**Data Structures and Algorithms I**

**Spring 2018**

**Homework #2**

1. For each of the following descriptions of something stored in memory when a program that was coded in C++ is running, state whether the memory is part of **static memory**, an **activation record** in the stack, or the **heap** (i.e., dynamic memory). For those entities that are stored in an activation record, state whether the memory resides in the **first, second, or third section of the activation record**, according to the breakdown discussed in class. For some of these parts, you may have to speculate about the most likely way that a class is implemented.
2. A char that is a global variable. **Static memory**
3. A char that is a local variable of a protected member function of a class **Stack[3]**
4. A double that is a parameter of a member function of a class. **Stack[1]**
5. An int that is a private data member of an object that is a local variable of a regular function **Stack[3]**
6. A double that is a protected data member of an object that is a parameter of a private member function of a class **Stack[1]**
7. A character that is stored in an old C-style string if the string is stored in a fixed-size array of characters that is declared locally in a private member function of a class **Heap**
8. A character that is stored in a C++ string object if the string object is a parameter of a regular function **Heap (depending on size, the string may actually be in a buffer, not allocated)**
9. A character that is stored in a C++ string object if the string object is a global variable **Static**
10. The value that a reference to an int refers to, if the reference is a parameter of a pubic member function of a class, and the argument passed to it is a local variable of a private member function of a class **Heap**
11. An int that is a private data member of an object that is part of a C++ vector of objects if the vector is declared locally in a regular function **Stack[3]**
12. A double that is part of a C++ vector of doubles if the vector is a parameter of a regular function **Stack[1]**
13. An integer that is a public data member of an object that is part of a C++ list of objects if the list is a local variable in a regular function **Stack[3]**
14. An integer that is a private data member of an object that is pointed to by a pointer in a C++ list of pointers to objects if the **object was dynamically allocated** and the list of objects is a local variable in a public member function of a class **Heap**
15. The previous value of a register used to point to the start of the second section of the current activation record, if the value has been backed up before the currently active regular function was called **Stack[2]**
16. The memory address stored by a pointer to an object (i.e., the value of the pointer, not the value of the thing that it points to) if the pointer is a global variable and the object that it points to is a local variable in a regular function **Static**
17. The memory address stored by a pointer to a int (i.e., the value of the pointer, not the value of the thing that it points to) if the pointer is a protected data member of an object, and the object is a local variable of a regular function, and the pointer points to a global int **Stack[3]**
18. A double that is a protected data member of a dynamically allocated object if the object is pointed to by a pointer that is a parameter of a public member function of a class **Heap**
19. Answer the following questions concerning lists, stacks, and queues:
20. Consider three implementations of a list of integers (each implemented as a C++ class). **The first implementation relies on a singly linked list**, and the object stores pointers to the start and end of the list. **The second implementation relies on a doubly linked list**, and the object stores pointers to the start and end of the list. **The third implementation relies on a dynamically allocated array of integers** (that can be resized to grow larger when necessary), and the object stores a pointer only to the start of the list. Assume that all implementations have been implemented in a reasonable fashion, that they all separately store the number of items currently in the list, that the first two implementations (relying on linked lists) do not use sentinel nodes, and that the third implementation separately stores the current capacity of the list. For each of three implementations, specify the worst-case running time of each of the following operations using big-Theta notation in terms of N, where N is the number of items in the list. If the average-case big-Theta running time is different, specify that also.
    1. An operation that inserts a new specified integer at the start of the list.

1. Constant 2. Constant 3. Constant

* 1. An operation that inserts a new specified integer after an integer item specified by a pointer. (That is, for the first two implementation, the routine is provided with a pointer to a node in the list, and an integer to insert after that node; for the third implementation, the routine is provided with a pointer to an integer in the dynamic array, and an integer to insert after that item.)

1. Constant 2. Constant 3. N(linear)

* 1. An operation that inserts a new specified integer before an integer item specified by a pointer. (That is, for the first two implementation, the routine is provided with a pointer to a node in the list, and an integer to insert before that node; for the third implementation, the routine is provided with a pointer to an integer in the dynamic array, and an integer to insert before that item.)

1. N (linear) 2. Constant 3. N (linear)

* 1. An operation that deletes an integer from the end of the list. (If the list is empty, the operation should not do anything.)

1. Constant 2. Constant 3. Constant

* 1. An operation that returns the value of the integer at position N/2 (rounded down if N is odd) in the list. (If the list is empty, the operation should throw an exception.)

1. N+N/2 2. N+N/2 3. Constant

1. If the recursive routine used to compute Fibonacci numbers (covered in class as part of a previous topic) is run for N = 50, is stack space likely to run out? Explain your answer.

Yes, stack space will certainly run out. Because each instance of the fibonacci() function calls fibonacci(n-1) and fibonacci(n-2), each function will call 2 additional functions, essentially creating 2^50 activation records

1. Assume you have access to an implementation of a stack class that provides worst-case constant time push and pop operations. Explain how to use two such stacks to implement a data structure that supports the same push and pop operations as the provided stack class and also a getMinValue operation that returns the value of the minimum item in the data structure (but does not alter the data structure), supporting all operations in worst-case constant time. You can assume that the values of items are comparable to each other using the standard comparison operators for less-than and greater-than.

One Stack1 will hold the actual elements of the stack that are pushed or popped. Stack2 will hold the current smallest elements of Stack1. Whenever a new value is pushed onto Stack1, it will be compared to the value at the top of Stack2 (which will be the most recent addition to Stack2 because of the LIFO nature of stacks). If the new element is smaller or equal to the the top of Stack2, it will be pushed onto Stack2, in addition to Stack1. If the new element is larger than the top of Stack2, it will simply be added to Stack1. When a pop occurs, we will check if the value being popped is the same as the element at the top of Stack2. If it is indeed the same value, it will be popped from both stacks. Otherwise we will only pop it from Stack1.

An example of program is shown in the charts below

|  |  |
| --- | --- |
| **PUSHES** | |
| Stack1 | Stack2 |
| 5 | 5 |
| 65 | 5 |
| 465 | 45 |
| 4465 | 445 |

|  |  |
| --- | --- |
| **POPS** | |
| Stack1 | Stack2 |
| 4465 | 445 |
| 465 | 45 |
| 65 | 5 |
| 5 | *5* |