CS 2420: Introduction to Algorithms and Data Structures

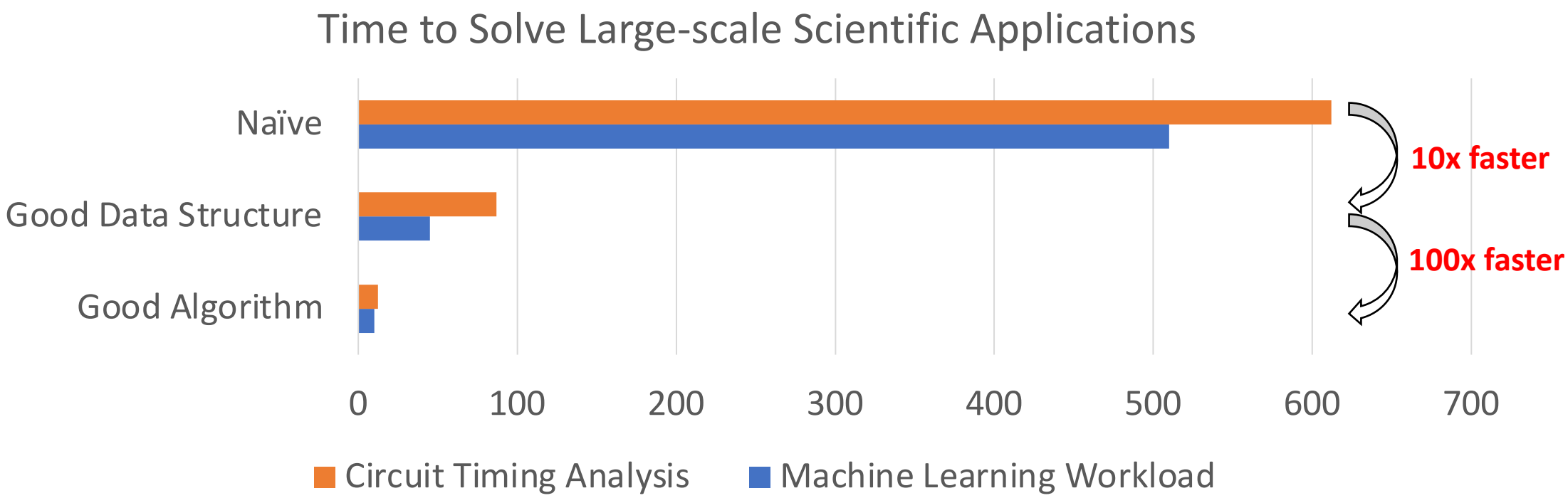
Dr. Tsung-Wei (TW) Huang

Department of Electrical and Computer Engineering

University of Utah, Salt Lake City, UT

Why Algorithms and Data Structures?

- It's critical to advance your application performance



Course Goal

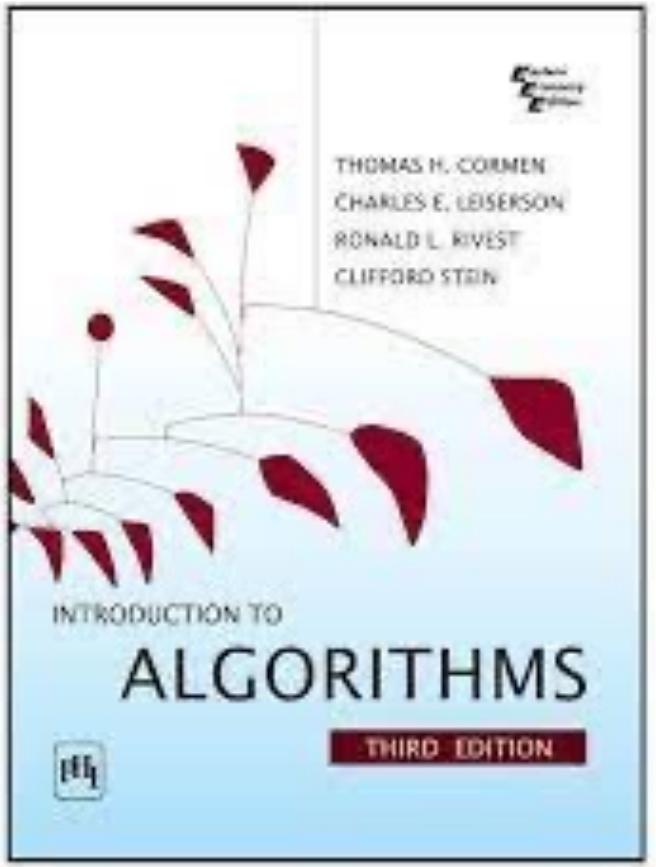
- In this course, we will look at:
  - *Algorithms* for solving problems efficiently
  - *Data structures* for efficiently storing, accessing, and modifying data
- We will see that all data structures have trade-offs
  - There is no *ultimate* data structure...
  - The choice depends on our requirements

