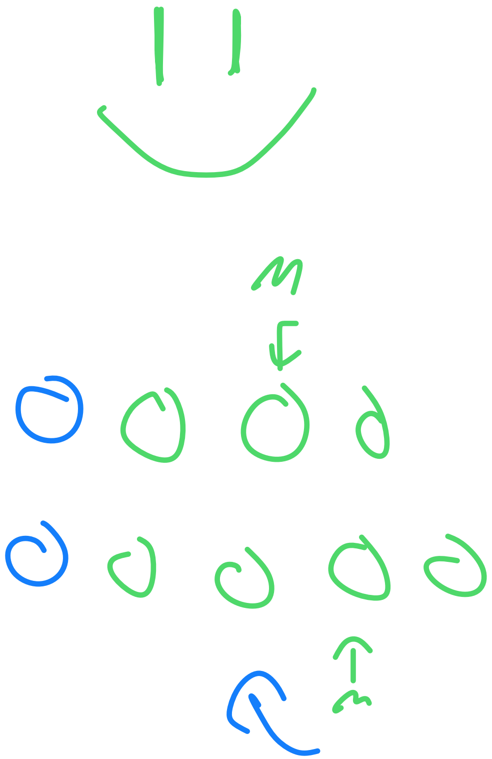
Data Structures

Queues and Iterators (and Trees)

CS 225
Brad Solomon & G Carl Evans

September 11, 2023



UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Announcements

Honors Class 199-225: First class today at CIF - 4039 @ 5 PM

Exam 1 starts today!

mp_stickers due date is Tuesday for everyone

No late day!

mp_lists releases today (or tomorrow)

Learning Objectives

Review the queue data structure

Introduce and explore iterators

Introduce trees

Queue Data Structure

0 head

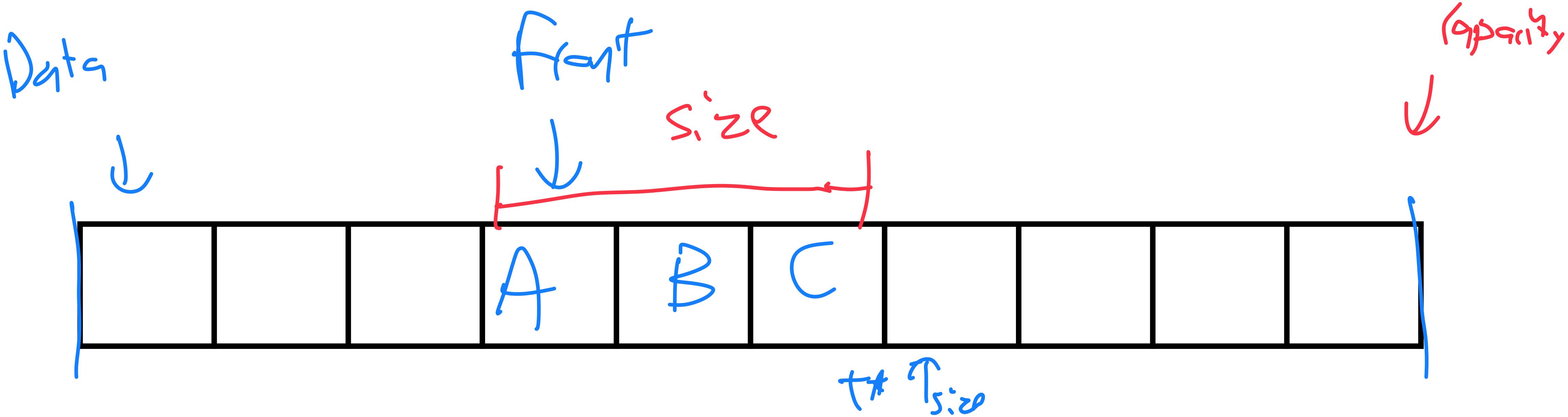
0 tail

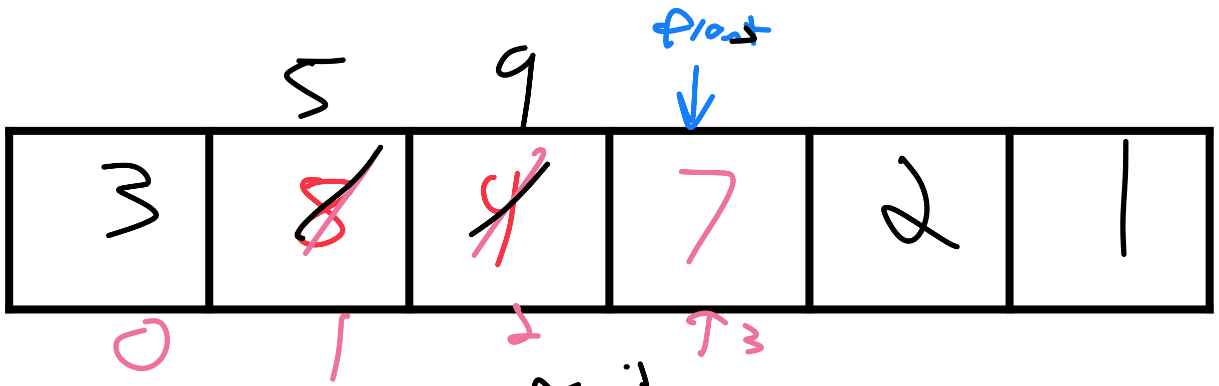
What do we need to track to maintain a queue with an array list?

T* → unsigned integers

↳ front =

↳ size =





			front		
	5	9			
3	~~8~~	~~4~~	7	2	1
0	1	2	3		

→ If size < capacity

Enqueue(D): insert at (front+size) % capacity

↳ size ++ % capacity

If size == 0, return null)

Dequeue(D): remove at front

↳ front ++; % capacity

↳ size --;

Be aware of edge cases

Size: ~~0~~ ~~3~~ ~~2~~ ~~2~~ 1

Front: ~~0~~ ~~2~~ 3

```
Queue<int> q;
q.enqueue(3);
q.enqueue(8);
q.enqueue(4);
q.dequeue();
q.enqueue(7);
q.dequeue();
q.dequeue();
q.enqueue(2);
q.enqueue(1);
q.enqueue(3);
q.enqueue(5);
q.dequeue(); <-
q.enqueue(9);
```

