ECE430.217 Data Structures

Data Structures and Algorithms

Weiss Book Chapter 2

Byoungyoung Lee

https://compsec.snu.ac.kr

byoungyoung@snu.ac.kr

Lecture slides were prepared based on Douglas W. Harder's notes (**dwharder@alumni.uwaterloo.ca).**

Outline

This topic will describe:

- The concrete data structures that can be used to store information
- The basic forms of **memory allocation**
  - Contiguous
  - Linked
  - Indexed
- The **prototypical examples** of these: arrays and linked lists
- **Other data structures**:
  - Trees
  - Hybrids
  - Higher-dimensional arrays
- Finally, we will discuss the run-time of queries and operations on arrays and linked lists

Memory Allocation

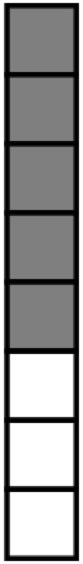
Memory allocation can be classified as either
- Contiguous
- Linked
- Indexed

Prototypical examples:
- Contiguous allocation: arrays
- Linked allocation: linked lists

Contiguous Allocation

An array stores n objects in a contiguous space of memory

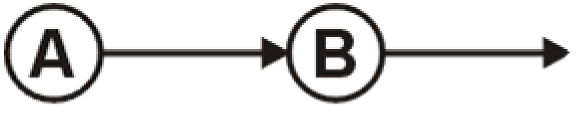
Unfortunately, if more memory is required, a request for new memory usually requires **copying all information into the new memory**



Linked Allocation

Linked storage such as a linked list associates two pieces of data with each item being stored:

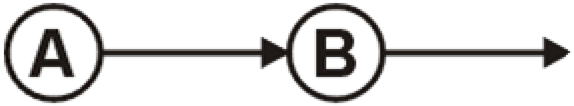
- The object itself, and
- A reference to the next item
  - In C++, the reference is the address of the next node



Linked Allocation

This is a class describing such a node

```
template <typename Type>
class Node {
    private:
        Type node_value;
        Node *next_node;
    public:
         // ...
};
```



Linked Allocation

The operations on this node must include:

- Constructing a new node
- Accessing (retrieving) the value
- Accessing the next node

```
Node( const Type& = Type(), Node* = nullptr );
Type value() const;
Node *next() const;
```

Pointing to nothing has been represented as:

C	`NULL`
Python	`None`
Java/C#	`null`
C++ (old)	`0`
C++ (new)	**`nullptr`**
Symbolically	**Ø**

Linked Allocation

For a linked list, however, we also require an object which links to the first object

The actual linked list class must store two pointers

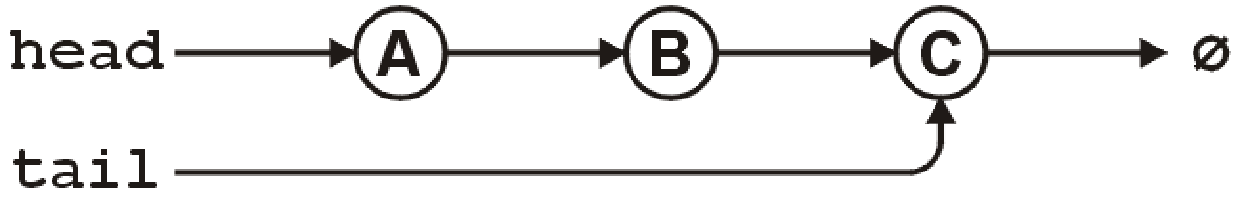
- A head and tail:

```
Node *head;
Node *tail;
```

Optionally, we can also keep a count

```
int count;
```

The `next_node` of the last node is assigned `nullptr`



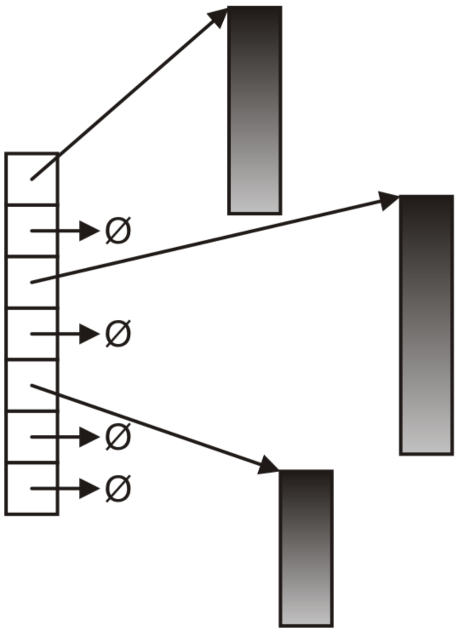
Linked Allocation

The class structure would be:

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

Indexed Allocation

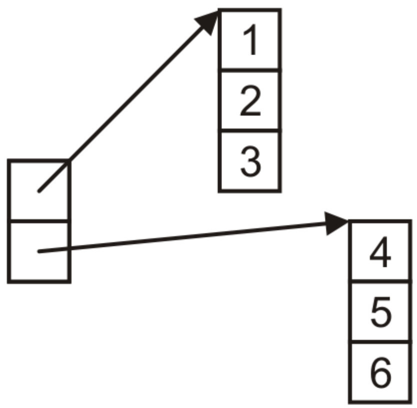
With indexed allocation, an array of pointers (possibly NULL) link to allocated memory locations

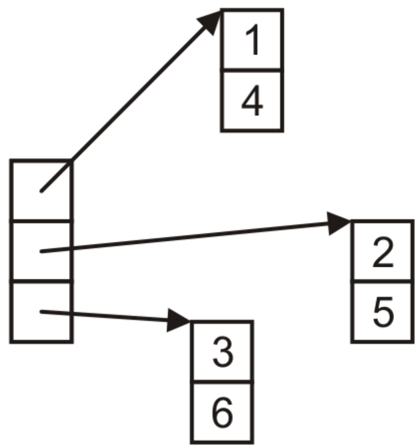


Indexed Allocation

Matrices can be implemented using indexed allocation:

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





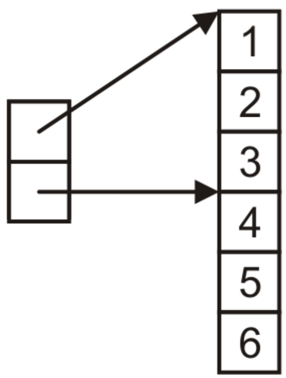
Indexed Allocation

Matrices can be implemented using indexed allocation

– Most implementations of matrices (or higher-dimensional arrays) use indices pointing into a single contiguous block of memory

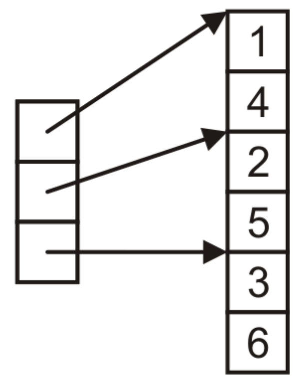
$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

Row-major order



C

Column-major order



Matlab, Fortran

Other Allocation Formats

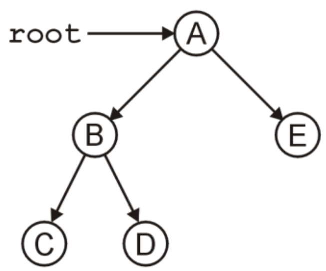
We will look at some variations or hybrids of these memory allocations including:

- Trees
- Graphs
- Deques (linked arrays)

Trees

The linked list can be used to store linearly ordered data
- What if we have multiple *next* pointers?

A rooted tree is similar
to a linked list but with **multiple next pointers**



Trees

A tree is a variation of a linked list:

- Each node points to an arbitrary number of subsequent nodes
- Useful for storing hierarchical data
- Useful for storing sorted data
- Usually we will restrict ourselves to trees where each node points to at most two other nodes

