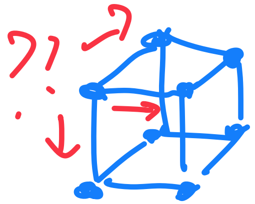
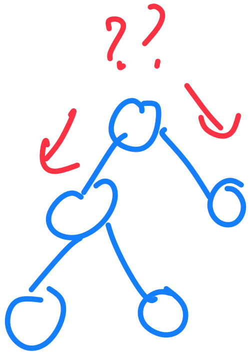
Data Structures

Iterators and Tree Fundamentals

CS 225
Brad Solomon

Friday :) September 13, 2024

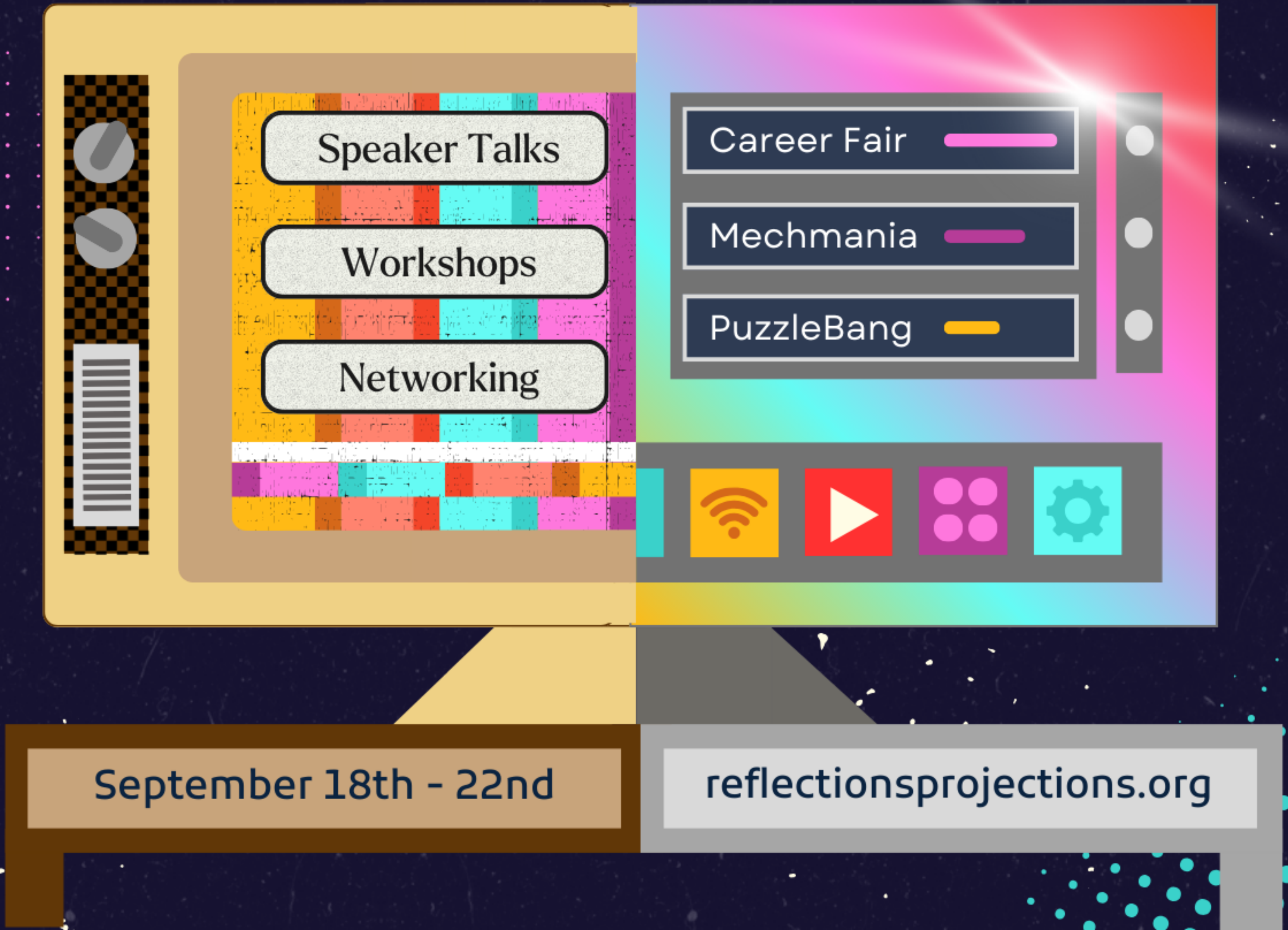




UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

reflections | projections



Learning Objectives

Queues?

