Data Structures
Unsorted Arrays

Phil Tayco

Slide version 1.6

Sep. 9, 2021

Arrays

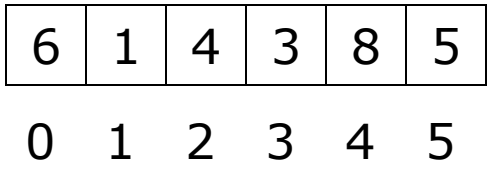
Our first traditional data structure

- If you are using the recommended textbook supplement, this material is covered in chapter 2
- Arrays in modern programming languages have different forms (ArrayList, dynamic memory allocated arrays, dictionaries, etc.)
- Depending on the language, many of the constraints we discuss may appear to be addressed
- We look at a more traditional view of an array and its design intentions versus a specific programming languages implementation

Arrays

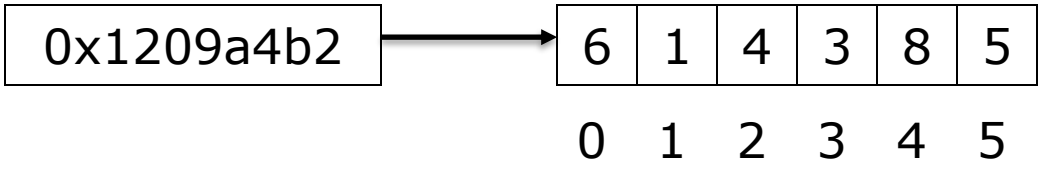
Definition

- This structure begins with reserving a specified amount of space for n number of elements
- Each element is the same data type
- Direct random access of any element is possible
- Element location is referred to as an “index” with the first index starting at a value of 0
- Example, an array of 6 integers:



Arrays

```
int[] numbers = new int[6];
numbers[0] = 6;
numbers[4] = 8;
```



- numbers is a memory address that is a reference to the actual location of the array
- The size of an int is known by the compiler
- Getting to the location of numbers[4] is a simple formula:
  - Memory address value + (4 * size of int)
- Direct access to any array element is very fast

Arrays

Considerations

- A specified amount of space for n number of elements must be reserved. This means you must consider maximum capacity at creation
  - This implies that some space may not be used
- Direct access to any element is consistent
  - This suggests fast performance to getting to any data element
- Array structure model can apply to other similar types of situations (e.g. reserving seats at a restaurant, spaces for cars in a parking lot, etc.)
- Let's analyze how the key functions perform with this data structure

Arrays

Insert

- Adding an element into an array requires 2 essential steps
  - Ensuring there is enough space in the array to add the new record
  - Adding the record while preserving the design intent of the array (i.e. does it remain unsorted or sorted)
- Many times, an additional variable representing how many elements are actually in use is maintained along with the array
  - The array may be reserved to hold 100 elements (capacity) but the number of elements in use may be 5 (current size)

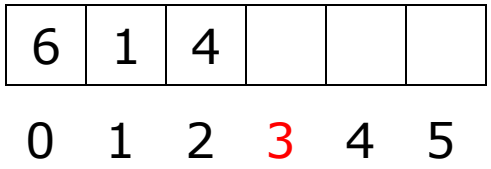
Arrays

Insert unordered 1: append algorithm

- If the order is insignificant, the key step after ensuring there is enough space is to find the next available space and add the new record there
- An effective way to maintain this is to use the current size variable as the index for the next available space to add a record
- Before append, if the current size is already equal to the array capacity, the append cannot be performed
- If insert is performed, the current size must be incremented

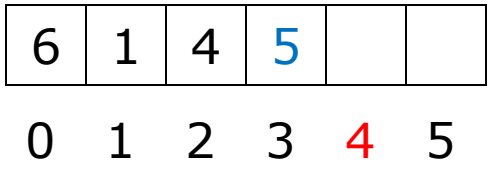
Arrays

array before append:



currentsize 3

array after append(5);



currentsize 4

Arrays

Sample code for append unordered:

```
// Given int[] numbers = new int[100];
// Given int currentSize = 0;

boolean append(int element)
{
   if (currentSize == numbers.length)
       return false; // Array is at maximum capacity

   numbers[currentSize++] = element;
       return true;
}
```

Arrays

Append unordered analysis

- We will analyze the performance using number of comparison operations performed
- Notice that whether the array is full or empty or somewhere in between, the check to see if the list is full is always performed
- There are no other comparison operations performed, so any time the append function is executed, there will always be one comparison done
- O() notation analysis looks at the performance of the algorithm as the list grows in size – will the number of comparisons change as the array gets larger in size?