Class Logistics

- Staff
  - Instructor: T-W Huang (tsung-wei.huang@utah.edu)
  - TA: I will help you directly
- Class schedule: 9 AM –10:20 AM Mon/Wed
- Class zoom: https://utah.zoom.us/j/2468214418
- Class GitHub: https://github.com/tsung-wei-huang/cs2420
- Lab
  - We will do the lab after each Wed class, since we have <10 students
  - No need to zoom in on Friday
  - You still need to enroll in the lab section (class number: 19145)

Scoring and Textbook

- Lab assignment (after every Wed's class): 60%
- Take-home exams: 40%
  - Two midterms
  - One final
- Textbook: Introduction to Algorithms
  - 3rd Edition (The MIT Press)
  - This is the "bible" of algorithm book



Example: Power of Data Structure

- Consider searching an element in an array
  - What is the index of 6?
  - What is the index of 17?
  - What is the index of 96?

7	11	6	55	98	45	16	96	46

Example: Power of Data Structure (cont'd)

- Consider searching an element in a *sorted* array
  - What is the index of 6?
  - What is the index of 17?
  - What is the index of 96?

7	11	6	55	98	45	16	96	46

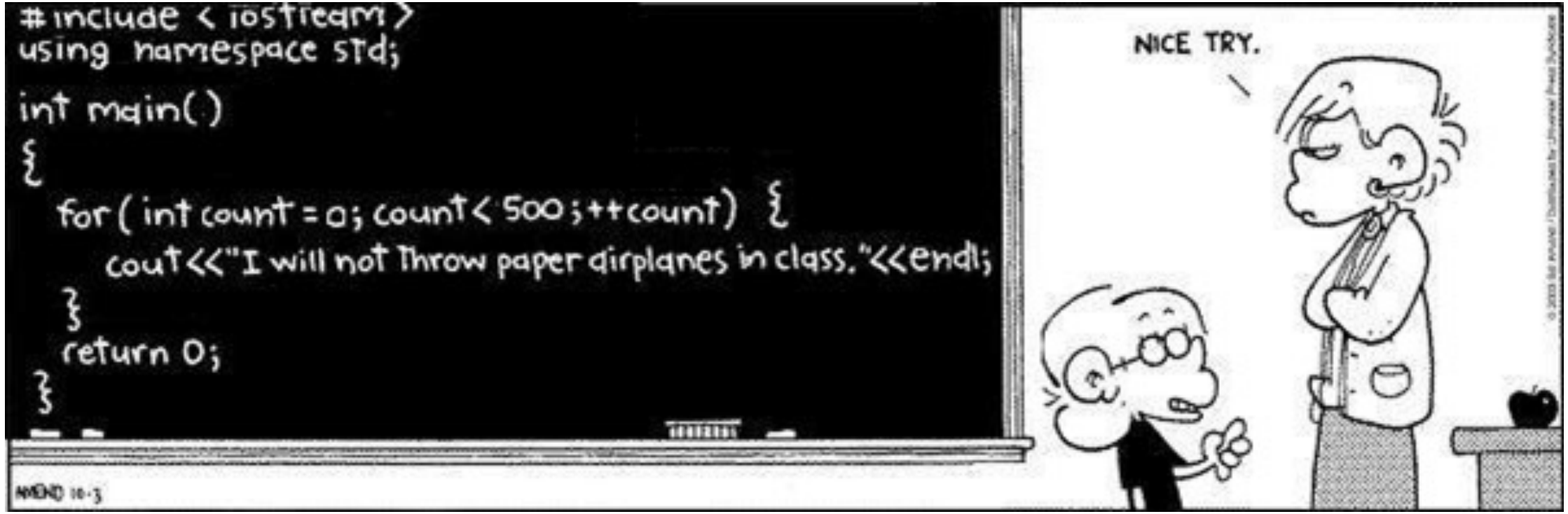
6	7	11	16	45	46	55	96	98

BINARY SEARCH ALGORITHM

Explained

Programming Language: C++

- Online C++ editor: https://www.onlinegdb.com/



Data Structure = Memory Manipulation

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

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

Contiguous Memory Allocation

- An array stores n objects in a single contiguous space of memory
- Unfortunately, if more memory is required, a request for new memory usually requires copying all information into the new memory
  - In general, you cannot request for the operating system to allocate to you the next n memory locations

