Data Structures
Sorted Arrays

Phil Tayco

Slide version 1.3

Sep. 7, 2021

Sorted Arrays

Definition

- As the name implies, a sorted array is an array where the elements maintain some sense of order
- This order can be anything that makes sense within the context of the data elements used:
  - Numerical sequential order of a key value like social security numbers
  - Reverse alphabetical order of strings like student last names
  - Function call order managing code execution
  - Numeric and mathematical symbol order for calculating equations
  - Character sequences representing compressed or encoded text
- Regarding the use of search, insert, update, and delete functions, the main reason to maintain some order to the data is to improve the efficiency of the search function

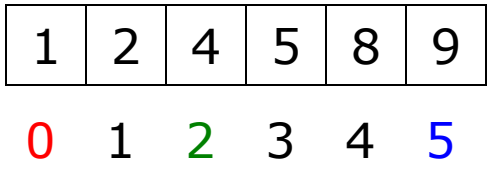
Sorted Arrays

Binary Search

- A linear search in a non sorted array performs at O(n) in the worst case
- By establishing an order, the binary search algorithm can be applied and significantly improves performance in the worst case
- The algorithm:
  - In an array of n elements, go to index [n/2]
  - If the record there is the one you want, you are done
  - If the record value there is smaller than your search value, all records less than the current record can be ignored – set your search range of elements to [n/2+1…n] and return to step 1
  - Otherwise, set your range of elements to [0…(n/2)-1] and return to step 1
  - Repeat this loop until you have 0 elements (record is not found) or record is found
- Review the example and the code in the next 2 slides noting the use of the index variables (color coded)

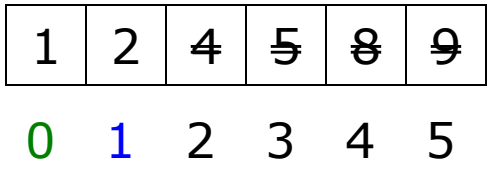
Sorted Arrays

binarySearch(2), first pass:



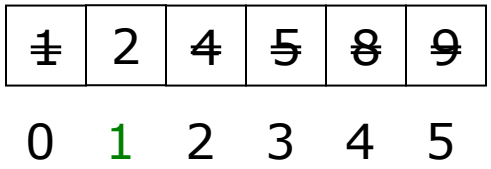
Lo = 0, hi = 5, cur = 2

binarySearch(2), second pass:



Lo = 0, hi = 1, cur = 0

binarySearch(2), third pass:



Lo = 1, hi = 1, cur = 1

Sorted Arrays

```
int binarySearch(int searchValue)
{
   int lowIndex = 0;
   int highIndex = currentSize - 1;
   int currentIndex;

   while (highIndex >= lowIndex)
   {
        currentIndex = (lowIndex + highIndex) / 2;
        if (numbers[currentIndex] == searchValue)
               return currentIndex;
        else if (numbers[currentIndex] > searchValue)
               highIndex = currentIndex - 1;
        else
               lowIndex = currentIndex + 1;
   }

   return -1;
}
```

Sorted Arrays

Binary search analysis

- Using the comparison operation as a unit of measure, each iteration of the while loop can be seen at worst as 3 comparisons performed
- Worst case scenario is an element not found:
  - 10 elements: 3 * (4 while loop iterations) + 1
  - 100 elements: 3 * (7 while loop iterations) + 1
  - 1000 elements: 3 * (10 while loop iterations) + 1
- What is the formula that captures the relationship between the size of the list and the number of iterations?
  - Remember we want the category of performance for O(), so we're more interested in the growth rate of the relationship and less on the exact mathematical formula

Sorted Arrays

Exponential formulas

- Consider the value of the exponent from given a base and the result
  - $8 = 2^3$
  - $16 = 8 * 2 = 2^4$
  - $32 = 16 * 2 = 2^5$
  - $64 = 32 * 2 = 2^6$
  - …
- With a base of 2, as the exponent increases by 1, the result doubles in size

Sorted Arrays

- These formulas can be restated in the form of a logarithm:
  - $\log_2 8 = 3$
  - $\log_2 16 = 4$
  - $\log_2 32 = 5$
- The key pattern is that as n doubles in size, the exponent result increments by one
- We can apply this with arrays treating the array size as n and the number of comparison sets (e.g. while loop iterations) as the exponent result
  - $\log_2 n$ = number of while loop iterations

Sorted Arrays

O(log n)

