

# Unit #0: Introduction

CPSC 221: Algorithms and Data Structures

Will Evans and Jan Manuch<sup>1</sup>

2016W1

hello  
hello

---

<sup>1</sup>Thanks to Steve Wolfman for the content of most of these slides with additional material from Alan Hu, Ed Knorr, and Kim Voll.

# Unit Outline

- ▶ Course logistics
- ▶ Course overview
- ▶ Fibonacci Fun
- ▶ Arrays
- ▶ Queues
- ▶ Stacks
- ▶ Deques

# Course Information

## Instructors

Will Evans	will@cs.ubc.ca	ICCS X841
Jan Manuch	jmanuch@cs.ubc.ca	ICCS 247

## TAs

Alexander Lim	Chris Hunter	David Zheng	Harman Gakhal
Henry Chee	Jason Zeng	Jin Ziyang	Jordan Coblin
Michael Zhang	Mike Spearman	Muzhi Ou	Nancy Chen
Oliver Zhan	Patience Shyu	Sharon Yang	Xing Zeng
	Zheng Dong		

## Office hours

See [www.ugrad.cs.ubc.ca/~cs221](http://www.ugrad.cs.ubc.ca/~cs221)

*Thurs after class*

## Texts

Epp Discrete Math, Koffman Data Structs C++

## Course Work

No late work; may be flexible with advance notice

10% Labs

15% Programming projects (~ 3)

15% Written homework (~ 3)

20% Midterm exam

40% Final exam

Must pass the final and combo of labs/assignments to pass the course.

# Collaboration

You may work in groups of two people on:

- ▶ Labs
- ▶ Programming projects
- ▶ Written homework

You may also collaborate with others as long as you follow the rules (see the website) and **acknowledge** their help on your assignment.

Don't violate the collaboration policy.

# Course Mechanics

- ▶ Web page: [www.ugrad.cs.ubc.ca/~cs221](http://www.ugrad.cs.ubc.ca/~cs221)
- ▶ Piazza:  
<https://piazza.com/ubc.ca/winterterm12016/cpsc221/home>
- ▶ UBC Connect site: [www.connect.ubc.ca](http://www.connect.ubc.ca)
- ▶ Labs are in ICCS X350
  - ▶ Use the Xshell program on the lab machines to ssh into a undergrad Unix machine (e.g. lulu.ugrad.cs.ubc.ca)
- ▶ Programming projects will be graded on UNIX/g++

encourage  
non-anonymous  
posting

# What is a Data Structure?

List              Tree  
Array              Queue  
Map              Graph  
Stack              HashTable  
Heap

A method of storing data that provides, through a set of operations, a way to manipulate and access the data.

# Observation

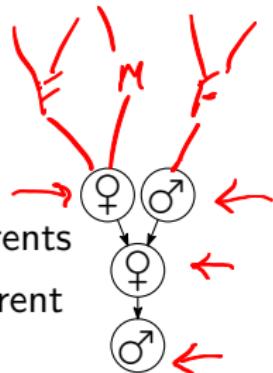
You

- ▶ All programs manipulate data
  - ▶ programs process, store, display, gather data
  - ▶ data can be information, numbers, images, sound
- ▶ The programmer must decide how to store and manipulate data
- ▶ This choice influences the program in many ways
  - ▶ execution speed
  - ▶ memory requirements
  - ▶ maintenance (debugging, extending, etc.)

## Goals of the Course

- ▶ Become familiar with some of the fundamental data structures and algorithms in computer science
  - ▶ Learn when to use them
- ▶ Improve your ability to solve problems abstractly
  - ▶ Data structures and algorithms are the building blocks
- ▶ Improve your ability to analyze algorithms
  - ▶ Prove correctness
  - ▶ Gauge, compare, and improve time and space complexity
- ▶ Become modestly skilled with C++ and UNIX, but this is largely on your own!

