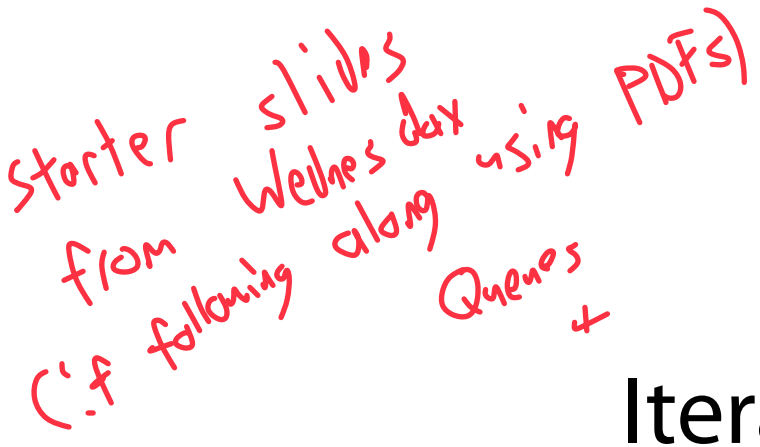


Data Structures

Iterators & Exam 1 Review

CS 225

Brad Solomon

September 11, 2026

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Announcements → Schedule

Reminder: Labs are due on Sunday!

MP is due on Monday (late day Tuesday for 93%)

Next MP (MP_lists, focus on iterators and lists) out Monday

Exam 1 (9/16 — 9/18)

Autograded MC and one coding question

Manually graded short answer prompt

Practice exam ~~will be released~~ out since Wednesday on PL

Topics covered can be found on website

Register now

https://courses.engr.illinois.edu/cs225/exams/

Learning Objectives

Introduce

Finish discussing queue implementation details

Discuss the importance of iterators

Review content seen on exam 1 (and how to prepare)

Introduce trees and the tree ADT (Time permitting)

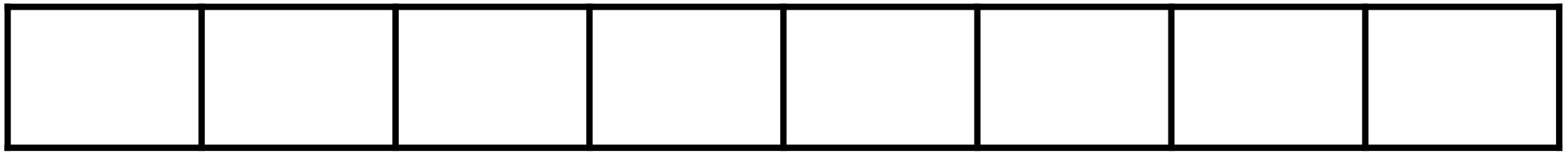
Stack and Queue

Interface Implement

Stack as array

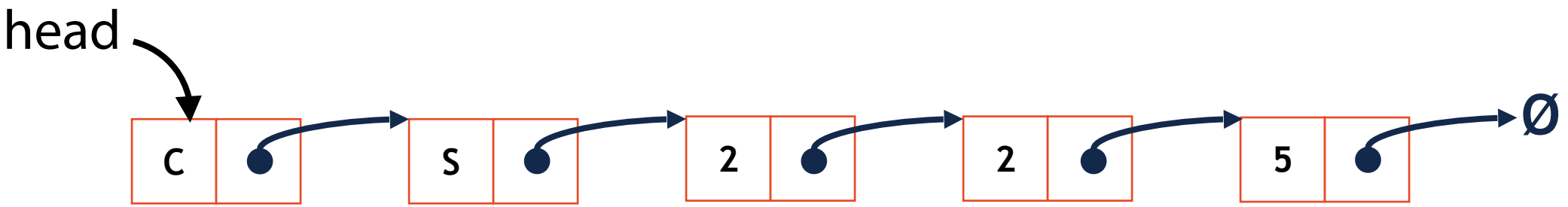
Taking advantage of special cases in lists / arrays

insertBack O(1)



insertFront O(1)

Stack as linked list



Stack ADT

- [Order]: LIFO (Last in first out)

- [Implementation]: Array (such as std::vector)

Linked List also works using insert / remove Front

- [Runtime]: O(1) Push, Pop, Top

If using array, O(1)* if we need to resize.

Queue Data Structure

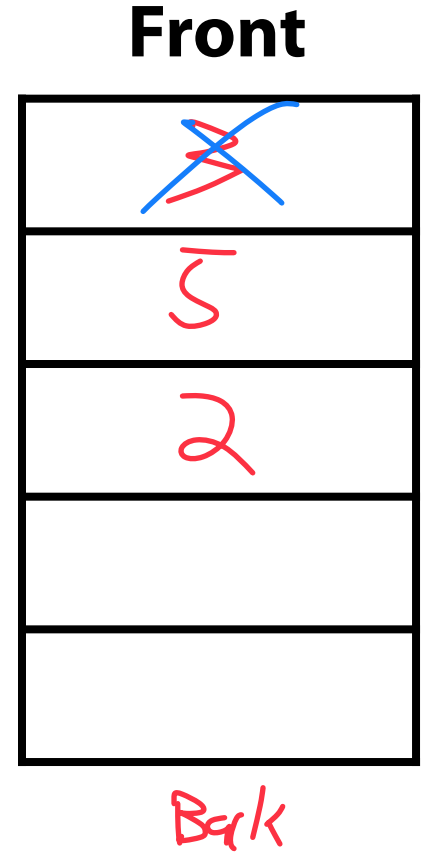
A **queue** stores an ordered collection of objects (like a list)

However you can only do three* operations:

Enqueue: Put an item at the back of the queue

Dequeue: Remove the front item of the queue

Front: Return the front item of the queue



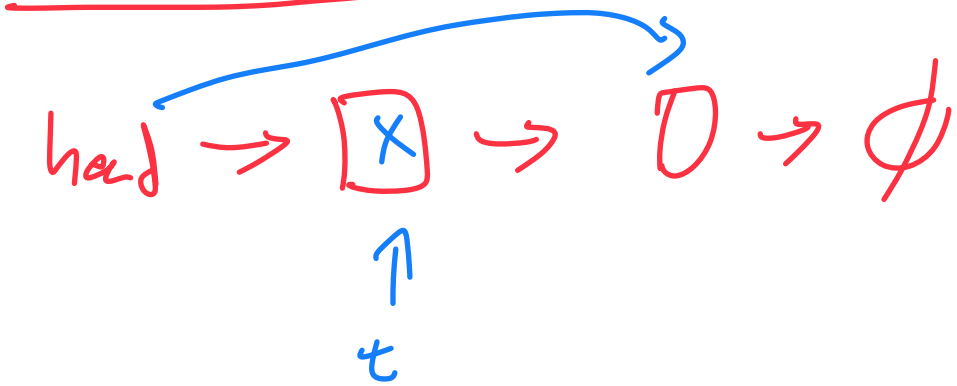

```
enqueue(3); enqueue(5); dequeue(); enqueue(2)
```