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++ that reference is the address of the next node

- Memory is not necessary to be contiguous

Linked Allocation (cont'd)

- This is a class describing such a node

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

Linked Allocation (cont'd)

- The operations on each item must include:
  - Constructing a new node
  - Accessing (retrieving) the value
  - Accessing the next pointer

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

Pointing to nothing has been represented as:

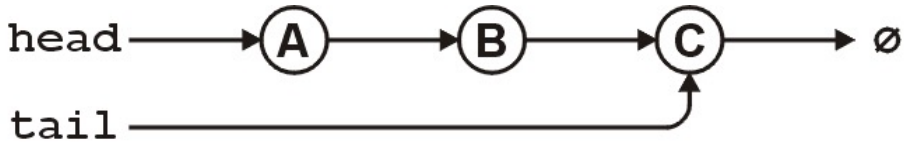
C	`NULL`
Java/C#	`null`
C++ (old)	`0`
C++ (new)	`nullptr`
Symbolically	∅

Linked Allocation (cont'd)

- 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;
```



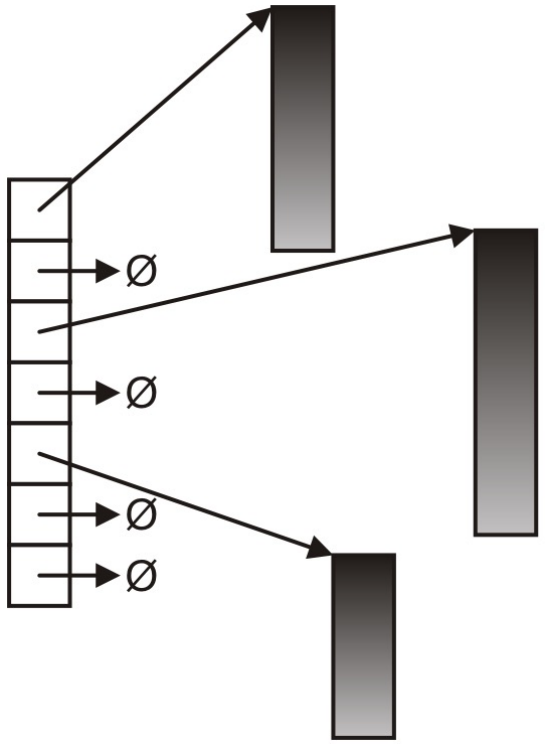
- Optionally, we can also keep a count