Capacity: 6 ✓

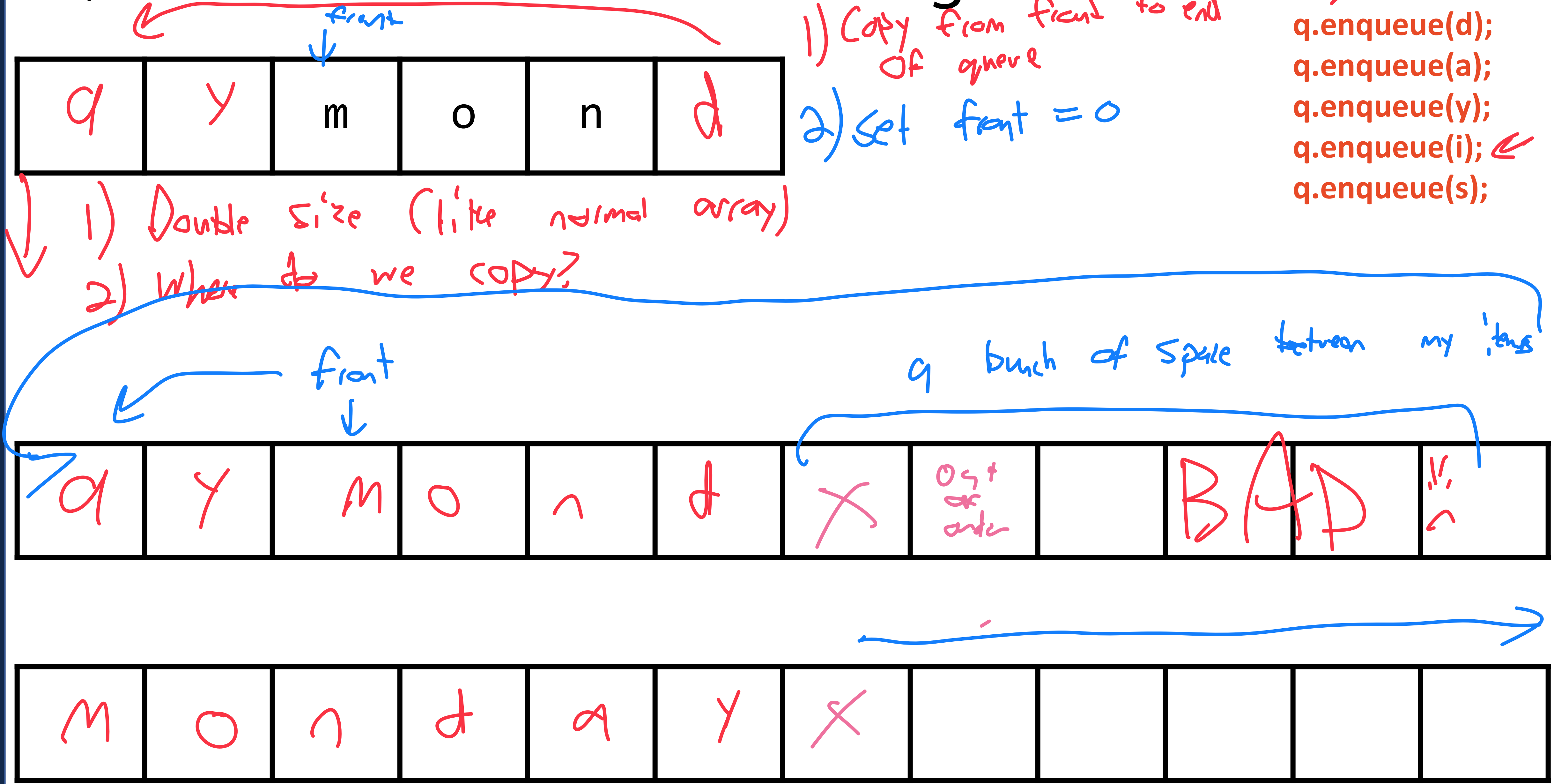
Queue Data Structure: Resizing

chk in dequeue

```
Queue<char> q;
... 3 wo/4
q.enqueue(d);
q.enqueue(a);
q.enqueue(y);
q.enqueue(i);
q.enqueue(s);
```

1) Copy from front to end of queue

2) set front = 0



1) Double size (like normal array)

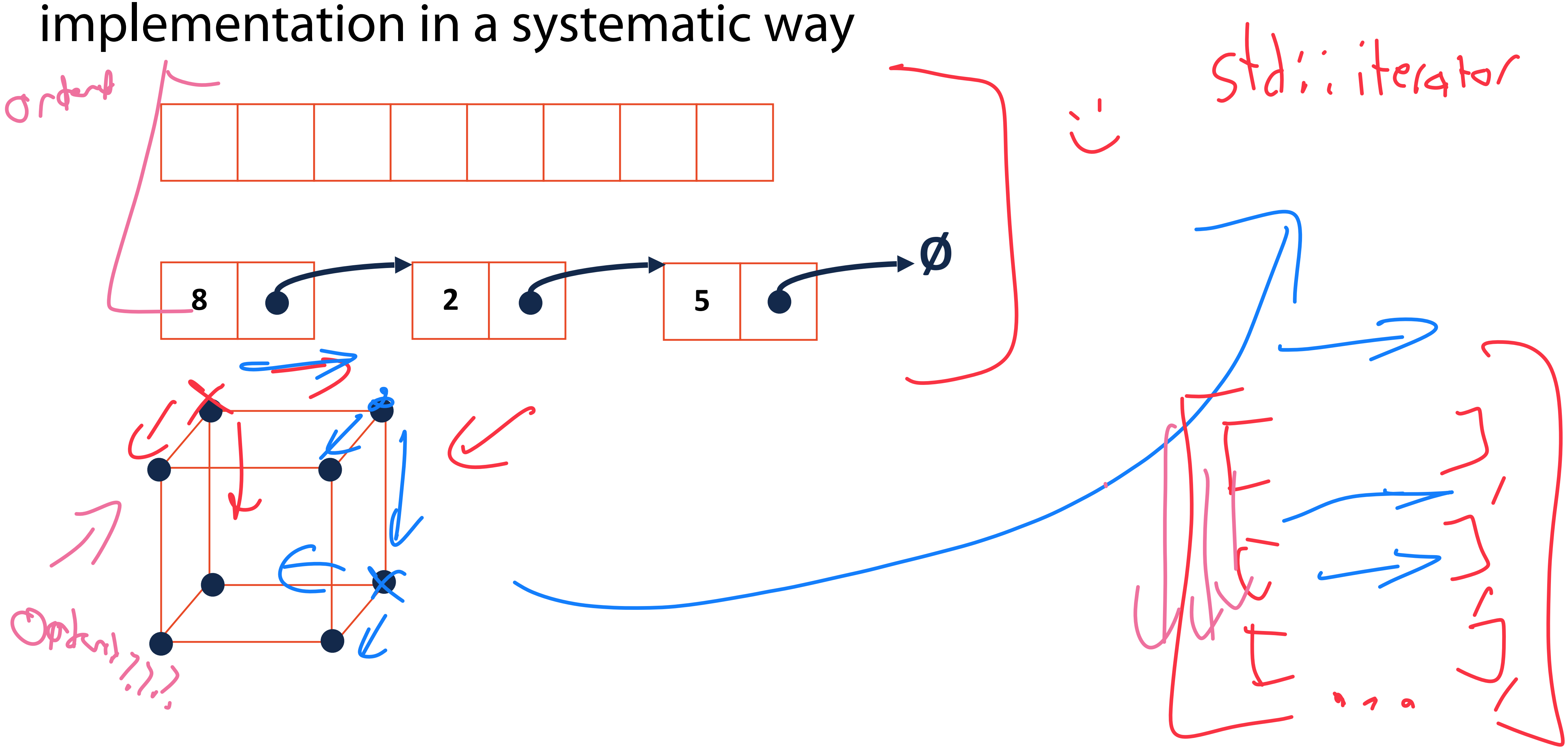
2) When do we copy?