## Analysis Example: Fibonacci numbers



Bee ancestry:

1. Fertilized egg becomes a female bee with two parents
2. Unfertilized egg becomes a male bee with one parent

How many great-grandparents does a male bee have?  
great-great-grandparents? ...

Fibonacci numbers: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

First two numbers are 1; each succeeding number is the sum of the previous two numbers.

# Recursive Fibonacci

Problem: Calculate the  $n$ th Fibonacci number.

Recursive definition:

$$fib_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ fib_{n-1} + fib_{n-2} & \text{if } n \geq 3 \end{cases}$$

C++ code:

```
int fib(int n) {
    if (n <= 2) return 1;
    else         return fib(n-1) + fib(n-2);
}
```

Too slow!

# Iterative Fibonacci

Idea: Use an array

```
int fib(int n) {  
    int F[n+1];  
  
    F[0]=0; F[1]=1; F[2]=1;  
    for( int i=3; i<=n; ++i ) {  
        F[i] = F[i-1] + F[i-2];  
    }  
    return F[n];  
}
```

(We don't really need the array.)

Can we do better?

## Fibonacci by formula

Idea: Use a formula (a *closed form solution* to the recursive definition.)

$$fib_n = \frac{\varphi^n - (-\varphi)^{-n}}{\sqrt{5}}$$

where  $\varphi = (1 + \sqrt{5})/2 \approx 1.61803$ .

```
#include <cmath>
int fib(int n) {
    double phi = (1 + sqrt(5))/2;
    return (pow(phi, n) - pow(-phi,-n))/sqrt(5);
}
```

Sadly, it's **impossible** to represent  $\sqrt{5}$  exactly on a digital computer.

Can we do better?

## Fibonacci with Matrix Multiplication

A

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 + 1 \\ 1 \end{pmatrix} = \begin{pmatrix} fib_3 \\ fib_2 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \end{pmatrix} = \begin{pmatrix} fib_4 \\ fib_3 \end{pmatrix}$$

$\begin{pmatrix} 3 \\ 2 \end{pmatrix}$

$$\underbrace{\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-2} \begin{pmatrix} 1 \\ 1 \end{pmatrix}}_{=} = \begin{pmatrix} fib_n \\ fib_{n-1} \end{pmatrix}$$

How do we calculate  $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-2}$  ?

## Repeated Squaring

$$\begin{matrix} 2^5 & + & 2^3 & + & 2^1 & 2^0 \\ \hline 1 & 1 & 1 & 1 & 1 & 1 \end{matrix}$$

$$= 63$$

$$2^6 - 1$$

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

$$\begin{aligned} A \times A &= A^2 \\ A^2 \times A^2 &= A^4 \\ A^4 \times A^4 &= A^8 \\ A^8 \times A^8 &= A^{16} \\ A^{16} \times A^{16} &= A^{32} \end{aligned}$$

:

$$\begin{array}{rcccl} n & \times & i & \underline{\quad} & m \\ 43 & 39sec & - & - & \\ \vdots & X & & & \\ \end{array}$$

$$\begin{array}{rcccl} & 2^{20} & & 7sec & - \\ 2 & 2^{22} & & 2:39sec & - \\ \vdots & & & X & \\ \end{array}$$

$$\begin{array}{rcccl} & 2^{25} & & 16sec & \\ & . & & (3sec) & \end{array}$$

Example:  $A^{\textcircled{n}} = A^{64} \times A^{32} \times A^4$ . 8 instead of 99 multiplications.

Generally, about  $\log_2 n$  multiplications.

$$2^{27} \quad (18.5\text{ sec})$$

Is this better than iterative Fibonacci?