Queue Data Structure

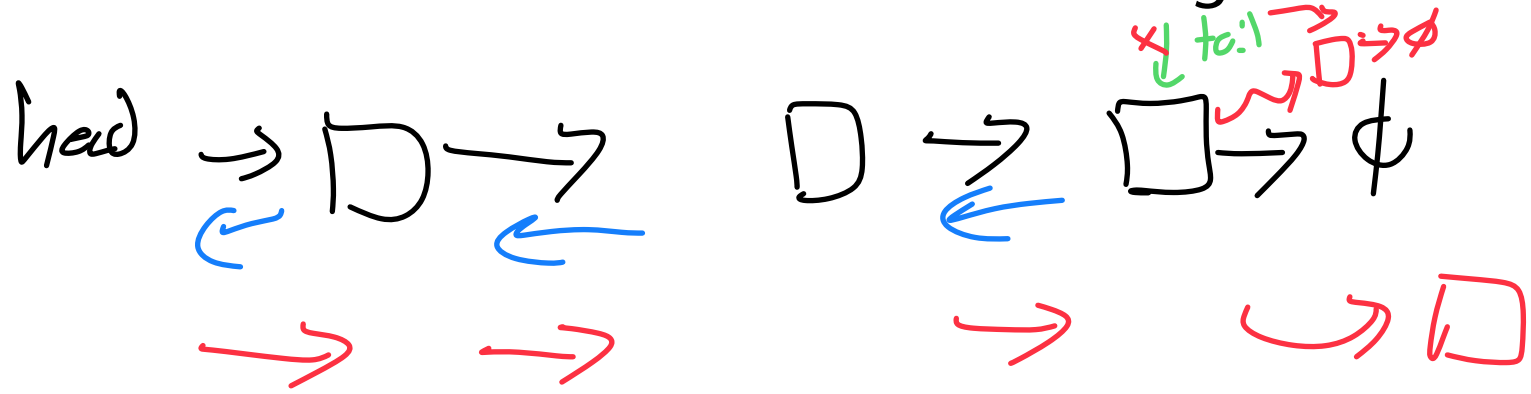
The queue is a **first in — first out** data structure (FIFO)

What data structure excels at removing from the front?

Linked List



Can we make that same data structure good at inserting at the end?



Doubly linked list does not give O(1) insertBack

tail pointer makes O(1) insertBack

Queue Data Structure

The C++ implementation of a queue is also a vector or deque — why?

1) Engineering
   ↳ the time to follow pointers is not free

2) We can improve the array (we have the tech....)

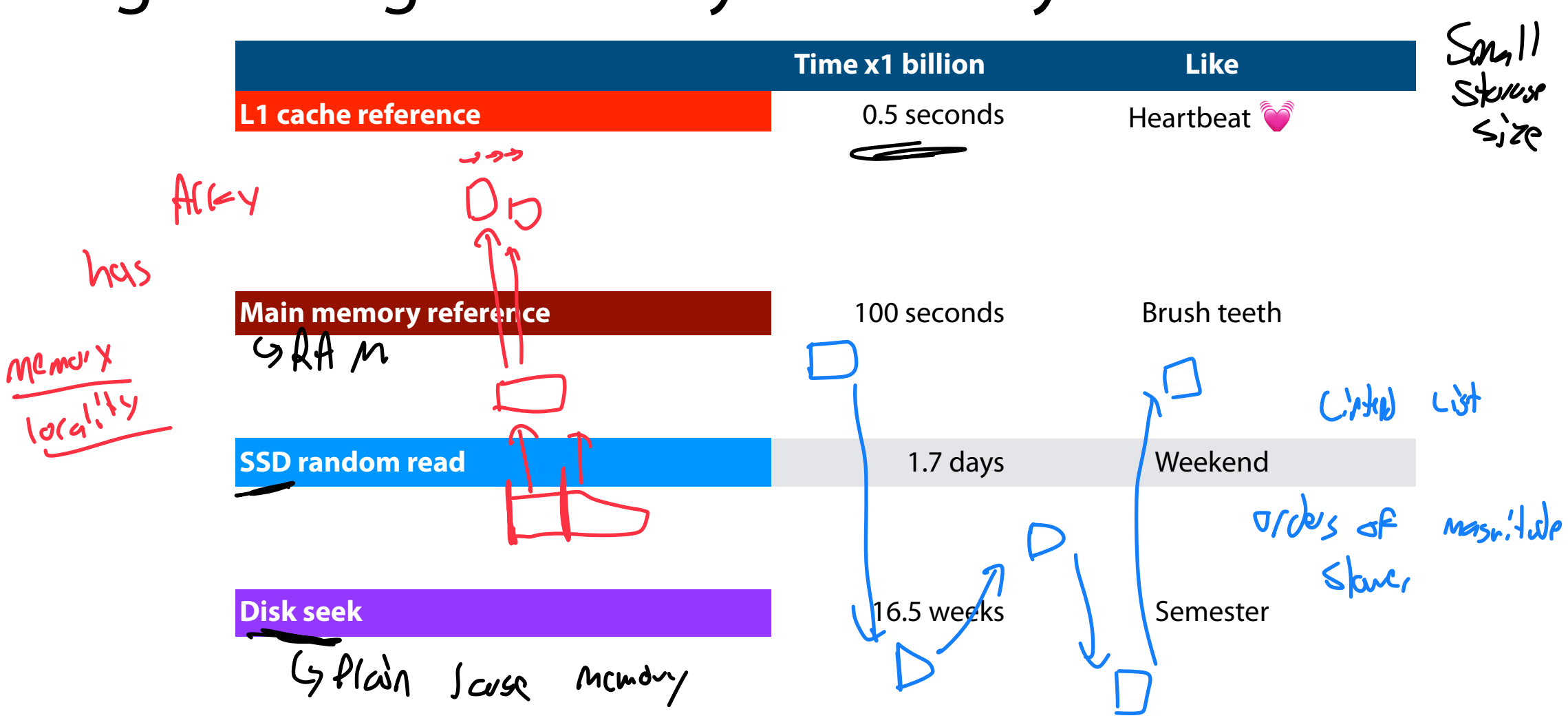
Engineering vs Theory Efficiency

	Time x1 billion	Like
L1 cache reference	0.5 seconds	Heartbeat 💓
Branch mispredict	5 seconds	Yawn 😯
L2 cache reference	7 seconds	Long yawn 😯😯😯
Mutex lock/unlock	25 seconds	Make coffee ☕
Main memory reference	100 seconds	Brush teeth
Compress 1K bytes	50 minutes	TV show 📺
Send 2K bytes over 1 Gbps network	5.5 hours	(Brief) Night's sleep 🛌
SSD random read	1.7 days	Weekend
Read 1 MB sequentially from memory	2.9 days	Long weekend
Read 1 MB sequentially from SSD	11.6 days	2 weeks for delivery 📦
Disk seek	16.5 weeks	Semester
Read 1 MB sequentially from disk	7.8 months	Human gestation 🐣
Above two together	1 year	🌍☀️
Send packet CA->Netherlands->CA	4.8 years	Ph.D. 🎓

(Care of https://gist.github.com/hellerbarde/2843375)

Engineering vs Theory Efficiency

	Time x1 billion	Like
L1 cache reference	0.5 seconds	Heartbeat 💗
Main memory reference	100 seconds	Brush teeth
SSD random read	1.7 days	Weekend
Disk seek	16.5 weeks	Semester
Send packet CA->Netherlands->CA	4.8 years	Ph.D. 🎓



(Care of https://gist.github.com/hellerbarde/2843375)

