CS 225

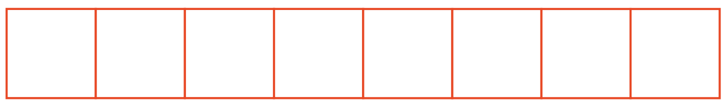
Data Structures

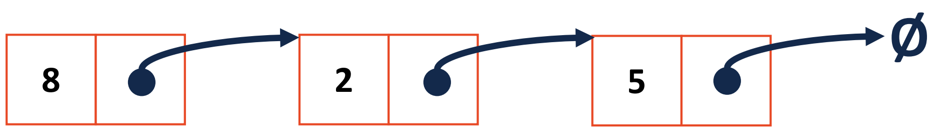
Sept. 26 – Trees

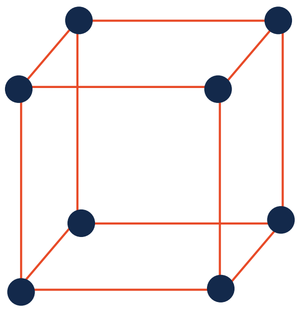
Wade Fagen-Ulmschneider

Iterators

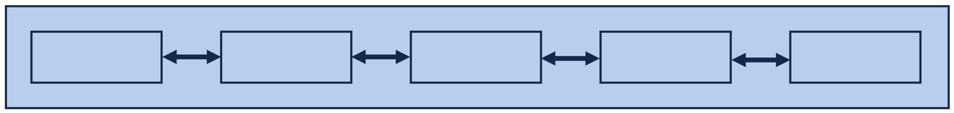
Suppose we want to look through every element in our data structure:

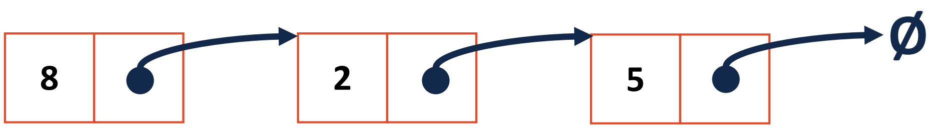


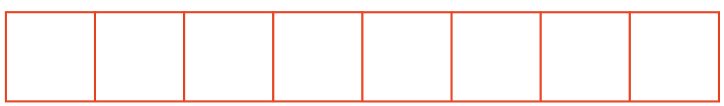


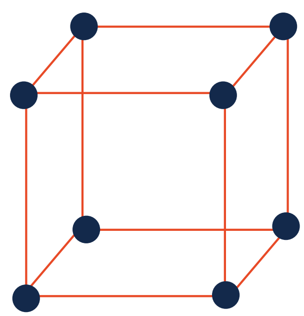


Iterators encapsulated access to our data:









Cur. Location	Cur. Data	Next
`ListNode *`		
`index`		
`(x, y, z)`		

Iterators

Every class that implements an iterator has two pieces:

1. [Implementing Class]:

Iterators

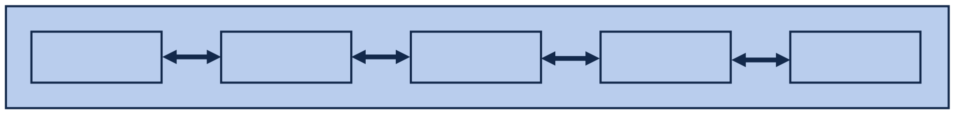
Every class that implements an iterator has two pieces:

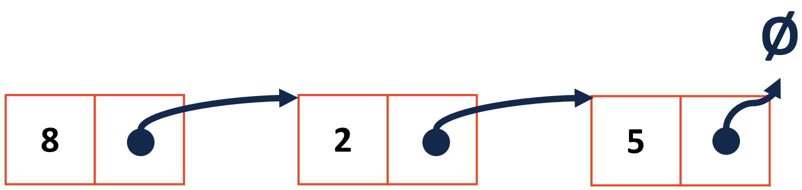
2. [Implementing Class' Iterator]:

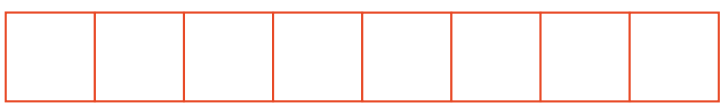
- Must have the **base class:** `std::iterator`

- **std::iterator** requires us to minimally implement:

Iterators encapsulated access to our data:



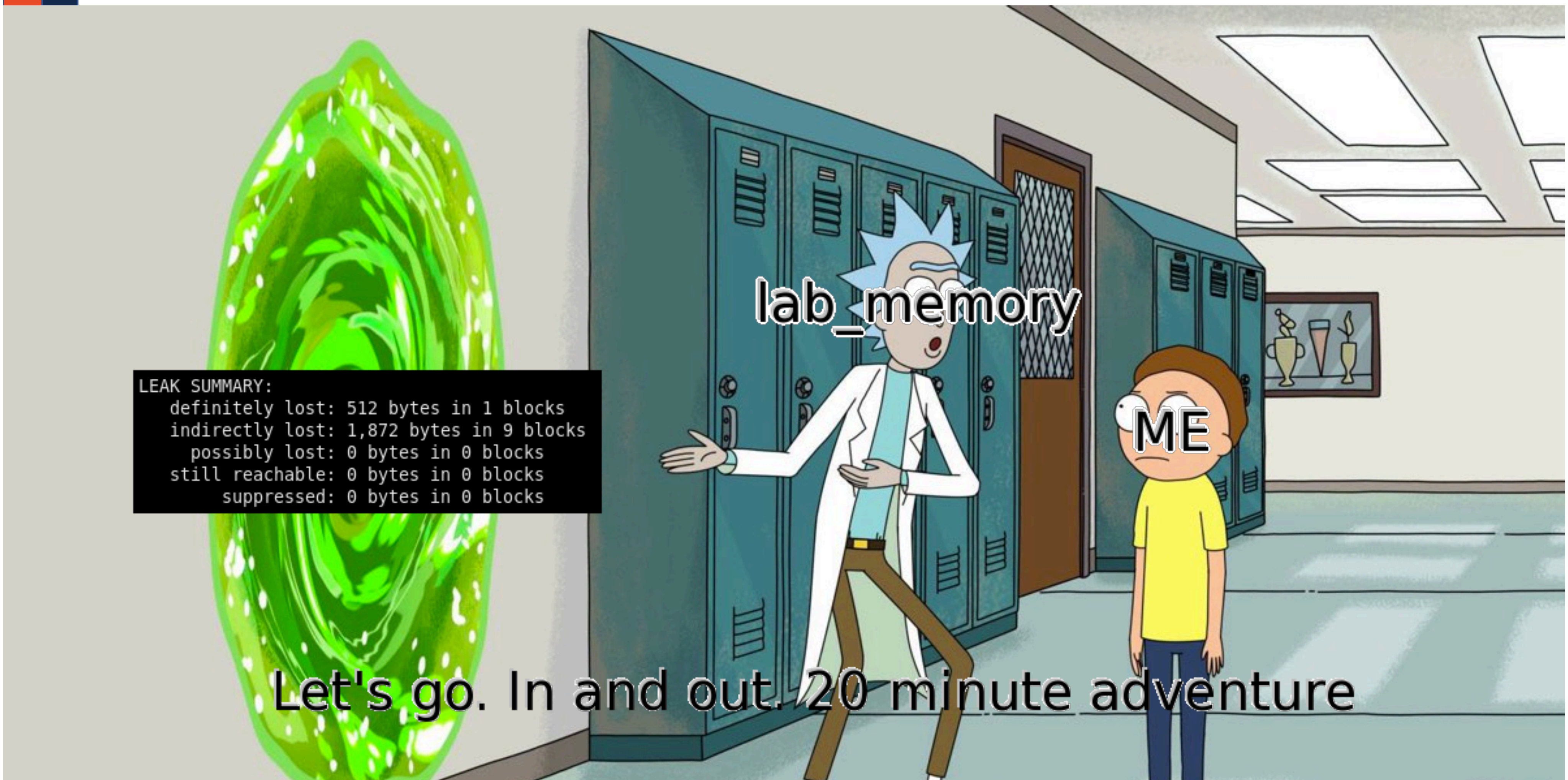


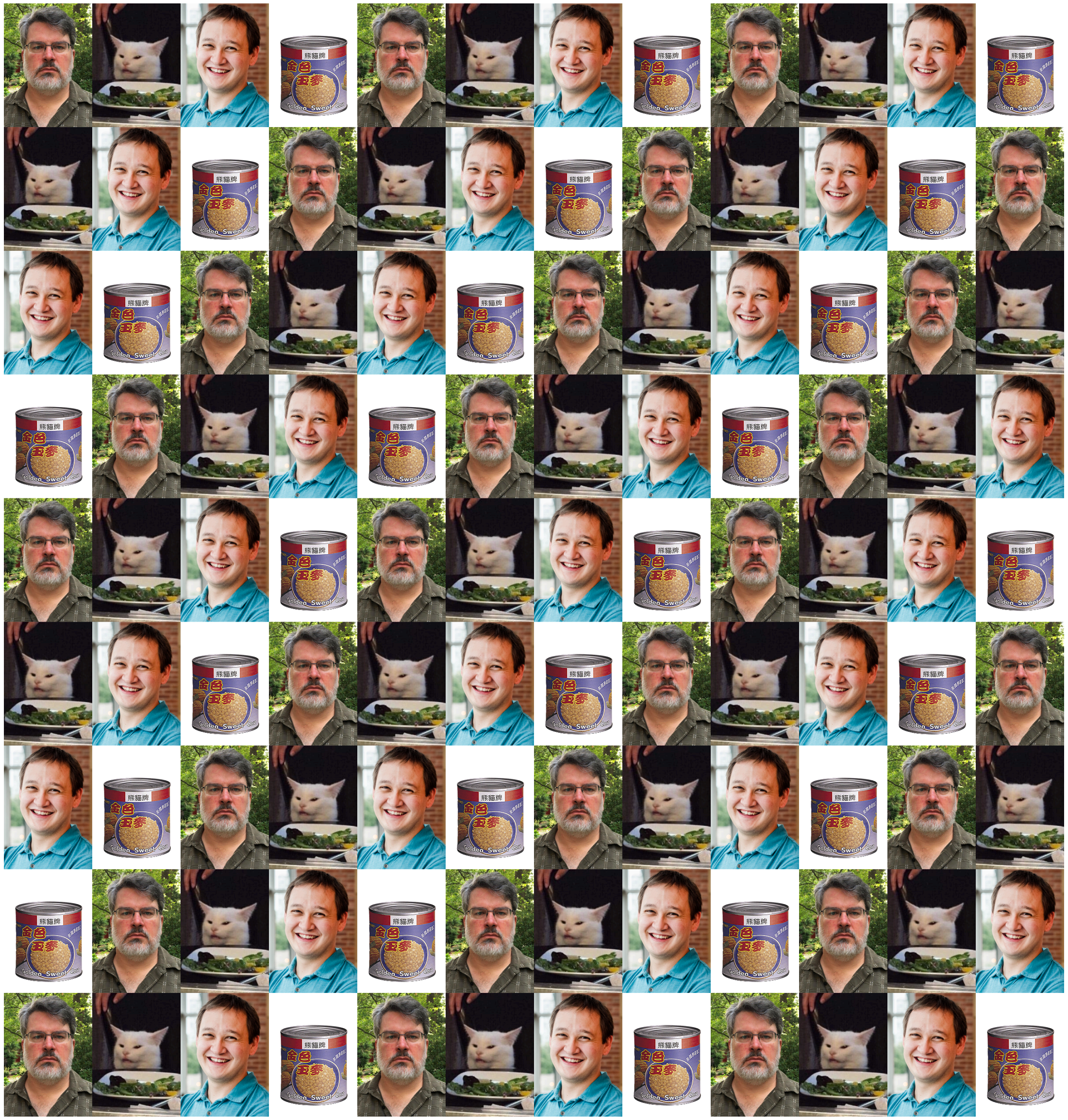


::begin	::end

```
Cube rubix;
```

RubikCube.cpp


lab_memory
LEAK SUMMARY:
definitely lost: 512 bytes in 1 blocks
indirectly lost: 1,872 bytes in 9 blocks
possibly lost: 0 bytes in 0 blocks
still reachable: 0 bytes in 0 blocks
suppressed: 0 bytes in 0 blocks
ME
Let's go. In and out. 20 minute adventure