:

# Abstract Data Type

## Abstract Data Type

Mathematical description of an object and the set of operations on the object

### Example: **Dictionary ADT**

- ▶ Stores pairs of strings: (word, definition)
- ▶ Operations:
  - ▶ Insert(word,definition)
  - ▶ Delete(word)
  - ▶ Find(word)

*← returns def. for word*

## Another Example: Array ADT

- ▶ Store things like integers, (pointers to) strings, etc.
- ▶ Operations:

- ▶ Initialize an empty array that can hold  $n$  things.

```
thing A[n];
```

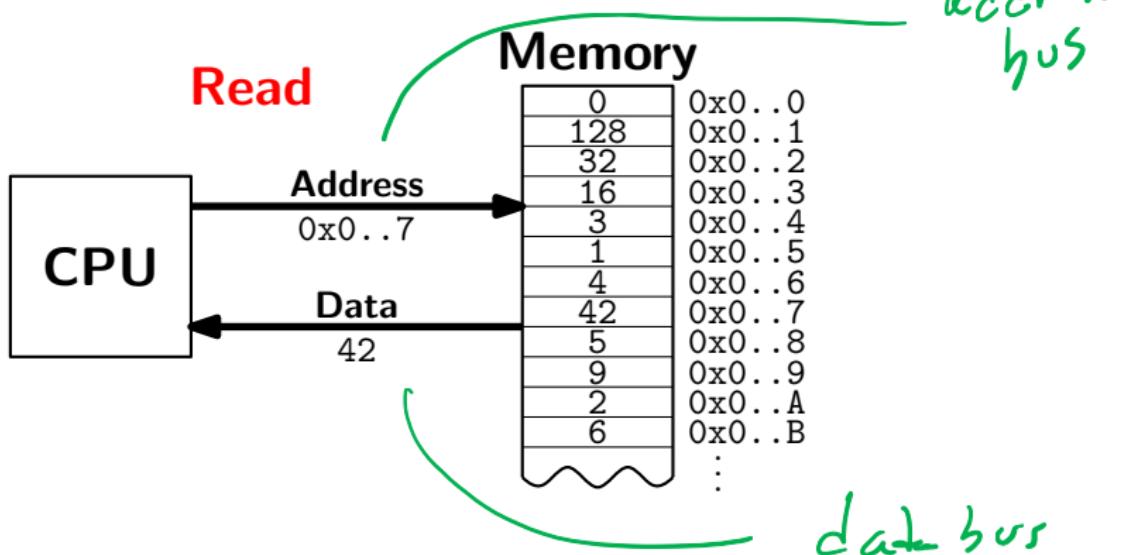
- ▶ Access (read or write) the  $i$ th thing in the array  
 $(0 \leq i \leq n - 1)$ .

```
thing1 = A[i]; Read
```

```
A[i] = thing2; Write
```

# Why Arrays?

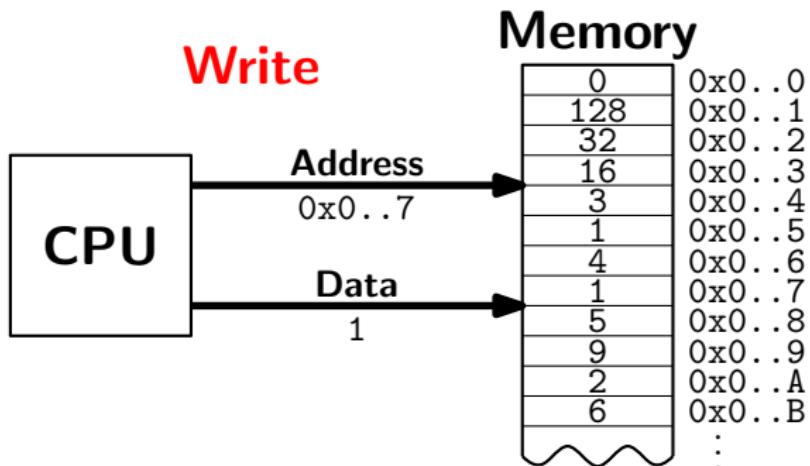
- ▶ Computer memory is an array.  
Read: CPU provides address  $i$ ,  
memory unit returns the data stored at  $i$ .