Queue Data Structure

```
q.enqueue(8);
q.enqueue(4);
q.dequeue();
```

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

Lets add a new variable front

↳ Data is the first allocated mem cell as front item

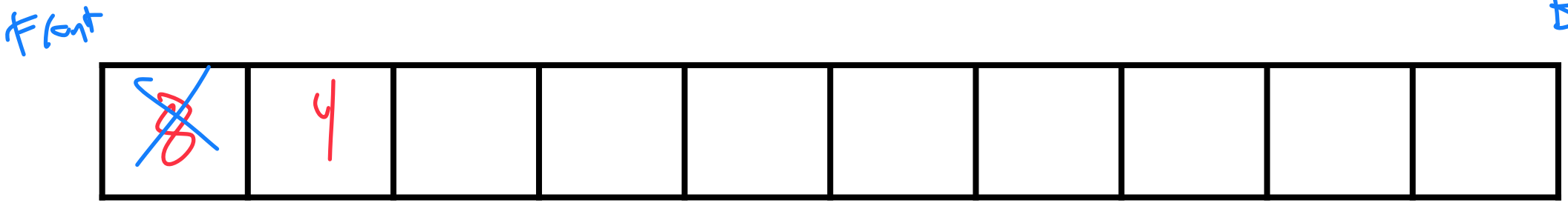
↳ No longer guranteed to be front item

Front

Data

Size

Capacity

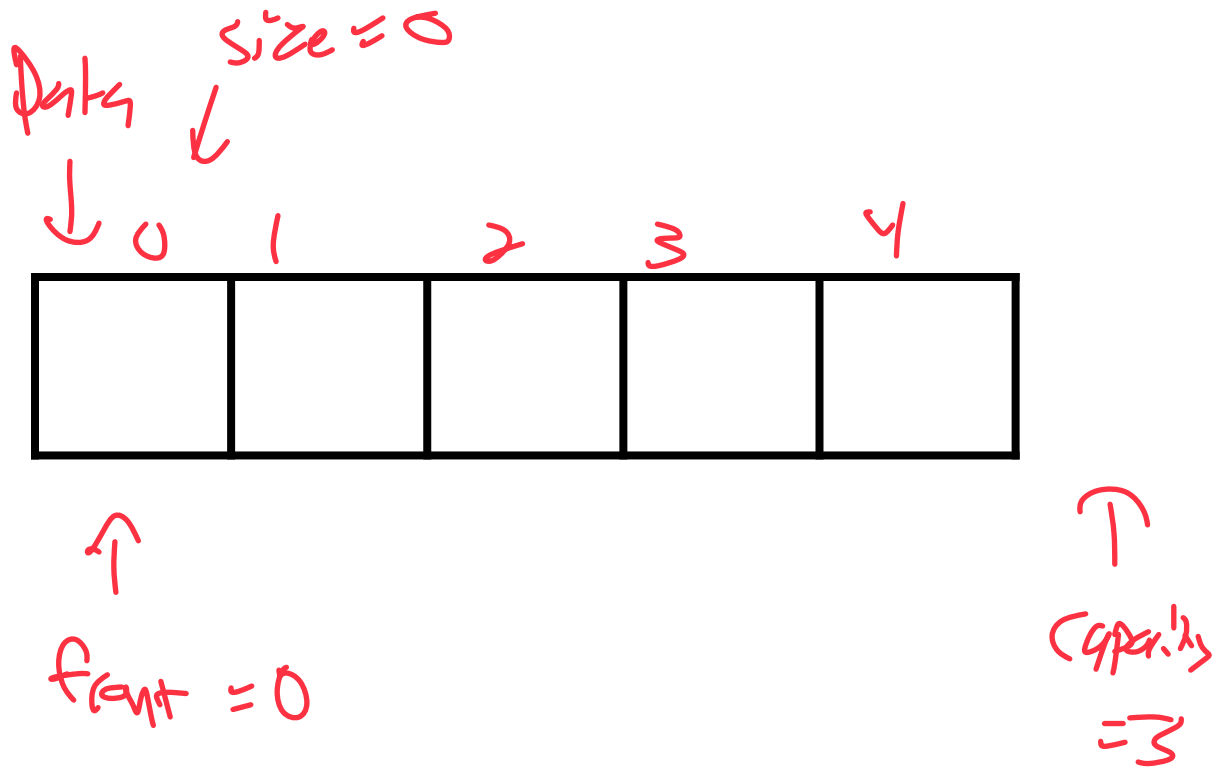


Queue Data Structure

Unlike the array list, it is easier to implement a Queue using unsigned ints

Queue.h

```
 1  #pragma once
 2
 3  template <typename T>
 4  class Queue {
 5    public:
 6      void enqueue(T e);
 7      T dequeue();
 8      bool isEmpty();
 9
10    private:
11      T *data_;
12      unsigned size_;
13      unsigned capacity_;
14      unsigned front_;
15  };
```