- As such, the mathematical pattern that this relationship is most similar to is a logarithm
- O(log n) is used to represent this category of efficiency
- It is significantly faster than O(n) but still not as efficient as O(1)
- Often this category is used when algorithms use a "divide and conquer" approach
  - The data set is divided in half each time we repeat a loop iteration

Sorted Arrays

So sorted arrays are better, right? Well…

- Recall worst case scenarios for unsorted arrays:
  - Insert: O(1)
  - Update, Search and Delete: O(n)
- For sorted arrays, the Search significantly improves to O(log n), but what about Insert, Update and Delete?
- Because the search is O(log n) and update and delete are using the search function, they too must be O(log n), which should be an improvement!
- Insert is O(1) so at the very least, we would want to keep insert in the O(1) category for sorted arrays

Sorted Arrays

Shifty results

- In order to keep the search at O(log n), we must keep the array ordered after an insert, update or delete is performed
- We must also maintain having no holes in the array
  - Insert needs to search for the right location to add the correct element and then shift any necessary elements Update must perform a search, and then the new value it is changed to may require moving the element to a new location and shifting other elements
  - Delete must perform a search, and then the elements must also be shifted over it so that we don't leave a hole

Sorted Arrays

Shifty results

- “Shift” appears to be a common theme which is worst case O(n)
- This means even though update and delete can use the binary search algorithm to perform its search, the worst case scenario of a shift keeps it in the O(n) category
- Furthermore for insert, the new element to add could require shifting all elements in the array to make space for the insert – this makes insert degrade from O(1) to O(n)
- There may be some optimizations and situations to look at though. Let’s examine each function individually

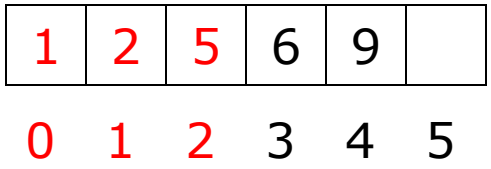
Sorted Arrays

Insert

- The algorithm requires searching for the correct location in the array for where the new element needs to be placed
- Once found, the elements to the right of it are shifted to make room for the new record
- We have two ways to search for this correct location: linear or binary search
- Let's start with linear search. Here we search for the insert spot starting at the first element
- We increment our current index until we find a value greater than the new value to be added
- We then shift the elements by 1 to the right of the current index and then add the new value

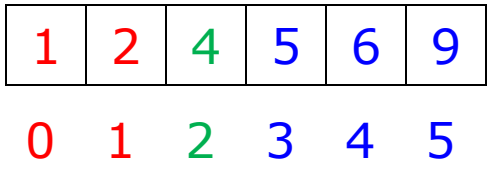
Sorted Arrays

Linear search in insert(4):



currentSize = 5

Shift at end of insert(4):



currentSize = 6

Sorted Arrays

```
boolean insertLinearSearchAndShift(int element)
{
   if (currentSize == numbers.length)
       return false;

   int currentIndex = 0;
   while (numbers[currentIndex] < element &&
              currentIndex < currentSize)
       currentIndex++;

   int insertIndex = currentIndex;
   for (int n = currentSize; n > insertIndex; n--)
       numbers[n] = numbers[n-1];
   currentSize++;

   numbers[insertIndex] = element;
   return true;
}
```

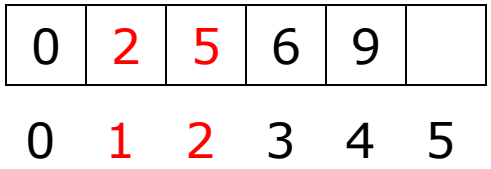
Sorted Arrays

Insert analysis

- The combination of the two loops together end up counting iterations that will go through the entire list linearly no matter where the new element goes
- This guarantees an O(n) performance
- For very large size lists, a guaranteed O(n) may not be good
- Will a binary search for the insert location first then be a better choice?

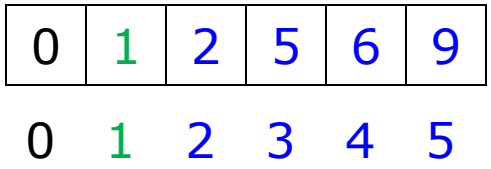
Sorted Arrays

Binary search in insert(1):



currentSize = 5

Shift at end of insert(1):



currentSize = 6

Sorted Arrays

```
boolean insertBinarySearchAndShift(int element)
{
   if (currentSize == numbers.length)
        return false;

   int lowIndex = 0;
   int highIndex = currentSize - 1;
   int currentIndex = 0;

   while (highIndex >= lowIndex)
   {
        currentIndex = (lowIndex + highIndex) / 2;
        if (currentIndex == 0)
               break;
        else if (numbers[currentIndex] > element &&
            numbers[currentIndex - 1] <= element)
               break;
        else if (numbers[currentIndex] > element)
               highIndex = currentIndex - 1;
        else
               lowIndex = currentIndex + 1;
   }
```

