COS 226 Bob's Notes: Stacks, Queues, Deques, and Steques
Fall 2010
R. Tarjan, Preceptor

In these notes I'll talk about linear data structures, which support various sets of operations on a list or sequence. I'll also say a bit about amortization (a big word – look it up).

The standard linear data structures are the stack, which supports last-in, first-out access, and the queue, which supports first-in, first-out access. Both of these are restrictions of a more general data structure, the double-ended queue, or deque (terminology suggested by Knuth, pronounced to rhyme with deck), which supports insertion and deletion at both ends. Implementing a deque is the first part of the second programming assignment. I'll use the following names for the deque update operations, in contrast to the longer names used in the assignment: do not get confused by the difference in terminology.

push(item): add the item to the front of the deque (called addFirst in the assignment)
inject(item): add the item to the back of the deque (called AddLast in the assignment)
pop(): remove and return the item on the front of the deque (called removeFirst in the assignment)
eject(): remove and return the item on the back of the deque (called removeLast in the assignment)

This data structure is symmetric: if in any sequence of deque operations we swap push and inject, and swap pop and eject, this does not change the outcome (the sequence of items returned by the pop and eject operations). The item on the front of the deque may also be called the first item or the top item; the item on the back of the deque may also be called the last item or the bottom item.

A stack is just a deque whose only update operations are push and pop (inject and eject are disallowed). A queue is a deque whose only update operations are inject and pop (push and eject are disallowed). We generally use the "top-bottom" terminology with stacks and the "front-back" or first-last" terminology with queues, but this is merely a convention. (The use of different terms for the same thing is common in mathematics and many sciences, so we might as well get used to it; in my notes I will generally use my favorite terminology and try to point out where it differs from what is in the other course materials.)

It is natural to consider other subsets of the update operations. Allowing only one operation produces a data structure that is either always empty (nothing can be added) or a black hole (nothing can be removed), so to get a useful data structure we need to allow at least one of push and inject, and at least one of pop and eject. Push plus eject gives us a queue; inject plus eject

gives us a stack. (Why?) More interesting are subsets of three out of the four operations. This gives us two new data structures. If we disallow inject (or equivalently push), we get what Knuth calls an input-restricted deque. If we disallow eject (or equivalently pop) we get what knuth calls an output-restricted deque. Of these two, the latter seems to be significantly more useful: it is a combination of a stack and a queue, and I prefer to call it a stack-ended queue, or steque (pronounced to rhyme with deque).

Before looking at the implementation of these data structures, let us study their power, and in particular what kind of permutations they can generate: this is the subject of the two problems in exercise 3. Suppose we do 2N update operations on an initially empty queue, consisting of N inject operations and N intermixed pop operations, such that the k^th inject operation adds item k – 1 to the queue (so that the N inject operations add 0 through N – 1 in increasing order to the queue), and when each pop occurs the queue is non-empty (the k^th pop occurs after at least k injects). The sequence of outputs is a permutation of 0 through N – 1. What permutations are possible? Only one, the identity permutation: the items leave the queue in the same order they enter. If we define an inversion in a permutation to be a pair a < b such that b occurs before a (possibly with other numbers in between). Another way to say that a queue can generate only identity permutations (of which there is one for each possible value of N) is this: a queue can generate a given permutation if and only if the permutation contains no inversions. (Why?) Incidently, the number of inversions in a permutation is one way to measure its "unsortedness."

What happens with a stack? What permutations on 0 through N – 1 can we generate by starting with an initially empty stack and doing N pushes, the k^th of which pushes k – 1, intermixed with N pops, such that for all k the k^th pop is preceded by at least k pushes? As we discussed in class, if a permutation can be generated in this way, the unique way to do it is by repeating the following step: if the next item in the output permutation is on top of the stack, pop it; otherwise, push the next input item. Eventually either the desired permutation will be completely generated or the next item to be output will be on the stack but not on top. In this case there is no way to generate the desired permutation. Each push or pop is forced; there are no choices. Thus we can test in linear time whether a given permutation can be generated: we execute the forced sequence of pushes and pops.