Queue ADT

- [Order]: First in first out FIFO
- [Implementation]: LL w/ head and tail
  circular array list*
  ↳ worst case for array is $O(n)$
  b/c resize
- [Runtime]: enqueue
  dequeue $O(1)$*

Iterators

We want to be able to loop through all elements for any underlying implementation in a systematic way



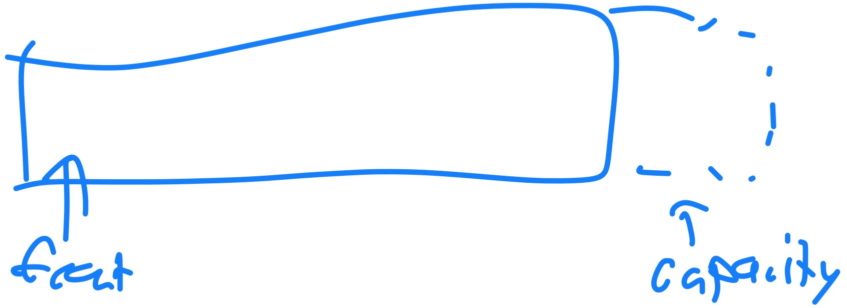
Iterators

list

For a class to implement an iterator, it needs two functions:

`Iterator begin()` ← iterator points at start (my first element)

`Iterator end()` ← returns 1 past the end of my data structure



pointer past end

Iterators

class List
private
linked list

The actual iterator is defined as a class **inside** the outer class:

1. It must be of base class **`std::iterator`** (std::vector)

2. It must implement at least the following operations:

`Iterator& operator ++()` -pre-increment (~~move~~ Point to next object)

`const T & operator *()` - de-reference

`bool operator !=(const Iterator &)` - check if two iterator pointers are same

Iterators

This is truncated code

iterator tip

Future assignments will have you write custom iterators:

```cpp
template <class T>
class List {

    class ListIterator : public
std::iterator<std::bidirectional_iterator_tag, T> {
      public:

        ListIterator& operator++();

        ListIterator& operator--()

        bool operator!=(const ListIterator& rhs);

        const T& operator*();
    };

    ListIterator begin() const;

    ListIterator end() const;
};
```

ptr

built-in overloads

ops in inner class

getters in main class

Array[++i]

increments before

array[i++]

look up i and then increment

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) { /* 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;
}
```

List<T>::ListIterator it

```
std::vector<Animal> zoo;


/* Full text snippet */

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


/* Auto Snippet */

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

/* For Each Snippet */

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

for each animal in zoo

cast Animal

Trees

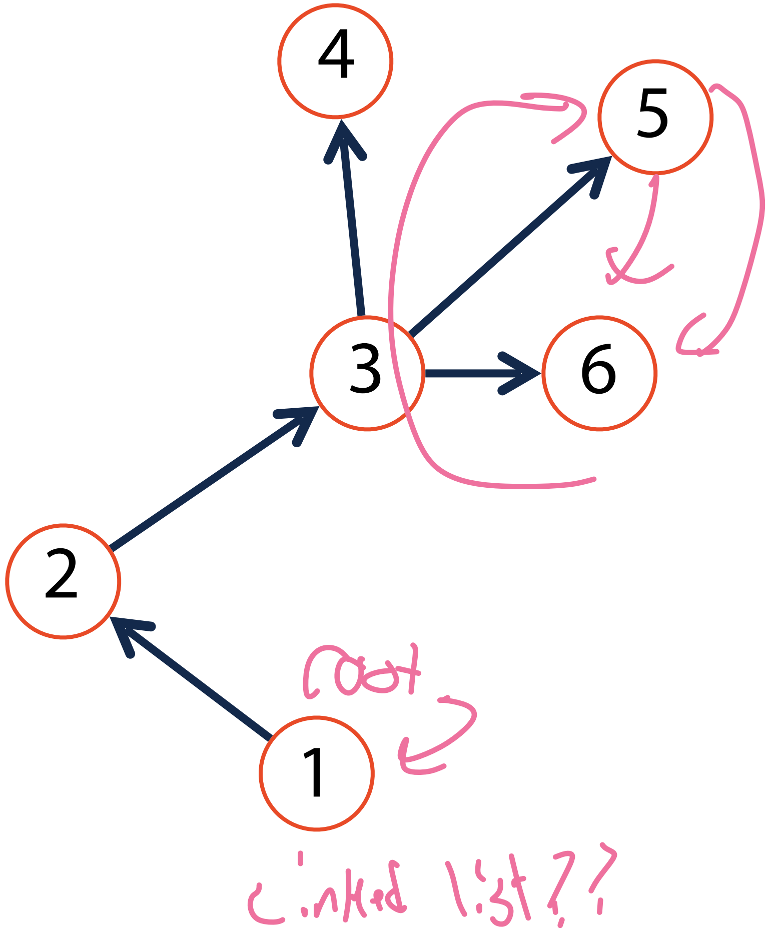
A non-linear data structure defined recursively as a collection of nodes where each node contains a value and zero or more connected nodes.