Arrays

- No! Since there will always be a comparison performed regardless of the size of the list, we categorize this as constant performance, or O(1)
  - Note that the 1 in O(1) does not mean 1 comparison operation
  - The O(1) represents that the performance of the algorithm as the list grows in size is constant
  - If you have another algorithm that say performs 5 comparisons, if it always performed that amount regardless of list size, we would still say it's O(1)
- Question: Would it even be possible to eliminate the need to perform a comparison?
  - No. A capacity check for an array must always be done
- Question: What are we assuming about "holes" in the array?
  - This algorithm assumes there are no unused elements between used ones in the array

Arrays

Examine another insert methods?

- Given this ideal performance, it may seem unnecessary to explore other algorithms
- It’s good practice and science to keep an open mind and take a look at other ideas
- Using Big-O to show other algorithm performance gives us definitive and comparable information for more effectively comparing algorithms against each another

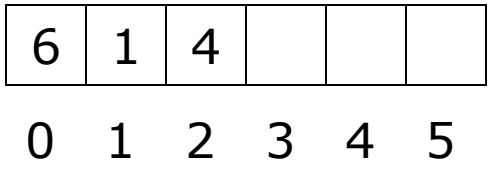
Arrays

Insert unordered algorithm 2: insertFront

- In the previous algorithm, the new element gets added to the end of the array
- What if we added it to the beginning of the array?
  - All elements “in front” of the new record are shifted over to the next spot
  - The need to check for capacity must still be performed

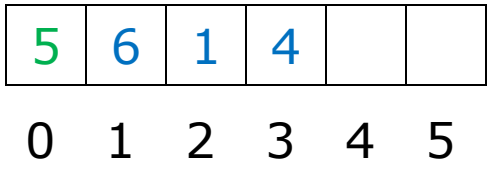
Arrays

array before insert:



currentsize 3

array after insert(5);



currentsize 4

Arrays

Sample code for insert unordered 2:

```
boolean insertFront(Type element)
{
   if (currentSize == numbers.length)
       return false; // Array is at maximum capacity

   for (int n = currentSize; n > 0 ; n--)
       numbers[n] = numbers[n-1];

   numbers[0] = element;
   currentSize++;
   return true;
}
```

Arrays

insertFront unordered analysis

- The code starts the same as append with the capacity check and then contains additional code for shifting elements (a loop)
- With this algorithm, the number of elements in use makes a difference – the more elements in use, the more elements need to be shifted
- Scenarios come into play here
  - What's the best case scenario(s)? No elements to shift
  - What's the worst case? Array only has 1 available space left (maximum number of elements to shift)
  - Note that another best case scenario is if the array is completely full (no insert or shift performed)

Arrays

- Best cases are O(1) performance because the array is either empty or full and the loop to shift elements does not occur
- Worst case performance is when only one space is available. This requires the loop to run the maximum number of iterations
- How many iterations are done depends on how many elements are in use
  - 5 elements shifted requires 5 iterations of the loop
  - 10 elements shifted requires 10 iterations of the loop
  - n elements shifted requires n iterations of the loop
- Each iteration contains a comparison operation – therefore as there are n elements in use, the number of comparisons is also around n

Arrays

- Recall that O() notation categorizes performance – it does not show the actual number of operations performed
- In the worst case for insertFront, the number of comparisons is proportional to the size of the list
- The proportion in this case is linear – meaning as n grows in size, the number of compare operations grows at a consistent rate
- We capture linear performance as O(n)
  - Whether the actual formula is 2n or n+1 or 3(n+1), we will still categorize it as O(n) to indicate that the performance rate related to the size of the list is linear