```
int count;
```

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

Indexed Allocation

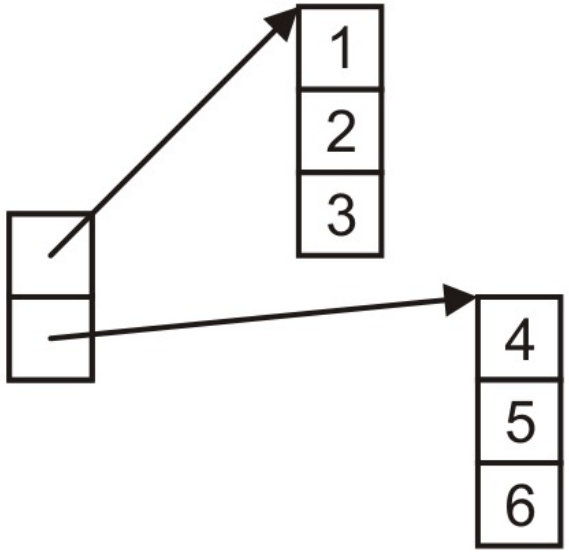
- With indexed allocation, an array of pointers (possibly NULL) link to a sequence of allocated memory locations
- Used in the C++ standard template library
- You will see indexed allocations in many applications



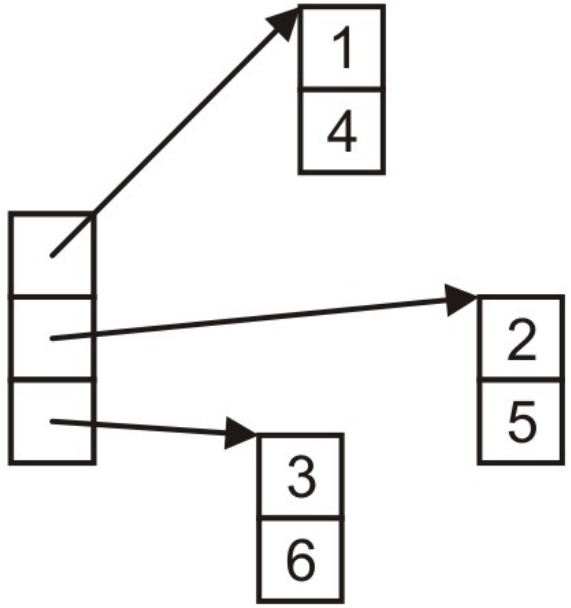
Indexed Allocation (cont'd)

- Matrix can be implemented using indexed allocation

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



Row Major



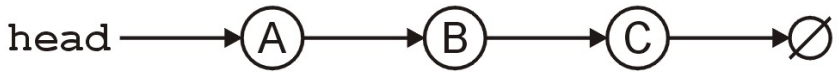
Column Major

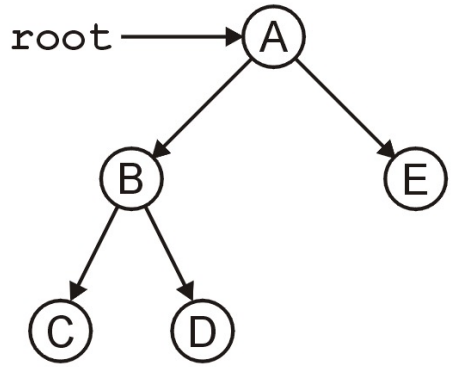
Other Allocation Formats

- We will look at some variations or hybrids of these memory allocations including:
  - Trees
  - Graphs
  - Deques (linked arrays)
  - inodes

Tree

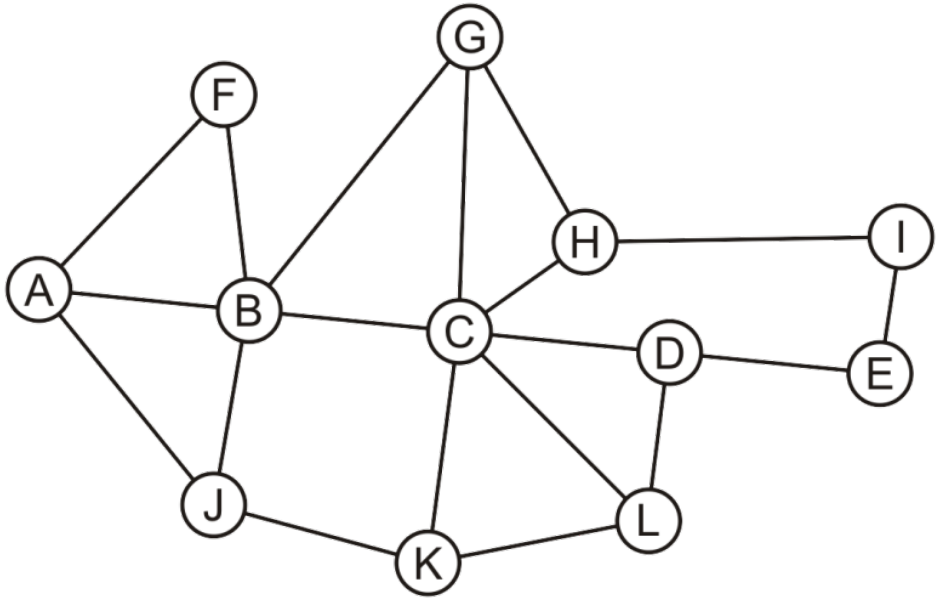
- A tree is a variation on a linked list:
  - Each node points to an arbitrary number of subsequent nodes
  - Useful for storing hierarchical data
  - We will see that it is also useful for storing sorted data
  - Usually we will restrict ourselves to trees where each node points to at most two other nodes
    - i.e., binary tree





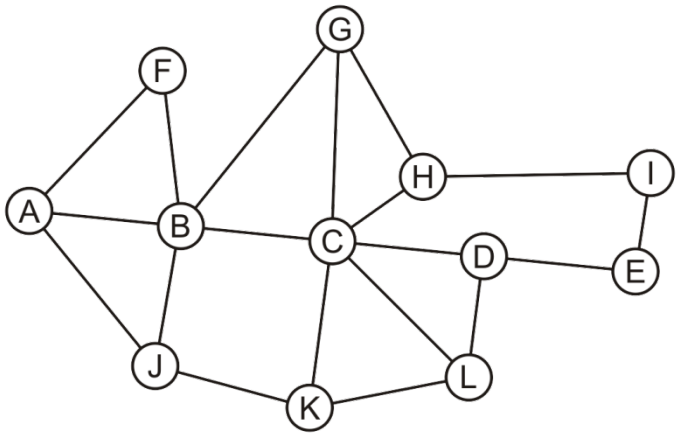
Graph

- Graph is a data structure that allows 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
  - For example, consider the network
    - 12 vertices (nodes)
    - 19 edges (node-to-node connections)



Graph in Adjacency Matrix

- 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*
- Pros and Cons?

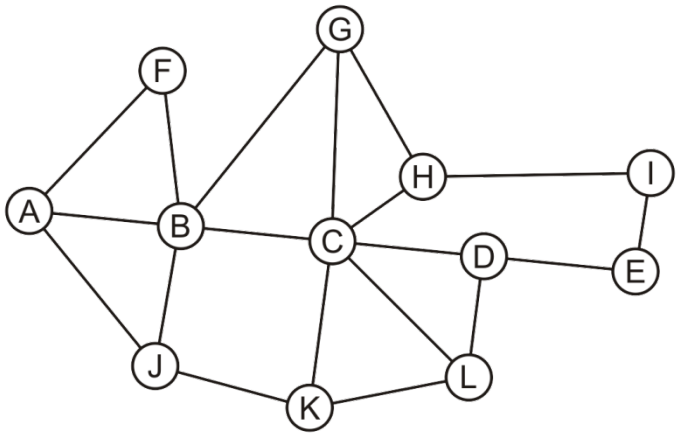


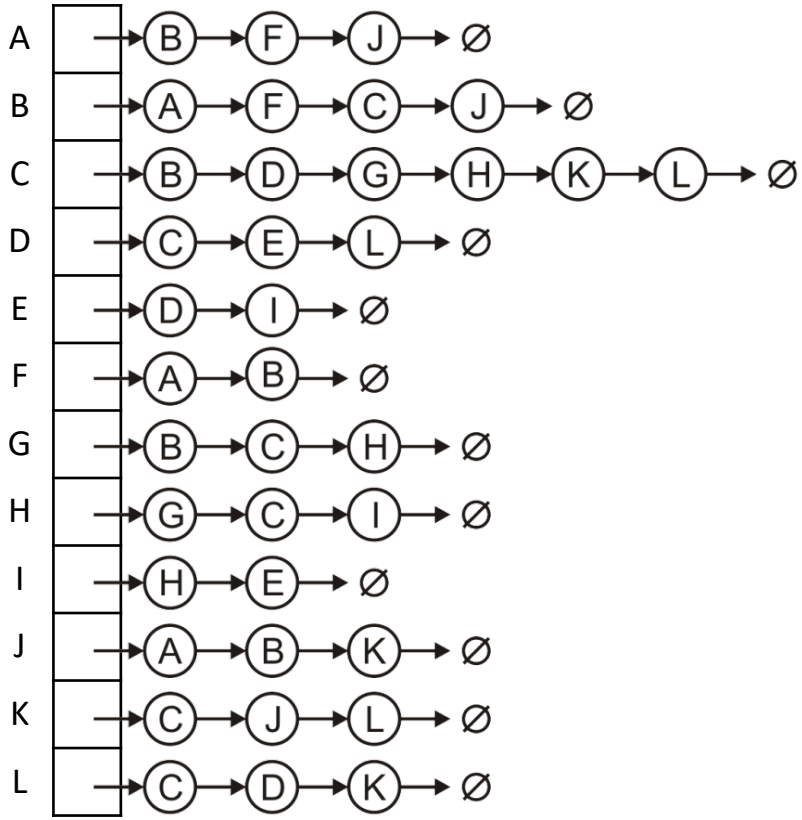
	A	B	C	D	E	F	G	H	I	J	K	L
A		×				×				×		
B	×		×			×	×			×		
C		×		×			×	×			×	×
D			×		×							×
E				×					×			
F	×	×										
G		×	×					×				
H			×				×		×			
I					×			×				
J	×	×									×	