↳ Key, value

[In CS 225] a tree is also:

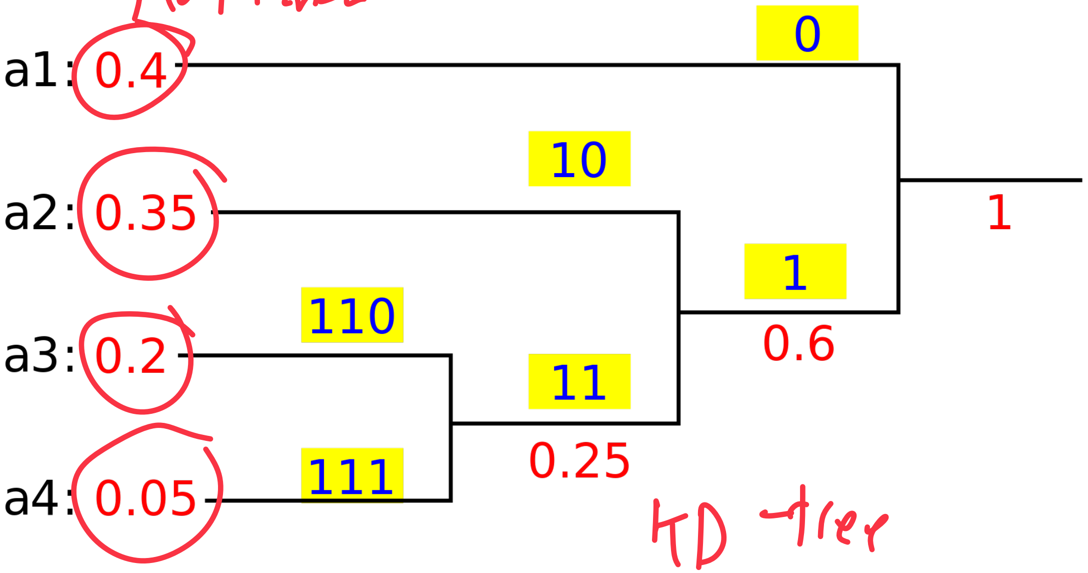
1) acyclic – has no cycles
has no path from node to itself