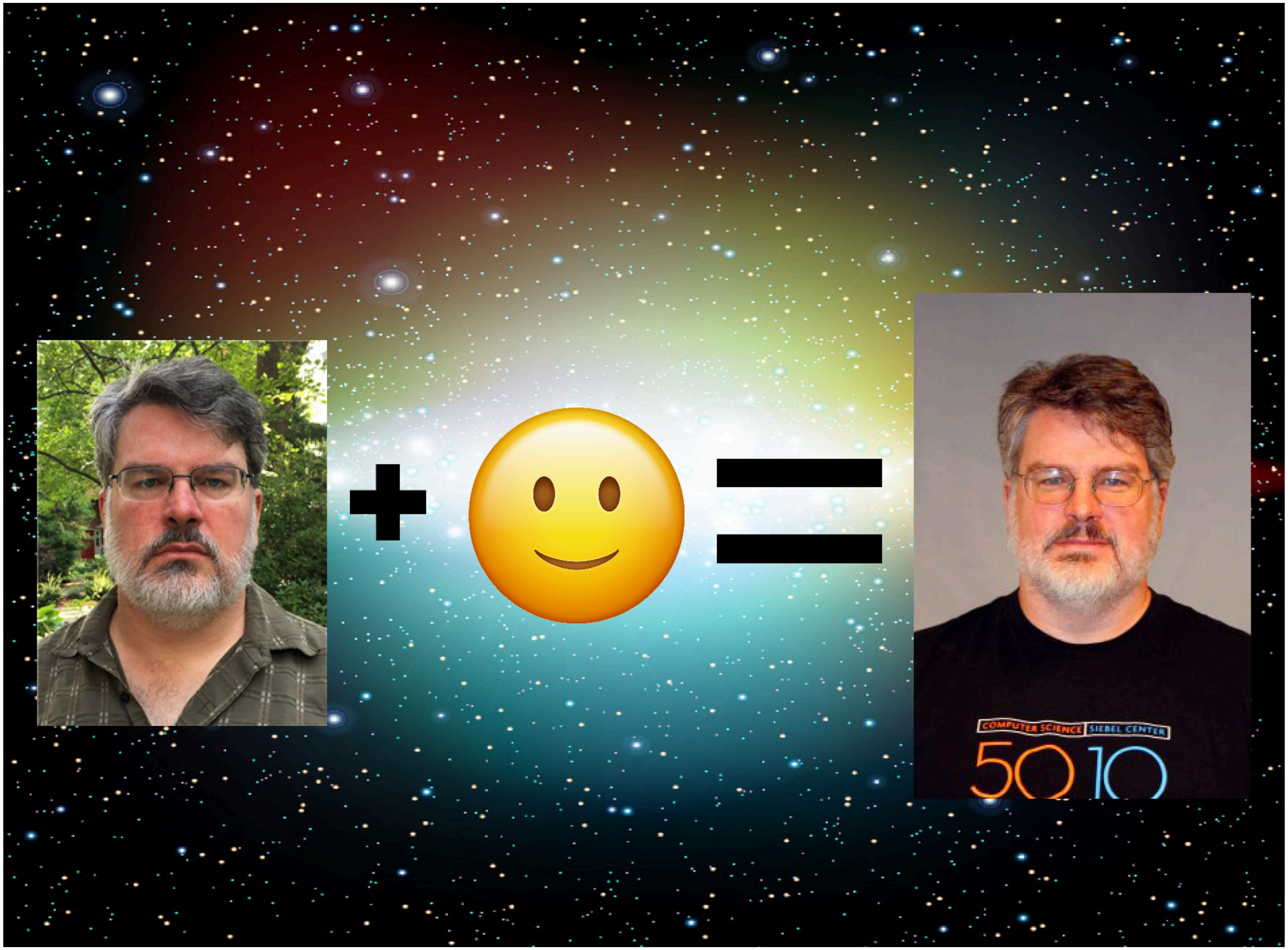



CS @ ILLINOIS
COMPUTER SCIENCE
ECE Illinois
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Hits CTRL + S to save code
2 seconds later

My brain:


Use the sticker program to create art
Use the sticker program to make memes
Use the sticker program to make memes about other people making memes


COMPUTER SCIENCE
SIEBEL CENTER
50
10

stlList.cpp

```
#include <list>
#include <string>
#include <iostream>

struct Animal {
  std::string name, food;
  bool big;
  Animal(std::string name = "blob", std::string food = "you", bool big = true) :
    name(name), food(food), big(big) { /* nothing */ }
};

int main() {
  Animal g("giraffe", "leaves", true), p("penguin", "fish", false), b("bear");
  std::vector<Animal> zoo;

  zoo.push_back(g);
  zoo.push_back(p);   // std::vector’s insertAtEnd
  zoo.push_back(b);

  for ( std::vector<Animal>::iterator it = zoo.begin(); it != zoo.end(); it++ ) {
    std::cout << (*it).name << " " << (*it).food << std::endl;
  }

  return 0;
}
```

stlList.cpp

```
#include <list>
#include <string>
#include <iostream>

struct Animal {
  std::string name, food;
  bool big;
  Animal(std::string name = "blob", std::string food = "you", bool big = true) :
    name(name), food(food), big(big) { /* nothing */ }
};

int main() {
  Animal g("giraffe", "leaves", true), p("penguin", "fish", false), b("bear");
  std::vector<Animal> zoo;

  zoo.push_back(g);
  zoo.push_back(p);   // std::vector's insertAtEnd
  zoo.push_back(b);

  for ( auto it = zoo.begin(); it != zoo.end(); it++ ) {
    std::cout << (*it).name << " " << (*it).food << std::endl;
  }

  return 0;
}
```

stlList.cpp

```cpp
#include <list>
#include <string>
#include <iostream>

struct Animal {
  std::string name, food;
  bool big;
  Animal(std::string name = "blob", std::string food = "you", bool big = true) :
    name(name), food(food), big(big) { /* none */ }
};

int main() {
  Animal g("giraffe", "leaves", true), p("penguin", "fish", false), b("bear");
  std::vector<Animal> zoo;

  zoo.push_back(g);
  zoo.push_back(p);   // std::vector's insertAtEnd
  zoo.push_back(b);

  for ( const Animal & animal : zoo ) {
    std::cout << animal.name << " " << animal.food << std::endl;
  }

  return 0;
}
```

For Each and Iterators

```
for ( const TYPE & variable : collection ) {
  // ...
}
```

```
std::vector<Animal> zoo;
…
for ( const Animal & animal : zoo ) {
    std::cout << animal.name << " " << animal.food << std::endl;
}
```

For Each and Iterators

```
for ( const TYPE & variable : collection ) {
  // ...
}
```

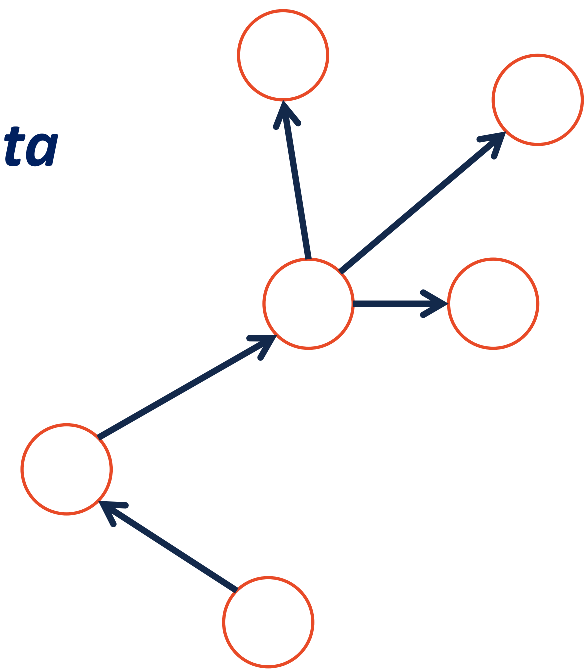
```
std::vector<Animal> zoo;
…
for ( const Animal & animal : zoo ) {
    std::cout << animal.name << " " << animal.food << std::endl;
}
```

```
std::multimap<std::string, Animal> zoo;
…
for ( const Animal & animal : zoo ) {
    std::cout << animal.name << " " << animal.food << std::endl;
}
```

Trees

"The most important non-linear data structure in computer science."