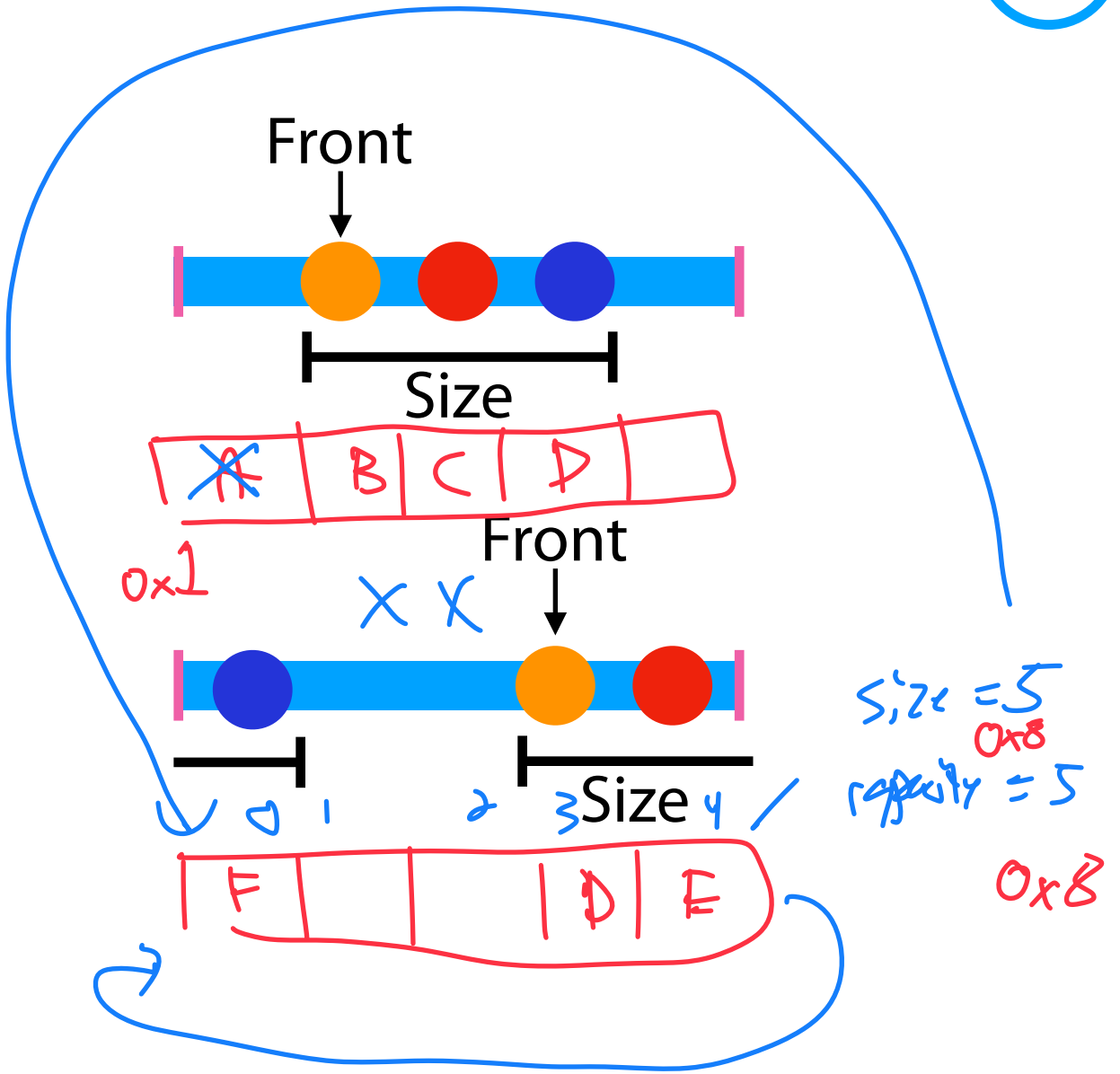


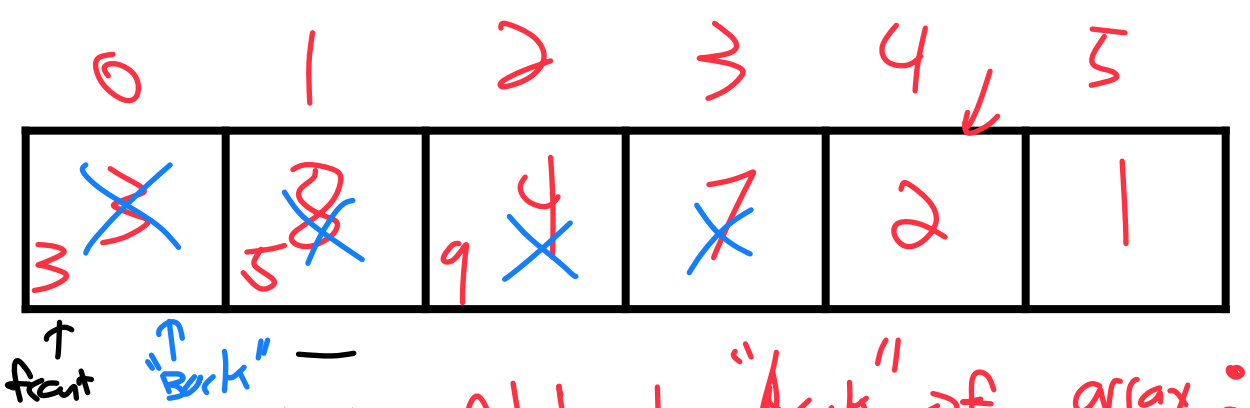
(Circular) Queue Data Structure

Queue.h

```
 1  #pragma once
 2
 3  template <typename T>
 4  class Queue {
 5    public:
 6      void enqueue(T e);
 7      T dequeue();
 8      bool isEmpty();
 9
10    private:
11      T *data_;
12      unsigned capacity_;
13      unsigned size_;
14      unsigned front_;
15  };
```

int ints





66%

F, S
0,0 –
0, 1 –
0, 2

```
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);
```

Enqueue(D): Add to "back" of array: (Front + size) % capacity

size ++

Dequeue(): Remove the front item (@ index front)

front ++ % capacity

size --

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

Front: ~~0~~ ~~1~~ ~~2~~ 3

Capacity: 6

Both actions have required checks

Enqueue(D):(as long as size != capacity)

Add data to ‘back’ of queue

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

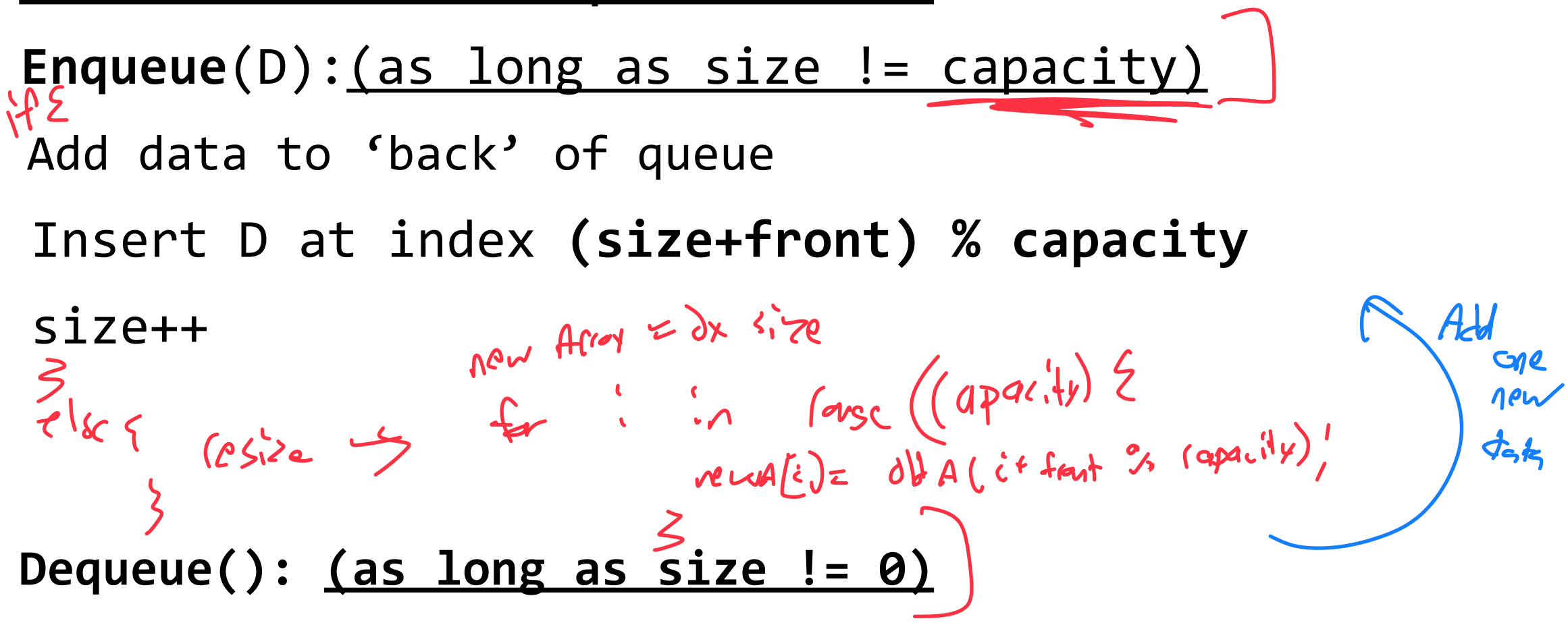
size++

Dequeue(): (as long as size != 0)