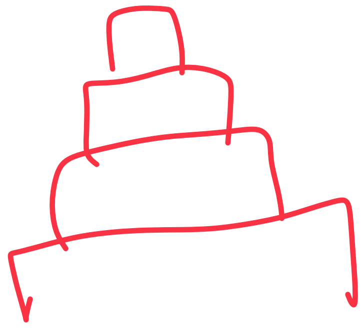
Discuss the importance of iterators

Review trees and binary trees

Practice tree theory with recursive definitions and proofs

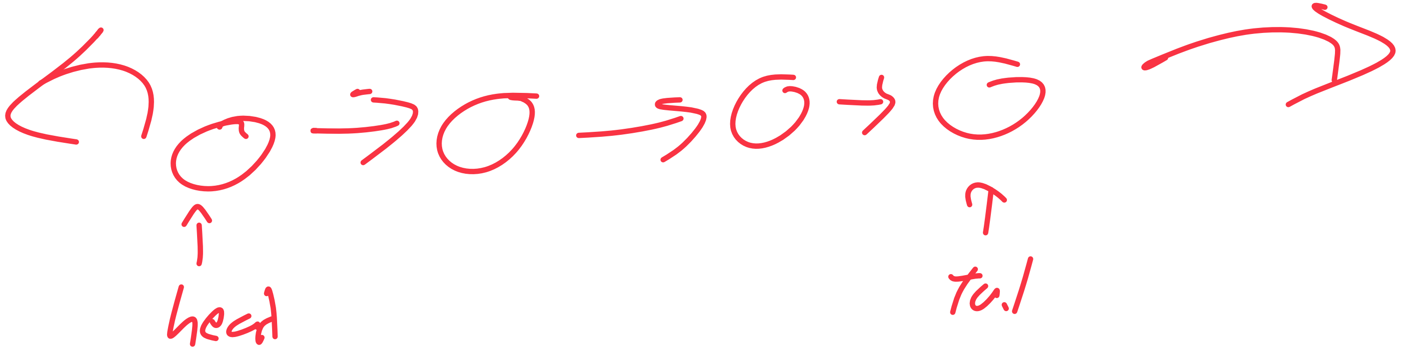
Discuss the tree ADT

Stack ADT

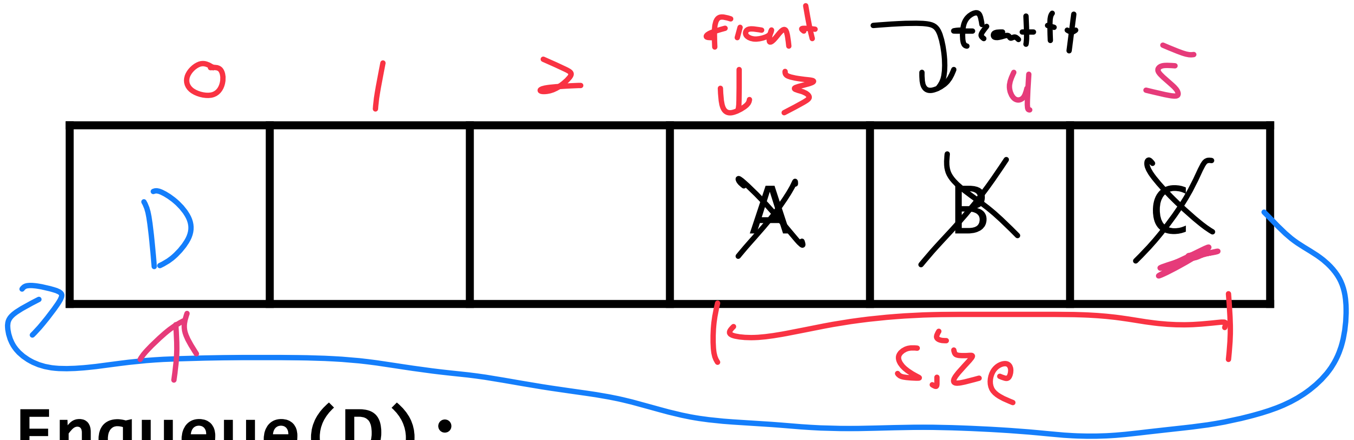


- [Order]: LIFO
- [Implementation]: Array (such as std::vector)
- [Runtime]: O(1) Push and Pop

Queue ADT



- [Order]: FIFO
- [Implementation]: Circular Queue as Array
- [Runtime]: O(1)



Enqueue(D):

Insert D at index **(size+front) % capacity**

$(3 + 3) \% 6 = 0$

size++

Dequeue(): Remove data at index front

front = **(front+1) % capacity**

$(5+1) \%$ capacity

size--

size++ until size == capacity

Queue<int> q;
...
q.enqueue(D);
q.dequeue();
q.dequeue();
q.dequeue();
q.dequeue();
q.enqueue(E);

