CMSC 420: Lecture 2
Some Basic Data Structures

Basic Data Structures: Before we go into our coverage of complex data structures, it is good to remember that in many applications, simple data structures are sufficient. This is true, for example, if the number of data objects is small enough that efficiency is not so much an issue, and hence a complex data structure is not called for. In many instances where you need a data structure for the purposes of prototyping an application, these simple data structures are quick and easy to implement.

Abstract Data Types: An important element to good data structure design is to distinguish between the functional definition of a data structure and its implementation. By an *abstract data structure* (ADT) we mean a set of objects and a set of operations defined on these objects. For example, a *stack* ADT is a structure which supports operations such as *push* and *pop* (whose definition you are no doubt familiar with). A stack may be implemented in a number of ways, for example using an array or using a linked list. An important aspect of object-oriented languages, like Java, is the capability to present the user of a data structure with an *abstract view* of its function without revealing the methods with which it operates. Java's *interface/implements* mechanism is an example. To a large extent, this course will be concerned with the various approaches for implementing simple abstract data types and the tradeoffs between these options.

Linear Lists: A *linear list* or simply *list* is perhaps the most basic of abstract data types. A list is simply an ordered sequence of elements $\langle a_1, a_2, \ldots, a_n \rangle$. We will not specify the actual *type* of these elements here, since it is not relevant to our presentation. (In Java this would be handled through *generics*.)

The *size* or *length* of such a list is n. Here is a very simple, minimalist specification of a list:

`init()`: Initialize an empty list

`get(i)`: Returns element a_i

`set(i,x)`: Sets the ith element to x

`length()`: Returns the number of elements currently in the list

`insert(i,x)`: Insert element x just prior to element a_i (causing the index of all subsequent items to be increased by one).

`delete(i)`: Delete the ith element (causing the indices of all subsequent elements to be decreased by 1).

I am sure that you can imagine many other useful operations, for example searching the list for an item, splitting or concatenating lists, generating an iterator object for enumerating the elements of the list.

There are a number of possible implementations of lists. The most basic question is whether to use *sequential allocation* (meaning storing the elements sequentially in an array) or *linked allocation* (meaning storing the elements in a linked list). (See Fig. 1.) With linked allocation there are many other options to be considered. Is the list singly linked (each node pointing to its successor in the list), doubly linked (each node pointing to both its successor and predecessor), circularly linked (with the last node pointing back to the first)?

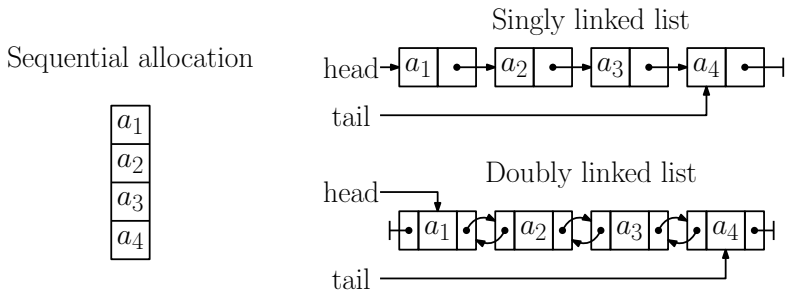

Fig. 1: Common types of list allocation.

Stacks, Queues, and Deques: There are a few very special types of lists. The most well known are of course *stacks* and *queues*. We'll also discuss an interesting generalization, called the *deque* (see Fig. 2):

Stack: Supports insertion (*push*) and removal (*pop*) from only one end of the list, called the stack's *top*. Stacks are among the most widely used of all data structures, and we will see many applications of them throughout the semester.

Queue: Supports insertion (called *enqueue*) and removal (called *dequeue*), each from opposite ends of the list. The end where insertion takes place is called the *tail*, and the end where removals occur is called the *head*.

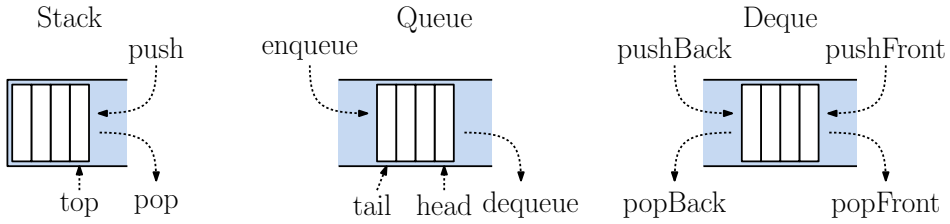

Fig. 2: Stack, Queue, and Deque.

