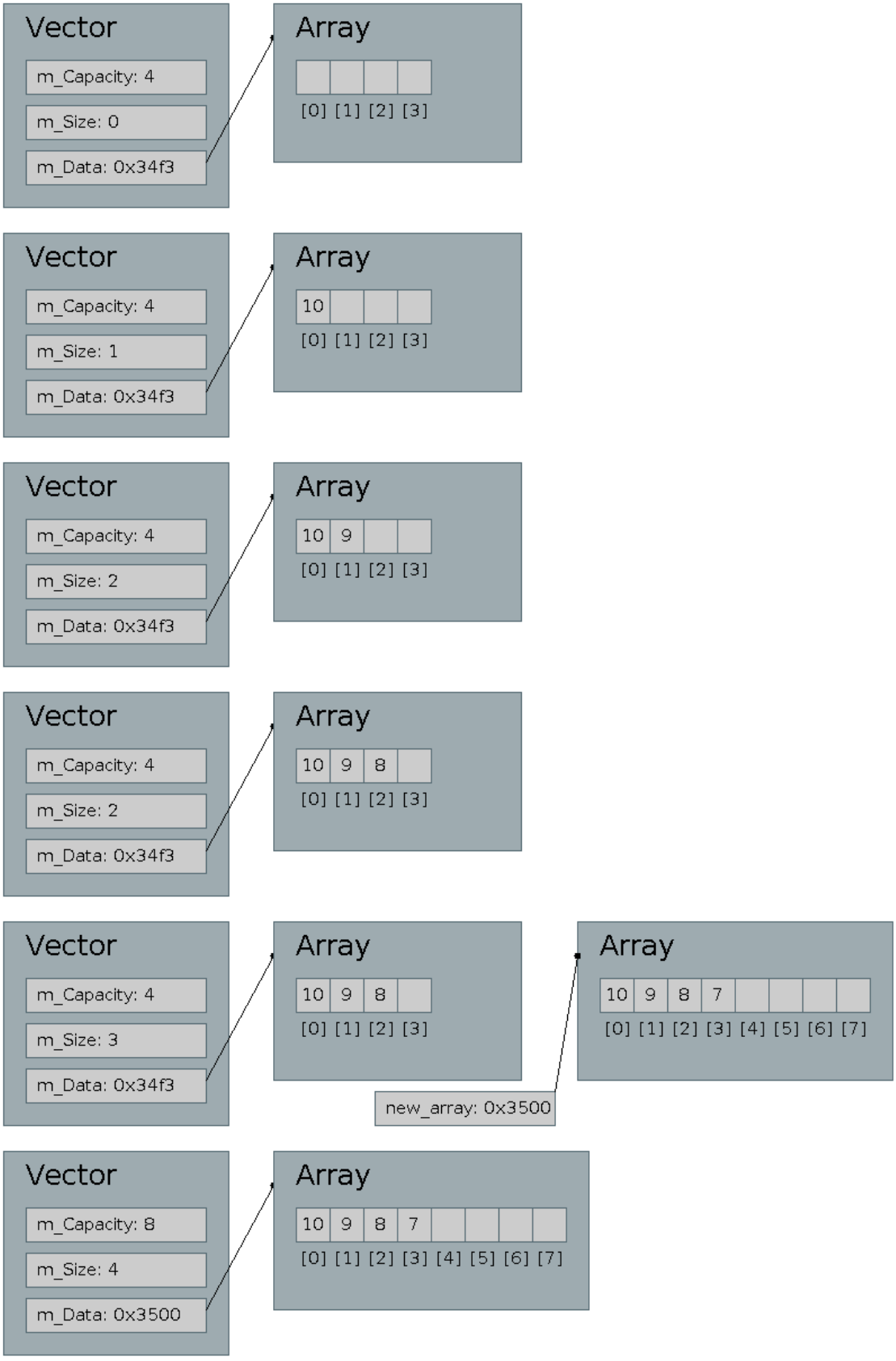


5. set the tail to the new_node



vector

A vector is an array that can automatically grow or shrink.



Amortized Complexity

This is the aggregate method.

- In a vector data structure adding an element to the back has a Ω(1) and O(size_vector).
- We want to have a complexity that is more tightly bounded
- Amortized Complexity attempts to do this

Cost of insertion into vector															
1	1	1	4												
1	1	1	8												
1	1	1	1	1	1	1	16								
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	32

The starting capactity is 4 and we double the capacity when full.

