© 2013 by Douglas Wilhelm Harder. All rights reserved.



## 2.2 Data Structures and Algorithsms

This course is concerned with the efficient allocation and manipulation of data. In general, the information you find here is applied to computers, but often it applies to real life, as well. As an example, we will learn sorting algorithms in this course, and while people understand these algorithms, when it comes to something like sorting a stack of examination papers, they will resort to exceptionally inefficient algorithms.

Before we can discuss the algorithms, let us look at the two basic forms of storing information on a computer: contiguous and node based. Arrays and linked lists represent prototypical examples of each, respectively. We will then see other data structures such as trees, hybrids, and higher-dimensional arrays. We will conclude by discussing the general idea of run-time analysis and introduce the balance of the course.

### 2.2.1 Data Allocation

Data can be stored on a computer in generally one of three means: through contiguous, linked or indexed allocation of memory. We will look at each of these three.

#### 2.2.1.1 Contiguous Allocation

The definition of contiguous, *adj*., given by the Oxford English Dictionary is

> touching, in actual contact, next in space; meeting at a common boundary, bordering, adjoining;

while Meriam-Webster defines it as

> Touching or connected throughout in an unbroken sequence.

The prototypical example of contiguous allocation is the array: an array of size $n$ occupies some multiple of $n$ bytes in memory and that memory is allocated as a single block. Accessing an entry in an array is a pointer operation: the location of the $k^{\text{th}}$ item is the base address of the array plus $k$ times the size of each entry. Such addressing is usually performed in a single operation in assembly language.

One issue with contiguous allocation is may not be easily extensible: suppose $n$ bytes have been allocated in memory, but additional memory is required—the array must be expanded. Because memory allocation is performed by an operating system independent of the program making the request for more memory, it is usually the case that the memory immediately following the array has been allocated to some other data structure or even some other process. Consequently, it is usually necessary to allocate a new array of larger size and copy all entries over. Once this is done, the memory for the old array is deallocated.

© 2013 by Douglas Wilhelm Harder. All rights reserved.



Figure 1 shows an array of size four which is first filled. Next, if additional memory is required, a new array (in this case, double the size) is allocated, the information is copied over, and the old array is deallocated.

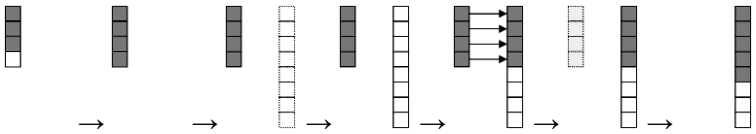


**Figure 1. Additional items being added to an array initially of size four.**

### 2.2.1.2 Linked Allocation

Linked allocation has each item of data stored together with a reference to successor or related data. Usually this reference will be in the form of an address, though other mechanisms exist.

The prototypical example of linked allocation is a linked list. The classic means of displaying a linked list is to display the data in stored in the node and an arrow pointing to the next node (or `NULL`, also represented by Ø and even by, engineers, as ground, ⏚).


In reality, this would be represented by a class, such as

```
template <typename Type>
class Node {
    private:
        Type element;
        Node *next_node;

    public:
        Node( const Type& = Type(), Node* = nullptr );
        Type retrieve() const;
        Node *next() const;
};
```

For example, `Node<int> new_node( 42, nullptr );` would define the variable `new_node` to be an object storing an integer[1] (in this case, 42) and a pointer storing the address[2] of the next node in the linked list (in this case the null pointer). The minimalist collection of member functions would return the value being stored (`new_node.retrieve()` returns `42`) and the address being stored in the node (in this case, `new_node.next()` returns `nullptr`). The next node in a linked list is also called the *successor*.

[1] Usually four bytes, but system dependent.
[2] Four bytes on a 32-bit system and eight bytes on a 64-bit system.

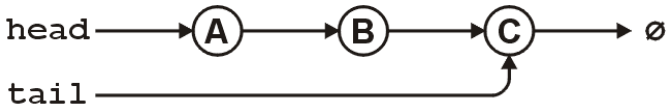
© 2013 by Douglas Wilhelm Harder. All rights reserved.



Aside: The symbol for a "pointer to nothing", or a "pointer that does not refer to a valid object" varies between languages:

| | |
|---|---|
| C | `NULL` |
| Java/C# | `null` |
| C++ (old) | `0` |
| C++ (new) | `nullptr` |
| Symbolically | Ø |

The `Node` class stores an individual node, but it is still necessary to have a data structure that allows the user to access and manipulate the linked list. At a minimum, such a class must store the address of the first node in the linked list (the *head*), but it is also useful to store the address of the last node (the *tail*), as well.




