Lecture 4: 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 C++ and Java is the capability to present the user of a data structure with an abstract definition of its function without revealing the methods with which it operates. To a large extent, this course will be concerned with the various options 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 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 C++ this might be done through the use of *templates*. In Java this can be handled by defining the elements to be of type *Object*. Since the Object class is a superclass of all other classes, we can store any class type in our list.)

The *size* or *length* of such a list is n. There is no agreed upon specification of the list ADT, but typical operations include:

`get(i)`: Returns element a_i.

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

`length()`: Returns the length of 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 search the list for an item, split or concatenate lists, return a sublist, make the list empty. There are often a number of programming language related operations, such as returning an iterator object for the list. Lists of various types are among the most primitive abstract data types, and because these are taught in virtually all basic programming classes we will not cover them in detail here.

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. With linked allocation there are many other options to be considered. Is the list singly linked, doubly linked, circularly linked?

Another question is whether we have an *internal list*, in which the nodes that constitute the linked list contain the actual data items a_i, or we have an *external list*, in which each linked list item contains a pointer to the associated data item.

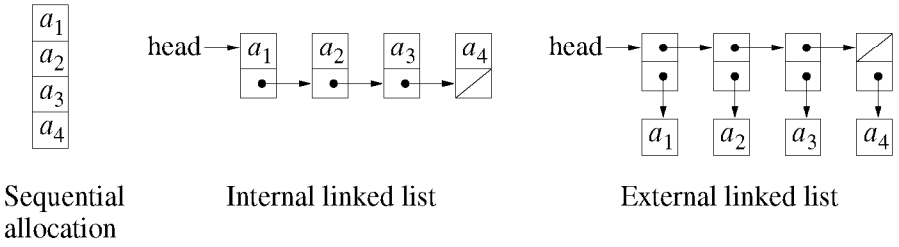

Figure 2: Common types of list allocation.

Multilists and Sparse Matrices: Although lists are very basic structures, they can be combined in various nontrivial ways. One such example is the notion of a *multilist*, which can be thought of as two sets of linked lists that are interleaved with each other. One application of multilists is in representing sparse matrices. Suppose that you want to represent a very large $m \times n$ matrix. Such a matrix can store $O(mn)$ entries. But in many applications the number of nonzero entries is much smaller, say on the order of $O(m+n)$. (For example, if $n = m = 10,000$ this might means that only around 0.01% of the entries are being used.) A common approach for representing such a *sparse matrix* is by creating m linked lists, one for each row and n linked lists, one for each column. Each linked list stores the nonzero entries of the matrix. Each entry contains the row and column indices $[i][j]$ as well as the value stored in this entry.

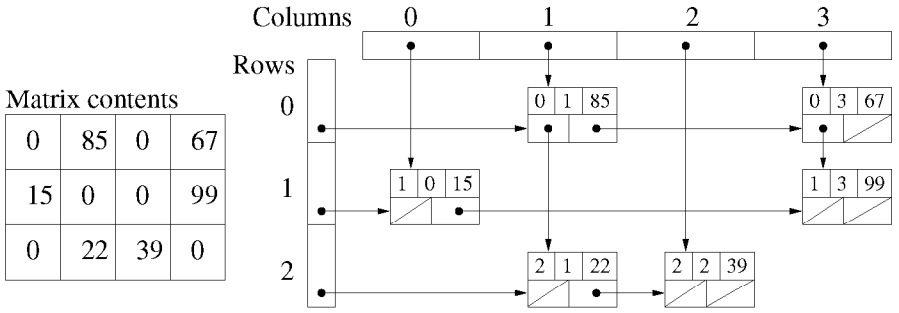

Figure 3: Sparse matrix representation using multilists.

Stacks, Queues, and Deques: There are two very special types of lists: *stacks* and *queues* and their generalization, called the *deque*.

Stack: Supports insertions (called *pushes*) and deletions (*pops*) from only one end (called the *top*). Stacks are used often in processing tree-structured objects, in compilers (in processing nested structures), and is used in systems to implement recursion.

Queue: Supports insertions (called *enqueues*) at one end (called the *tail* or *rear*) and deletions (called *dequeues*) from the other end (called the *head* or *front*). Queues are used in operating systems and networking to store a list of items that are waiting for some resource.

Deque: This is 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, where you can deal off the top or the bottom. A deque supports insertions and deletions from either end. Clearly, given a deque, you immediately have an implementation of a stack or queue by simple inheritance.

Both stacks and queues can be implemented efficiently as arrays or as linked lists. There are a number of interesting issues involving these data structures. However, since you have probably seen these already, we will skip the details here.

Graphs: Intuitively, a *graph* is a collection of vertices or nodes, connected by a collection of edges. Graphs are extremely important because they are a very flexible mathematical model for many application problems. Basically, any time you have a set of objects, and there is some "connection" or "relationship" or "interaction" between pairs of objects, a graph is a good way to model this. Examples of graphs in application include *communication* and *transportation networks*, *VLSI* and other sorts of *logic circuits*, *surface meshes* used for shape description in computer-aided design and geographic information systems, *precedence constraints* in scheduling systems. The list of application is almost too long to even consider enumerating it.

Definition: A *directed graph* (or *digraph*) $G = (V, E)$ consists of a finite set V, called the *vertices* or *nodes*, and E, a set of *ordered pairs*, called the *edges* of G. (Another way of saying this is that E is a binary relation on V.)

Observe that *self-loops* are allowed by this definition. Some definitions of graphs disallow this. Multiple edges are not permitted (although the edges (v, w) and (w, v) are distinct).

Figure 4: Digraph and graph example.