This algorithm leads to another way to characterize permutations generable by a stack. Whereas the permutations generable by a queue are all those without a certain forbidden pair, namely an inversion, those generable by a stack are all those without a certain forbidden triple. The forbidden triple is a < b < c, with c occurring first, a second, and b third, possibly with other items between c and a and between a and b. In explaining this (in both precepts!), your loyal preceptor had a classic "brain freeze." I plan to do better here. There is a lesson here: anyone can have a brain freeze. Answering questions in real time in front of a group is hard. Be aware of this the next time you have an oral exam. Be prepared, even over-prepared. If you have a

brain freeze, call a time-out (if possible) breathe a few times to give yourself a chance to clear your head, and think. If this doesn't work, admit failure and move on, but come back to the question later if you have the opportunity.

Returning to forbidden triples, suppose a permutation contains a forbidden triple. Since item c must be popped before a and b, but a and b must be pushed before c, both a and b must be on the stack when c is pushed. Since a must be pushed before b, item b is above item a when c is pushed. But a must be popped before b, which is impossible. Thus a permutation that contains a forbidden configuration is not generable.

We want to prove an equivalence, namely that a permutation is generable IF AND ONLY IF it does not contain a forbidden triple. We just proved that a permutation containing a forbidden triple is not generable. This is the "only if" direction of the equivalence. Actually, it is the "contrapositive" of the only if direction, but if you remember your Boolean logic you will know that an implication is equivalent to its contrapositive. If you don't remember Boolean logic, or have never seen it, then you have one more cool thing to learn(!), but for now let me answer your question,"What is a contrapositive?" Given a logical implication, "if statement A is true then statement B is true," its contrapositive is, "if statement B is false then statement A is false." Boolean logic tells us that an implication is true if and only if its contrapositive is true. This comes in handy when trying to verify an implication: sometimes it is easier to verify the contrapositive than the original statement. It also comes in handy when trying to verify an equivalence, as we are trying to do here: an equivalence consists of two implications.

Back to the task at hand. We shall prove the contrapositive of the "if" direction. Suppose a permutation is not generable. Run the algorithm above. Then the algorithm must get stuck. That is, the next item in the output permutation is on the stack but not on top. Call this item a and the item immediately above it on the stack b. Item b must follow item a in the output permutation, since otherwise the algorithm would have got stuck earlier. Furthermore $a < b$ since a was added to the stack first. Let c be the next item in the output permutation when b was added to the stack. Item c cannot be a, b, or any other item on the stack when b is pushed: if it were a, a would be popped; if it were b, b would be pushed and then popped, and if it were another item on the stack then the algorithm would already be stuck. Thus must not yet have been pushed, which means that $c > b$. Of the three items a, b, c, c occurs first in the output permutation. Thus a, b, c is a forbidden triple. This gives us the "if" direction of the equivalence. This also proves that the algorithmic test for generability is correct, since if the algorithm gets stuck there must be a forbidden triple, and a permutation with a forbidden triple is not generable.

The problem of characterizing generable permutations becomes harder for more-powerful data structures. In precept I suggested you think about this problem for the case of two stacks in

parallel: the allowed updates are to push the next input item to one of the stacks, or to pop an item from one of the stacks and add it to the output. For two stacks the sequence of update operations is not forced: each item has a choice of stacks. It would seem that one would need to search an exponential number of push and pop sequences to determine if a given permutation is generable. In fact, there is a linear-time algorithm to test whether a sequence is generable by two stacks; the data structure it uses is related to a key part of the linear-time algorithm I developed with John Hopocroft for my Ph.D. thesis. These are worthy topics for COS 423, but they are "beyond the scope" of COS 226, as they say.

A problem very similar but not identical to the two-stacks problem is that of characterizing permutations generable by a deque. There is a linear-time algorithm for this problem that is a variant of the algorithm for two stacks. Every permutation generable by a deque is generable by two stacks, but not vice-versa. (Exercise: prove this.) Indeed, a pair of stacks is almost functionally equivalent to a deque, as I discuss below.

Instead of thinking about generable permutations, we can turn the question around and ask what permutations can be sorted by a stack, or two stacks, or a deque. That is, the input is a permutation of 0 through N – 1, and the output is 0 through N – 1 in increasing order. The generability question for a particular data structure is equivalent to the sortability question if each insertion operation has a corresponding deletion operation, and vice-versa: we obtain sortability from generability by reversing the roles of the input and the output and of the insertion and deletion operations. (Exercise: verify this.) A data structure for which one cannot do this is the steque. (Why?) One of my first papers studied sortability of networks of queues and stacks. One explanation for my brain freeze in precept was confusing generability and sortability: the forbidden triple for stack generability maps to a DIFFERENT forbidden triple for stack sortability. (What is it?)