- David Knuth, The Art of Programming, Vol. 1

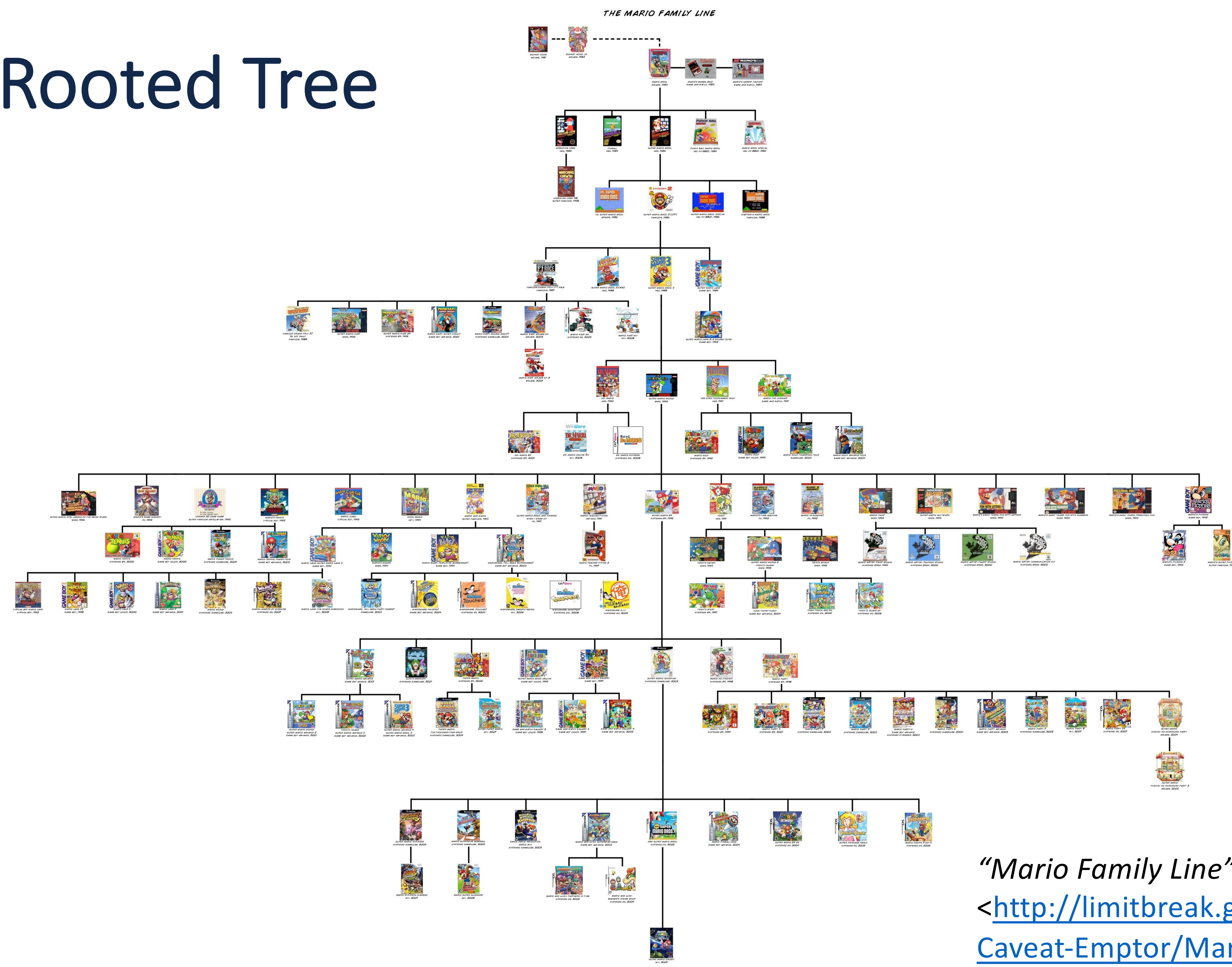


A tree is:

-

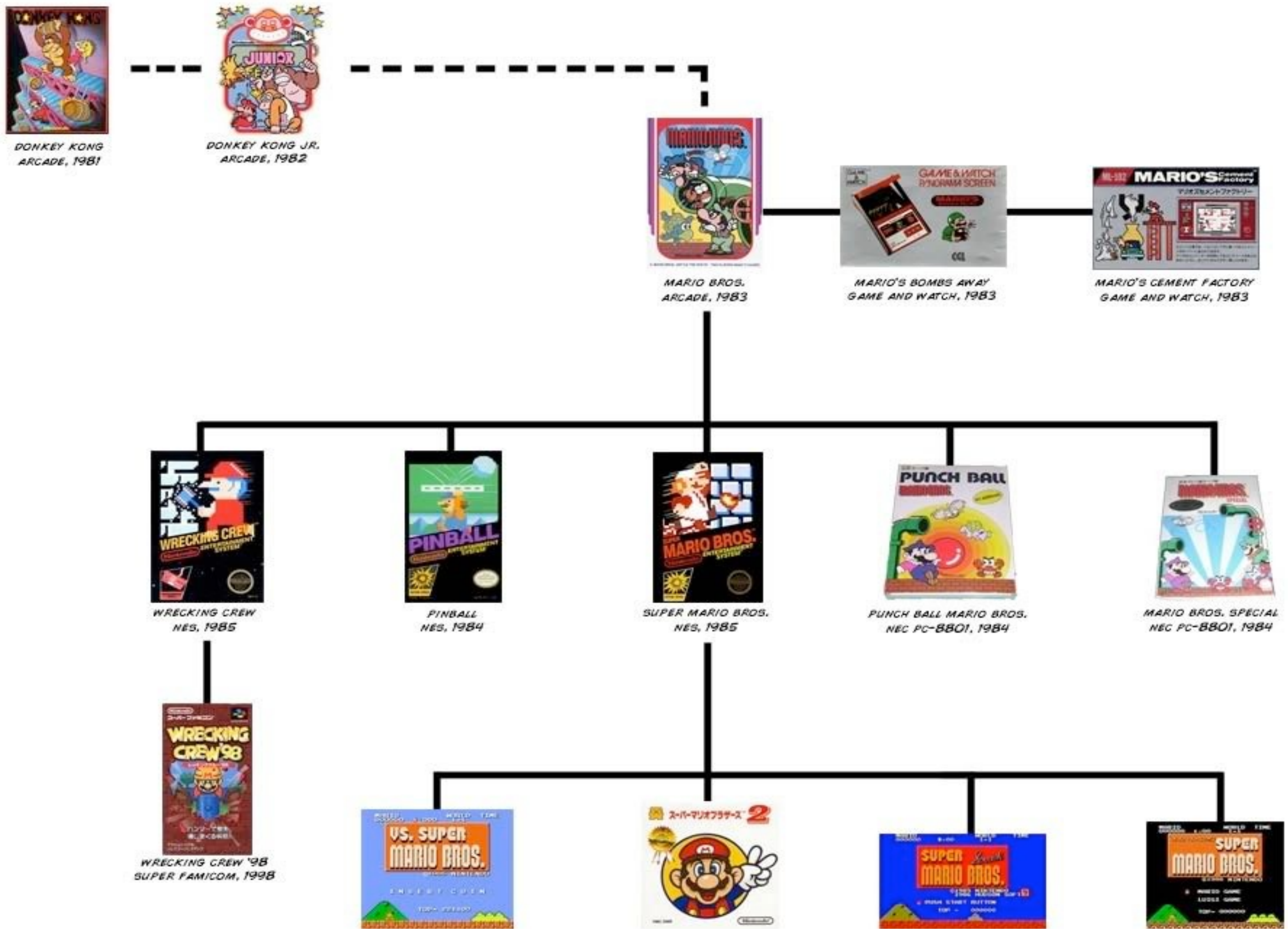
-

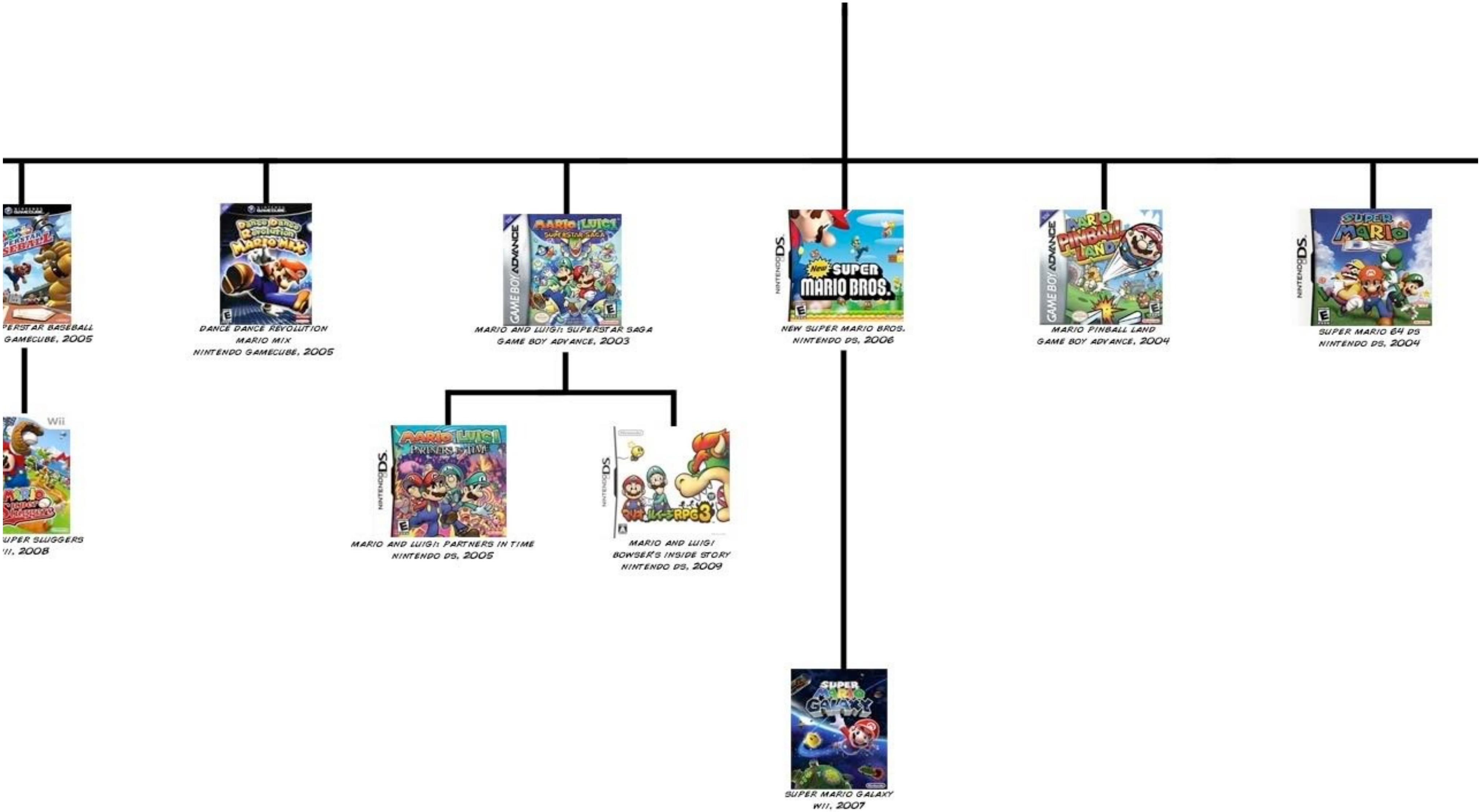
A Rooted Tree



"Mario Family Line" <http://limitbreak.gameriot.com/blogs/Caveat-Emptor/Mario-Family-Line>

THE MARIO FAMILY LINE

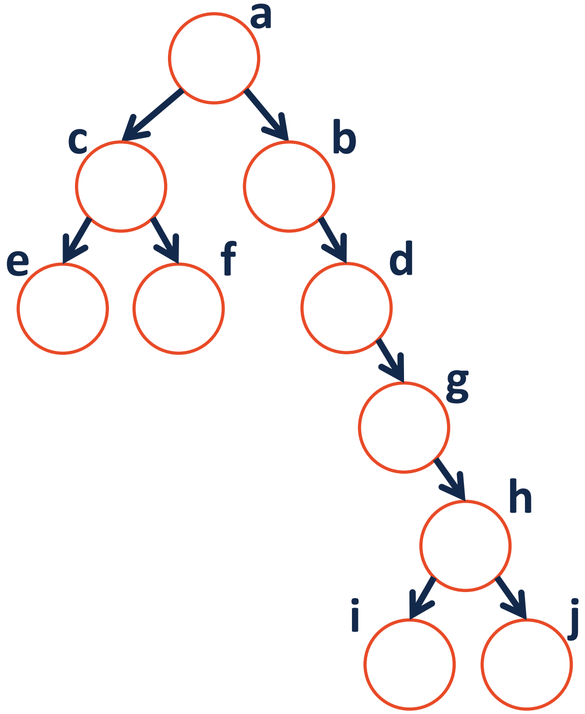



PERSTAR BASEBALL
GAMECUBE, 2005
DANCE DANCE REVOLUTION
MARIO MIX
NINTENDO GAMECUBE, 2005
MARIO AND LUIGI: SUPERSTAR SAGA
GAME BOY ADVANCE, 2003
NEW SUPER MARIO BROS.
NINTENDO DS, 2006
MARIO PINBALL LAND
GAME BOY ADVANCE, 2004
SUPER MARIO 64 DS
NINTENDO DS, 2004
UPER SLUGGERS
2008
MARIO AND LUIGI: PARTNERS IN TIME
NINTENDO DS, 2005
MARIO AND LUIGI
BOWSER'S INSIDE STORY
NINTENDO DS, 2009
SUPER MARIO GALAXY
WII, 2007

More Specific Trees

We'll focus on **binary trees**:

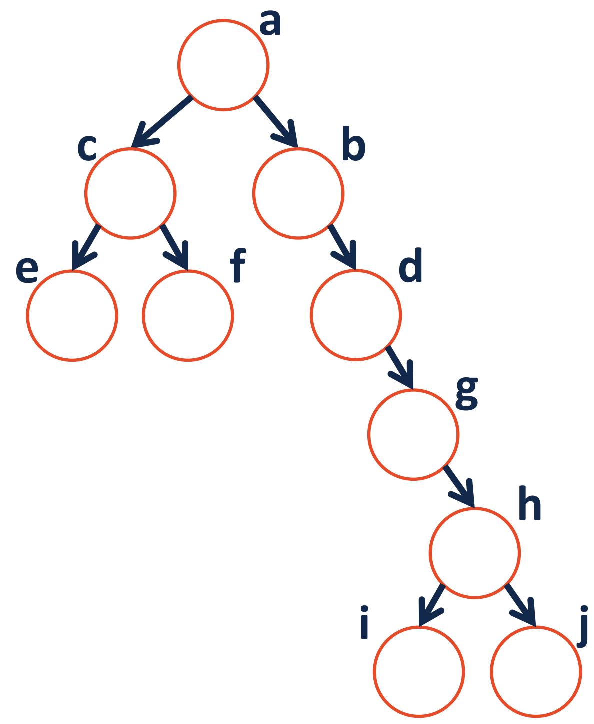
- A binary tree is **rooted** – every node can be reached via a path from the root