Arrays

- Now let's compare append to insertFront
- append does not have any different scenarios to deal with – whether the array is empty or full, the performance is O(1)
- insertFront's best case scenarios are O(1) but in the worst case, it is O(n)
- Clearly then, in the worst case comparable scenario, append is faster than insertFront because O(1) is significantly better than O(n)
- Looking at the code, this is probably intuitively obvious – what O() notation does, though, is provide systematic and mathematic notation to make the comparisons more formal

Arrays

A shifty problem

- The insertFront algorithm’s main problem is the use of the shift. If we want to still go with the idea of inserting the new element at the beginning of the array, to improve performance we must find a way to eliminate the shift
- Note now our overall context is that the data in the array can have unordered elements - we may be able to take advantage of that as well

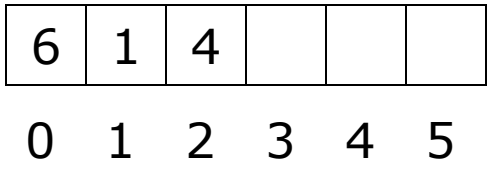
Arrays

Updated algorithm

- Check for maximum capacity as usual
- Instead of shifting elements, move the first element after the last element
- After the move, insert the new element at the first index

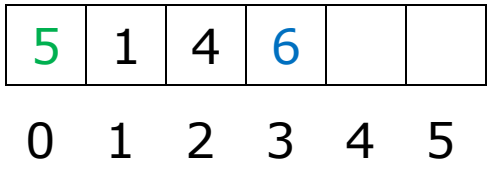
Arrays

array before insert:



currentsize 3

array after insert(5);



currentsize 4

Arrays

Sample code for insertFrontNoShift:

```
boolean insertFrontNoShift(Type element)
{
   if (currentSize == numbers.length)
       return false; // Array is at maximum capacity

   numbers[currentSize++] = numbers[0];
   numbers[0] = element;
   return true;
}
```

Arrays

- Keeping the same scenarios and analyzing comparison operations, the performance of insertFrontNoShift matches append at O(1) for all scenarios
  - No matter how large n approaches in the array, the number of comparisons performed is constant
- This is a good example of analyzing an algorithm (insertFront) and then finding a way to improve and demonstrate it using the Big-O process
  - We went from worst case scenario O(n) to O(1) and still be able to insert at index 0
- Does this mean insertFront using the shift has no use at all? Perhaps not, but it does help us understand the impact of a shift in situations where the order of the elements in an array need to be preserved (more on that later!)

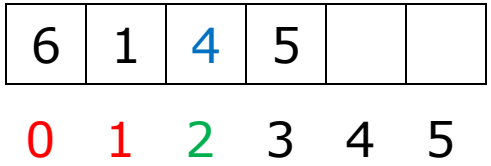
Arrays

Search

- As discussed in the introduction, the search in an unordered list is an O(n) operation in the worst case using comparisons as the operation type
- Worst case situation is the record is found in the last location or does not exist in a full array
- Best case is the record is found on the first try

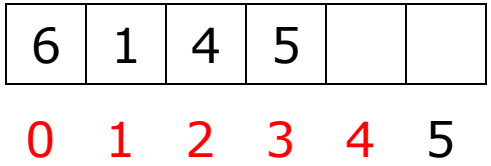
Arrays

search(4):



Returns 2

search(3);



Returns -1

Arrays

Sample code for linear search:

```
int linearSearch(int element)
{
   for (int n = 0; n < currentSize; n++)
       if (numbers[n] == element)
             return n;

   return -1;
}
```

Arrays

Linear search analysis

- Clearly, this is O(n) performance. Notice the actual number of comparisons
  - 1 comparison to control the loop
  - 1 comparison to check between element and array index
- The actual number of comparisons in the worst case is 2 * n. Why isnt this referred to as O(2n)?
- Remember our analysis is to first identify the algorithm category – linear performance is categorized as O(n)
- The actual number of comparisons is still important, but we focus on performance category first

Arrays

Can we do better?