Now let's review ways of implementing a stack, queue, steque, or deque using either a linked or an array representation. We have seen how to represent a stack as a singly linked list of item, accessed by a pointer to the first item, and how to extend this representation to a queue by adding a pointer to the last item. The latter representation in fact supports a steque: push, pop, and inject all take constant time. An alternative that avoids two access pointers is to make the list circular: the last item points to the first. Access is by a pointer to the last item. Each of the operations push, pop, and inject can still be done in constant time. Either the linear represention with two access pointers or the circular representation with one access pointer also supports catentation of two steques (or stacks, or queues) in constant time. (Exercise: implement catenable steques as circular singly-linked lists.)

Singly-linked lists are not powerful enough to support all four deque operations in constant time. The most straightforward way to add a constant-time eject operation to the steque operations is

to make the list doubly-linked, either linear with two access pointers or circular with one access pointer.

The alternative to a linked representation is an array representation. We have seen how to represent a stack in an array, with a stack of size k in positions 0 through k – 1, with 0 the bottom and k – 1 the top. If we use wraparound, we can represent a queue, a steque, or even a deque in one array, as long as the number of items in the data structure does not exceed the size of the array: if the array has size N and the data structure contains k items, the items are in positions j mod N, j + 1 mod N, …, j + k – 1 mod N.

With an array representation, we need to think a bit about memory management. One problem that definitely requires a solution is overflow, when an insertion causes the size of the data structure to exceed the number of array positions. A lesser issue that we may or may not want to address is underflow: the size of the data structure may shrink to be much less than the size of the array, in which case we are wasting space. The solution to both problems proposed in class is on overflow to copy the entire data structure into an array of twice the size, and on underflow, defined to be a deletion that reduces the size of the data structure to ¼ the array size, copy the data structure into an array of half the size. Each such copying takes time linear in the size of the data structure, say k, but our hope is that over a sequence of operations the total cost of copying is linear in the number of operations, constant per operation. That is, we hope to amortize the copying over the operation sequence.

As noted in class, it does NOT work to define underflow to be when the data structure shrinks to half the size of the array, because an alternating sequence of insertions and deletions would result in overflow on each insertion and underflow on each deletion, causing each operation to take linear, not constant time. We need to introduce "hysteresis" into the implementation, so that each time a copy must be done it will have been preceded by enough operations to pay for it. This is the reason to define underflow to be ¼ full; 1/3 or 1/10 would work just as well though the constants would be different.

Kevin claimed in class that if we implement a deque, say, in this way, then the total time for m deque operations, starting from an empty deque, is proportional to m. Let's prove this. It helps to have a formal framework for studying such situations. We want to average the cost per operation over a sequence of operations, in the hope that the cost of expensive operations will be balanced by the cost of cheap ones. We call this king of time-averaging "amortization." The dictionary definition of amortization is to pay off a loan in equal monthly installments. We apply this concept to the analysis of algorithms. Imagine the computer as being coin-operated. We shall call our coins "credits." Each unit of computer time costs one credit. (A unit of computer time could be a second, a minute, or an hour; we shall adjust the constant factor as needed.) We want to do some sequence of operations. We assign to each operation a certain

number of credits; this number is the "amortized time" of the operation. We do an operation by spending the available credits. If the operation is fast, we may not need all the credits allocated to the operation. In this case we can save the unused credits to spend on future operations. On the other hand, if the operation consumes all the credits allocated to it and it needs more time, then we must either spend previously saved credits or borrow credits. Our goal is to show that, no matter what sequence of operations occurs, the credits allocated to the operations are enough to pay for the running time of all the operations, including any borrowing that may be required. Then the total running time is at most the sum of the amortized times of all the operations. Thus we are justified in using the amortized time of an operation as a conservative estimate of its actual running time. Our goal is to find an assignment of credits to operations (as few as necessary) that smoothes out variations in running times, using fast operations to pay for slow ones. In many situations, including the one we are about to analyze, no borrowing is necessary.