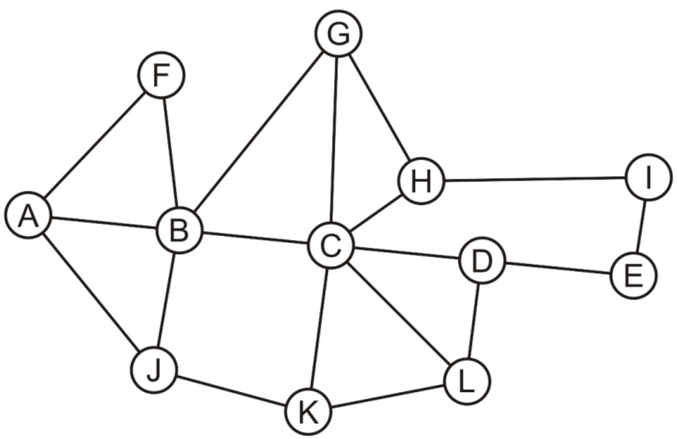
Graphs

Suppose we allow arbitrary relations between any two objects in a container

– Given n objects, there are $n^2 - n$ possible relations

- If we allow symmetry, this reduces to $\dfrac{n^2 - n}{2}$

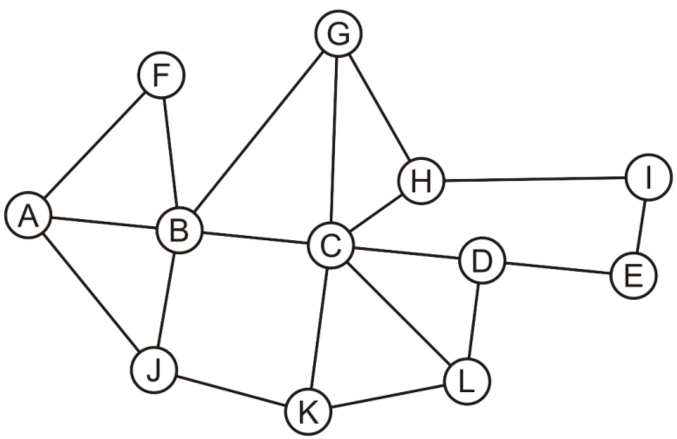
– For example, consider the network



Graphs in Two-dim. Arrays

Suppose we allow arbitrary relations between any two objects in a container

- We could represent this using **a two-dimensional array**
- In this case, the matrix is *symmetric*

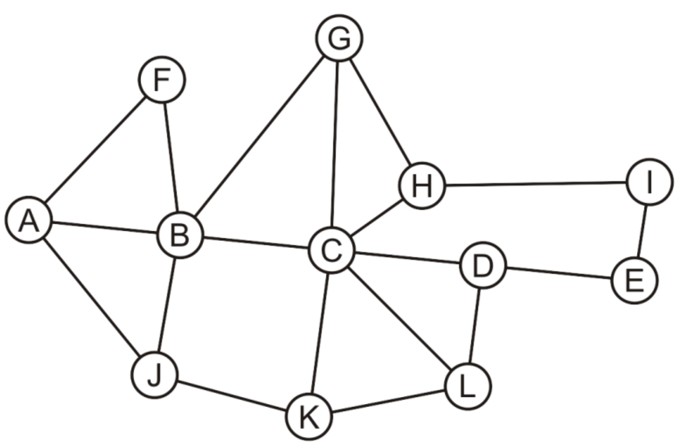


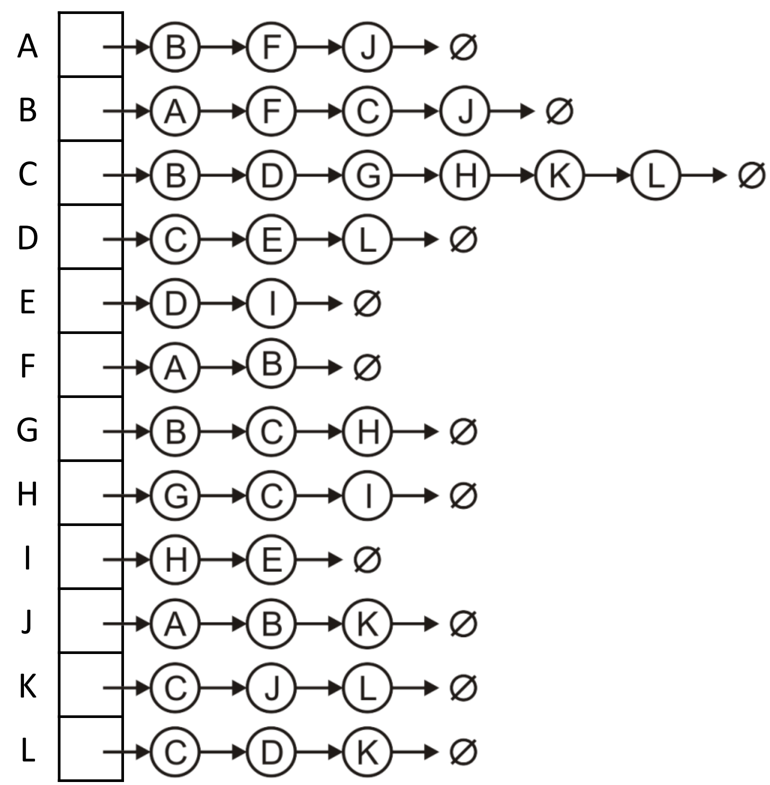
	A	B	C	D	E	F	G	H	I	J	K	L
A		×				×				×		
B	×		×			×	×			×		
C		×		×			×	×			×	×
D			×		×							×
E				×					×			
F	×	×										
G		×	×					×				
H			×				×		×			
I					×			×				
J	×	×									×	
K			×							×		×
L			×	×							×	