Remove data at index front

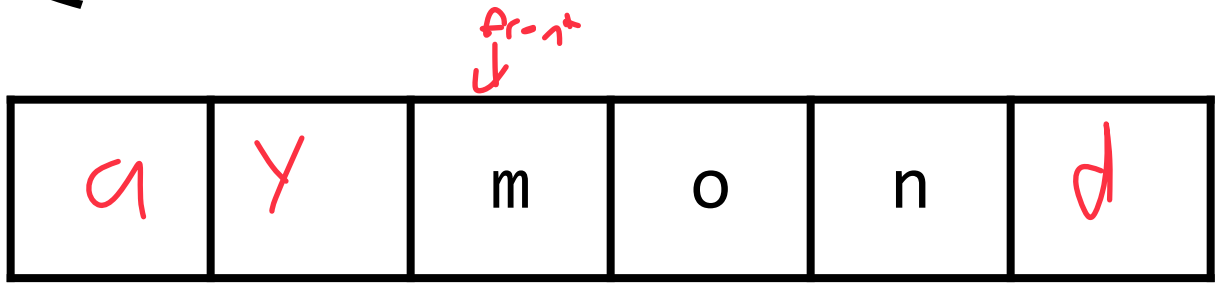
front = **(front+1) % capacity**

size--

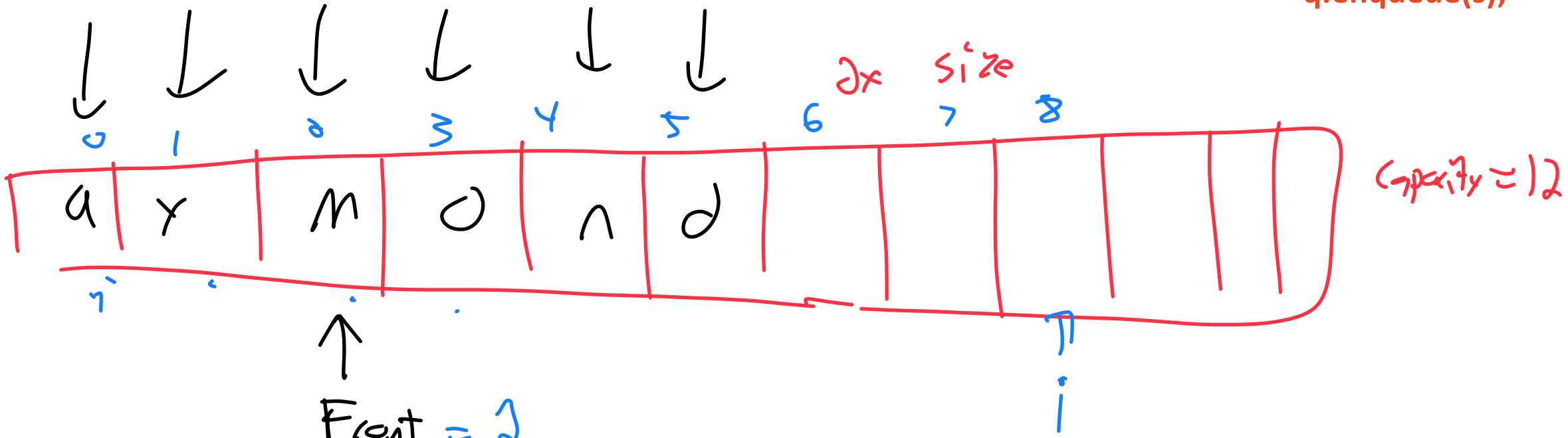


Queue Data Structure: Resizing

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



capacity = 6



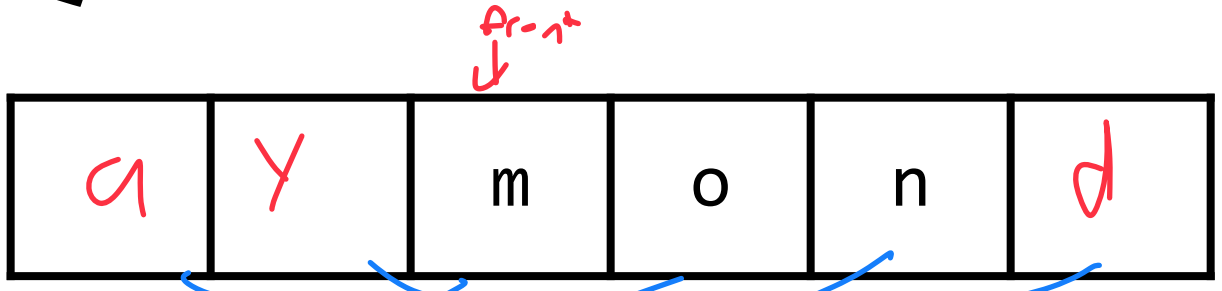
Front = 2

size = 6

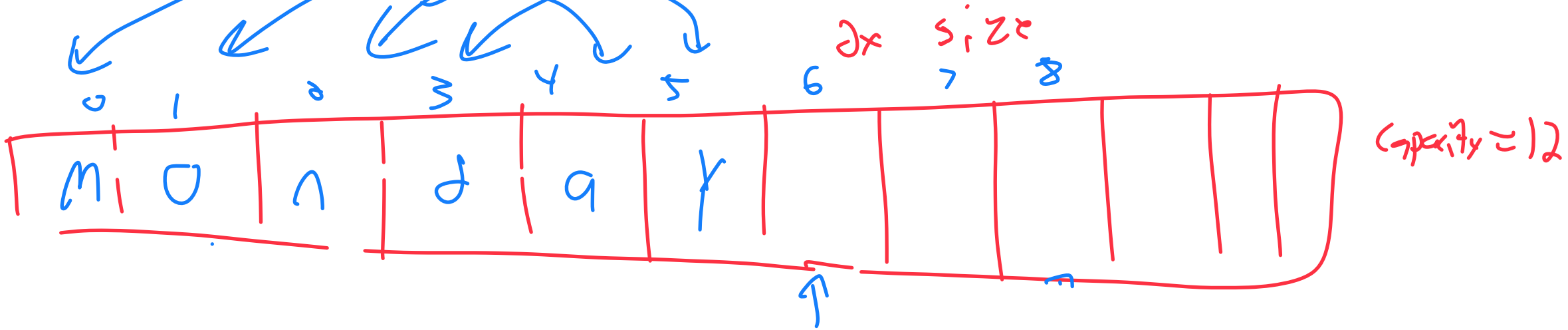
Oh no! More problems!

Queue Data Structure: Resizing

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



capacity = 6



Front = 0

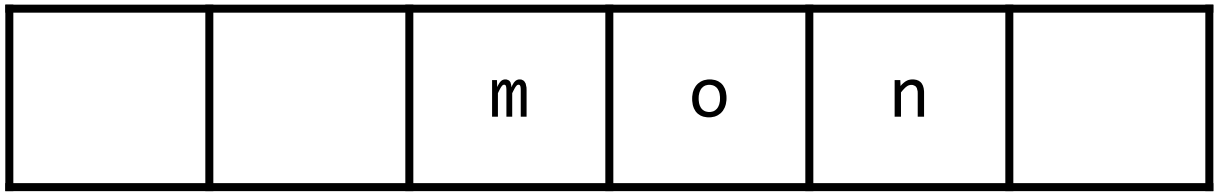
size = 6

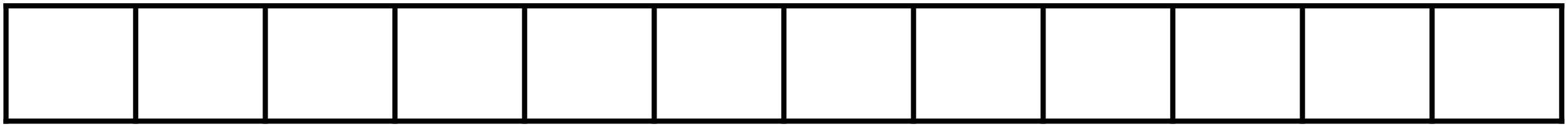
O(n) to resize (same as array)

O(1)* amortized

Queue Data Structure: Resizing

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





Queue ADT

- [Order]: First in - First out (FIFO)