Let us apply this idea to an array implementation of a deque. We need a way to pay for copying using credits saved from earlier operations. To keep track of saved credits, we define a "credit invariant" that relates the state of the data structure (how close it is to needing to be copied) to the number of saved credits. Let N be the number of array positions and k the number of deque items. N is a power of two; $N/4 < k <= N$. We assume that a deque operation that does not do a copy takes one unit of time and a deque operation that does do a copy takes 2N time in the case of an overflow and N/2 time in the case of an underflow; that is, time equal to the size of the new array. (One can adjust the constant factors to account for copying into the new array and disposal of the old array, but this just changes the constant factor in the analysis.) Our credit invariant is: the number of saved credits is at least $4|k – N/2|$. We allocate five credits to each deque operation.

Claim: Any sequence of deque operations starting from an empty deque maintains the credit invariant. Thus we never run out of credits, and the total time for M deque operations is at most 5M.

Proof: Initially the stack is empty and there are no saved credits. The first operation must be a push or inject, which gets five credits. If we initialize N = 2, the push or pop takes 2 units of time. After the push or inject, k = N/2, so the credit invariant requires no saved credits; 3 credits are left over. Suppose the credit invariant holds before some deque operation that does not cause overflow or underflow. The operation takes one unit of time and the credit invariant increases or decreases by four credits, so the five credits given to the operation suffice to do the operation and maintain the invariant. Suppose the credit invariant holds before a push or inject that causes an overflow. The time for the operation is 2N, which is at least the number of saved credits by the credit invariant. The copying doubles N; after the operation, k = N/ + 1, so the credit invariant requires four saved credits. These come from the five given to the operation, leaving one left over. The analysis of an operation that causes an underflow is similar.

Exercise: Do the analysis for the underflow case.

Amortization is a very powerful tool in the analysis of algorithms. Although the preceding example may be the only one we see in COS 226, there are many more examples in COS 423, including the analysis of the path compression technique used in the union-find data structure. The credit-based framework for amortization is called the "banker's paradigm."

One can efficiently store more than one linear data structure in a single array. The simplest instance of this is two stacks in one array: the first stack starts at the lowest position and grows up, the second stack starts at the highest position and grows down. As long as the sum of the sizes of the two stacks does not exceed the array size, each stack operation takes constant time in the worst case. One can use the doubling/halving idea to handle overflow and underflow. This makes the constant time per operation amortized rather than worst case. This idea generalizes substantially: for any given integer k, one can store k deques in a single array so that the amortized time per deque operation is proportional to k (thus constant if k is constant), and such that the space used remains linear in the total size of all the deques.

Problem: Verify this.

A natural question to ask when one has a data structure with operations that have good amortized time bounds but not good worst-case time bounds is whether the amortized bounds can be made worst-case. The answer is, sometimes yes, sometimes no. In the case of an array representation of a deque or set of deques in Java, creating an array of size N takes ~N time, and there is no way around this, so we are stuck. On the other hand, if creation of an array of any size takes constant time, but the array is uninitialized (its entries are unspecified and could be anything), then one can "deamortize" the array implementation of a deque so that each deque operation takes constant time in the worst case. The same is true of the set-of-deques representation mentioned in the previous paragraph.

Problem: Verify this.

I'll close with a few problems on the relationships among some of these data structures.

Problem: Implement two stacks using a single deque so that each stack operation takes a constant number of deque operations in the worst case.

Problem: Implement a steque using two stacks so that each steque operation takes a constant amortized number of stack operations.

Problem: Implement a deque using a stack and a steque so that each deque operation takes a constant amortized number of stack and steque operations.

Problem: Implement a deque using three stacks so that each deque operation takes a constant amortized number of stack operations.

Problem: Implement a deque using a constant number of stacks so that each deque operation takes a constant number of stack operations in the worst case. You can assume the existence of a method that returns the size of a stack in constant time, and your simulation can do side calculations to determine what stack operations to do, as long as they take constant time. The solution to this problem gives a solution of the first part of the second programming assignment that uses singly-linked instead of doubly-linked lists, but the conceptual, programming, and time overhead for the deamortization is high.

I had originally planned to include here some notes on mathematical induction, but these notes are already long enough, and there is much good material on induction the web. See for example http://www.math.utah.edu/mathcircle/notes/induction.pdf