Size: ~~3~~ ~~4~~ ~~3~~ ~~2~~ 1

Front: ~~3~~ ~~4~~ ~~5~~ ~~6~~ 0 1

unsigned ints

Capacity: 6

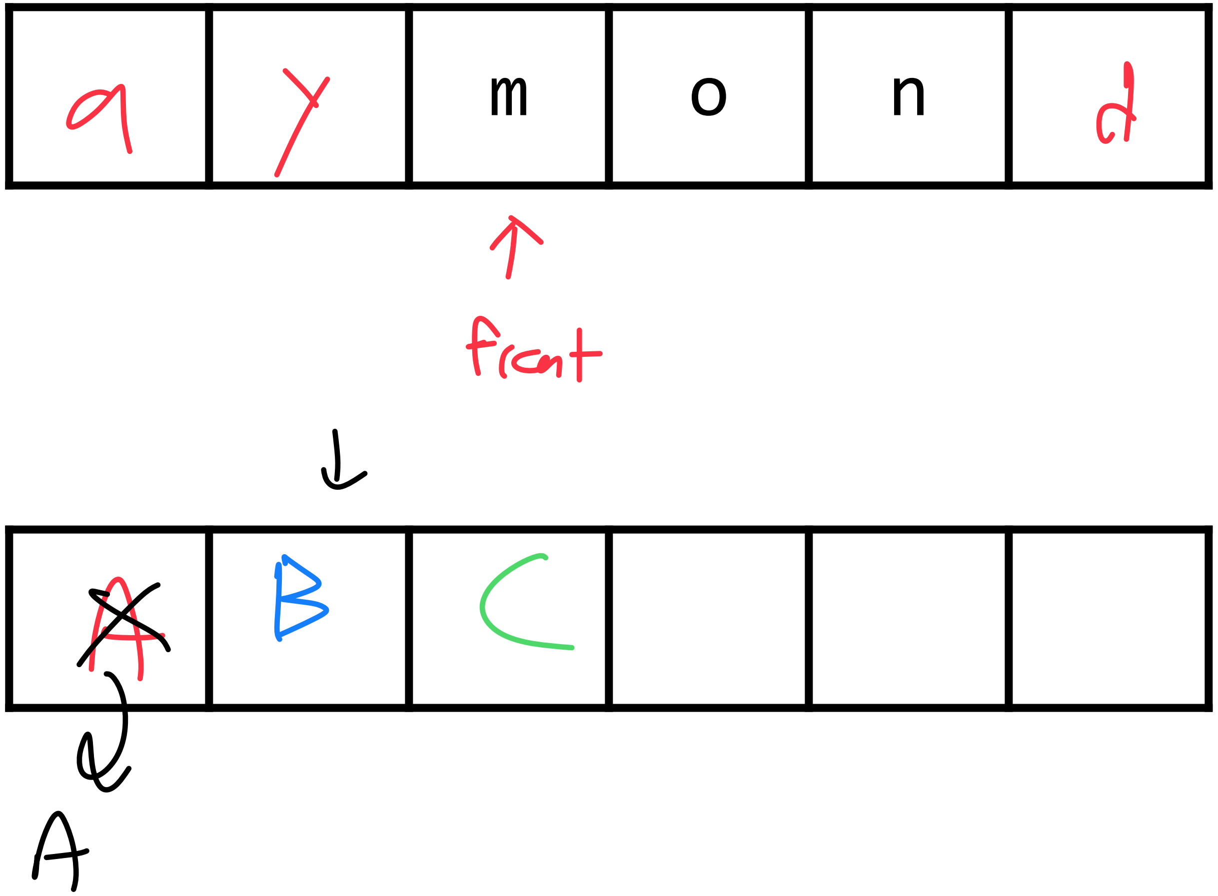
Queue Data Structure: Resizing

		m	o	n	

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

Queue Data Structure: Resizing

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



front: ~~0~~ 1

size: ~~0~~ ~~1~~ ~~2~~ 3

~~3~~ A
~~2~~ B
~~1~~ C

dequeue()

Queue Data Structure: Resizing

Queue<char> q;

...

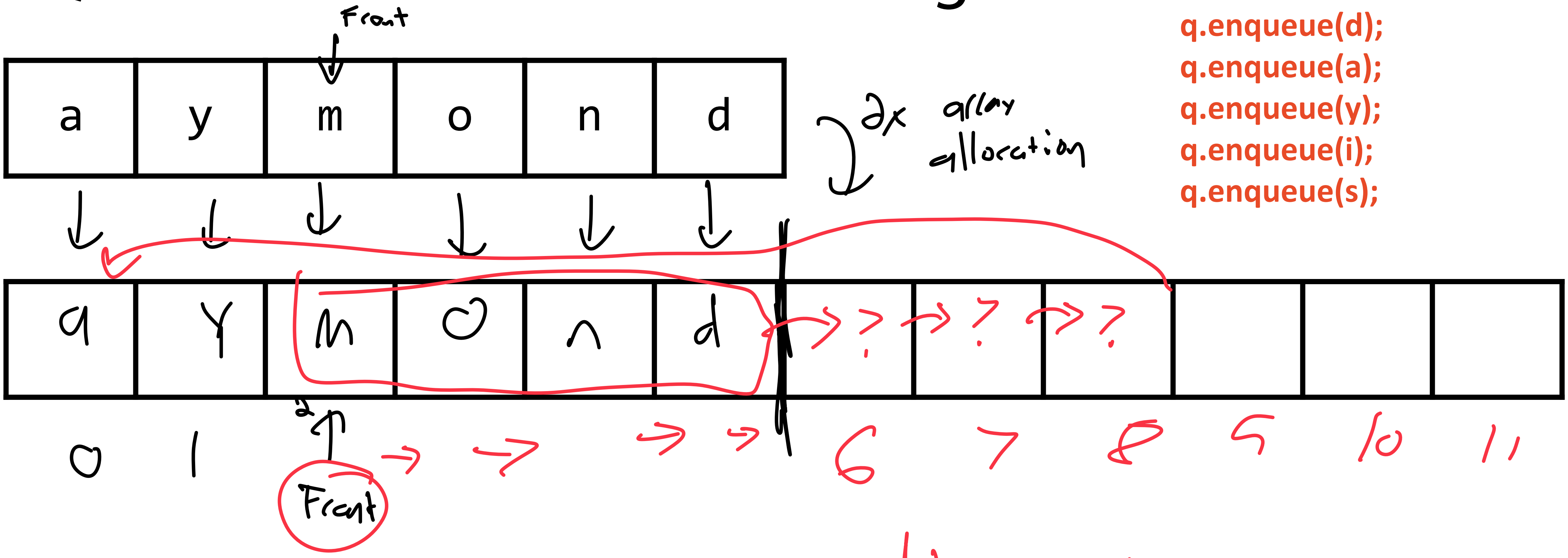
q.enqueue(d);

q.enqueue(a);

q.enqueue(y);

q.enqueue(i);

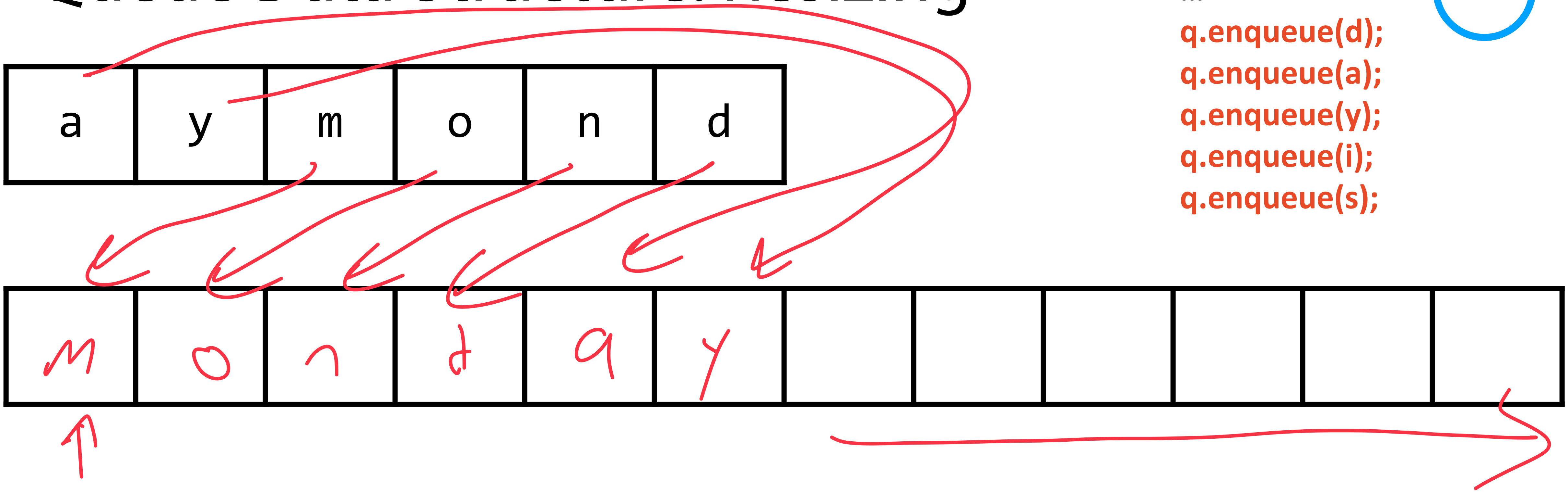
q.enqueue(s);