Definition: An *undirected graph* (or *graph*) $G = (V, E)$ consists of a finite set V of vertices, and a set E of *unordered pairs* of distinct vertices, called the edges. (Note that self-loops are not allowed).

Note that directed graphs and undirected graphs are different (but similar) objects mathematically. We say that vertex v is *adjacent* to vertex u if there is an edge (u, v). In a directed graph, given the edge $e = (u, v)$, we say that u is the *origin* of e and v is the *destination* of e. In undirected graphs u and v are the *endpoints* of the edge. The edge e is *incident* (meaning that it touches) both u and v.

In a digraph, the number of edges coming out of a vertex is called the *out-degree* of that vertex, and the number of edges coming in is called the *in-degree*. In an undirected graph we just talk about the *degree* of a vertex as the number of incident edges. By the *degree* of a graph, we usually mean the maximum degree of its vertices.

When discussing the size of a graph, we typically consider both the number of vertices and the number of edges. The number of vertices is typically written as n or V, and the number of edges is written as m or E. Here are some basic combinatorial facts about graphs and digraphs. We will leave the proofs to you. Given a graph with V vertices and E edges then:

In a graph:

- $0 \leq E \leq \binom{n}{2} = n(n-1)/2 \in O(n^2)$.
- $\sum_{v \in V} \mathit{deg}(v) = 2E$.

In a digraph:

- $0 \leq E \leq n^2$.
- $\sum_{v \in V} \mathit{in\text{-}deg}(v) = \sum_{v \in V} \mathit{out\text{-}deg}(v) = E$.

Notice that generally the number of edges in a graph may be as large as quadratic in the number of vertices. However, the large graphs that arise in practice typically have much fewer edges. A graph is said to be *sparse* if $E \in O(V)$, and *dense*, otherwise. When giving the running times of algorithms, we will usually express it as a function of both V and E, so that the performance on sparse and dense graphs will be apparent.

Adjacency Matrix: An $n \times n$ matrix defined for $1 \leq v, w \leq n$.

$$A[v,w] = \begin{cases} 1 & \text{if } (v,w) \in E \\ 0 & \text{otherwise.} \end{cases}$$

If the digraph has weights we can store the weights in the matrix. For example if $(v,w) \in E$ then $A[v,w] = W(v,w)$ (the weight on edge (v,w)). If $(v,w) \notin E$ then generally $W(v,w)$ need not be defined, but often we set it to some "special" value, e.g. $A(v,w) = -1$, or ∞. (By ∞ we mean (in practice) some number which is larger than any allowable weight. In practice, this might be some machine dependent constant like `MAXINT`.)

Adjacency List: An array $\mathit{Adj}[1 \ldots n]$ of pointers where for $1 \leq v \leq n$, $\mathit{Adj}[v]$ points to a linked list containing the vertices which are adjacent to v (i.e. the vertices that can be reached from v by a single edge). If the edges have weights then these weights may also be stored in the linked list elements.

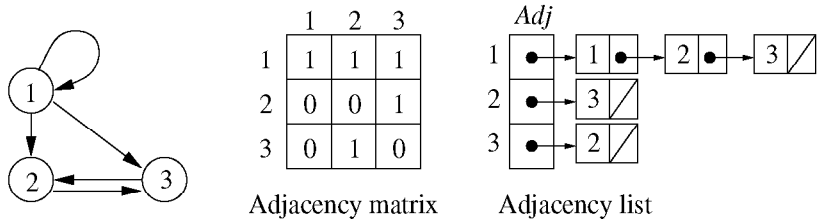

Figure 5: Adjacency matrix and adjacency list for digraphs.

We can represent undirected graphs using exactly the same representation, but we will store each edge twice. In particular, we representing the undirected edge $\{v,w\}$ by the two oppositely directed edges (v,w) and (w,v). Notice that even though we represent undirected graphs in the same way that we represent digraphs, it is important to remember that these two classes of objects are mathematically distinct from one another.

This can cause some complications. For example, suppose you write an algorithm that operates by marking edges of a graph. You need to be careful when you mark edge (v,w) in the representation that you also mark (w,v), since they are both the same edge in reality. When dealing with adjacency lists, it may not be convenient to walk down the entire linked list, so it is common to include *cross links* between corresponding edges.

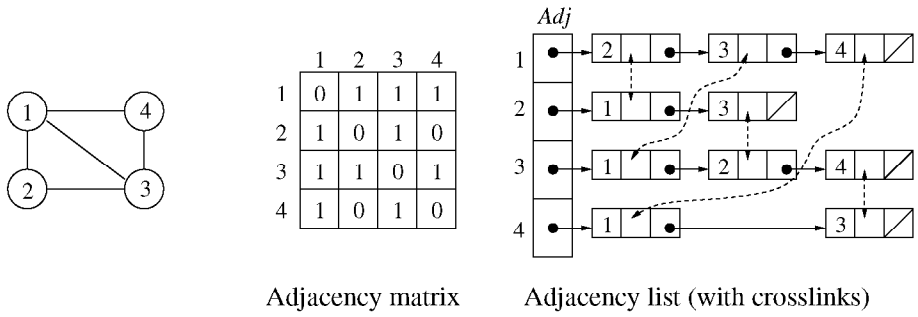

Figure 6: Adjacency matrix and adjacency list for graphs.

An adjacency matrix requires $O(V^2)$ storage and an adjacency list requires $O(V + E)$ storage. The V arises because there is one entry for each vertex in Adj. Since each list has *out-deg*(v) entries, when this is summed over all vertices, the total number of adjacency list records is $O(E)$. For sparse graphs the adjacency list representation is more space efficient.