Sorted Arrays

```
    int insertIndex = currentIndex;

    if (element > numbers[insertIndex])
        insertIndex++;

    for (int n = currentSize; n > insertIndex; n--)
        numbers[n] = numbers[n-1];
    currentSize++;

    numbers[insertIndex] = element;
    return true;
}
```

Sorted Arrays

- With the linear search approach, we have a guaranteed O(n), no matter the state of the data in the array
- For the binary search approach are there best and worst case scenarios with the data?
- In the best case, search results in no shift (new element is added at end of the array) making the performance at best be O(log n)
- In the worst case, the search ends up at the beginning of the array, followed by an n-size shift
- Though the worst case is categorically O(n), additional analysis shows that situation also starts with an O(log n) search, which is worse than the linear search O(n)

Sorted Arrays

Algorithm analysis

- Algorithm 1 is always O(n) while algorithm 2 ranges from O(log n) to O(log n) + O(n)
- Categorically, both algorithms are O(n) in worst case. Which one to use?
  - Insertion of random values suggests algorithm 2 will be preferred sometimes
  - Algorithm 1 provides predictable consistency on performance time but does it matter versus algorithm 2 with many insertions over time?
- Regardless categorically, insert is worst case O(n) for sorted arrays

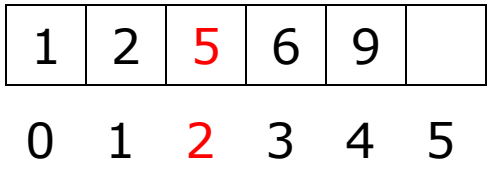
Sorted Arrays

Update

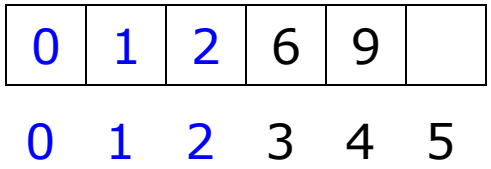
- This algorithm uses the search to find the value to change
- Once the value is changed, the new value may need to be moved to the correct spot to keep the array sorted
- Once the correct spot is found, elements must be shifted to make room for the updated value
- Because of the shift, we can take advantage of this by combining the search with the shift
  - If the updated value is to the left, linearly move in that direction shifting elements at the same time until you hit the right spot
  - If the updated value is to the right, find the correct location and then shift the rest of the elements over to add the new updated value
- Like the insert, we could use either binary search or linear search to find the value to update. Then, the new value is placed which will involve a linear shift

Sorted Arrays

Binary search in update (5, 0):



We change the 5 to a 0 and then place it in the correct location and shift other elements:



Sorted Arrays

Code for update (uses binary search function):

```
boolean updateBinarySearchMoveAndShift(int oldValue, int newValue)
{
    int recordIndex = binarySearch(oldValue);
    if (recordIndex == -1)
          return false;

    int nextIndex = 0;
    if (newValue > oldValue) // newValue is to the left
    {
          nextIndex = recordIndex + 1;
          if (nextIndex == currentSize)
                    return true;

          while(nextIndex < currentSize && newValue > numbers[nextIndex])
          {
                    numbers[nextIndex-1] = numbers[nextIndex];
                    nextIndex++;
          }
          nextIndex--;
    }
```

Sorted Arrays

Code for update (uses binary search function):

```
    else   // newValue is to the right
    {
          if (recordIndex == 0)
                    return true;
          nextIndex = recordIndex - 1;

          while(nextIndex >= 0 && newValue < numbers[nextIndex])
          {
                    numbers [nextIndex+1] = numbers[nextIndex];
                    nextIndex--;
          }
          nextIndex++;
    }
    numbers[nextIndex] = newValue;
    return true;
}
```

Sorted Arrays

Update analysis

- The use of the binary search makes the algorithm perform at least at O(log n)
- The worst case scenario here is the update of a value from one end of the array to the other
- The shift is unavoidably performed in a linear way adding a worst case O(n)
- Like the insert, the challenge is the linear shift. We can try different ideas with update to improve search and shift, but the worst case still leads to an O(n) performance
- Question: What if started with a linear search followed by a shift? What's the worst case scenario here?
  - Linear search could be finding the value to change at the end of the array (O(n)), and then the new value may cause a shift of all elements (another O(n))
- Next, what about delete?