Broken circular queue! Not this!

Queue Data Structure: Resizing

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

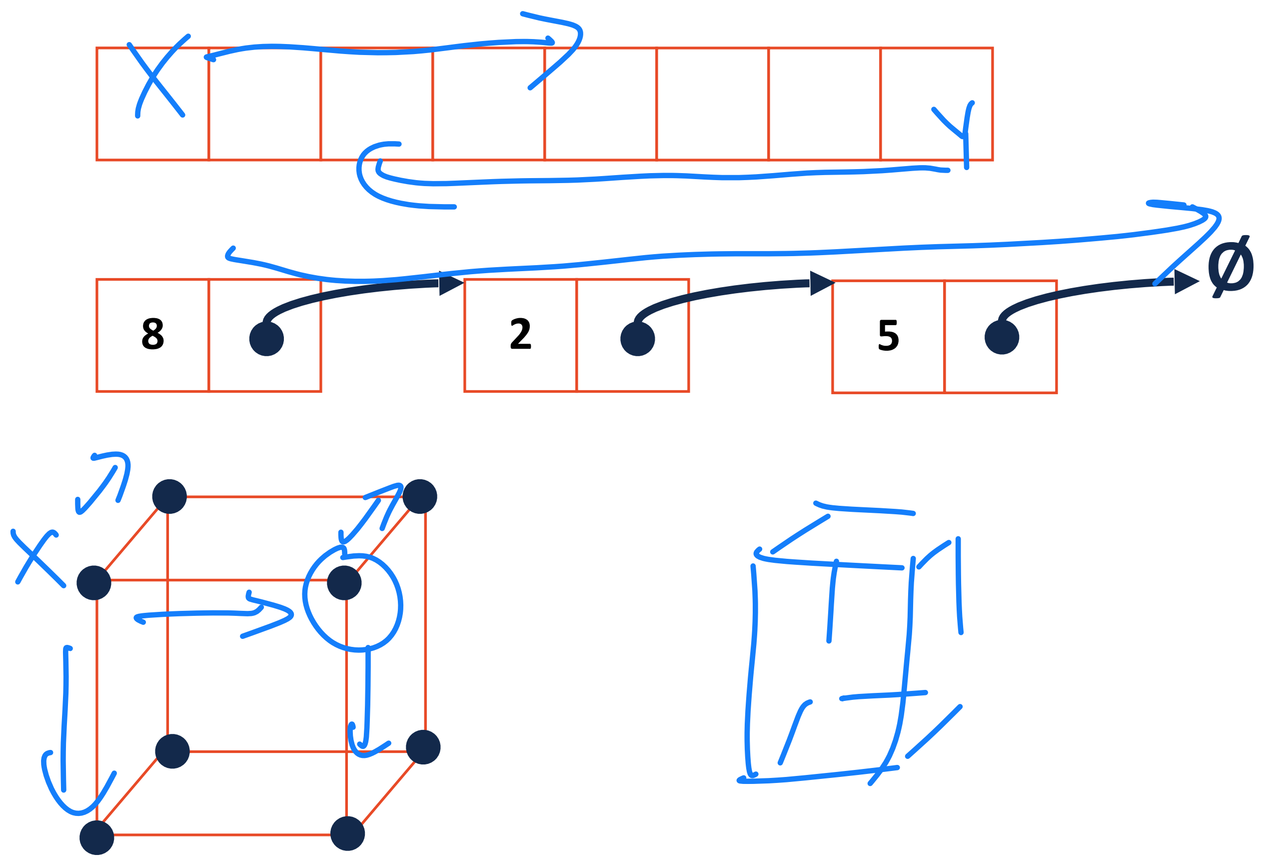


Front = 0

$O(1)^*$ amortized time

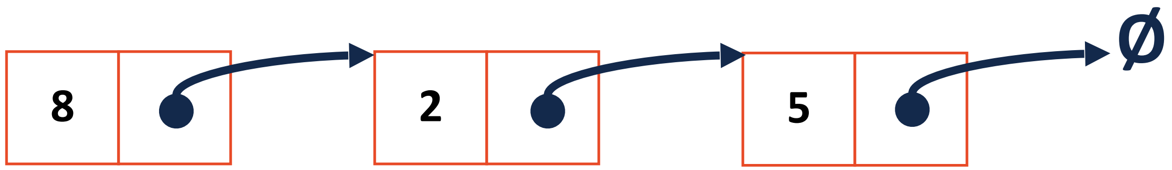
Iterators → mp_lists

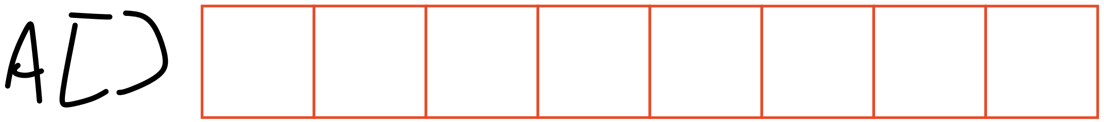
We want to be able to loop through all elements for any underlying implementation in a systematic way