Graphs in Array of Linked Lists

Suppose we allow arbitrary relations between any two objects in a container

- Alternatively, we could use a hybrid: **an array of linked lists**

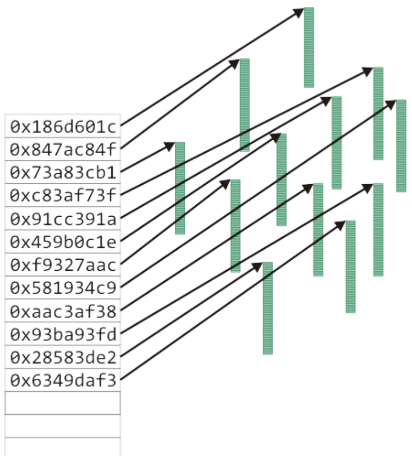




Hybrid data structures

The Unix inode was used to store information about large files

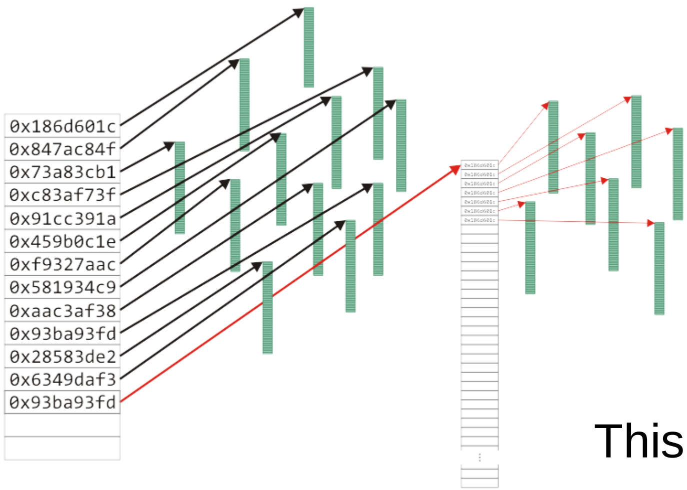
- The first twelve entries can reference the first twelve blocks (48 KiB)