# Why Arrays?

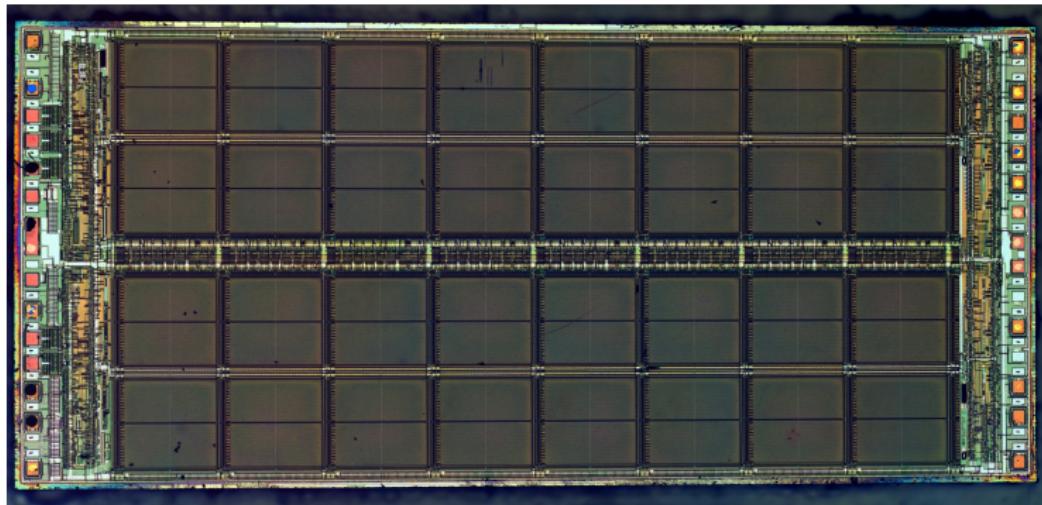
- ▶ Computer memory is an array.

Write: CPU provides address  $i$  and data  $d$ ,  
memory unit stores data  $d$  at  $i$ .



# Why Arrays?

- ▶ Computer memory is an array. Every bit has a physical location.



<http://zeptobars.ru/en/read/how-to-open-microchip-asic-what-inside> licensed under Creative Commons Attribution 3.0 Unported.

# Why Arrays?

- ▶ Computer memory is an array.
- ▶ Simple and fast.
- ▶ Used in almost every program.
- ▶ Used to implement other data structures.

# Array limitations

- ▶ Need to know size when array is created.

Fix: Resizeable arrays.

If the array fills up, allocate a new, bigger array and copy the old contents to the new array.

- ▶ Indices are integers 0,1,2,...

Fix: Hashing.

(more later)

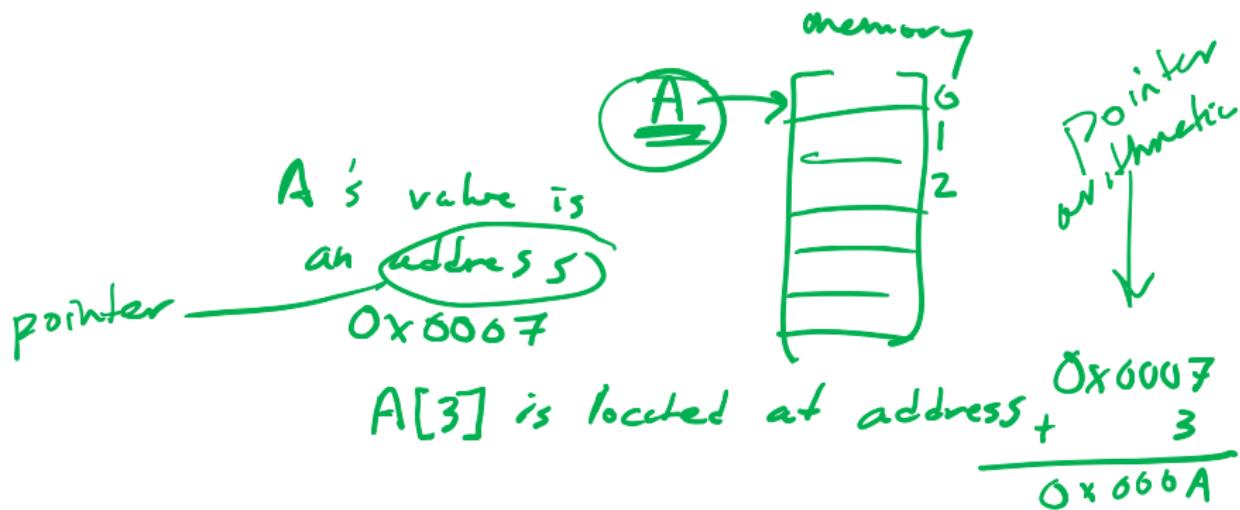
AWK has  
associative arrays  
(indexed by anything)