K			×							×		×
L			×	×							×	

Graph in Adjacency List

- Suppose we allow arbitrary relations between any two objects in a container
  - Alternatively, we could use a hybrid: an array of linked lists

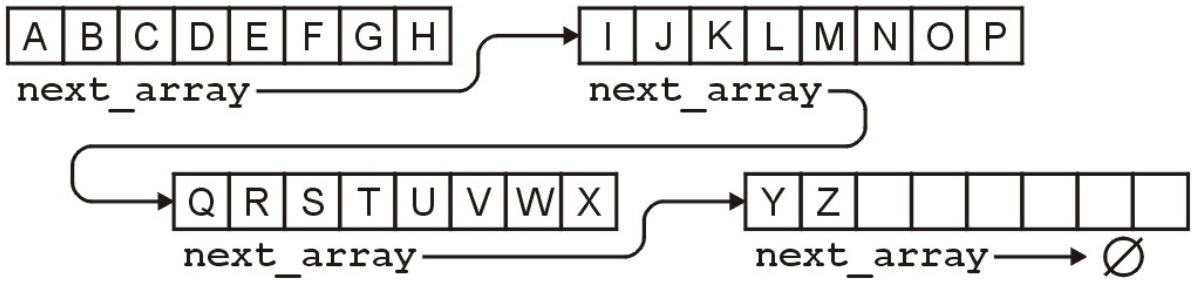
- Pros and Cons?





Linked Arrays

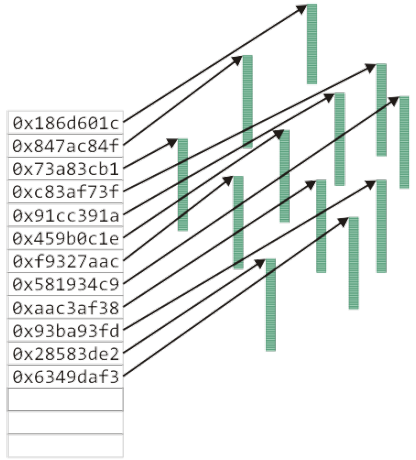
- Other hybrids are linked lists of arrays
  - Something like this is used for the C++ STL `deque` container
- For example, the alphabet could be stored either as:
  - An array of 26 entries, or
  - A linked list of arrays of 8 entries



Hyper Data Structure

The Unix inode was used to store information about large files

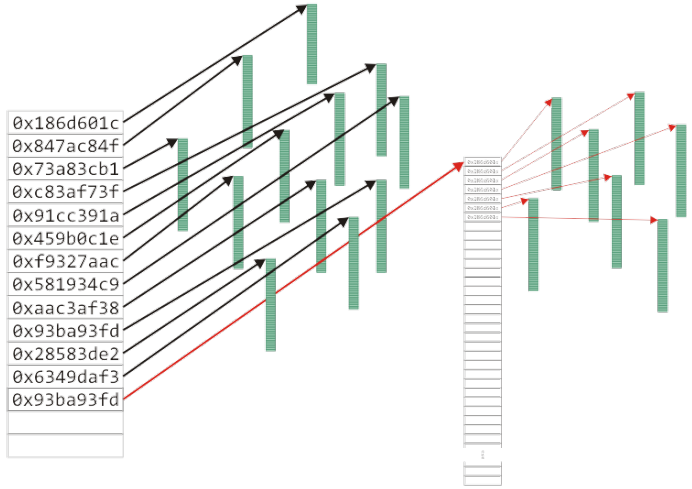
- The first twelve entries can reference the first twelve blocks (48 KiB)



Hyper Data Structure (cont'd)

The Unix inode was used to store information about large files

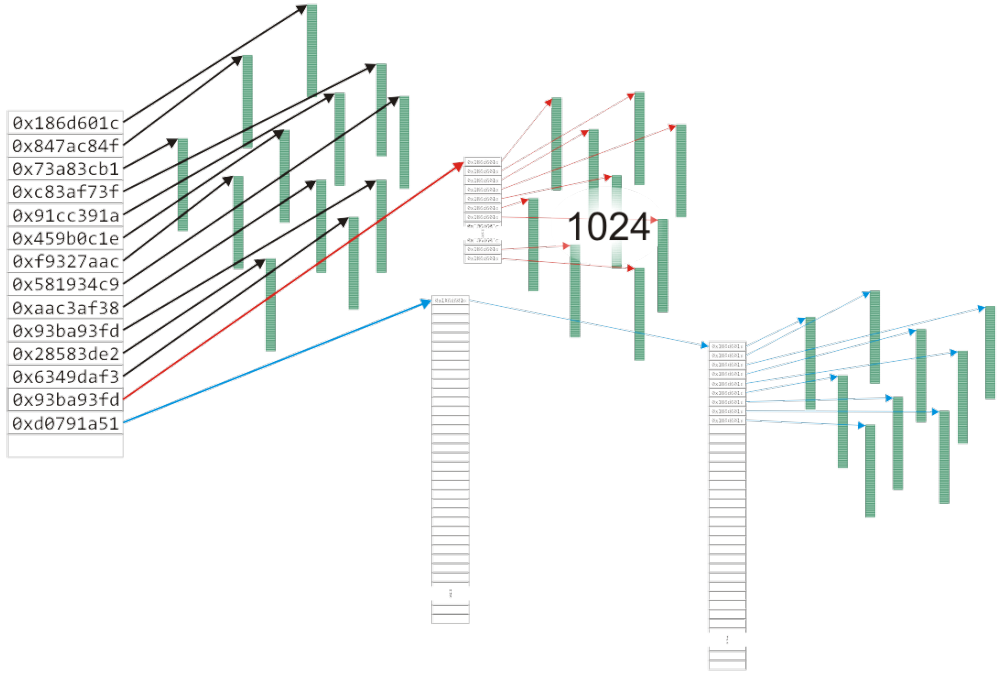