More Specific Trees

We'll focus on **binary trees**:

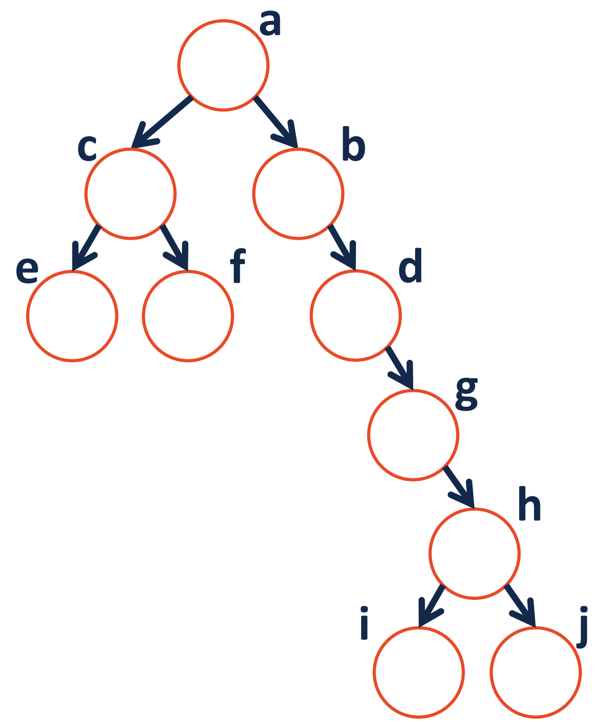
- A binary tree is **acyclic** – there are no cycles within the graph



More Specific Trees

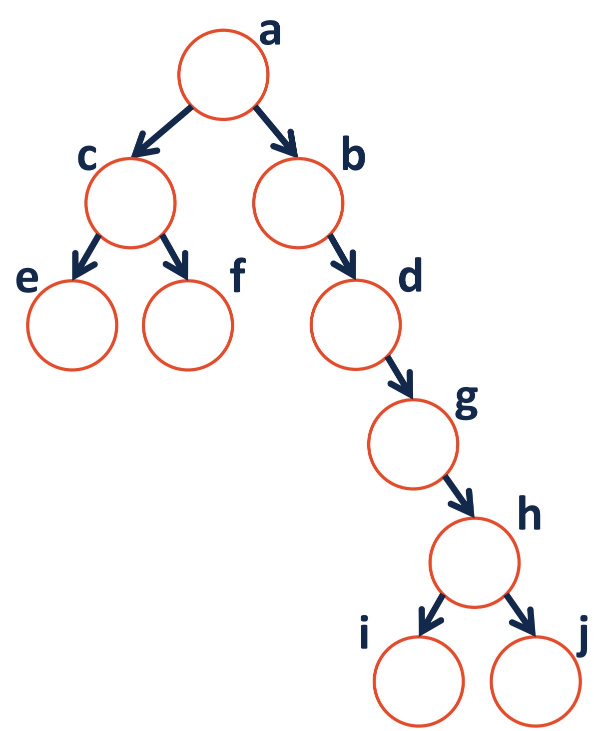
We'll focus on **binary trees**:

- A binary tree contains **two or fewer children** – where one is the "left child" and one is the "right child":



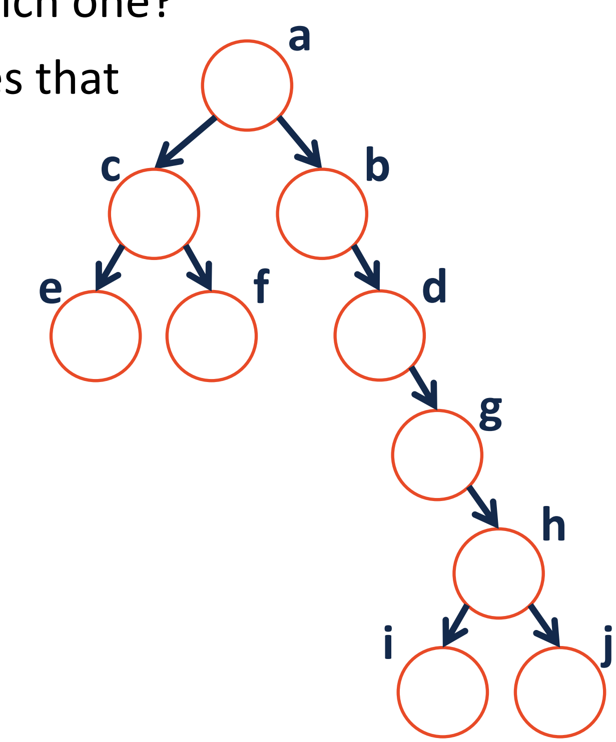
Tree Terminology

- What's the longest **English word** you can make using the **vertex** labels in the tree (repeats allowed)?



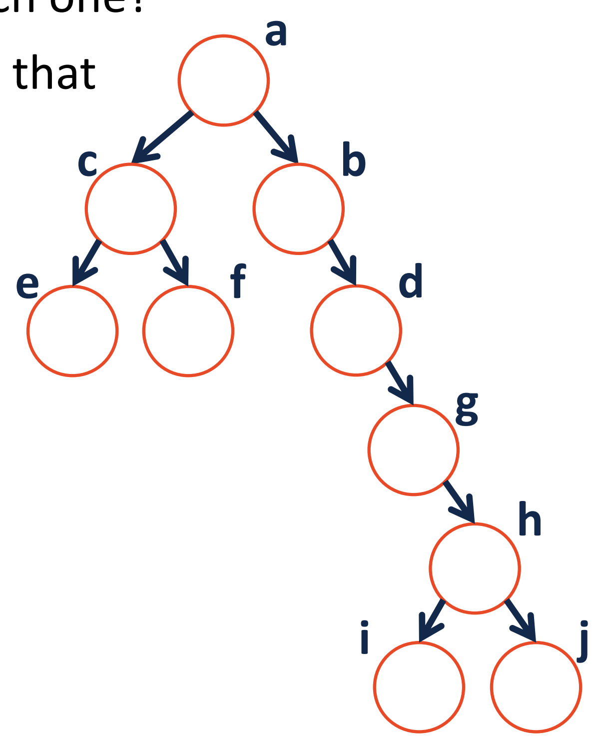
Tree Terminology

- Find an **edge** that is not on the longest **path** in the tree. Give that edge a reasonable name.
- One of the vertices is called the **root** of the tree. Which one?
- Make an "word" containing the names of the vertices that have a **parent** but no **sibling**.
- How many parents does each vertex have?
- Which vertex has the fewest **children**?
- Which vertex has the most **ancestors**?
- Which vertex has the most **descendants**?
- List all the vertices is b's left **subtree**.
- List all the **leaves** in the tree.