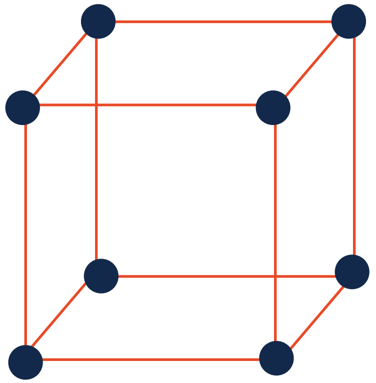


Iterators

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





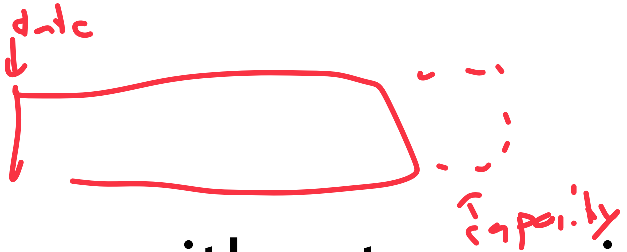


*it

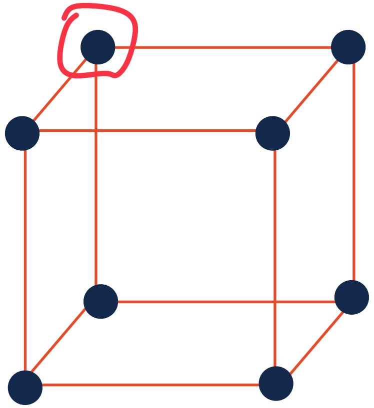
Cur. Location	Cur. Data	Next
ListNode * curr	curr->data	curr->next
unsigned index	A[i] getData(i)	index++
Some form of (x, y, z)	??	??

Iterators

Cube myCube



Iterators provide a way to access items in a container without exposing the underlying structure of the container



```
Cube::Iterator it = myCube.begin();

while (it != myCube.end()) {
     std::cout << *it << " ";
     it++;
}
```

(iterator) != (iterator)

*it ← de reference

Iterators

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

`Iterator begin()` – return type iterator

↳ points to start position

`Iterator end()` ← return type iterator

↳ points to one mem address past the end of class

Iterators

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

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

2. It must implement at least the following operations:

`Iterator& operator ++()` – move to next item

`const T & operator *()` ← return the data/value at current pos

`bool operator !=(const Iterator &)` ← check if iterators are equal

Iterators

i++
use it at curr & then increment

Here is a (truncated) example of an iterator:

```
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;
};
```

Pre increment

++it
increment first and then use new value

Implementing this gives iterator access

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;
}
```

Animal

stlList.cpp

A = [ ]

A[i++] vs A[++i)

```
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 << (*it).name << " " << (*it).food << std::endl;
  }

/* For Each Snippet */

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

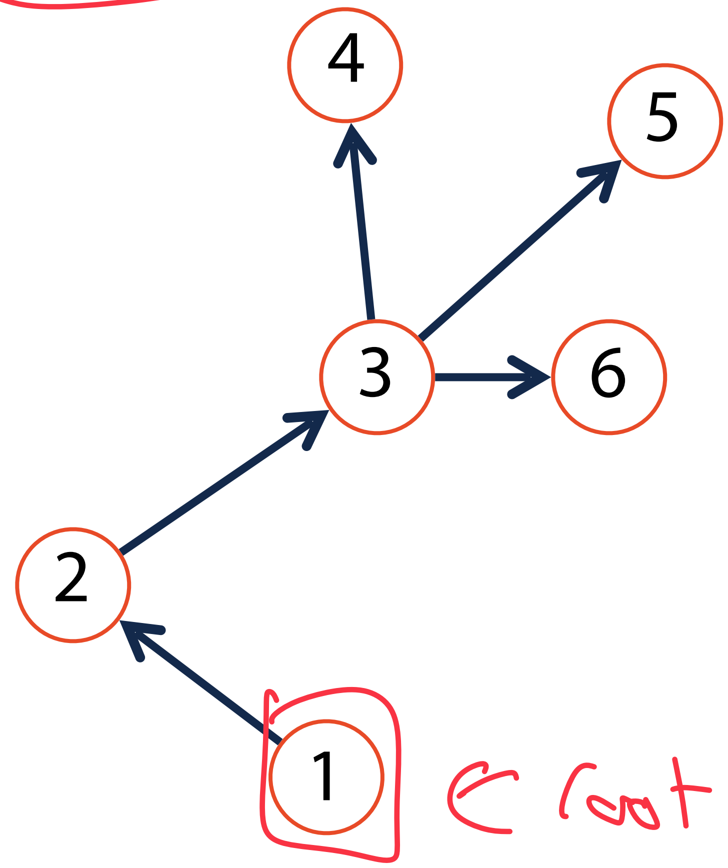
Trees

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

[In CS 225] a tree is also:

1) Acyclic — No path from node to itself

2) Rooted — A specific node is labeled root

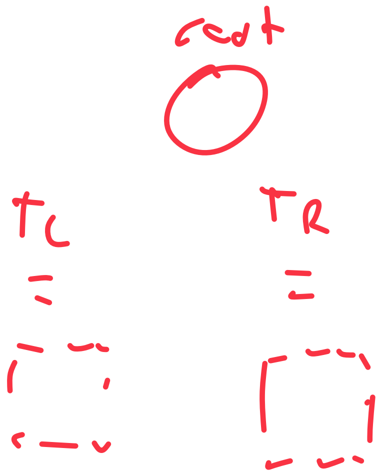


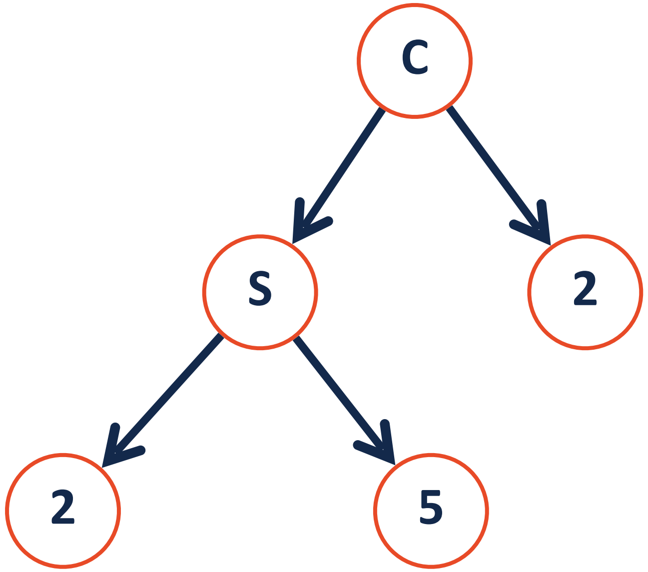
Binary Tree

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

1. $T = \emptyset$

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

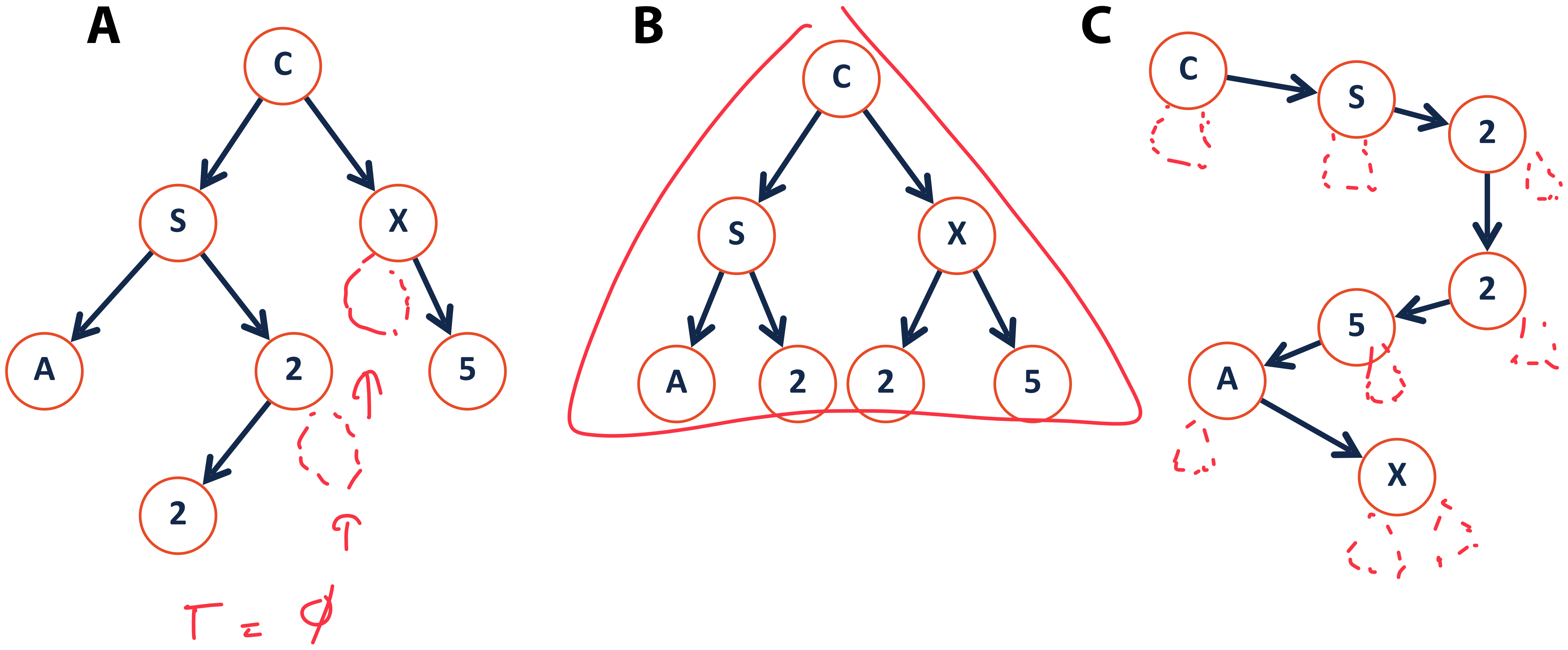




Which of the following are binary trees?

75% say "All"
20% "A+B"

Join Code: 225



Binary Tree

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

1.

On Monday!

↳ Started just last slide

(Tree ADT brainstorm)

2.

3.

Binary Tree: full

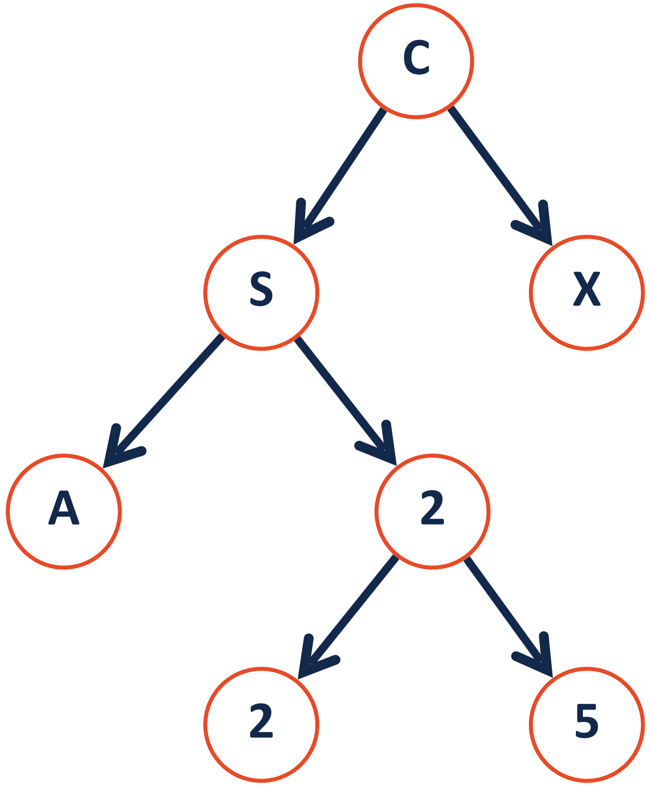
A **full tree** is a binary tree where every node has either 0 or 2 children

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

1.