- The first twelve entries can reference the first twelve blocks (48 KiB)



Hyper Data Structure (cont'd)

The Unix inode was used to store information about large files

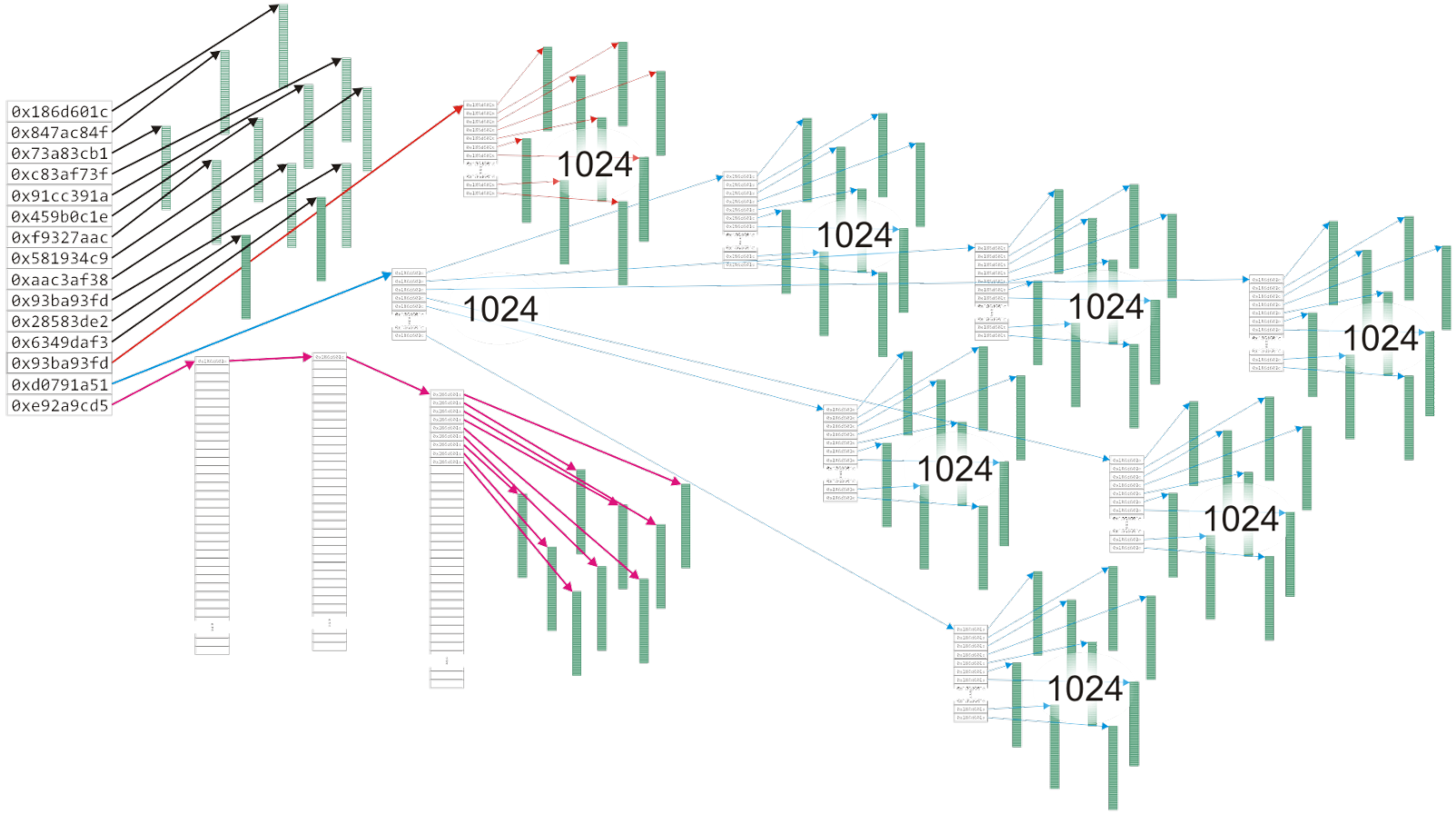
- The first twelve entries can reference the first twelve blocks (48 KiB)



Hyper Data Structure (cont'd)

The Unix inode was used to store information about large files

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



Algorithms 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?

Operation Efficiency Table

- We will us 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 Sorted Arrays

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

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

* only if the array is not full

Operations on Singly-Linked Lists

If the array is not sorted, only one operations changes:

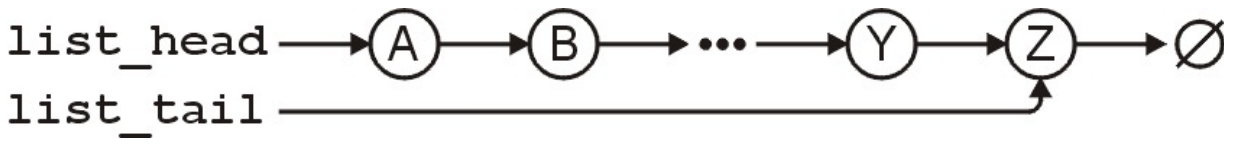
	Front/1^{st}	Arbitrary Location	Back/n^{th}