Deque: This data structure is a generalization of stacks and queues, called a *double-ended queue* or "*deque*" for short. It supports insertions and removals from either end of the list.

The name is actually a play on words. It is written like "d-e-que" for a "double-ended queue", but it is pronounced like *deck*, because it behaves like a deck of cards, since you can deal off the top or the bottom.

Both stacks and queues can be implemented efficiently as arrays or as linked lists. Note that when a queue is implemented using sequential allocation (as an array) the head and tail pointers chase each other around the array. When each reaches the end of the array it wraps back around to the beginning of the array (see Fig. 3).

Dynamic Storage Reallocation: When sequential allocation is used for stacks and queues, an important issue is what to do when an attempt is made to insert an element into an array that is full. When this occurs, the usual practice is the allocate a new array of twice the size as the existing array, and then copy the elements of the old array into the new one. For example, if the initial stack or queue has eight elements, then when an attempt is made to

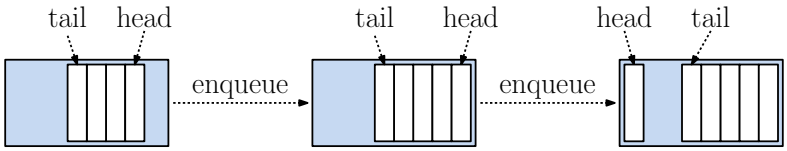

Fig. 3: Head/tail wrap-around in a queue.

insert a ninth element, we allocate an array of size 16, copy the existing eight elements to this new array, and then add the new element (see Fig. 4 below). When we fill this up, we then allocate an array of size 32, and when it is filled an array of size 64, and so on.

To understand whether such an approach is efficient, we will view the problem from the perspective of an amortized analysis.

Amortized Analysis: In an *amortized analysis* of a data structure, rather than the (worst-case) cost of each individual operation, we consider the average cost of maintaining the data structure over a long sequence of operations. The idea is that many operations are very cheap to perform, but occasionally we need to perform a major reorganization, which can be quite costly. More formally, suppose that the running time of any sequence of m data structure operations (starting from an empty structure) is given by a function $T(m)$. Then we *amortized cost* of each operation is defined to be $T(m)/m$. Amortization is a way of establishing that, when taken as a whole, the costly reorganizations do not dominate the cheap ones. (The term comes from economics. It refers to spreading out expensive payments over small installments.)

To make this more formal in the case of a dynamic stack, let's define *cost model*, which described the "actual cost" of each operation:

Cheap: Each time we are asked to perform a push or pop, and the array is not reallocated, it costs one (+1) work unit for the operation.

Expensive: If a push would cause the array to overflow, we reallocated the current array of size n into a new array of size $2n$, and it costs $2n$ work units to perform the reallocation (which involves allocating the new array, initializing it, copying the n elements over).[1] Finally, it costs 1 additional work unit to perform the actual push operation.

For example, suppose we start with an array of size $n = 1$, and we perform a sequence of m pushes and pops. What is the total cost of all these operations? If we never did a reallocation (just alternated one push and one pop), the overall cost would be exactly m, and this is the lowest possible. But how high might the cost be? Suppose we just keep pushing elements. Suppose, for example, that we push 17 elements. When we push the second element, we overflow the array of size 1 and double the array size to $n = 2$. When we push the 3rd element, we overflow the array of size 2 and double the array size to $n = 4$. Following this pattern, we see that we will overflow with push numbers 5, 9, and 17, resulting in the

[1] You might wonder why we charged $2n$ for the reallocation cost and not, say n, since this would represent the cost of copying the elements from the old stack to the new stack. The convenient answer is that it doesn't matter, since the there is only a constant factor difference between them, and we will be happy with an asymptotic analysis of the running time. But this is always the case. If you were to increase the array size by some other function of n, the amortized cost may be different depending on whether the actual cost is based on the array size before reallocation versus the array size after reallocation.

creation of new arrays of sizes 8, 16, and 32. The overall cost is 17 for the individual pushes and $62 = 2 + 4 + 8 + 16 + 32$ for the reallocations, for a total of $17 + 62 = 79$.

Tokens, Charging, and Amortized Cost: How can we analyze the overall cost for an arbitrary sequence of pushes and pops? We will show that in any sequence of m pushes and pops, the overall cost cannot exceed $5m$. (Intuitively, we pay a factor of at most 5 for the luxury of never running out of space in our stack.)