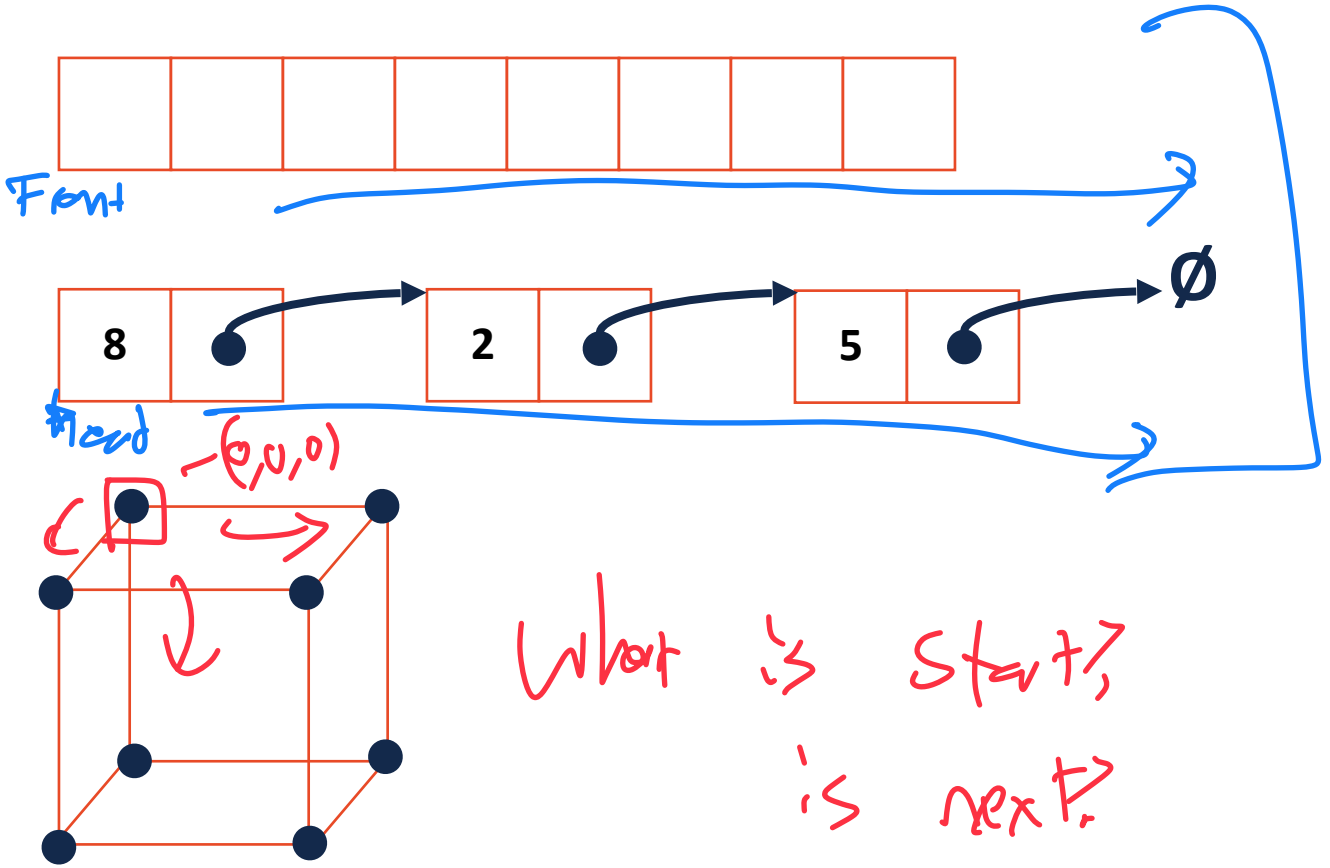
- [Implementation]: trivial as Linked List w/ tail pointer

  Array as circular queue

- [Runtime]: O(1) / O(n)* if array resize

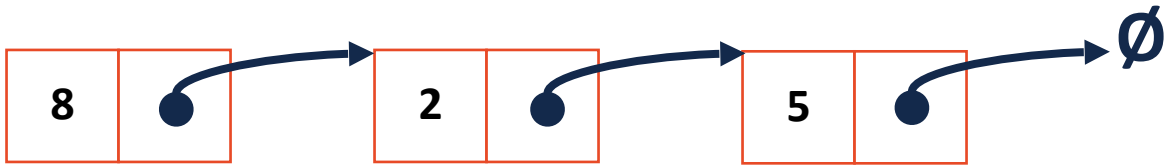
Iterators

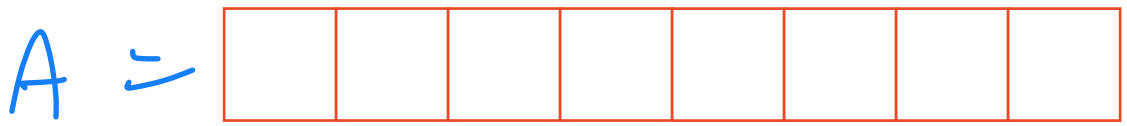
We want to be able to loop through all elements for any underlying implementation in a systematic way

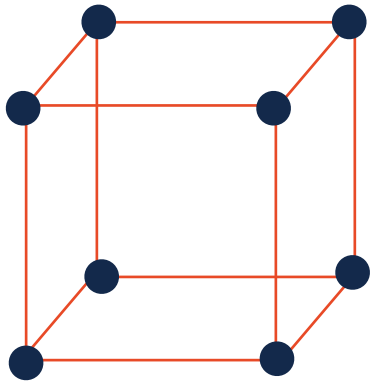


Iterators

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







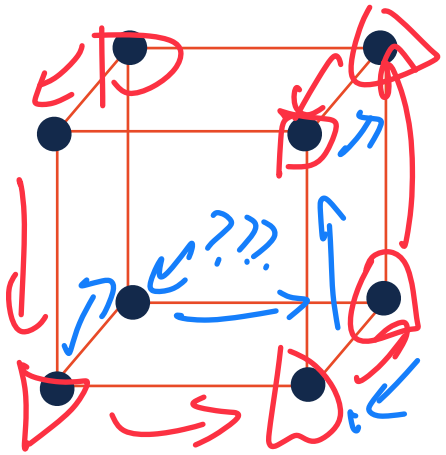
Cur. Location	Cur. Data	Next
`ListNode * curr`	curr->data	curr->next
`unsigned index`	A[i]	A[i+1]

`Some form of`
 	
`(x, y, z)`
	???	???