- The inclination is to see if we can develop an algorithm that performs better than this
- Significantly better means a better category of performance. For linear search, O(log n) and O(1) are the only ones better than O(n)
- As discussed in the introduction slides, because the content of the array is unordered, we have to check every element in the worst case making it impossible for linear search to achieve anything better than O(n) for unsorted arrays

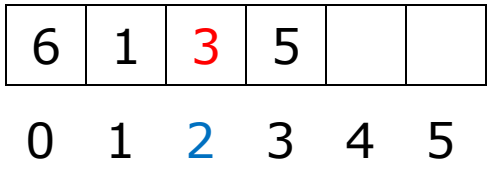
Arrays

Update

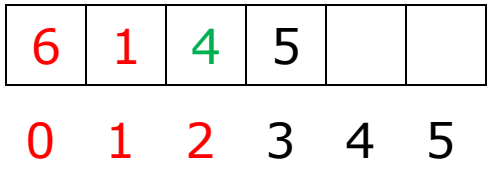
- The search algorithm has a significant implication with update (and delete)
- Update is essentially a search for an element and if found, modifying it while still maintaining the design intent of the structure
- In this case, the design intent is an unordered list making a modification of a record simple (there's no need to check if the new element needs to move somewhere else)
- We can use the search algorithm in our update

Arrays

Before update (3, 4):



After update(3, 4);



Returns true

Arrays

Sample code for update:

```
boolean update(int oldValue, int newValue)
{
   int searchIndex = linearSearch(oldValue);

   if (searchIndex == -1)
        return false; // Element to update is not found

   numbers[searchIndex] = newValue;
   return true;
}
```

Arrays

Update analysis

- Because the algorithm uses the search algorithm, the performance initially depends on the Big O of the search
- Since the search is linear in the worst case, update performance will at least be O(n) as well
- Are there additional comparisons to consider?
  - Notice after search is performed, there is one more comparison done to check if the record was found
  - Technically, this algorithm is O(2n + 1), but again, as far as a category is concerned, this is still O(n)
- Thus, the update performance is also linear in the worst case

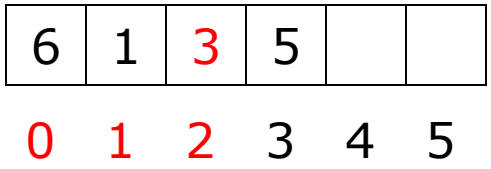
Arrays

Delete

- Now the delete. In most data structures, the delete function is usually found to be the most complex and is often treated last among the 4 functions
- The design intent of this structure is still unordered, so maintaining order after removing an element does not need to be considered
- Intuitive algorithm is to do a search for the element to delete and if found, remove it from the array

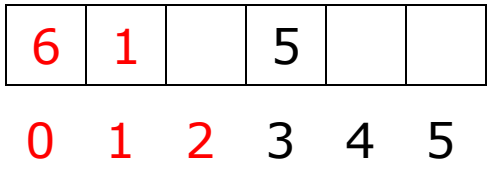
Arrays

Basic search and delete of 3:



currentSize = 4

After basic search and delete of 3



currentSize = 3

Arrays

Hole-y Toledo!

- The problem here is now the array is in a situation that affects the other algorithms discussed (insert, search, and update)
- We must avoid having “holes” in the array because that impacts the use of the “currentSize” variable as the location of the next available element
- The delete algorithm must include how we can fix the hole without impacting the other algorithms

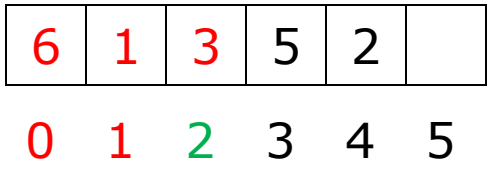
Arrays

The shifty approach to fix the hole

- One algorithm could use the following steps:
  - Use the search algorithm to find the element to remove
  - Shift over “to the left” any records after the record
  - Reduce the current size by 1
- Because of the insertFront and insertFrontNoShift algorithms, what does this solution suggest about the performance of this approach?