To show this, we will employ a *charging argument*. Here is a high-level overview. We will play the role of an accountant. Whenever the user asks us to perform a push or pop operation, we will assess the user a fixed charge of 5 work tokens. We will store our work tokens in a bank account, from which we will make withdrawals to pay for the actual costs (as defined above). Most of the time, the operation will not involved any reallocation. We will take one of the 5 work token to pay the actual cost and put the remaining 4 tokens in our bank account. We will show that, whenever it is time to perform a reallocation, we will always have enough work tokens in our bank account to pay the actual costs. Since we charge 5 tokens for each operation, it follows that the actual costs for any sequence of m operations never exceeds $5m$.

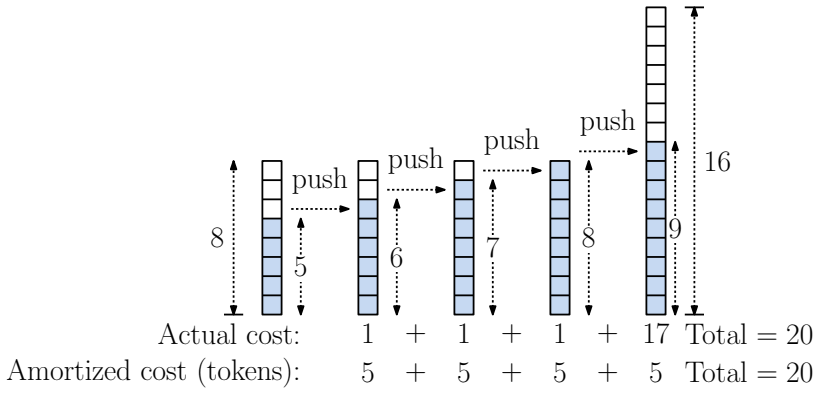

Fig. 4: Amortized analysis of doubling reallocation.

An example is shown in Fig. 4. Let suppose that the current array has size 8 and it contains 5 elements. We are asked to perform 4 consecutive pushes. The first 3 are all cheap and cost $1 + 1 + 1 = 3$. The last one causes the array to overflow to 9 elements, which necessitates that we reallocate to an array of size 16 for a total cost of $16 + 1 = 17$. The overall actual cost is $3 + 17 = 20$. We charged the user 5 tokens for each operation. Thus, the total in our bank account is $4 \cdot 5 = 20$, exactly what we need to pay for the actual costs! Did we just get lucky or is this always true? The following theorem shows that this is always true.

Theorem: Starting with an empty stack, any sequence of m push/pop operations to our dynamically allocated stack has a total actual cost of at most $5m$. (That is, the *amortized cost* of each stack operation is 5.)

Proof: Let's take the sequence of length m and break it into subsequences by splitting it just after each array reallocation. Let's call each subsequence a *run*. Each run, except possibly the last, consists of a sequence of cheap operations, and its last operation causes a reallocation. We will show that the number of tokens collected during each run suffices to pay the actual costs of the run. To simplify the analysis, we will ignore the first and last runs. (We will leave the full proof as an exercise.)

Consider the state of affairs at the start of a run. We have just reallocated an array of some size n, creating an array of size $2n$, and (because we just overflowed an array of size n) we know that the current array must have $n+1$ actual elements. When the run ends, we have just overflowed the $2n$ array, so there must be $2n+1$ elements in the stack. Thus, there must be *at least* $(2n+1)-(n+1)=n$ push and pop operations in this run. This means that we have collected a total of at least $5n$ work tokens in the run. We take away one token to pay for the actual cost of each operation, which leaves us at least $4n$ tokens in our bank account. The next reallocation results in doubling our array size to $4n$ for an actual cost of $4n$, but since we have at least $4n$ tokens in our bank account, we can pay for it.

In summary, by charging 5 work tokens for each of the m operations, we have enough accumulated tokens to pay the total actual costs for all these operations. This implies that the total actual costs cannot be more than $5m$, as desired.

Okay, we have just shown that doubling works. What about other possible strategies?

Linear growth: What if, rather than doubling, we reallocate by adding a fixed increment, say going from n elements to $n+c$ elements for some large integer constant $c \geq 1$? So, the allocated stack sizes would be multiples of c, $\langle c, 2c, 3c, \ldots, kc, \ldots\rangle$.