A minimal C++ class might be defined as

```
template <typename Type>
class List {
    private:
        Node<Type> *head;
        Node<Type> *tail;
        int count;
    public:
        // constructor(s)...
        // accessor(s)...
        // mutator(s)...
};
```

where we track the number of nodes in the linked list, as well.

© 2013 by Douglas Wilhelm Harder. All rights reserved.



### 2.2.1.3 Indexed Allocation

Indexed allocation has an array of pointers to other memory locations, some pointers perhaps being null, as shown in Figure 2.

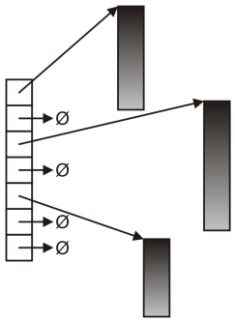


**Figure 2. Indexed allocation.**

Applications include the C++ STP, matrices, and computer engineers will see such allocation repeatedly in their course on operating systems (for example, inodes).

For example, matrices tend to use a hybrid of contiguous and indexed allocation. The matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

can be stored either by storing the rows or the columns, as shown in

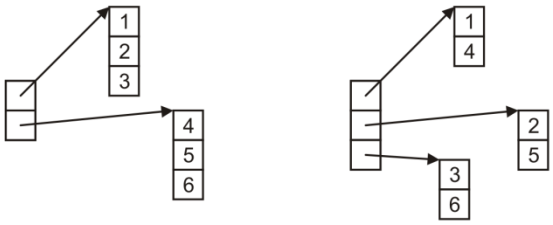



**Figure 3. Storing a matrix by row or by column.**

More ideally, the entries are stored in a contiguous block of memory, and the indices point into this structure, as shown in Figure 4.


**Figure 4. Row-major order and column-major order for storing matrices.**

© 2013 by Douglas Wilhelm Harder. All rights reserved.



C, C++ and python use row-major order while Matlab and Fortran use column-major order for storing matrices.

Example: The matrix

```
int M[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

would store the six entries in contiguous memory locations, as may be seen by

```
int *v = M[0];