2.

3.



Binary Tree: full

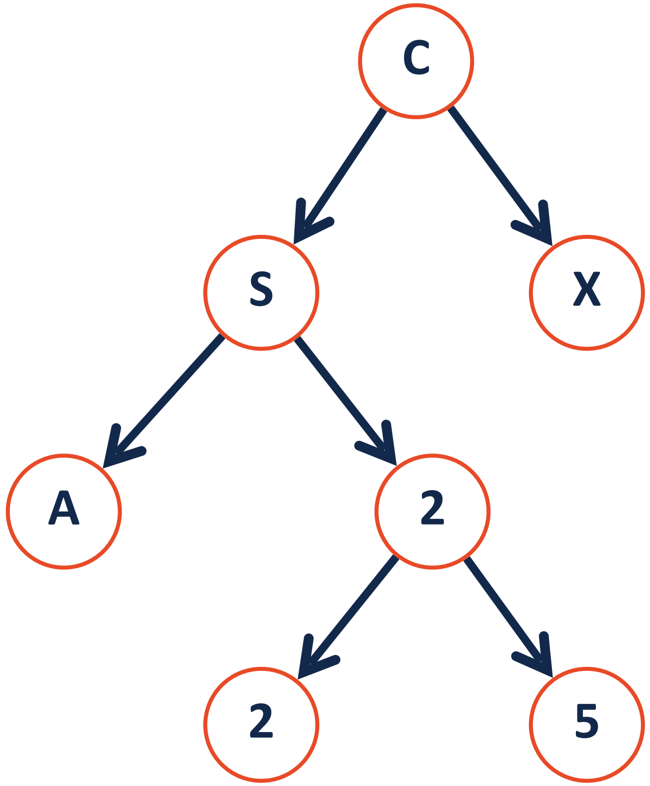
A **full tree** is a binary tree where every node has either 0 or 2 children

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

1. $F = \emptyset$

2. $F = (data, \emptyset, \emptyset)$

3. $F = (data, F_l \neq \emptyset, F_r \neq \emptyset)$



Binary Tree: perfect

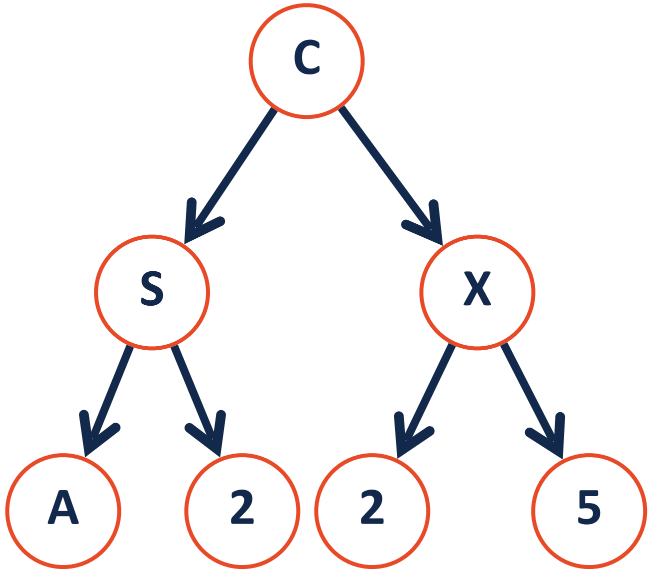
A **perfect tree** is a binary tree where…

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: perfect

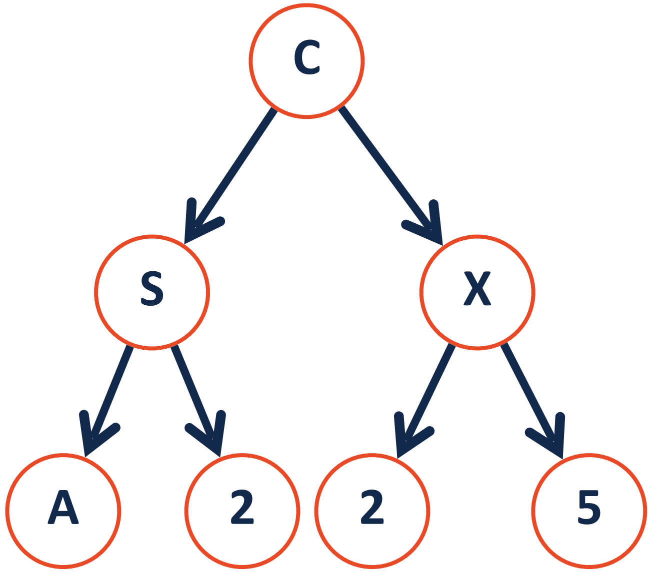
A **perfect tree** is a binary tree where…