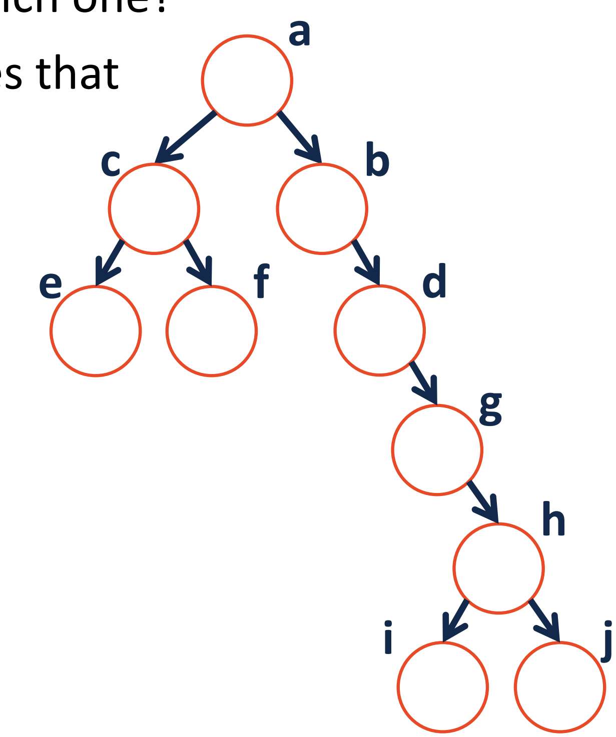
Tree Terminology

- Find an **edge** that is not on the longest **path** in the tree. Give that edge a reasonable name.
- One of the vertices is called the **root** of the tree. Which one?
- Make an “word” containing the names of the vertices that have a **parent** but no **sibling**.
- How many parents does each vertex have?
- Which vertex has the fewest **children**?
- Which vertex has the most **ancestors**?
- Which vertex has the most **descendants**?
- List all the vertices is b’s left **subtree**.
- List all the **leaves** in the tree.



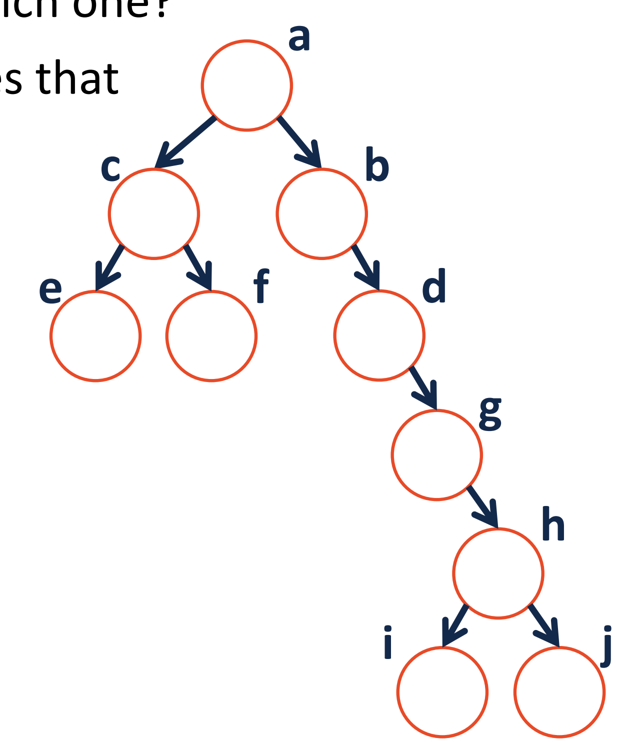
Tree Terminology

- Find an **edge** that is not on the longest **path** in the tree. Give that edge a reasonable name.
- One of the vertices is called the **root** of the tree. Which one?
- Make an "word" containing the names of the vertices that have a **parent** but no **sibling**.
- How many parents does each vertex have?
- Which vertex has the fewest **children**?
- Which vertex has the most **ancestors**?
- Which vertex has the most **descendants**?
- List all the vertices is b's left **subtree**.
- List all the **leaves** in the tree.



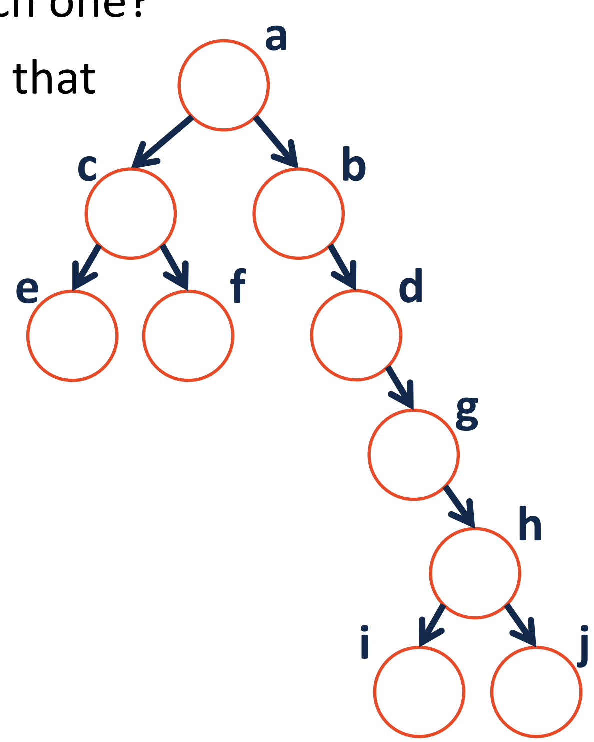
Tree Terminology

- Find an **edge** that is not on the longest **path** in the tree. Give that edge a reasonable name.
- One of the vertices is called the **root** of the tree. Which one?
- Make an "word" containing the names of the vertices that have a **parent** but no **sibling**.
- How many parents does each vertex have?
- Which vertex has the fewest **children**?
- Which vertex has the most **ancestors**?
- Which vertex has the most **descendants**?
- List all the vertices is b's left **subtree**.
- List all the **leaves** in the tree.



Tree Terminology

- Find an **edge** that is not on the longest **path** in the tree. Give that edge a reasonable name.
- One of the vertices is called the **root** of the tree. Which one?
- Make an "word" containing the names of the vertices that have a **parent** but no **sibling**.
- How many parents does each vertex have?
- Which vertex has the fewest **children**?
- Which vertex has the most **ancestors**?
- Which vertex has the most **descendants**?
- List all the vertices is b's left **subtree**.
- List all the **leaves** in the tree.