Hybrid data structures

The Unix inode was used to store information about large files

- The next entry is a pointer to an array that stores the next 1024 blocks

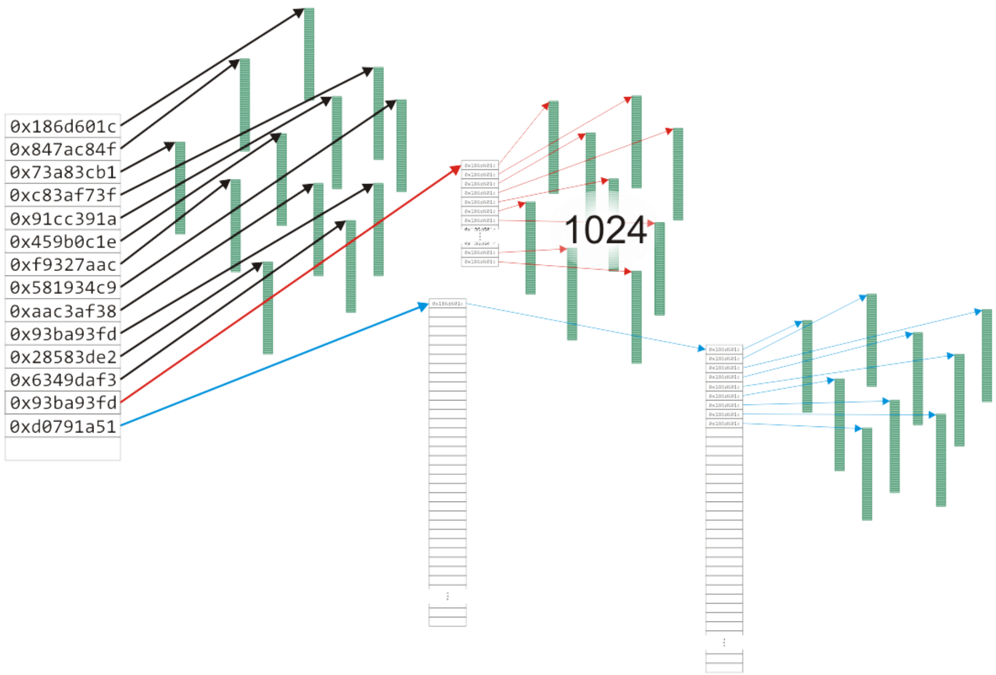


This stores files up to 4 MiB on a 32-bit computer

Hybrid data structures

The Unix inode was used to store information about large files

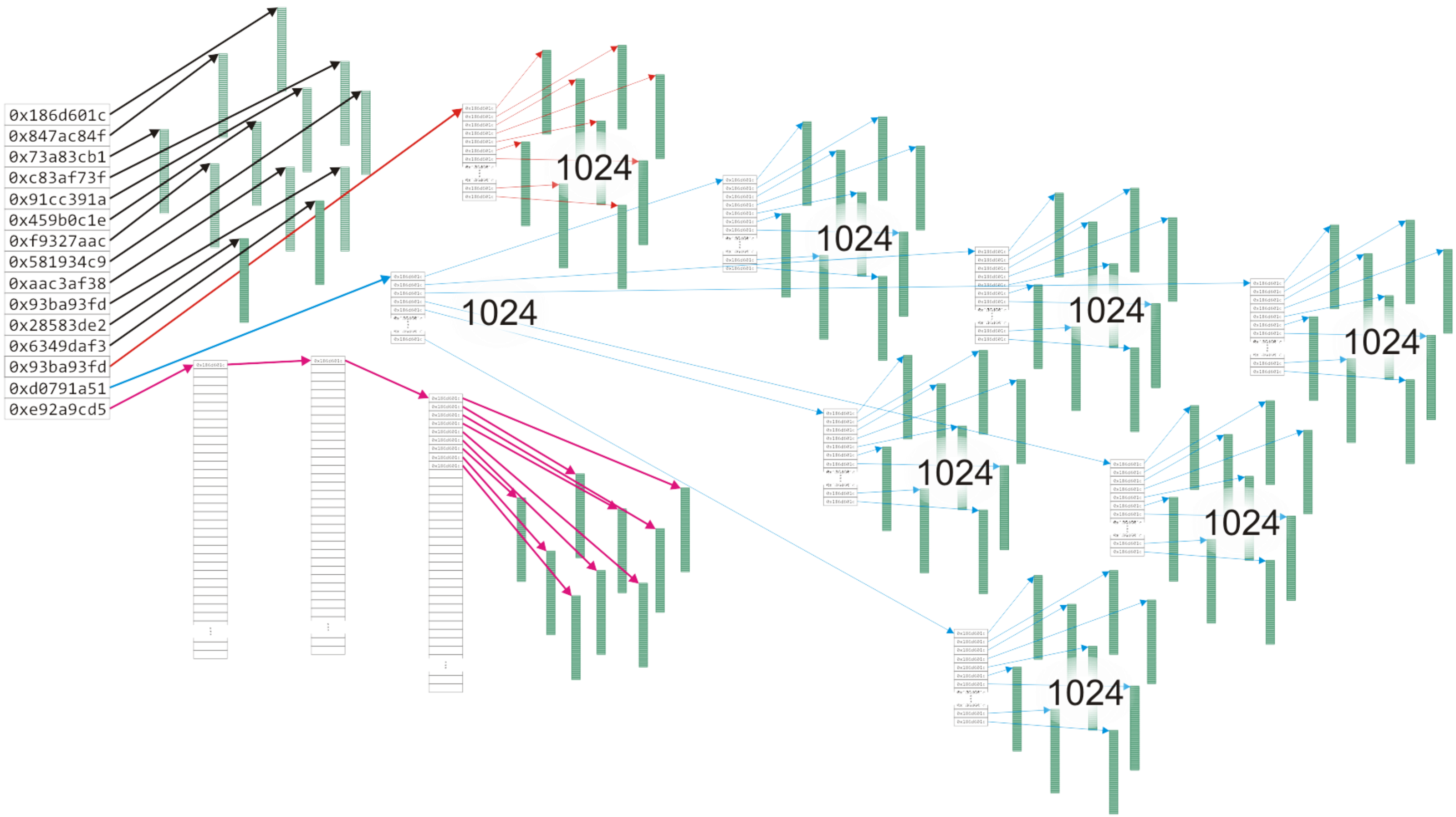
- The next entry has two levels of indirection for files up to 4 GiB