- T(n) = n + (n/2) - 1 + T(n/2)
- T(4) = 4 + 3

Solving for T(n) / n

- T(n) = (n + (n/2) - 1) + T(n/2)
- T(n) = (n + (n/2) - 1) + (n/2 + (n/4) - 1) + T(n/4)
- T(n) = (n + (n/2) - 1) + (n/2 + (n/4) - 1) + (n/4 + (n/8) - 1) + ... + T(n/n/4)
- T(n) = (n + (n/2) - 1) + (n/2 + (n/4) - 1) + (n/4 + (n/8) - 1) + ... + T(4)
- T(n) = (n + (n/2) - 1) + (n/2 + (n/4) - 1) + (n/4 + (n/8) - 1) + ... + 4 + 3
- T(n) = sumof: (n + n/2 + n/4 + ... +) for log_2(n-2) times + sumof: (n/2 + n/4 + n/8 + ... +) for log_2(n-2) times + sumof: (-1) for log_2(n-2) times
- T(n) = n + sumof(1) for log_2(n-3) times + n/16 + cn
- T(n) = c1*n + c2*log_2(n)
- T(n) / n = c1 * n/n + c2*log_2(n)/n
- T(n) / n = O(1)

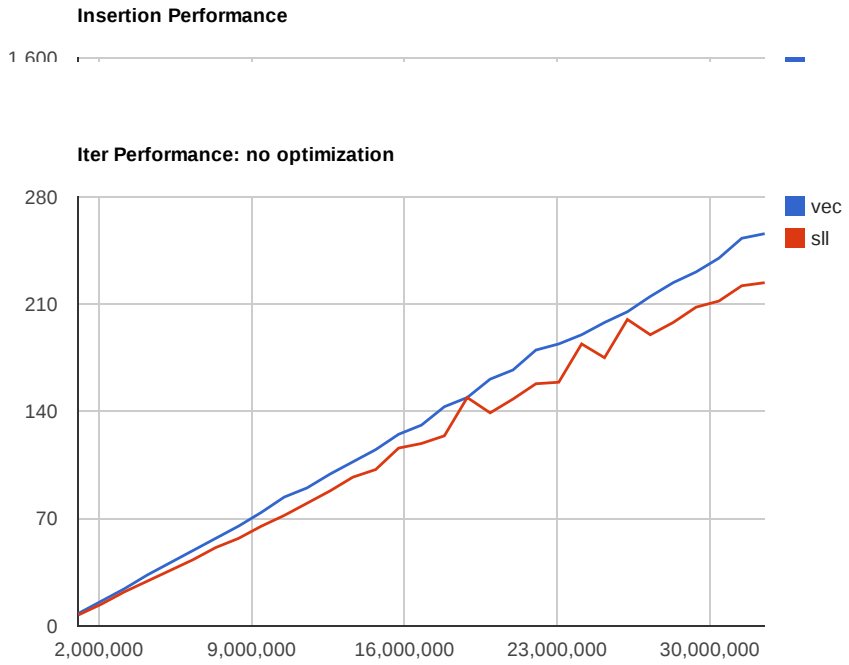
List vs. Vector Performance

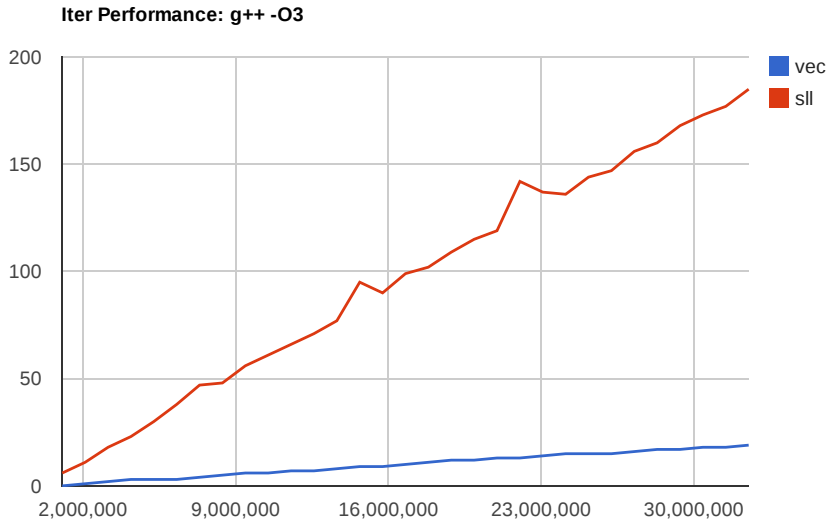
64 bit system:

- linked list value is 4 bytes
- linked list next is 8 bytes

Common cache line size: 64 bytes

- Linked list fits 64/12=5 items in cache before fetching new line
- Array holds 64/4=16 items in cache before fetching new line





Binary Search

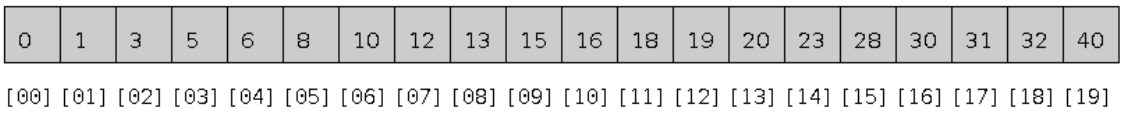
Binary Search requires that the collection has random access and is sorted. It has O(lgn) time complexity.

```
1:  // binarySearch.cpp - download here
2:
3:  int binarySearch(int * array, int n, int key){
4:    int low = 0;
5:    int mid;
6:    int high = n-1;
7:
```

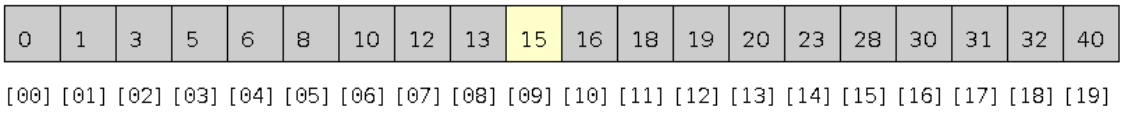
```
8:    while(low <= high){
9:      mid = (low + high) / 2;
10:     if(key < array[mid]){
11:       high = mid - 1;
12:     } else if(array[mid] < key){
13:       low = mid + 1;
14:     } else {
15:       return mid;
16:     }
17:   }
18:   return -1;
19: }
```

Question: Does the array contain 16?

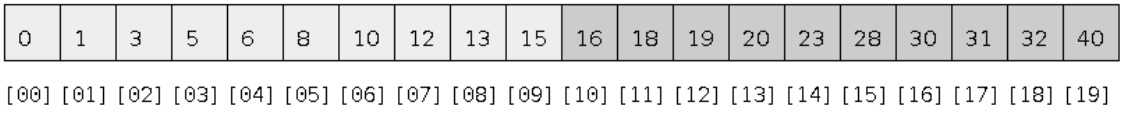
1. **start:** [low = 0, high = 19, n = 20]



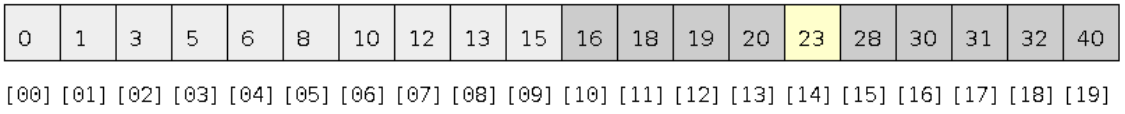
2. **iter1:** (low + high) / 2 = (0 + 19) / 2 = 9



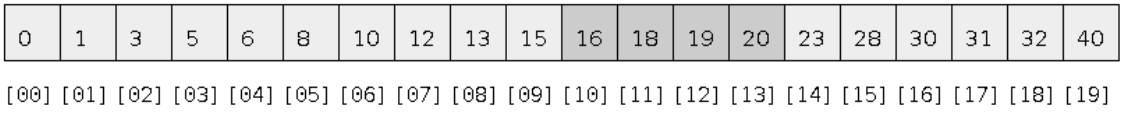
3. **iter1:** At mid (9), is 16 < 15? No. Go right. [low = 10, high = 19]



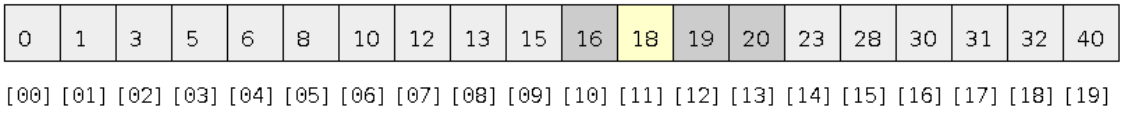
4. **iter2:** (low + high) / 2 = (10 + 19) / 2 = 14