for ( int i = 0; i < 6; ++i ) {
    cout << v[i] << ' ' << endl;
}
```

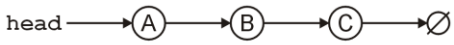
### 2.2.2 Other Allocation Formats

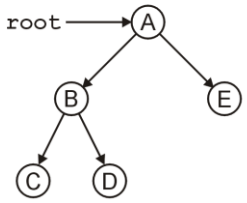
Most other means of allocating memory are a modification of the three we have seen. We will briefly look at

1. trees,
2. graphs,
3. linked arrays, and
4. inodes.

#### 2.2.2.1 Trees

A linked list has only one next pointer, in which case, it can be used to store linearly ordered data. If each node could store two or even more next pointers, they could each reference a different successor. This can be used, for example, to store a hierarchy. A tree that has one node specified as the *root* is called a *rooted tree*, though almost every tree we see in this class will be rooted.




Generally, a tree uses linked allocation, but we will see one case where a contiguous allocation is possible.

© 2013 by Douglas Wilhelm Harder. All rights reserved.



### 2.2.2.2 Graphs

A graph allows arbitrary relations between any two vertices in the graph, as shown in

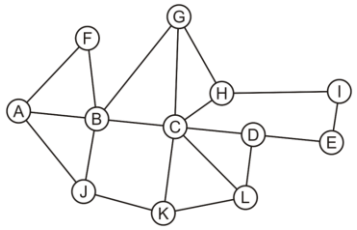
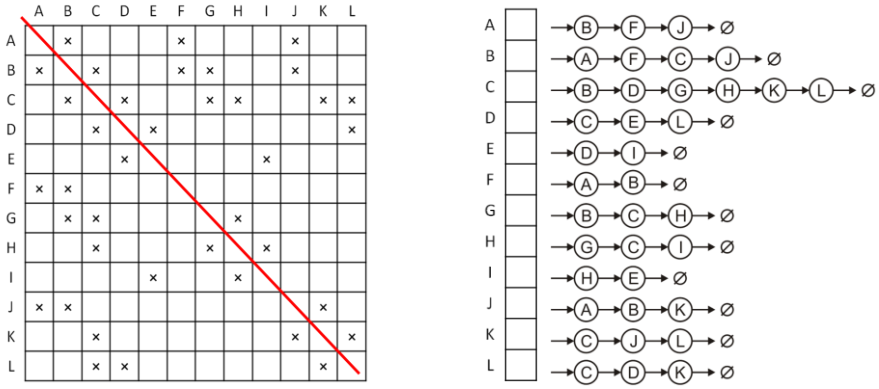


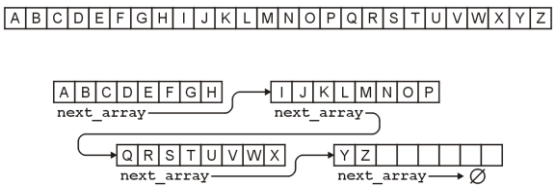

**Figure 5. A graph representing an adjacency relation.**

Such a mathematical structure could be stored either as a matrix or as an array of linked lists.




### 2.2.2.3 Linked Arrays

It is also possible to link arrays. Something similar to this (including indexing) is used with the C++ STL `deque` container. For example, the alphabet could be stored as a number of linked arrays:

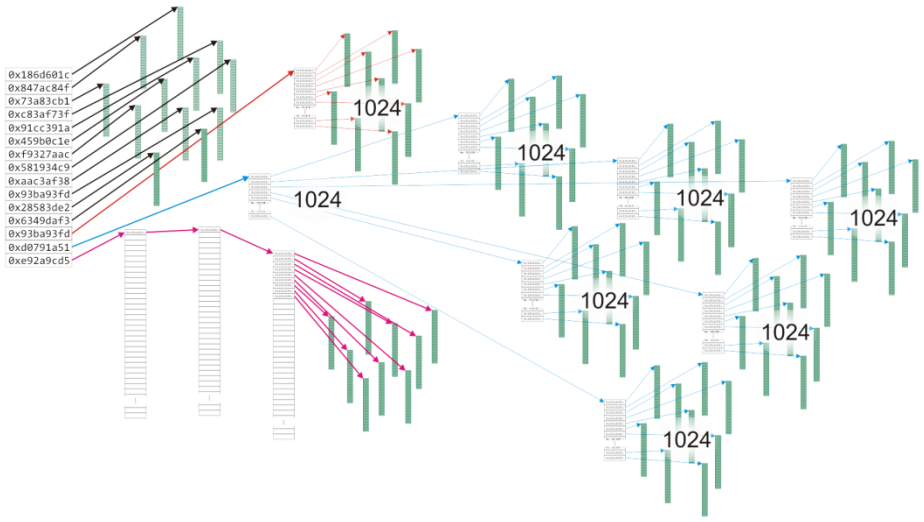


© 2013 by Douglas Wilhelm Harder. All rights reserved.



### 2.2.2.4 inodes

The Unix inode structure is used to store the addresses of the blocks of a file. In order to avoid requiring contiguous memory for each file, each file is broken into blocks (usually 4 KiB). The inode uses indexed memory allocation where the first twelve indices store the addresses of the first twelve blocks. Next, the thirteen address points to a block of 1024 addresses for the next 1024 blocks (if the file is bigger than 48 KiB in size). (Note: this would be on a 32-bit computer where each address is 4 bytes.) The fourteenth address points to a block of 1024 addresses that, in turn, point to blocks of 1024 addresses for files up to 4.05 GiB. The last entry allows for three levels of indirection for files up to approximately 4 TiB.




### 2.2.3 Algorithm Run Times

Having discussed memory allocation for data and some simple data structures, the next question is how efficiently can we access and manipulate these data structures. For any container of data, there may be many queries or operations we may wish to perform on that data: for example, sorting a list, finding the largest element, finding the shortest path between two points on a map, *etc*. To begin, let us consider three very simple operations on containers storing a sequence of entries such as in an array or a linked list. The operations include

1. finding an entry (determining membership and its location),
2. inserting a new item into container, and
3. erasing (or deleting) an item currently in the container.

As the items in the container are linearly stored, the run time may vary depending on whether the item is

1. the first item (at the “front”),
2. an item at an arbitrary location within the list, or
3. the last (or $n^{\text{th}}$) item (at the “back”).

Note: we are using the terms *front* and *back* as these are the member functions defined in the C++ STL.

© 2013 by Douglas Wilhelm Harder. All rights reserved.



### 2.2.3.1 Operations on Sorted Arrays

Suppose we have a sorted array: accessing the front or the back is trivial, we simply access the 1st and last items in the array. Because the list is sorted, we can also find an item using a binary search, which is also reasonably fast. Inserting or erasing an item at the back of the array is also very fast (and simple) unless the array is full (in which case, we must increase the capacity (or size) of the array). On the other hand, inserting or erasing the front item requires us to either copy all entries one cell forward or backward, respectively, and this must be done for each item in the list. For a large array, this is a relatively slow operation. Similarly, inserting or erasing an item at an arbitrary location may be quick (if we are placing it near the back of the array) or it may be slow, but on average, we will likely be copying half of the entries either forward or backward by one cell—still a relatively slow operation.