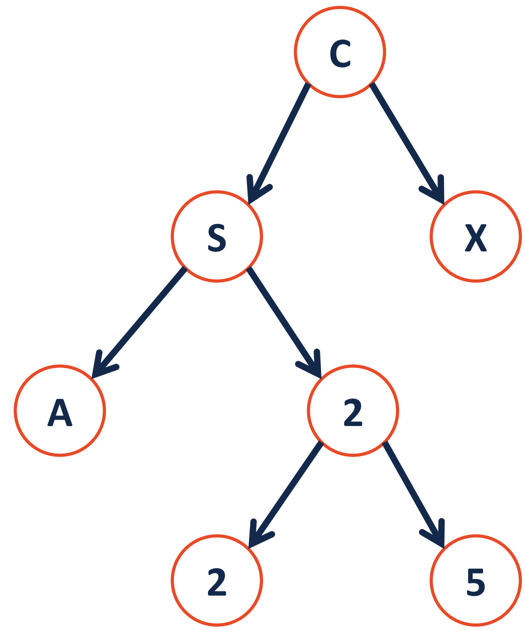
Binary Tree – Defined

A *binary tree* T is either:

-

OR

-

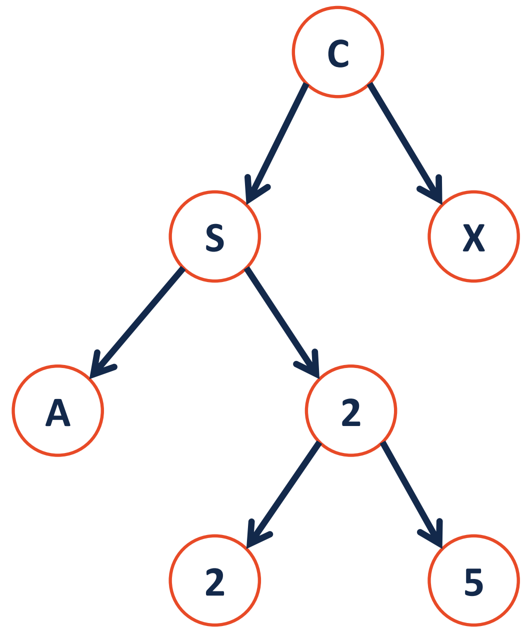


Tree Property: height

height(T): length of the longest path from the root to a leaf

Given a binary tree T:

height(T) =

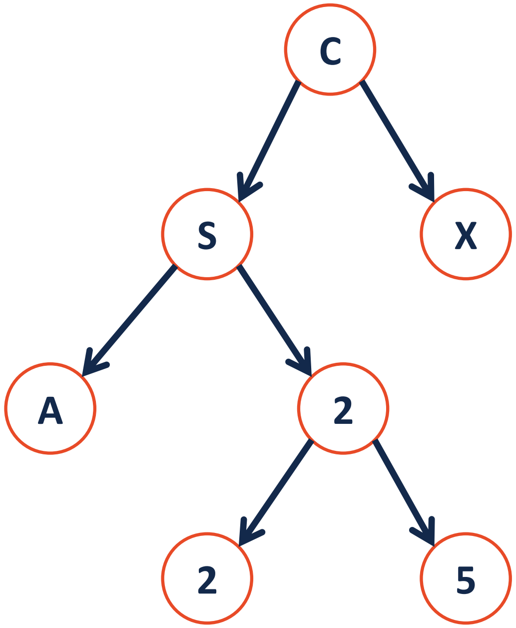


Tree Property: full

A tree ***F*** is **full** if and only if:

1.

2.

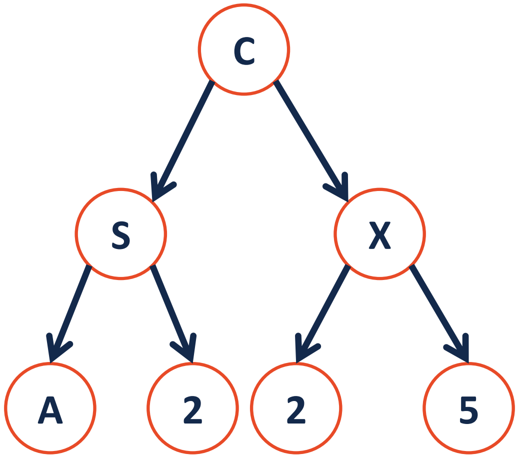


Tree Property: perfect

A **perfect** tree ***P*** is:

1.

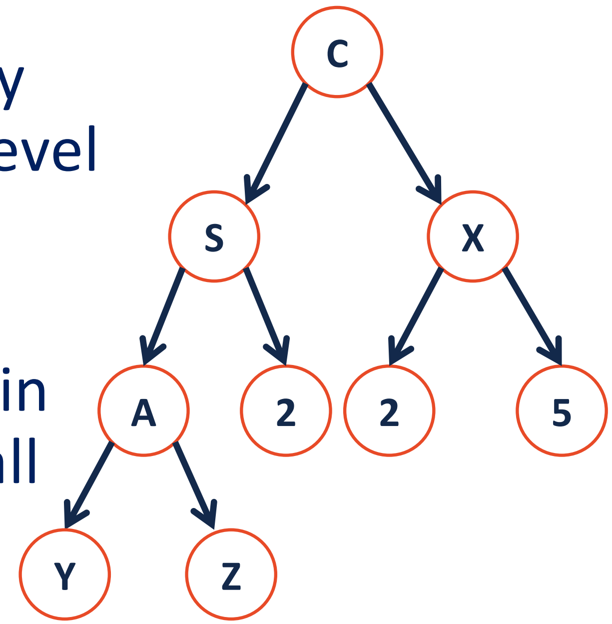
2.



Tree Property: complete

Conceptually: A perfect tree for every level except the last, where the last level if "pushed to the left".

Slightly more formal: For any level k in [0, h-1], k has 2^k nodes. For level h, all nodes are "pushed to the left".



Tree Property: complete

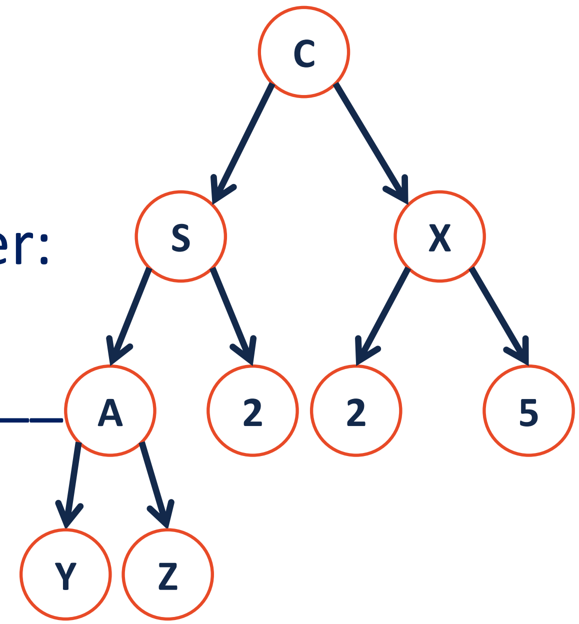
A **complete** tree ***C*** of height **h**, $\mathbf{C_h}$:

1. $\mathbf{C_{-1}}$ = {}
2. $\mathbf{C_h}$ *(where h>0)* **= {r, T_L, T_R}** and either:

T_L is ___________ and T_R is __________

OR

T_L is ___________ and T_R is __________



Tree Property: complete

Is every **full** tree **complete**?

If every **complete** tree **full**?