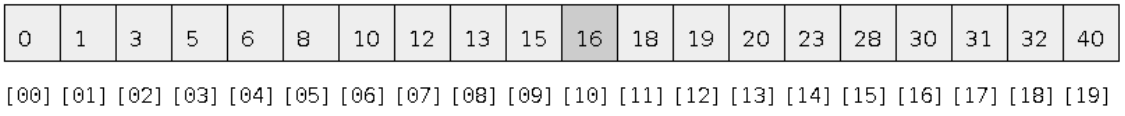
5. **iter2:** At mid (14), is 16 < 23? Yes. Go left. [low = 10, high = 13]



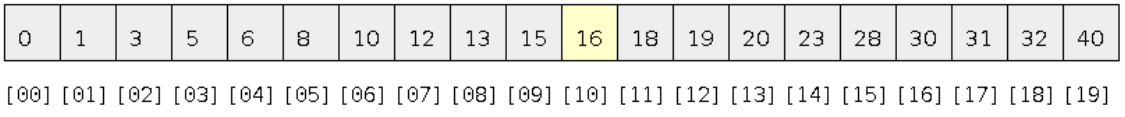
6. **iter3:** (low + high) / 2 = (10 + 13) / 2 = 11

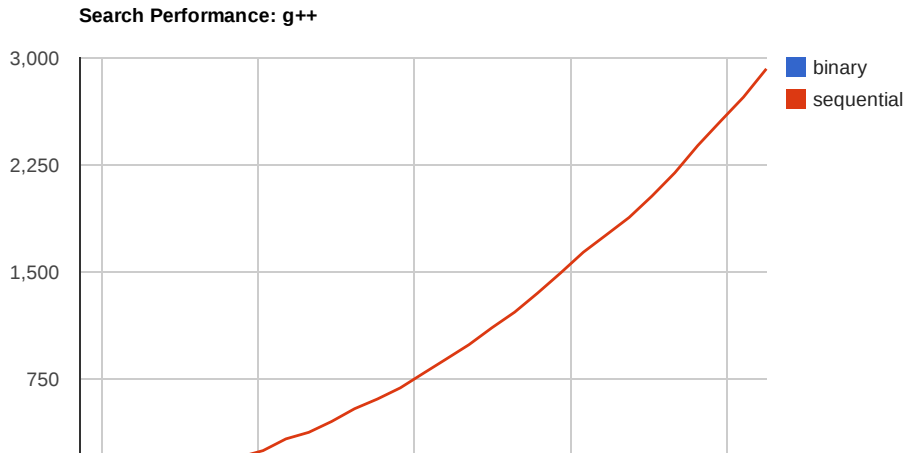


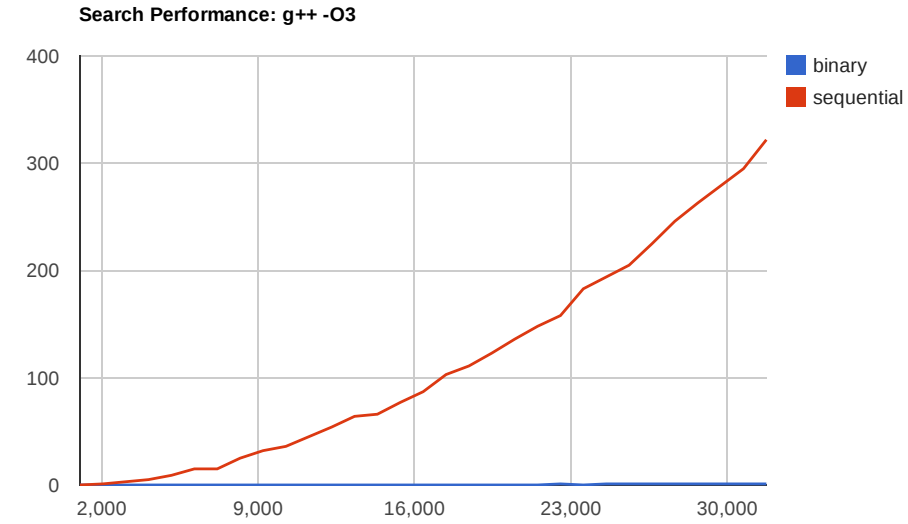
7. **iter3:** At mid (11), is 16 < 18? Yes. Go left. [low = 10, high = 10]



8. **iter4:** (low + high) / 2 = (10 + 10) / 2 = 10. At mid (10), 16 == 16. Found result.







References

1. Adam Drozdek. "Data Structures and Algorithms in C++"
2. Google Chart Tools