Project 1 – Worst-case Complexity Analysis

CIS-350 SUMMER 2021

with Dr. Jinhua Guo

Demetrius Johnson

27 May 2021

Worst-case Complexity Analysis:

***The following analysis are based on my implementation of each of these function which can be found in the .cpp file**

// LIFECYCLE

- **SortedArray(); //default constructor**
  - Worst case time: O(1)
  - Variables are initialized and memory is allocated through one execution cycle

- **SortedArray(const SortedArray <Object> &from); //copy constructor**
  - Worst case time: O(N)
  - N is the size (not capacity) of the sorted array
  - Need to do a deep copy because of dynamic memory allocation using a single for-loop

- **~SortedArray(); //destructor**
  - Worst case time: O(1)
  - Constant time since only 1 command is issued: delete

//OPERATORS

- **const SortedArray & operator= (const SortedArray &from); //overloaded operator=**
  - Worst case time: O(N)
  - N is the size (not capacity) of the sorted array
  - This function is nearly exactly like the copy constructor in its functionality; uses only 1 for-loop to perform a deep copy

- **const Object & operator[](int idx) const; //access operator []**
  - Worst case time: O(1)
  - Constant time; simply returning a reference to a known memory location

- **bool equals(const SortedArray <Object> &rhs); //equals**
  - Worst case time: O(N)
  - N is there size (not capacity) of the sorted array; worst case is when two sorted arrays are the same, then the size (N) of all elements in the arrays will be compared using a single for-loop

- **bool empty() const; //check if size == 0**
    - Worst case time: O(1)
    - A single execution is needed to see if size == 0 by checking the size variable
- **int size() const;**
    - Worst case time: O(1)
    - A single execution is needed to return the value stored by the size variable

- **int capacity() const;**
    - Worst case time: O(1)
    - A single execution is needed to return the value stored by the capacity variable

- **void reserve(int newCapacity); //reserve a capacity length**
    - Worst case time: O(2N) = O(N)
    - N is the size of the array (not capacity)
    - Two for loops are needed: but they are not nested, one loop is used to store the array elements, then in a separate loop the elements are copied into the new capacity array, thus 2*N executions will occur, which simplifies to N operations for time complexity

//DISPLAY METHODS

- **void print(ostream &out, char delimiter = ',') const; //print all elements**
    - Worst case time: O(N)
    - N is the size of the array (not capacity)
    - Function simply uses one for-loop to iterate through all elements in the array but stops at array size (not the capacity)

//SORTED ARRAY PROTOCOTLS

- **void clear ();**
    - Worst case time: O(1)
    - Simply deallocate memory and reallocate it and reset the integer variables takes constant time

- **void insert(const Object &obj); //insert a new element**
    - Worst Case time: O(N)
    - N is the size (not capacity) of the array
    - In the worst case, array will have to be resized which means the reserve function is called (which has a worst case time of O(N).
    - Then, we use a binary search to find the insertion point which has a worst case time of O(log_2(N)) == O(log(N))
    - Then, we have to slide all elements down in order to make space for the new element, which takes at the worst case (element needs to be inserted at front) O(N)
    - O(N + N + log_2(N)) = O(2N + log(N)) ➔ as N approaches infinity, the final worst case time is: **O(N)**

- **void deleteMin();**
  - Worst case time: O(N)
  - N is the size (not capacity) of the array
  - The element at the front of the array has to be deleted, and so N-1 elements must be shifted to the left after the deletion

- **void deleteMax();**
  - Worst case time: O(1)
  - Constant time since the greatest element in the sorted array is at the end of the array so no element shifts are necessary
  - Simply decrease size of the array by 1

- **const Object & findMin() const;**
  - Worst case time: O(1)
  - Constant time since list is sorted, we simply return the 1st element (element 0)
- **const Object & findMax() const;**
  - Worst case time: O(1)
  - Constant time since list is sorted, we simply return the highest (greatest) element

- **int binarySearch (const Object &obj);**
  - Worst case time: O(log(N))
  - Binary search starts with N elements to search (the size, not capacity of the array)
  - Each iteration of the loop halves the number of elements to search
  - Thus time is O(log_2(N)) which simplifies to O(log(N))