Find	Good	Bad	Good
Insert	Bad	Bad	Good* Bad
Erase	Bad	Bad	Good

* only if the array is not full

Operations on Singly-Linked Lists

However, for a singly linked list where we a head and tail pointer, we have:

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

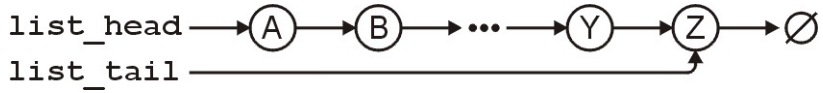


Operations on Lists

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	Bad	Good
Insert	Good	Good	Good
Erase	Good	Good	Bad

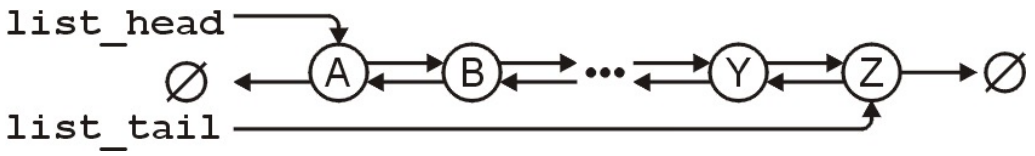
- Note, this requires a little bit of trickery: we must modify the value stored in the k^{th} node
- This is a common co-op interview question!



Operations on Lists

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

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



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


Use the right tool for the right job
Thank You
Dr. Tsung-Wei Huang
tsung-wei.huang@utah.edu