2) rooted – has a root
↳ Define some node to be root

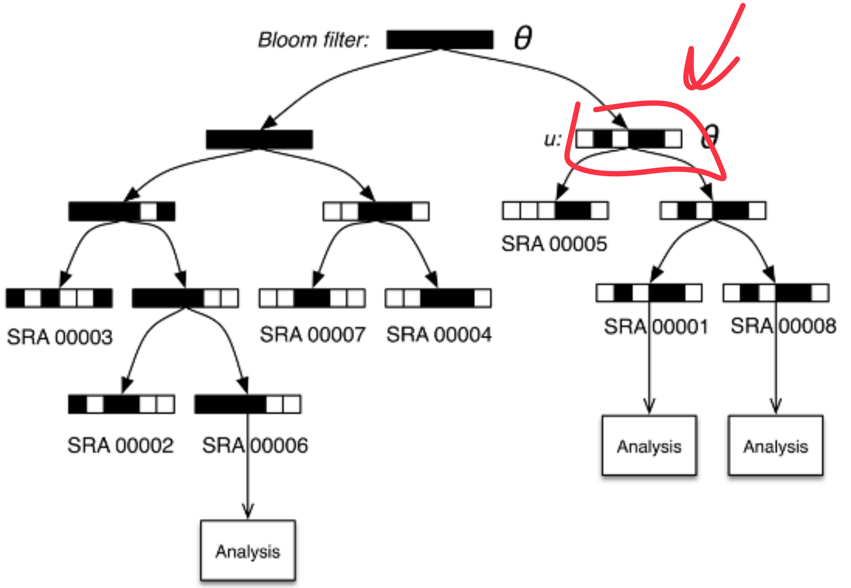


There are many *types* of trees

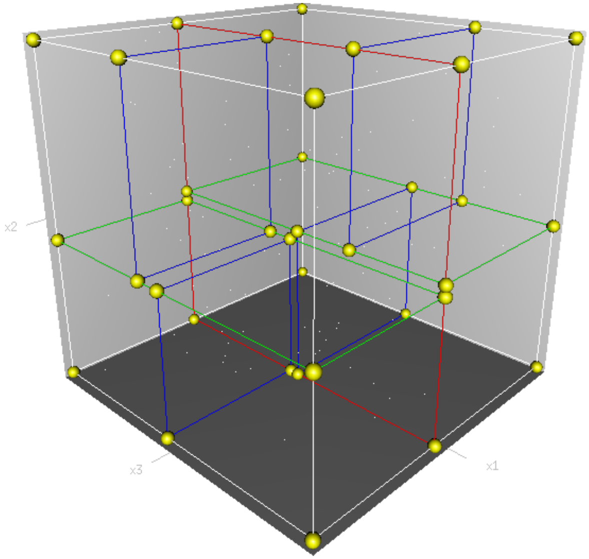
Huffman tree



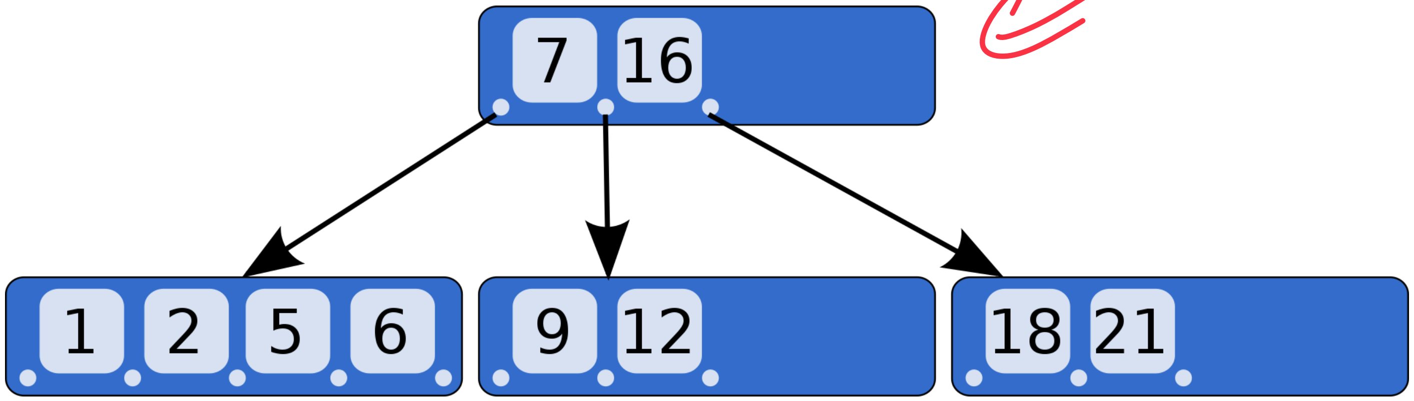
SBT



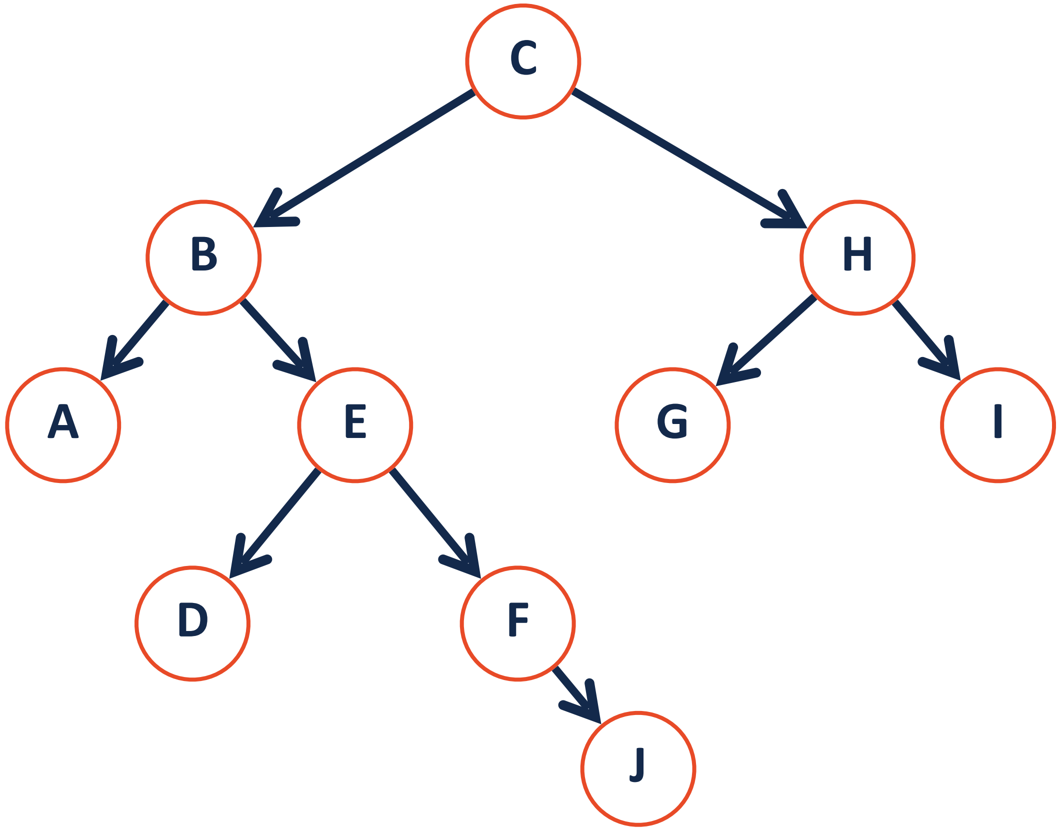
KD tree



B-tree



Tree Terminology

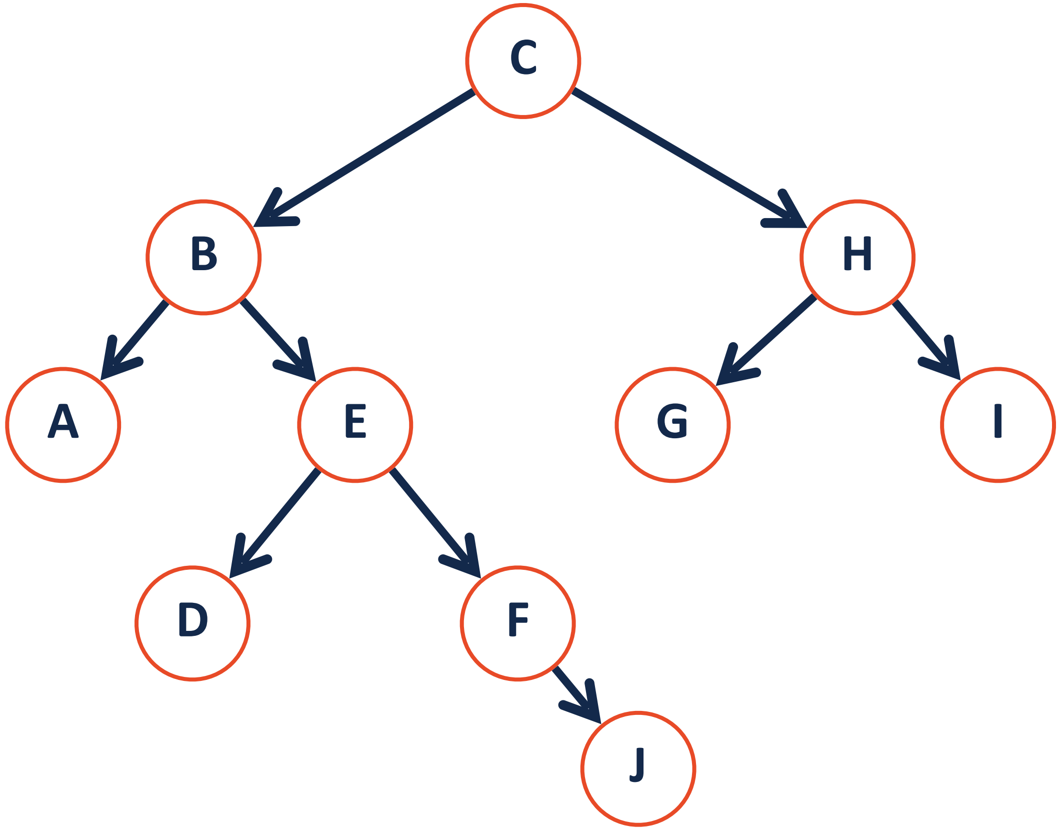


Node: The vertex of a tree

Edge: The connecting path between nodes

Path: A list of the edges (or nodes) traversed to go from node `start` to node `end`