If we tabulate this, we get the following table:

| | **Front / First Location** | **At an Arbitrary Location** | **Back / $n^{th}$ Location** |
|---|---|---|---|
| Find | **Good** | **Okay** | **Good** |
| Insert | **Bad** | **Bad** | **Good**[1] **Bad** |
| Erase | **Bad** | **Bad** | **Good** |

[1] only if the array is not full.

### 2.2.3.2 Operations on Unsorted Arrays

If we begin with an unsorted array, the only operation that changes is the time it takes to find an entry at an arbitrary location. Now we cannot use the order to find the entry and consequently we must search through all the entries. If we're lucky, we might find it quickly; however, if not, we may end up searching through most of the array. On average, we will search through half of the entries.

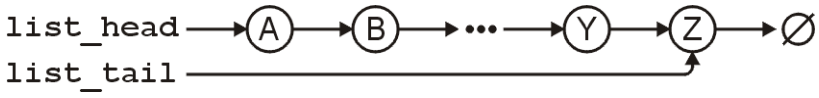
If we tabulate this, we get the following table:

| | **Front / First Location** | **At an Arbitrary Location** | **Back / $n^{th}$ Location** |
|---|---|---|---|
| Find | **Good** | **Bad** | **Good** |
| Insert | **Bad** | **Bad** | **Good**[1] **Bad** |
| Erase | **Bad** | **Bad** | **Good** |

[1] only if the array is not full.

### 2.2.3.3 Operations on Singly Linked Lists

Suppose we have a singly linked list with both head and tail pointers:




At this point, accessing, inserting at and erasing from the front is relatively easy. Similarly, with the tail pointer, accessing or inserting at the back of the linked list is also relatively easy. Unfortunately, erasing the last item is more difficult, as we must modify the next pointer of the second-last item. With a singly

© 2013 by Douglas Wilhelm Harder. All rights reserved.



linked list, the only way to find the second-last item is to step through the entire linked list—a relatively slow operation. Similarly, to find, insert or erase an element at an arbitrary location will require us to step through, on average, half the items—still a slow operation. Consequently, if we tabulate these observations, we get

| | **Front / First Location** | **At an Arbitrary Location** | **Back / $n^{th}$ Location** |
|---|---|---|---|
| Find | **Good** | **Bad** | **Good** |
| Insert | **Good** | **Bad** | **Good** |
| Erase | **Good** | **Bad** | **Bad** |

Note that if we have a pointer to the $k^{th}$ node and we want to insert or remove that node, it is possible to perform both an insertion at that location and an erase at that location quite quickly, only it requires some trickery. This is discussed here because this is a common question in interviews:

1. If you have a pointer to the $k^{th}$ node and you want to insert a new object at that location, insert a new node after this node, copy the entry of the $k^{th}$ node into the following node, and replace the value stored in the $k^{th}$ node with the new object.
2. If you have a pointer to the $k^{th}$ node and you want to erase that node, copy the value of the next node into this node and then proceed to delete the next node.

Now our run times are:

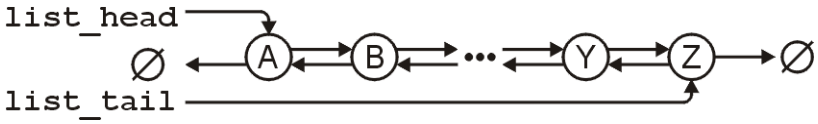
| | **Front / First Location** | **At an Arbitrary Location** | **Back / $n^{th}$ Location** |
|---|---|---|---|
| Find | **Good** | **Bad** | **Good** |
| Insert | **Good** | **Good** | **Good** |
| Erase | **Good** | **Good** | **Bad** |

© 2013 by Douglas Wilhelm Harder. All rights reserved.



### 2.2.3.4 Operations on Doubly Linked Lists

Suppose we have a doubly linked list with both head and tail pointers:




Now, this requires additional memory: one address for each node.[3] The only change this makes is that it is now easy to erase the item at the back of the list.

| | **Front / First Location** | **At an Arbitrary Location** | **Back / $n^{th}$ Location** |
|---|---|---|---|
| Find | **Good** | **Bad** | **Good** |
| Insert | **Good** | **Good** | **Good** |
| Erase | **Good** | **Good** | **Good** |

[3] We can actually do this without additional memory: xor the addresses of the previous and next nodes and then, while traversing the list forward, xor the address of this combined pointer with the address of the previous node to get the address of the next node. In this case, we sacrifice one additional operation to save memory.