Analysis of Stack and Queue Implementations

1. Differences in computational complexity

Table 1 compares the time complexity of each operation of the stack and queue implementations.

Table 1 -- Time complexity of the operations available to stack and queue implementations

	ArrayStack	ListStack	ArrayQueue	ListQueue
push	O(1)	O(1)	N/A	N/A
pop	O(1)	O(1)	N/A	N/A
top	O(1)	O(1)	N/A	N/A
size	O(1)	O(1)	O(1)	O(1)
isEmpty	O(1)	O(1)	O(1)	O(1)
enqueue	N/A	N/A	O(1)	O(1)
dequeue	N/A	N/A	O(n)	O(1)
front	N/A	N/A	O(1)	O(1)

The only divergence in asymptotic time complexity is the dequeue operation of ArrayQueue and ListQueue. This operation in ArrayQueue takes linear time because every element must be repositioned after the front element is removed. In ListQueue, this operation takes constant time because the header sentinel must only be modified to point to the next element.

The space complexity of all four implementations are O(n). Note, however, that the coefficients are higher for both list implementations because of the need for sentinels and pointers.

2. Advantages in scheduling situations

Each stack data structure works on a last-in-first-out basis so they would be suited to a newest-process-highest-priority scheduler. Although ListStack has more overhead than ArrayStack, they both take constant time to perform their operations. The advantage of ListStack comes from its ability to scale dynamically.

Each queue data structure works on a first-in-last-out basis so they would be suited to a first come, first served scheduler. ListQueue has more overhead than ArrayQueue but it has advantages in removing processes from the queue and scaling dynamically.

3. Choosing between implementations

Section 3 made a clear distinction on the scheduling situations where a stack or queue would be required but it did not detail the trade-offs of choosing one stack or queue implementation over the other. What follows is a policy—highlighting space and time complexity—on when to use a certain implementation.

- LIFO scheduling

  - ArrayStack – If the maximum number of processes is fixed, the combination of constant time operations and low memory overhead make this implementation ideal.
  - ListStack – If the maximum number of processes is unknown, the combination of constant time operations and dynamic memory allocation make this implementation ideal. A singly linked list was chosen specifically to reduce the overhead of storing pointers with respect to other list structures.
- FIFO scheduling
  - ArrayQueue – The linear dequeue operation makes this implementation ideal only if space constraints are sufficiently more pressing than time constraints.
  - ListQueue – If it is important that all operations take linear time and the additional overhead of storing pointers is not of great concern, this implementation is ideal. A singly linked list with header and trailer sentinels was chosen specifically to reduce the overhead of storing pointers with respect to other list structures.

4. Future extensions and improvements

The data structures implemented thus far do not lend themselves to sophisticated scheduling algorithms. Future extensions may require priority-based scheduling or access to arbitrary elements of the stack or queue. The data structure for a priority-based scheduler will need to keep a record of the priorities of its elements and, if we must have access to arbitrary elements, the data structure will need to implement such an operation with optimal space and time complexity.

Appendix I

Analysis of a priority queue implementation

The priority queue implementation, ProtoPriorityQueue, has a space complexity of O(n) which is just as good as the other implementations described in this paper. The following table shows the time complexity of the various operations which can be performed on this implementation of the priority queue.

Table 2 -- Time complexity of the operations available to the ProtoPriorityQueue implementation

size	O(1)
isEmpty	O(1)
min	O(1)
insert	O(n)
removeMin	O(1)

The insert operation is O(n) because there is a loop which enforces that the new element gets inserted in a position such that the queue is in sorted order. This allows all other operations to take constant time. It would be possible to optimize insertion further by implementing the underlying structure as a tree instead of a linked list. A tree would allow insertion in O(lg n) which would correspond to a significant speed up for large numbers of elements.