Tree Terminology



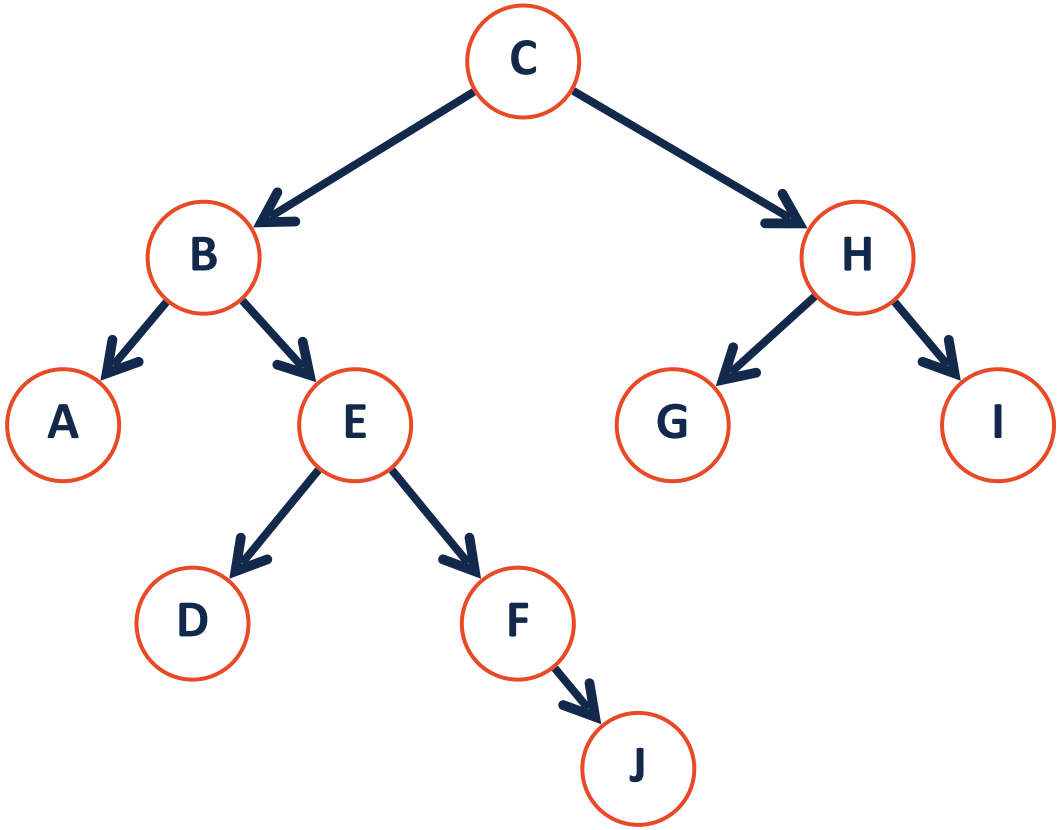
Parent: The precursor node to the current node is the 'parent'

Child: The nodes linked by the current node are it's 'children'

Neighbor: Parent or child

Degree: The number of children for a given node

Tree Terminology



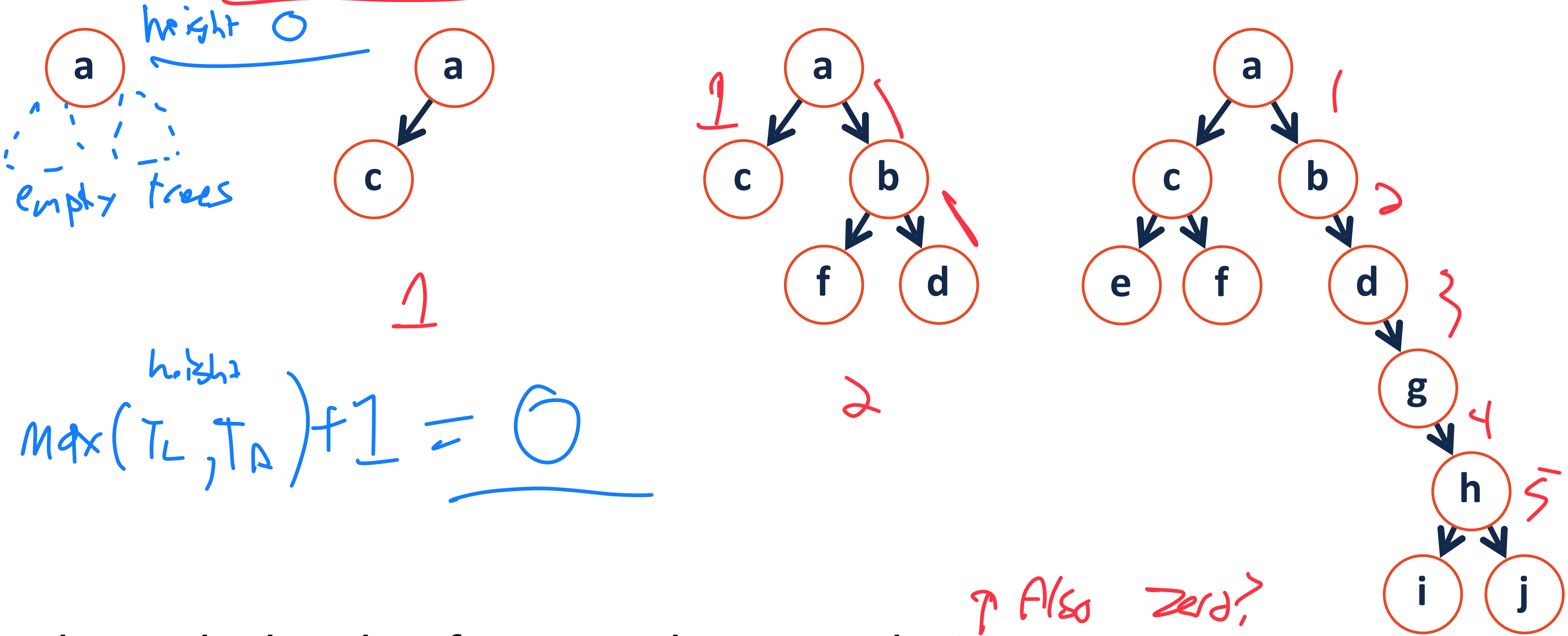
Root: The start of a tree (the only node with no parent).

Leaf: The terminating nodes of a tree (have no children)

Internal: A node with at least one child

Tree Terminology

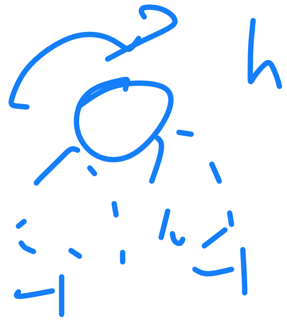
Height: the length of the longest path from the root to a leaf



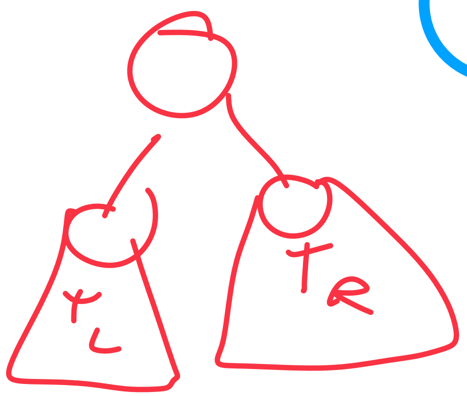
$$\max(\overset{\text{height}}{T_L, T_R}) + 1 = \underline{0}$$

What is the height of a tree with ***zero*** nodes? ↑ Also zero?