man awk

How would you implement the Array ADT?

memory management (truly)  
address arithmetic

convert  $i$  to mem address



# How would you implement the Array ADT?

Arrays in C++ *32-bit*

Create *int* A[100];

Access for( int i=0; i<100; i++ )

*A[i]* = (i+1) \* A[i-1];

*i*  
*\* (A + i)*      ↑

# How would you implement the Array ADT?

## Arrays in C++

```
Create int A[100];  
Access for( int i=0; i<100; i++ )  
    A[i] = (i+1) * A[i-1];
```

**Warning** No bounds checking!

# Data Structures as Algorithms

## Algorithm

a high level, language independent description of a step-by-step process for solving a problem

## Data Structure

a way of storing and organizing data so that it can be manipulated as described by an ADT

A data structure is defined by the algorithms that implement the ADT operations.

# Why so many data structures?

## Ideal data structure

fast, elegant, memory efficient

## Trade-offs

- ▶ time vs. space
- ▶ performance vs. elegance
- ▶ generality vs. simplicity
- ▶ one operation's performance vs. another's

## Data structures for Dictionary ADT

- ▶ List
- ▶ Skip list
- ▶ Binary search tree
- ▶ AVL tree
- ▶ Splay tree
- ▶ B-tree
- ▶ Red-Black tree
- ▶ Hash table

...

# Code Implementation

## Theory      Ideal

- ▶ abstract base class (interface) describes ADT
- ▶ descendants implement data structures for the ADT
- ▶ data structures can change without affecting client code

## Practice

- ▶ different implementations sometimes suggest different interfaces (generality vs. simplicity)
- ▶ performance of a data structure may influence the form of the client code (time vs. space, one operation vs. another)

if delete is slow, you may not call delete  
client

array  
or dictionary  
is full()

# ADT Presentation Algorithm

1. Present an ADT
2. Motivate with some applications
3. Repeat
  - 3.1 develop a data structure for the ADT
  - 3.2 analyze its properties
    - ▶ efficiency
    - ▶ correctness
    - ▶ limitations
    - ▶ ease of programming
4. Contrast data structure's strengths and weaknesses
  - ▶ understand when to use each one

# Queue ADT

## Queue operations

- ▶ create
- ▶ destroy
- ▶ enqueue
- ▶ dequeue
- ▶ is\_empty



## Queue property

If  $x$  is enqueued before  $y$  is enqueued, then  $x$  will be dequeued before  $y$  is dequeued.