Iterators

Our fwset interface

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



```
1  Cube::Iterator it = myCube.begin();
2
3  while (it != myCube.end()) {
4      std::cout << *it << " ";
5      it++;
6  }
7
```

first item

last (One past end)

Get next

Get current "data"

Iterators

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

```
Iterator begin()
```

```
Iterator end()
```

Iterators

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

`Iterator begin()`

Returns an Iterator object pointing at the 'first item'

`Iterator end()`

Returns an Iterator object pointing one entry past end of dataset

Iterators

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

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

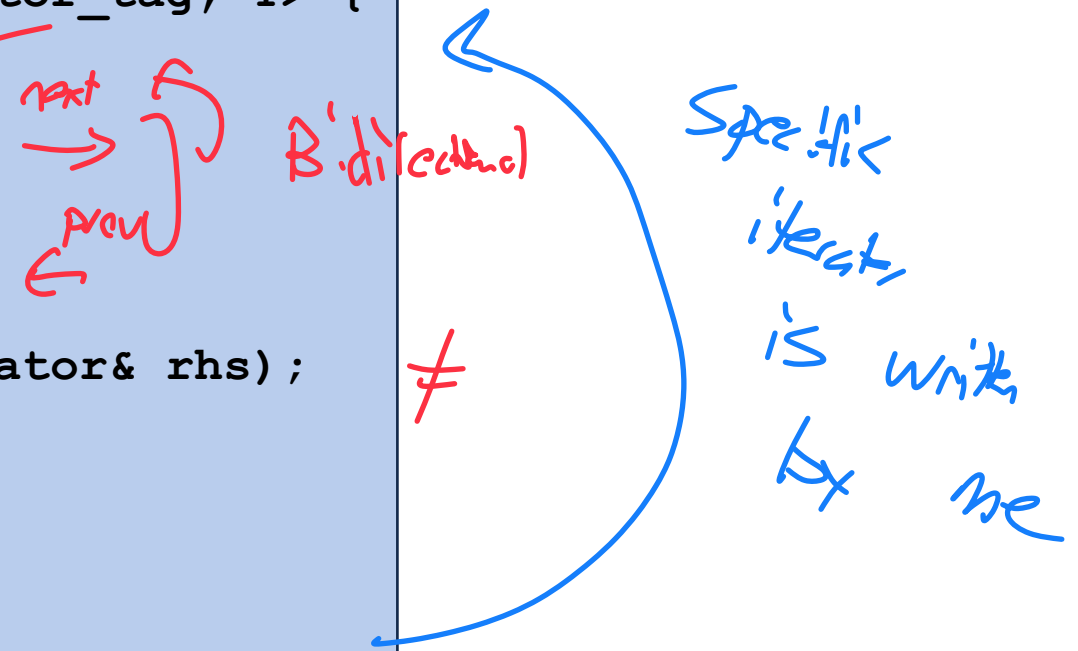
2. It must implement at least the following operations:

```
Iterator& operator ++()

const T & operator *()

bool operator !=(const Iterator &)
```

Iterators

Here is a (truncated) example of an iterator:

```
 1 template <class T>
 2 class List {
 3
 4     class ListIterator : public
   std::iterator<std::bidirectional_iterator_tag, T> {
 5       public:
 6
 7         ListIterator& operator++();
 8
 9         ListIterator& operator--()
10
11         bool operator!=(const ListIterator& rhs);
12
13         const T& operator*();
14     };
15
16     ListIterator begin() const;
17
18     ListIterator end() const;
19 };
```



MP_List Iterator

```
1  class ListIterator {
2    private:
3      // @TODO: graded in mp_lists part 1
4      ListNode* position_;

5    public:
6      …
7
8      ListIterator() : position_(NULL) { }
9      ListIterator(ListNode* x) : position_(x) { }
```

You write implementation for List in iterator

stlList.cpp

```
1  #include <list>
2  #include <string>
3  #include <iostream>
4
5  struct Animal {
6    std::string name, food;
7    bool big;
8    Animal(std::string name = "blob", std::string food = "you", bool big = true) :
9      name(name), food(food), big(big) { /* nothing */ }
10 };
11
12 int main() {
13   Animal g("giraffe", "leaves", true), p("penguin", "fish", false), b("bear");
14   std::vector<Animal> zoo;
15
16   zoo.push_back(g);
17   zoo.push_back(p);   // std::vector's insertAtEnd
18   zoo.push_back(b);
19
20   for ( std::vector<Animal>::iterator it = zoo.begin(); it != zoo.end(); ++it ) {
21     std::cout << (*it).name << " " << (*it).food << std::endl;
22   }
23
24   return 0;
25 }
```

iterator you define

call deref operator

You write what this is

```
1
2  std::vector<Animal> zoo;
3
4
5  /* Full text snippet */
6
7    for ( std::vector<Animal>::iterator it = zoo.begin(); it != zoo.end(); ++it ) {
8      std::cout << (*it).name << " " << (*it).food << std::endl;
9    }
10
11
12 /* Auto Snippet */
13
14   for ( auto it = zoo.begin(); it != zoo.end; ++it ) {
15     std::cout << (*it).name << " " << (*it).food << std::endl;
16   }
17
18 /* For Each Snippet */
19
20   for ( const Animal & animal : zoo ) {
21     std::cout << animal.name << " " << animal.food << std::endl;
22   }
23
24
25
```

For each animal in zoo

Exam 1 Review

Exam content focuses on fundamentals and lists

To prepare for MCQ:

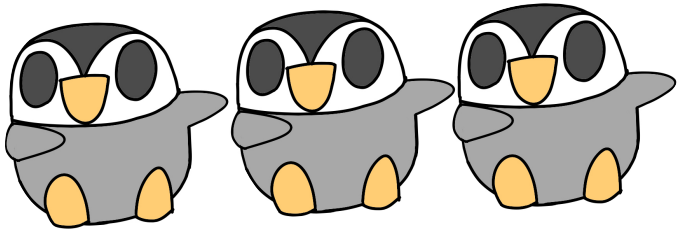
- Have a clear understanding of the list ADT

- Understand the Big O details behind linked list and array list

- Be able to apply this knowledge in new contexts

e.g. ‘What if I had a linked list and made this modification…’

Lists



The not-so-secret underlying implementation for many things

	Singly Linked List	Array
Look up **arbitrary** location	O(n)	O(1)
Insert after **given** element	O(1)	O(n)
Remove after **given** element	O(1)	O(n)
Insert at **arbitrary** location	O(n)	O(n)
Remove at **arbitrary** location	O(n)	O(n)
Search for an input **value**	O(n)	O(n)

Special Cases: insertFront insertBack (not full)

Exam 1 Review

To prepare for the free response question:

- Have a clear understanding of the list ADT

- Understand the Big O details behind linked list and array list

- Be able to apply this knowledge in new contexts

Brainstorm: What are some variations on linked lists / arrays?

FRQ Example Rubric

Points	Rubric Item	Description
2	Overall Big O is correct	The answer has the best correct Big O for the overall problem
3	Minimum necessary steps described	The explanation describes all steps needed to solve the question (even if the Big O of one or more steps is wrong).
1	Individual step Big O is correct	The explanation correctly describes the Big O of all steps. (To receive points, you must receive full credit for the minimum necessary steps)
2	Explanation of how the 'twist' affects Big O	Each problem has at least one adjustment beyond the core data structure. Your answer should explicitly state how these changed parameters / structural requirements affected (or did not affect) the runtime.
1	Answer is given in complete sentences	Even when practicing, write in full complete sentences.
1	Answer has appropriate structure	Start with a clear declaration of Big O. On a new line state the steps and their Big Os explicitly. Then start a new paragraph where you explain why. Make it easy for your graders

Exam 1 Review

Preparing for the coding question:

- Make sure you understand the fundamentals of debugging

- Review your coding assignments so far (debug / stickers)

- The coding question will be similar to that of practice exam

Brainstorm: What are some ways we could test your knowledge of PNGs / HSLAPixels in the context of arrays?

Questions?