Hybrid data structures

The Unix inode was used to store information about large files

- The last entry has three levels of indirection for files up to 4 TiB



Algorithm run times

Once we have chosen a data structure to store both the objects and the relationships, we must implement the queries or operations as algorithms

- The Abstract Data Type will be implemented as a class
- The data structure will be defined by the member variables
- The member functions will implement the algorithms

The question is, how do we determine the efficiency of the algorithms?

Operations

We will use the following matrix to describe operations at the locations within the structure

	Front/1^{st}	Arbitrary Location	Back/n^{th}
Find	?	?	?
Insert	?	?	?
Erase	?	?	?

Operations on Arrays

Given a sorted array, we have the following run times:

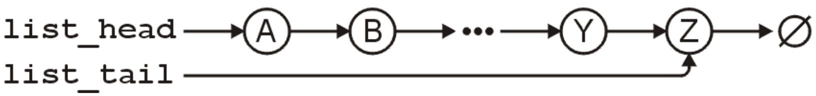
	Front/1st	Arbitrary Location	Back/nth
Find	Good	Good	Good
Insert	Bad	Bad	Good* Bad
Erase	Bad	Bad	Good

* only if the array is not full

Operations on Singly-linked Lists

For a singly linked list with a head and tail pointer, we have:

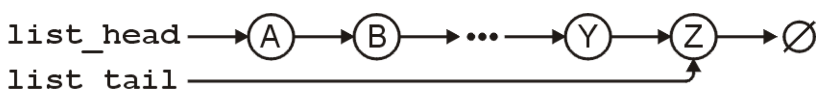
	Front/1st	Arbitrary Location	Back/nth
Find	Good	Bad	Good
Insert	Good	Bad	Good
Erase	Good	Bad	Bad



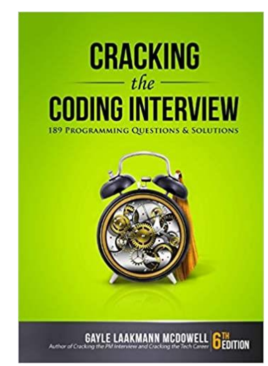
Operations on Singly-linked Lists

If we have a pointer to the k^{th} entry, we can insert or erase at that location quite easily

	Front/1^{st}	Arbitrary Location	Back/n^{th}
Find	Good	Good	Good
Insert	Good	**Good**	Good
Erase	Good	**Good**	Bad



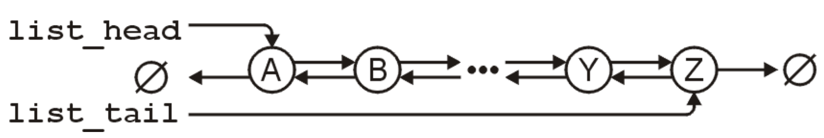
- Note, this requires a little bit of trickery: we must modify the value stored in the k^{th} node
- This is a common coding interview question!



Operations on Doubly-linked Lists

For a doubly linked list, one operation becomes more efficient:

	Front/1st	Arbitrary Location	Back/nth
Find	Good	Good	Good
Insert	Good	Good	Good
Erase	Good	Good	**Good**



Next Lecture

The next topic, **asymptotic analysis**, will provide the mathematics that will allow us to measure the efficiency of algorithms

It will also allow us to measure the memory requirements of both the data structure and any additional memory required by the algorithms

Summary

In this topic, we have introduced the concept of data structures

- We discussed contiguous, linked, and indexed allocation
- We looked at arrays and linked lists
- We considered
  - Trees
  - Two-dimensional arrays
  - Hybrid data structures
- We considered the run time of the algorithms required to perform various queries and operations on specific data structures:
  - Arrays and linked lists