FIFO: First In First Out

→ *x is removed from Queue*

## Applications of the Q

- ▶ Hold jobs for a printer
- ▶ Store packets on network routers
- ▶ Hold memory “freelists”
- ▶ Make waitlists fair
- ▶ Breadth first search

## Abstract Q Example

enqueue R  
enqueue O  
dequeue  
enqueue T  
enqueue A  
enqueue T  
dequeue  
dequeue  
enqueue E  
dequeue

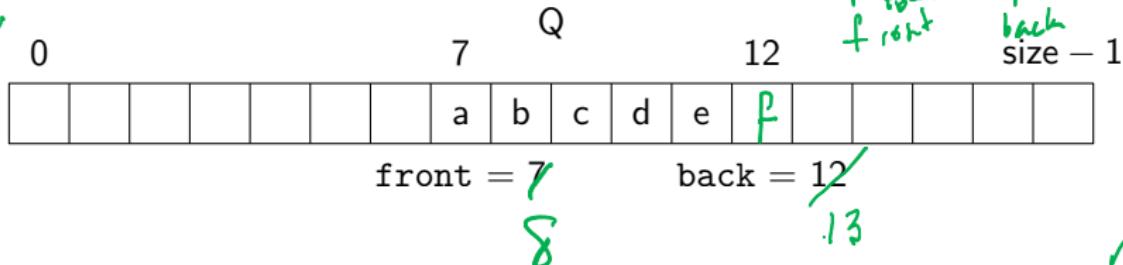
In order, what letters are dequeued?

- a. OATE
- b. ROTA
- c. OTAE
- d. None of these, but it **can** be determined from just the ADT.
- e. None of these, and it **cannot** be determined from just the ADT.



initial entries ?  
Size ?  
RRRR ?

# Circular Array Q Data Structure



```
void enqueue(Object x) {
    Q[back] = x;
    back = (back + 1) % size;
}
```

```
bool is_empty() {
    return (front == back);
}
```

If is\_full() explode();

```
Object dequeue() {
    x = Q[front];
    front = (front + 1) % size;
    return x;
}
```

```
bool is_full() {
    return (front ==
        (back + 1) % size);
}
```

If is\_empty() implode();

# Circular Array Q Example

	0	1	2	3	front	back	
enqueue R	<del>R</del>	O			0	0	
enqueue O					0	1	
dequeue					1	2	
enqueue T	T	0	/	A	1	2	
enqueue A					1	3	
enqueue T	T	E	T	/	1	0	
dequeue					1	1	
dequeue							
enqueue E	T	E	T	A			
dequeue							

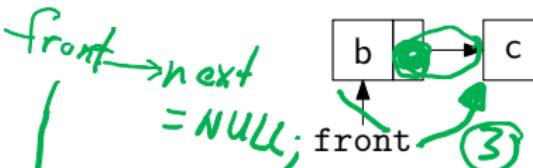
\* explode

What are the final contents of the array?

- a. RTE
- b. RTET
- c. TETA ✓
- d. TE ✓
- e. None

# Linked List Q Data Structure

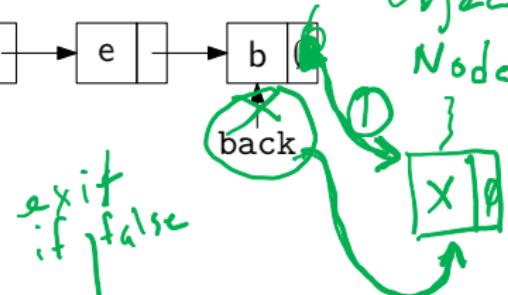
```
struct Node{  
    Object data;  
    Node* next;  
};
```



Node \*front, \*back;

```
void enqueue(Object x) {  
    if (is_empty())  
        front = back = new Node(x)  
    else {  
        back->next = new Node(x);  
        back = back->next;  
    }  
}
```

```
bool is_empty() {  
    return (front == NULL);  
}
```



```
Object dequeue()  
assert(!is_empty());  
Object ret = front->data;  
Node *temp = front;  
front = front->next; ③  
delete temp;  
return ret;
```

Why not ④ front = front->next ?

DIY memory management      memory leak

④ delete front;  
front = front->next ?