Exponential growth: What if, rather than doubling, we reallocated by adding a fixed constant factor, say going from n elements to $\lceil cn \rceil$ for some constant $c > 1$? So, the allocated stack sizes would be powers of c, $\langle 1, c, c^2, c^3, \ldots, c^k, \ldots\rangle$.

Doubly-exponential growth: What if, rather than doubling, we squared the size of the array, going from n to n^2? So, the allocated stack sizes would be $\langle c, c^2, c^4, c^8, \ldots, c^{2^k}, \ldots\rangle$.

We will leave the analyses of these cases as an exercise, but (spoiler alert!) the fixed-increment method will *not* yield a constant amortized cost, the constant-factor expansion will yield a constant amortized cost (but the factor 5 with change). The repeated squaring depends on which cost model we use (whether the cost depends on the size before reallocation or after reallocation).

Multilists and Sparse Matrices: Although lists are very basic structures, they can be combined in numerous nontrivial ways. A *multilist* is a structure in which a number of lists are combined to form a single aggregate structure. Java's `ArrayList` is a simple example, in which a sequence of lists are combined into an array structure. A more interesting example of this concept is its use to represent a *sparse matrix*.

Recall from linear algebra that a matrix is a structure consisting of n rows and m columns, whose entries are drawn from some numeric field, say the real numbers. In practice, n and m can be very large, say on the order of tens to hundred of thousands. For example, a physicist who wants to study the dynamics of a galaxy might model the n stars of the galaxy using an $n \times n$ matrix, where entry $A[i, j]$ stores the gravitational force that star i exerts on star j. The number of entries of such a matrix is n^2 (and generally nm for an $n \times m$ matrix). This may be impractical if n is very large.

The physicist knows that most stars are so far apart from each other that (due to the inverse square law of gravity), only a small number of matrices are significant, and all the others could be set to zero. For example, $n = 10,000$ but a star typically exerts a significant gravitational pull on only its 20 nearest stellar neighbors, then only $20/10,000 = 0.02\%$ of the matrix

entries are nonzero. Such a matrix in which only a small fraction of the entries are nonzero is called *sparse*.

We can use a multilist representation to store sparse matrices. The idea is to create $2n$ linked lists, one for each row and one for each column. Each entry of each list stores five things, its row and column index, its numeric value, and links to the next items in the current row and current column (see Fig. 5). We will not discuss the technical details, but all the standard matrix operations (such as matrix multiplication, vector-matrix multiplication, transposition) can be performed efficiently using this representation.

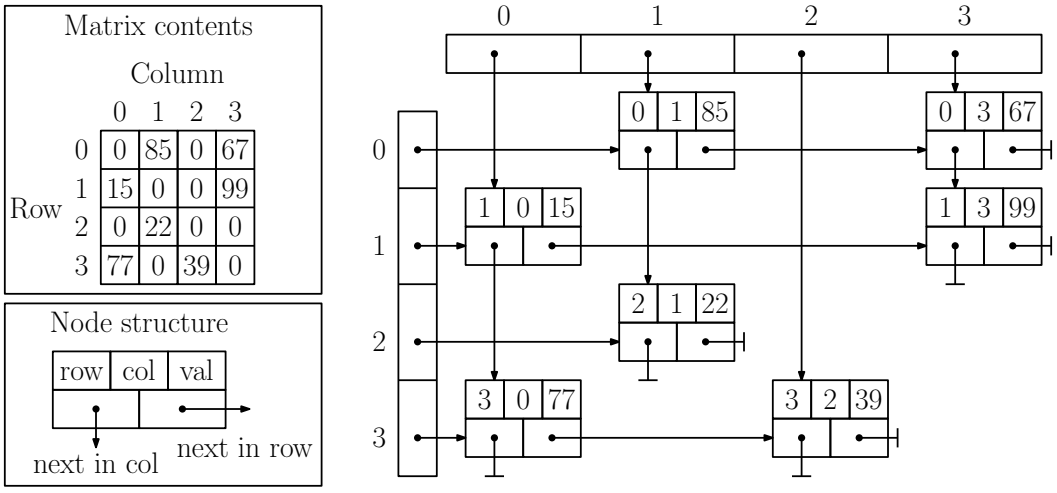

Fig. 5: Sparse matrix representation using a multilist structure.

As a challenge, you might think of how you would write a program to perform matrix multiplication using such a representation. The running time should be "sensitive" to the number of entries in the two matrices.