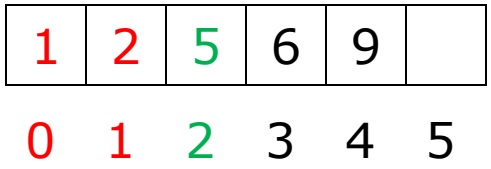
Sorted Arrays

Delete

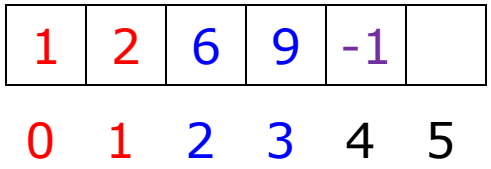
- Once again, we can either use a binary search or linear search to find the value to remove
- Once the value is found, elements from the right must be shifted to ensure there are no holes
- This leads to another situation similar to insert where we can go with a consistent O(n) and do a linear search followed by a shift

Sorted Arrays

Linear search in delete (5):



Shift at end of delete(5):



Note: in this example, we are using "-1" to represent an unused element – this value is arbitrary and can be anything depending on the context (this can be null or 0 or -99, etc.)

Sorted Arrays

```
boolean deleteLinear(int targetValue)
{
   int targetIndex = 0;

   while (targetIndex < currentSize)
   {
        if (numbers[targetIndex] != targetValue)
                targetIndex++;
        else
                break;
   }

   if (targetIndex == currentSize)
        return false;

   for (int n = targetIndex; n < currentSize - 1; n++)
        numbers[n] = numbers[n+1];

   numbers[--currentSize] = -1;
   return true;
}
```

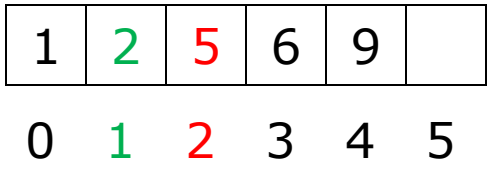
Sorted Arrays

Delete O(n) analysis

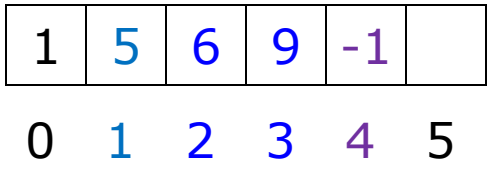
- The O(n) delete is a guaranteed O(n) solution. No matter the scenario
- As we did with insert, the search part of the delete algorithm could instead use a binary search
- Given what we analyzed with insert and update, do you think using binary search to find the element to delete will be a significant improvement?

Sorted Arrays

Binary search in delete (2):



Shift at end of delete(2):



Sorted Arrays

Code for delete using binary search:

```
boolean deleteBinary(int targetValue)
{
   int targetIndex = binarySearch(targetValue);

   if (targetIndex == -1)
        return false;

   for (int n = targetIndex; n < currentSize - 1; n++)
        numbers[n] = numbers[n+1];

   numbers[--currentSize] = -1;
   return true;
}
```

Sorted Arrays

Delete using Binary Search analysis

- The worst case situation is still O(n) solution, and arguably worse
- If the target value is the first element in the array, the search is a complete O(log n) followed by a complete O(n) shift
- The best case is like insertBinary at O(log n) in which the target value to remove is at the end of the array followed by no shift of elements
- Another case is if the element does not exist, this algorithm performs at O(log n) while the previous delete algorithm is always O(n)
- This means if deletes are occurring with elements that may not exist more often than not, deleteBinary may be preferred

Sorted Arrays

	Unsorted Arrays	Sorted Arrays
Search	O(n)	O(log n)
Insert	O(1)	O(n)
Update	O(n)	O(n)
Delete	O(n)	O(n)

Takeaways:

- Adding elements to unsorted data is preferred at O(1)
- Sorted data is preferred for doing searches
- Whether sorted or not, update and delete are O(n) in their worst case situations

Sorted Arrays

Summary

- Linear search based maintenance algorithms for insert and delete can guarantee O(n) performance – sometimes consistency is preferred
- Using a binary search based maintenance algorithm for insert, update, and delete ranges from O(log n) to O(log n) + O(n) – these can average out to O(n) with frequent use of these functions
- The big deal with the maintenance algorithms is the shift. When looking for better solutions and data structures, we look at weaknesses of one in hopes of improving on that
- The next data structure we will look at will address the shift along with another weakness of arrays, which is fixed memory management