Tree Height



height = 0



Binary Tree

↳ root = null

height(T) =

Base Case: height(empty) = -1

Recursive Step: get height(T_L) & height(T_R)

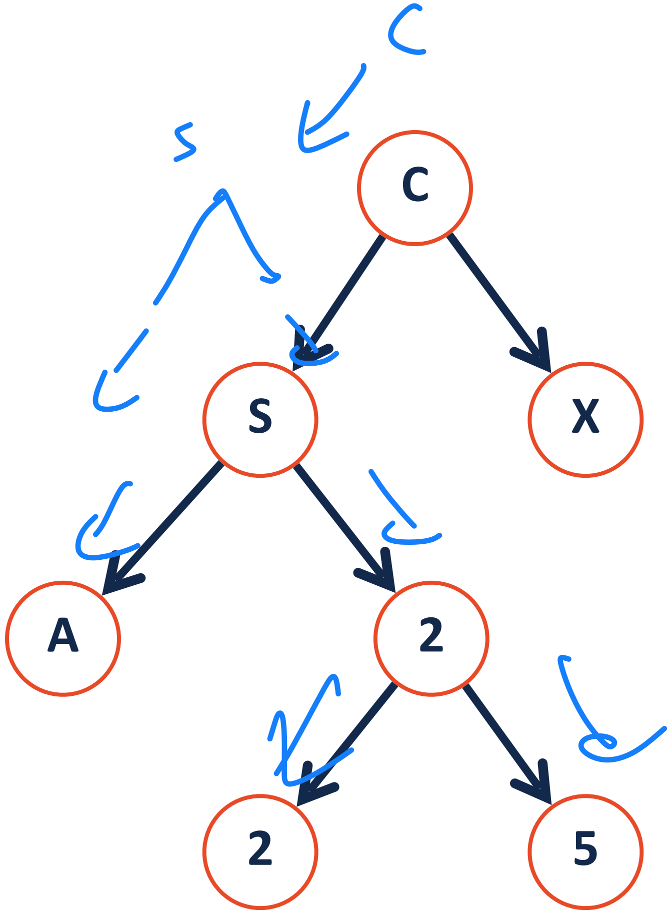
Combining: max [height(T_L), height(T_R)] +1

Binary Tree

A **binary tree** is a tree T such that:

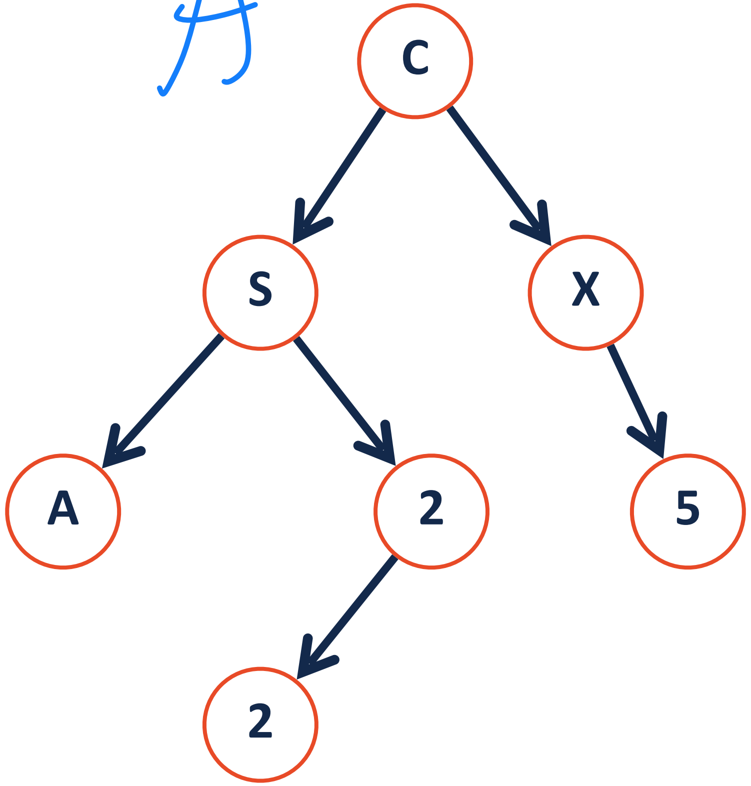
1. $T = \text{null} \; (\phi)$

2. $T = (\text{data}, T_L, T_R)$

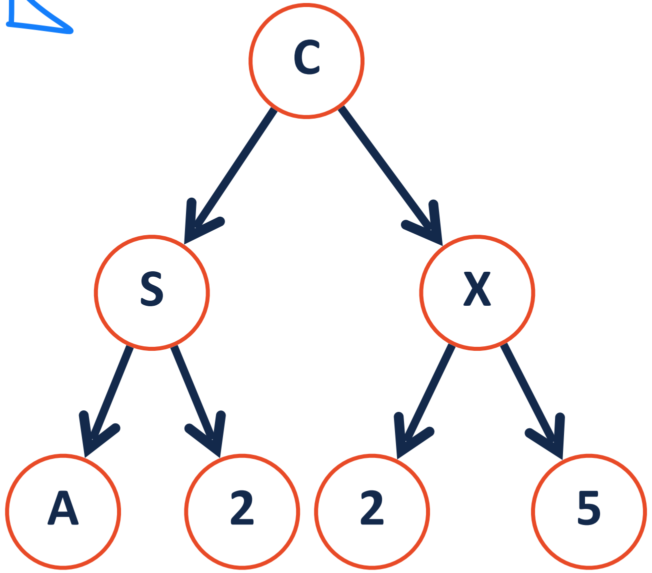


Which of the following are binary trees?

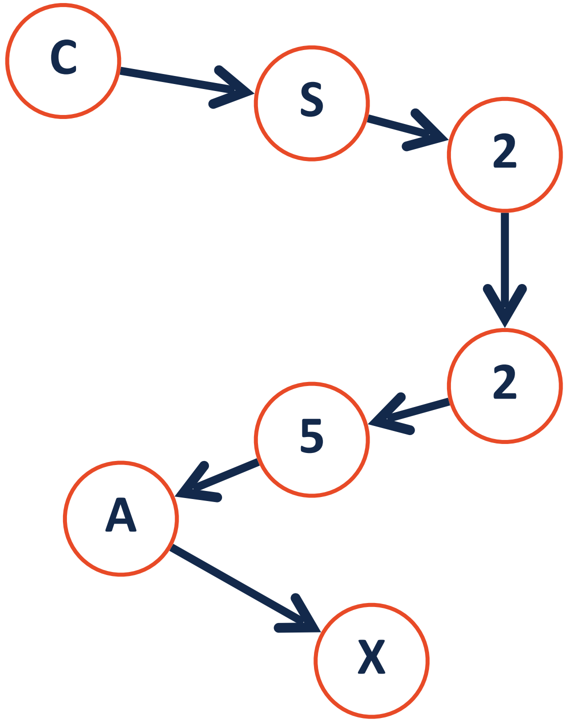
A



B



C



Binary Tree

Ended here

Lets define additional terminology for different **types** of binary trees!