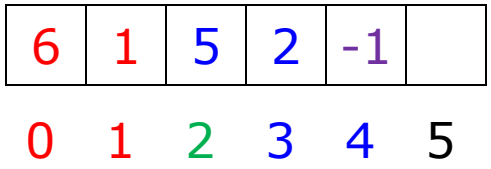
Arrays

Search in delete(3):



currentSize = 5

Shift at end of delete(3):



currentSize = 4

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.)

Arrays

Sample code for delete:

```
boolean deleteShift(int element)
{
   int searchIndex = linearSearch(element);

   if (searchIndex == -1)
        return false; // Element to delete is not found

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

   numbers[--currentSize] = -1; // “Remove” last element
}
```

Arrays

deleteShift analysis

- The algorithm starts with the search algorithm, which is O(n) in itself
- The loop after the search is a shift and implies another O(n) operation
- While the 2 apparent O(n) operations is still categorically O(n) as a whole, it feels like this could be a significant amount

Arrays

- However, the scenarios in which the search and shift apply actually end up performing consistently in all cases:
  - Best case search is O(1) followed by a worst case O(n) shift (first element found is deleted followed by shifting all elements in a full array)
  - Worst case search is O(n) followed by nothing to shift
- As a result, the category of the delete algorithm is still O(n), and a little digging into the details show that it is also a consistent O(n) in all scenarios
- This bit of information is useful for understanding the algorithm itself, as well as getting a feel for the process of analyzing any algorithm

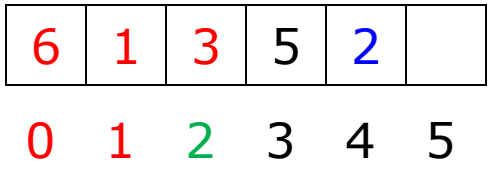
Arrays

deleteMove algorithm

- What happens if we tried to eliminate the shift?
- Like insertFrontNoShift, we can fix the hole by taking advantage of the fact that the array is unordered
  - Move the last element in the array (at the "currentSize" index location - 1) to the spot where the element was removed
  - Reduce the current size by 1

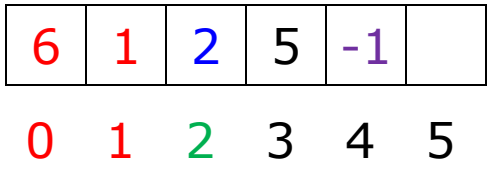
Arrays

Search in delete(3):



currentSize = 5

Move at end of delete(3):



currentSize = 4

Arrays

Sample code for 2nd delete:

```
boolean deleteMove(int element)
{
   int searchIndex = linearSearch(element);

   if (searchIndex == -1)
       return false; // Element to delete is not
   found

   numbers[searchIndex] = numbers[--currentSize];
   numbers[currentSize] = -1;

   return true;
}
```

Arrays

That worst case scenario though…

- In cases where the linearSearch is quick, deleteMove runs faster than deleteShift (i.e. we have a best case scenario)
- The worst case of the delete is still O(n) though because of the linear search through the entire array
- For both delete algorithms then, the worst case is O(n)
- Since they are in the same Big-O category, we can dig deeper:
  - Worst case search for both algorithms is O(n)
  - deleteMove is better than deleteShift for situations where the element to delete is near the front of the array

Arrays

Duplicates

- As we look at more data structures and algorithms, the question of how duplicate values are handled will recur and is also important to consider
- Duplicate values imply there is no difference between elements (if two elements have the same key value, they're not really different elements)
- As such, selecting between duplicate elements is arbitrary. If you have two elements with the same value, it should not matter which one you find in your algorithm
- Thus, for the search, insert, update and delete algorithms, duplicate values do not impact their performance

Arrays

Unordered Array Summary

- Worst case scenarios show the performances as:
  - Insert: O(1)
  - Update, Search and Delete: O(n)
- Holes are not intended in this data structure to ensure the currentSize variable is utilized properly
- Update and Delete are dependent on Search – if Search were somehow improved, Update and Delete could be positively impacted
- Search is improved dramatically if a sense of order is maintained. This is our next topic: Sorted arrays
- Do you think any tradeoffs will be made by sorting the data in the array? Think in terms of Big-O and worst case scenarios…