CMSC 420: Lecture 4
Some Basic Data Structures

Read: Chapt. 3 in Weiss.

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)?

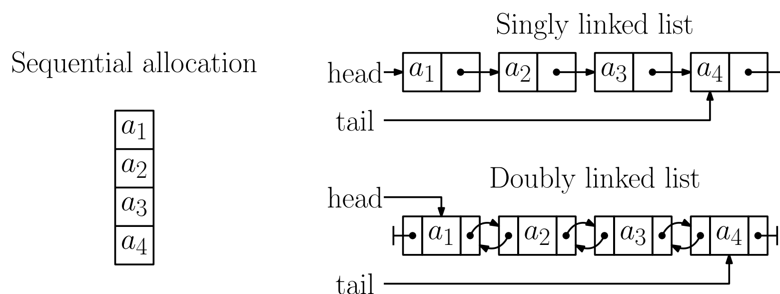

Fig. 1: Common types of list allocation.

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*.

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*.

Deque: This data structure is a combination 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.

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 8 elements, then when an attempt is made to insert a 9th element, we allocate an array of size 16, copy the existing 8 elements to this new array, and then add the new element. 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.

You might wonder, why do we double the array size? Why not, instead, just allocate an array with 100 additional elements? Why not be more aggressive and square the size of the array (jumping from 8 to 64 elements)?

If you have no additional knowledge regarding the access sequence, there is a good reason why increasing the size by a constant factor is the "right" thing to do. (Doubling itself is not essential. You could increase the size by another factor, such as 1.5 or 3.0, but the increase should be by a constant factor.)

This reason is related to the notion of *amortization*, which we introduced in the previous lecture. Remember that amortization means that the cost of accessing a data structure is summed over a long sequence of operations, and rather than reporting the cost of single operation, we instead report the average cost over all the operations.

Theorem: When doubling reallocation is used for stack/queue/deque operations, the amortized cost of each operation is $O(1)$.

Proof: Let us do the proof for stacks, since the generalization to the other structures is straightforward. Let us also assume that the initial allocation is of constant size (e.g., we always start with capacity for 8 elements). The initialization takes $O(1)$ time.

We will use a charging argument to show that the amortized cost is constant per operation. In particular, we will "amortize" the cost of reallocation among the push operations that came just before it.

Each time we do a push operation, we perform the operation and put 4 *work tokens* in a bank account. The operation itself takes only a constant time, and the 4 work tokens will be saved up for later. Now, suppose that the push operation requires that we run out of space. Suppose that the current array size is n. We we allocate a new array of size $2n$, which must be initialized and elements copied to it. Thus, the actual cost of performing this work is $2n$. We want to pay for this work from our bank account. Have we accumulated enough funds to do so? Well, the last time we reallocated we went from an array of size $n/2$ to an array of size n. In order to overflow this array, we must have performed at least $n/2$ additional pushes. Since each push allows us to place 4 tokens in our bank account, we have accumulated at least $4(n/2) = 2n$ tokens. Thus, we have enough to pay for the cost of reallocation.

Would this work if instead we had added 100 additional elements? The answer is no. If this list was really large (thousands), we would not accumulate enough tokens to pay for the reallocation. What if we increased by a much larger amount, say squaring the current array size. The good news is that provide us with enough tokens to pay for the reallocation, but if we were to stop the process right after the last reallocation, we would have a huge bank account (with $O(n^2)$ tokens), which would go to waste. So, doubling (and in general, increasing by a constant factor) is the perfect solution.

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. 2). 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.

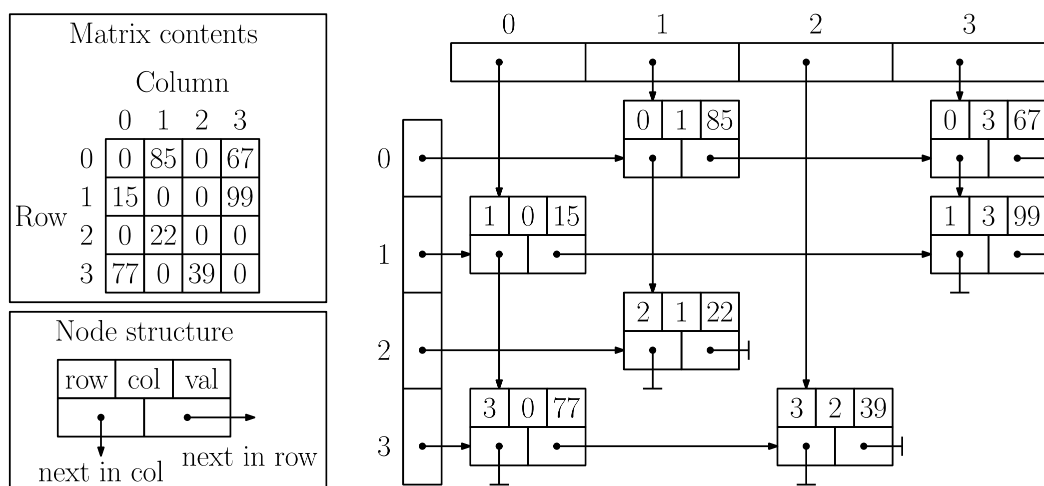

Fig. 2: Sparse matrix representation using a multilist structure.