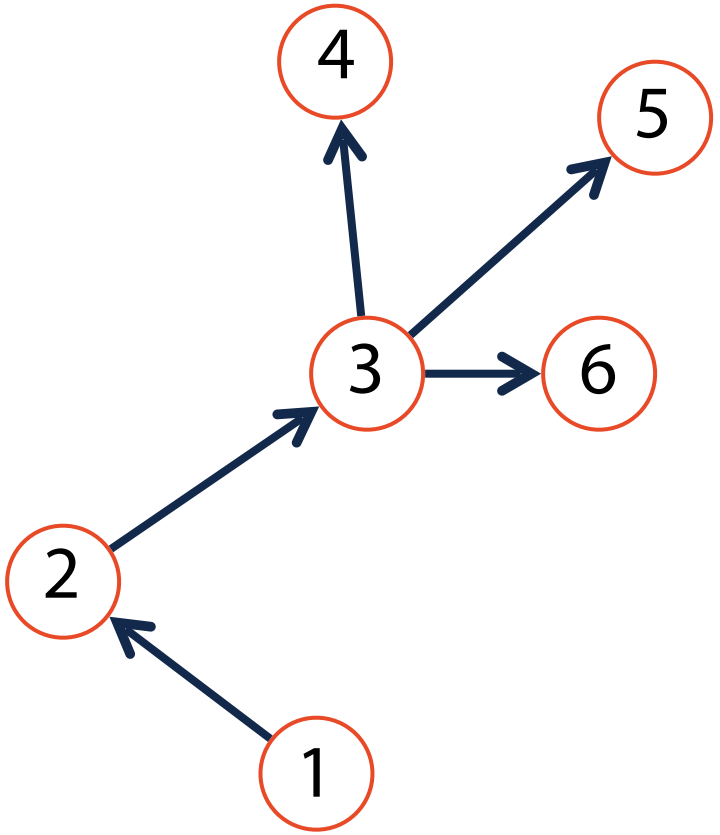
Trees

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

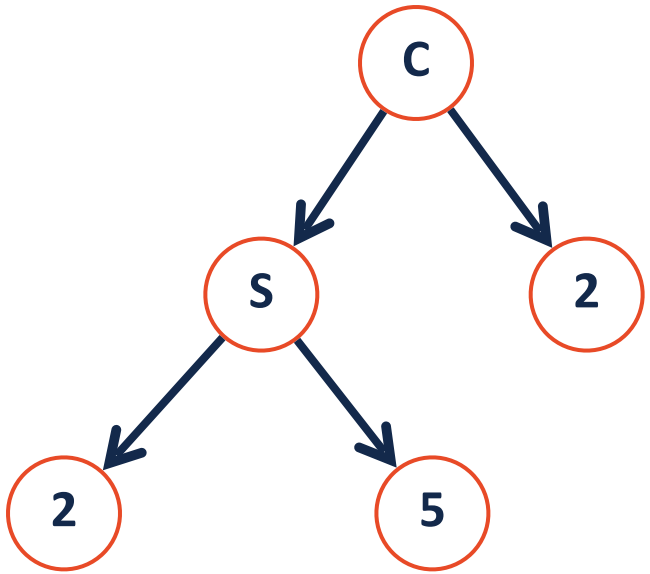


Binary Tree

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

1. $T = \emptyset$

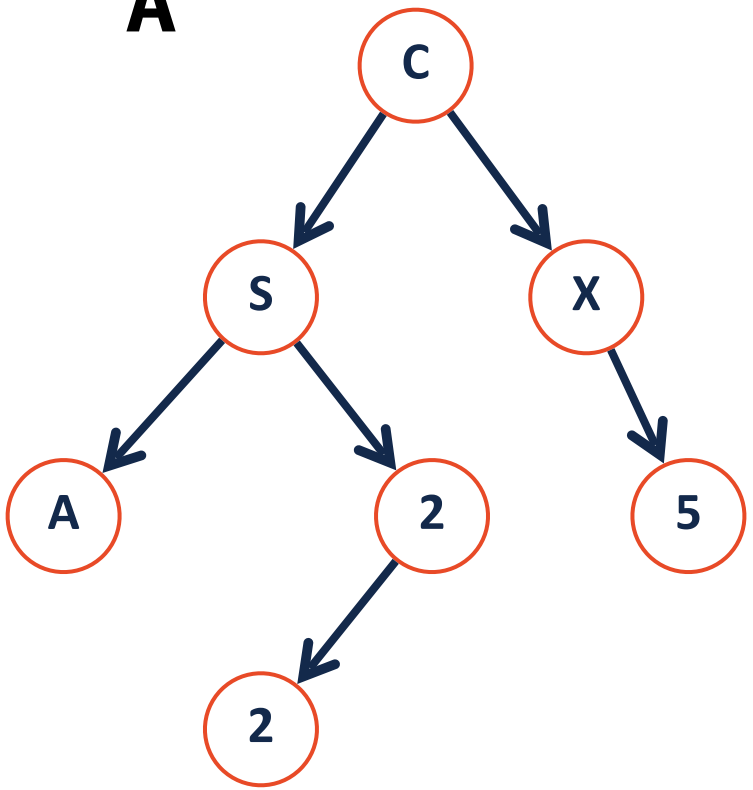
2. $T = (data, T_L, T_R)$



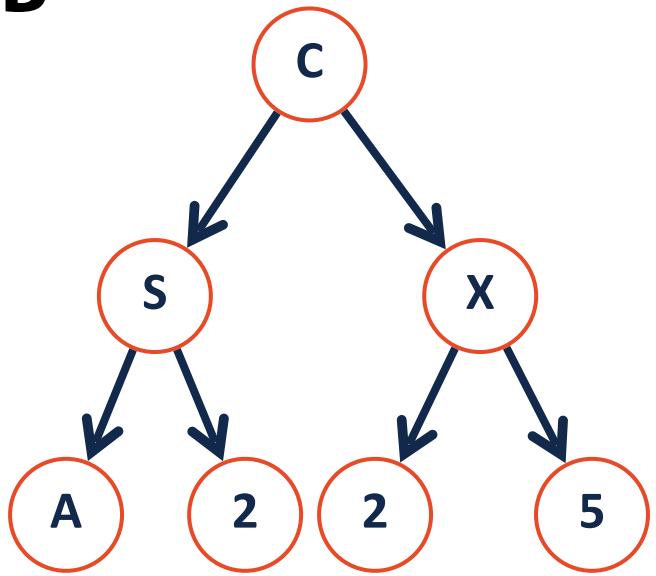
Which of the following are binary trees?

Join Code: 225

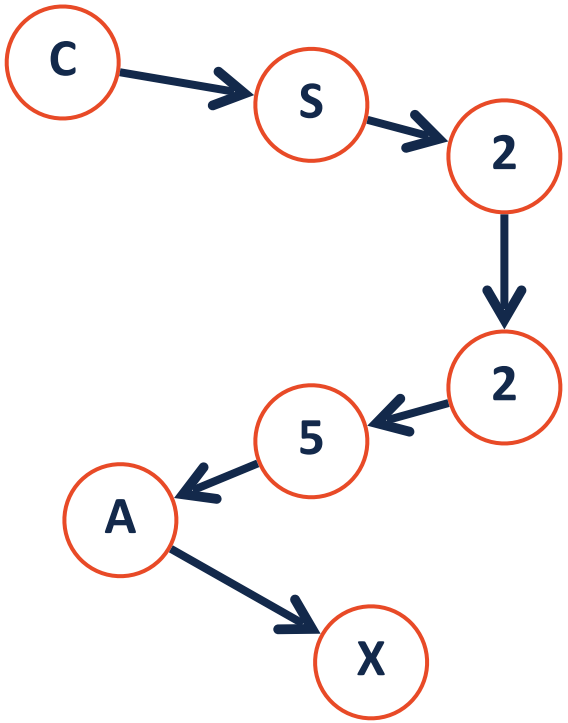
A



B



C



Tree ADT