1.

2.

3.

Binary Tree: full

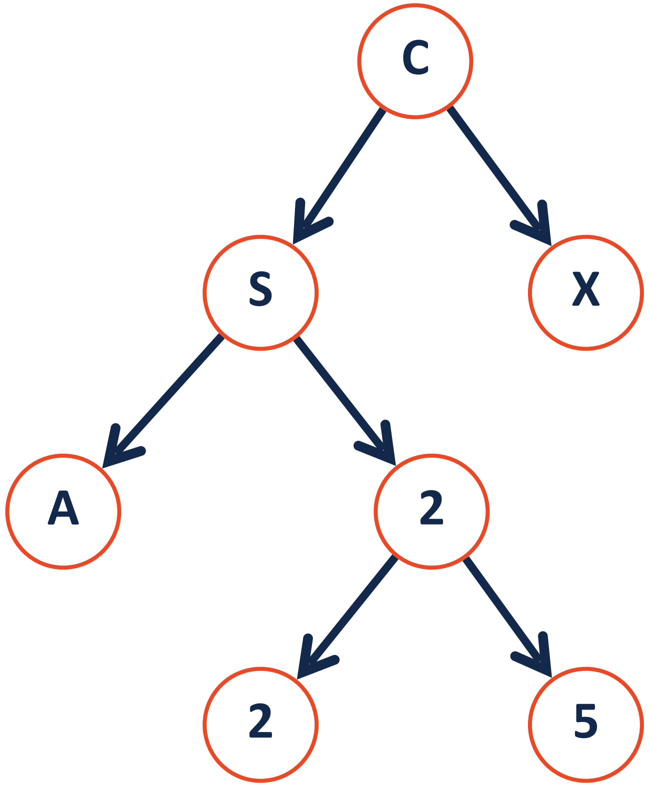
A **full tree** is a binary tree where every node has either 0 or 2 children

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

1.

2.

3.



Binary Tree: perfect

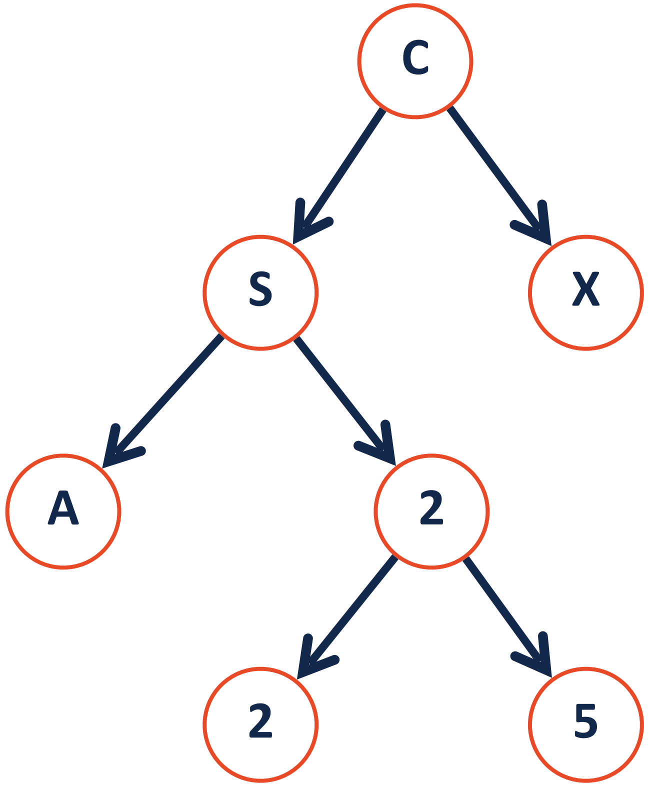
A **perfect tree** is a binary tree where…

Every internal node has 2 children and all leaves are at the same level.

A tree **P** is **perfect** if and only if:

1.

2.



Binary Tree: complete

A **complete tree** is a B.T. where…

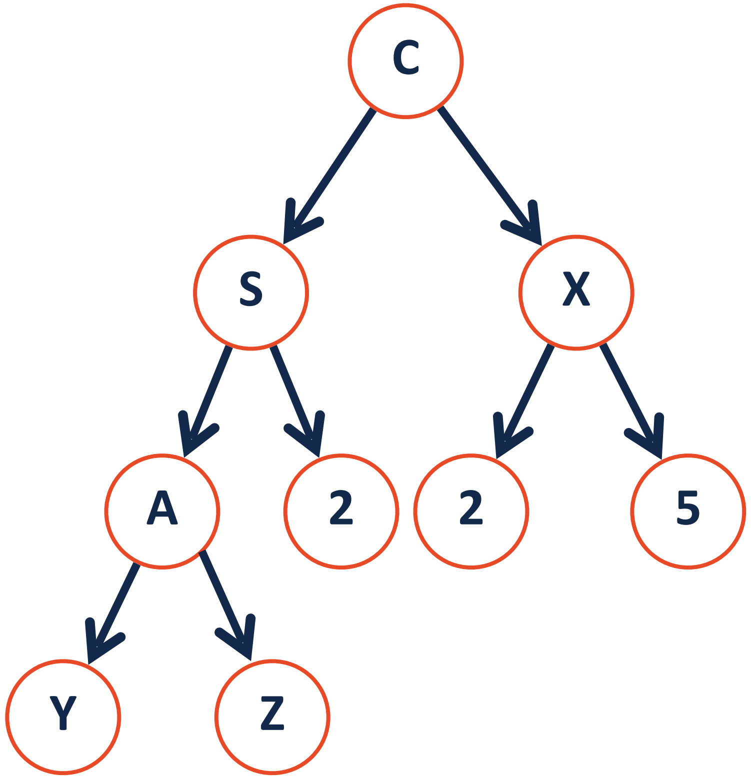
All levels are completely filled except the last (which is pushed to left)

A tree **C** is **complete** if and only if:

1.

2.

3.



Binary Tree

Why do we care?

1. Terminology instantly defines a particular tree structure

2. Understanding how to think ‘recursively’ is very important.

Binary Tree: Thinking with Types

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

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

For next time: Tree ADT and BinaryTree implementation