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

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

1. $P_h = (data, P_{h-1}, P_{h-1})$

2. $P_0 = (data, \emptyset, \emptyset) \equiv P_{-1} = \emptyset$



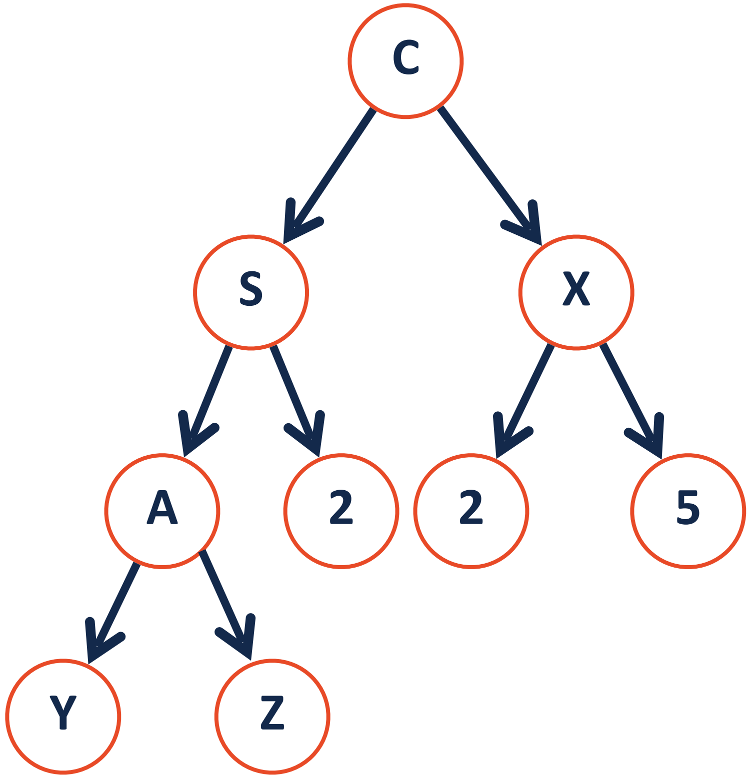
Binary Tree: complete

A **complete tree** is a B.T. where…

All levels except the last are completely filled.

The last level contains at least one node (and is pushed to left)

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



1.

2.

3.

Binary Tree: complete

A **complete tree** is a B.T. where…

All levels except the last are completely filled.

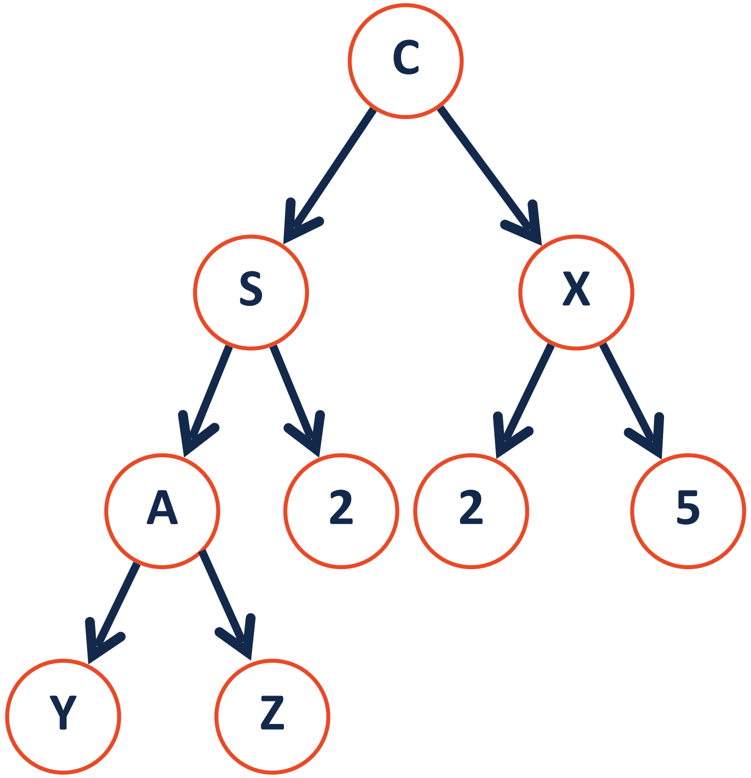
The last level contains at least one node (and is pushed to left)

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

1. $C_h = (data, C_{h-1}, P_{h-2})$

2. $C_h = (data, P_{h-1}, C_{h-1})$

3. $C_{-1} = \emptyset$



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**?

Binary Tree: Practicing Proofs

Theorem: If there are **n** objects in our representation of a binary tree, then there are _________ NULL pointers.

Binary Tree: Practicing Proofs

Theorem: If there are **n** objects in our representation of a binary tree, then there are **n+1** NULL pointers.

Base Case:

Binary Tree: Practicing Proofs

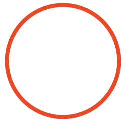
Theorem: If there are **n** objects in our representation of a binary tree, then there are **n+1** NULL pointers.

Base Case:

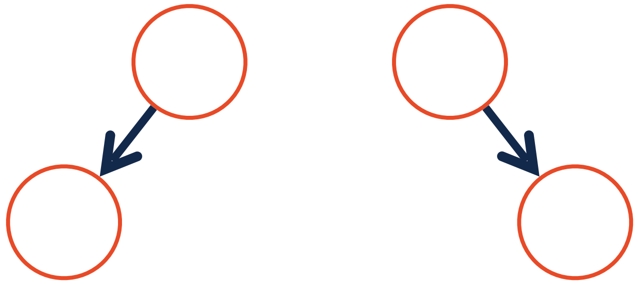
Let F(n) be the max number of NULL pointers in a tree of n nodes

N=0 has one NULL

N=1 has two NULL



N=2 has three NULL



Theorem: If there are **n** objects in our representation of a binary tree, then there are **n+1** NULL pointers.

Induction Step:

Theorem: If there are **n** objects in our representation of a binary tree, then there are **n+1** NULL pointers.

IS: Assume claim is true for $|T| \leq k-1$**, prove true for** $|T| = k$

By def, $T = r, T_L, T_R$. Let q be the # of nodes in T_L

Since r exists, $0 \leq q \leq k-1$. By IH, T_L has $q+1$ NULL

All nodes not in r or T_L exist in T_R. So T_R has $k-q-1$ nodes

$k-q-1$ is also smaller than k so by IH, T_R has $k-q$ NULL

Total number of NULL is the sum of T_L and T_R: $q+1+k-q = k+1$

Tree ADT

Tree Nodes

↳ get child(i)

↳ set Data()

↳ ?? insertLeft/Right/i ?

Tree

↳ height

↳ root

↳ insert (index)