Lecture 5

Christopher Godley

CSCI 2270 Data Structures

June 11th, 2018

Recap: Arrays - Search

- Algorithm complexity: What is the runtime of this algorithm?

```
1. searchArray(A, v)
2.    found = false
3.    index = -1
4.    x = 0
5.    while(!found and x <= A.end)
6.                if A[x] == v
7.                            found = true
8.                            index = x
9.                else
10.                           x ++
11.   return index
```

O(n)

2D Arrays: Search

- Algorithm complexity: What is the runtime of this algorithm?

```
1. searchArray(A, v)
2.    found = false
3.    index = -1
4.    x = 0
5.    while(!found and x <= A.end)
6.          if A[x] == v
7.                found = true
8.                index = x
9.          else
10.               x ++
11.   return index
```

O(n^2)

Arrays: Add element

- Remember: elements of an array are stored contiguously
- Adding at the end is easy if there is room
- Pre-Conditions:
  - An array, *A*
  - A new value to insert, *v*
  - The index you want to store the new value in, *index*
  - The number of elements already in the array, *numElements*
- Post-Conditions:
  - Array *A* is updated such that *A[index] = v*
  - *numElements* has been incremented

Arrays: Add element

- Algorithm complexity: What is the runtime of this algorithm?

```
1. insertArrayElement(A, v, index, numElements)
2.    For x = numElements-1 to index
3.        A[x+1] = A[x]
4.    A[index] = v
5.    numElements++
```

O(n)

See Example 2 in section 3.2.2 from the textbook

Arrays: Deleting element

- Pre-Conditions:
  - An array, *A*
  - The index containing the value to delete, *index*
  - The number of elements already in the array, *numElements*
- Post-Conditions:
  - The value at A[x] is overwritten
  - *numElements* is decreased by 1

Arrays: Deleting Element

- Algorithm complexity: What is the runtime of this algorithm?

```
1. deleteArrayElement(A, index, numElements)
2.     for x = index to numElements-2
3.         A[x] = A[x+1]
4.     numElements = numElements - 1
```

O(n)

Arrays: Doubling

- Pre-Conditions:
  - *A* and *B* are arrays the same type
  - *B* >= *A*
- Post-Conditions:
  - Every index of *B* is the same as the corresponding index in *A*
    - *B[0] = A[0], B[1] = A[1], etc…*

Arrays: Doubling

- Algorithm complexity: What is the runtime of this algorithm?

```
1. copyArray(A, B)
2.     for x = 0 to A.end
3.         B[x] = A[x]
```

O(n)

- Or

```
1. doubleArray(A)
2.     B.length = A.length*2
3.     for x = 0 to A.end
4.         B[x] = A[x]
5.     return B
```

Memory

- The Stack
  - Local variables
  - Limited space
    - Carefully managed to preserve space
- The Heap
  - Much larger than stack
  - Used for storing variables created dynamically during runtime
  - Variables using pointers
    - Why? (intuition?)

Heap

- Features:
  - Allocated memory stays allocated until specifically de-allocated
    - Every call to *malloc()* must have a corresponding *free()*
  - Dynamically allocated memory must be accessed through a pointer
    - We don’t **need** pointers to work in the stack
  - Allocate large arrays, structures, and objects on the heap