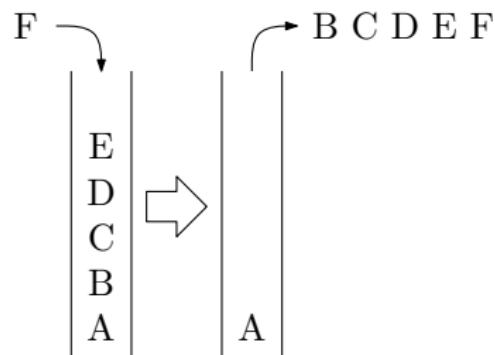
# Circular Array vs. Linked List

- Ease of implementation      same
- Generality      size / init — dynamic resize ✘
- Speed —      same except ✘ and ✘  
Memory use      next pointers increase memory  
                    ✘ Cache performance better for  
                    Array

# Stack ADT

## Stack operations

- ▶ create
- ▶ destroy
- ▶ push
- ▶ pop
- ▶ top
- ▶ is\_empty



## Stack property

if  $x$  is pushed before  $y$  is pushed, then  $x$  will be popped after  $y$  is popped.

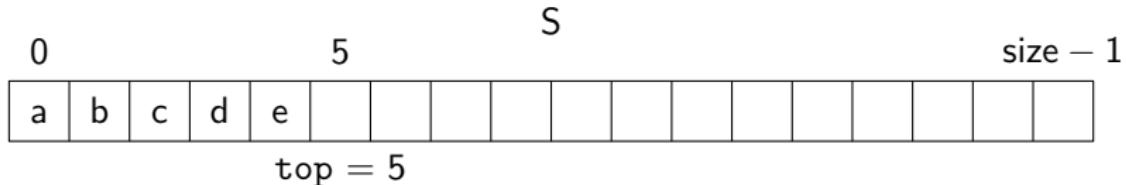
LIFO: Last In First Out

# Stacks in Practice

- ▶ Function call stack
- ▶ Removing recursion
- ▶ Balancing symbols (parentheses)
- ▶ Evaluating Reverse Polish Notation
- ▶ Depth first search

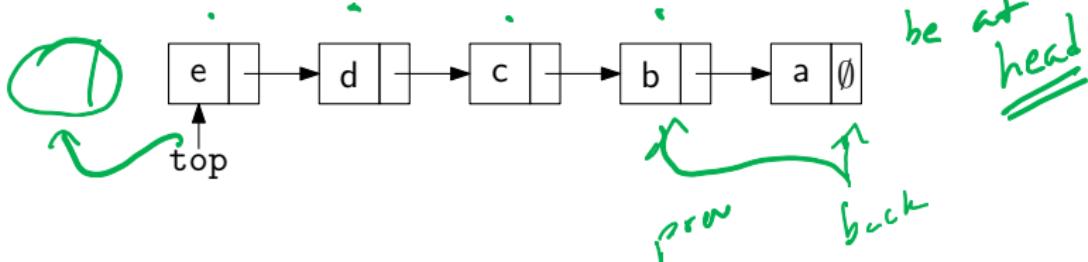
postscript  
java VM  
HP 1990s

# Array Stack Data Structure



```
void push(Object x) {          Object pop() {  
    assert(!is_full());           assert(!is_empty());  
    S[top] = x;                  top--;  
    top++;                      return S[top];  
}  
Object top() {                 }  
    assert(!is_empty());  
    return S[top-1];  
}  
bool is_empty() {              }  
    return( top == 0 );  
}  
bool is_full() {               }  
    return( top == size);  
}
```

# Linked List Stack Data Structure



```
void push(Object x) {  
    Node *temp = top;  
    top = new Node(x);  
    top->next = temp;  
}
```

```
Object top() {  
    assert(!is_empty());  
    return top->data;  
}
```

```
Object pop() {  
    assert(!is_empty());  
    Object ret = top->data;  
    Node *temp = top;  
    top = top->next;  
    delete temp;  
    return ret;  
}  
  
bool is_empty() {  
    return( top == NULL );  
}
```

# Deque ADT

icon

## Deque (Double-ended queue) operations

- ▶ create/destroy
- ▶ pushL/pushR
- ▶ popL/popR
- ▶ is\_empty

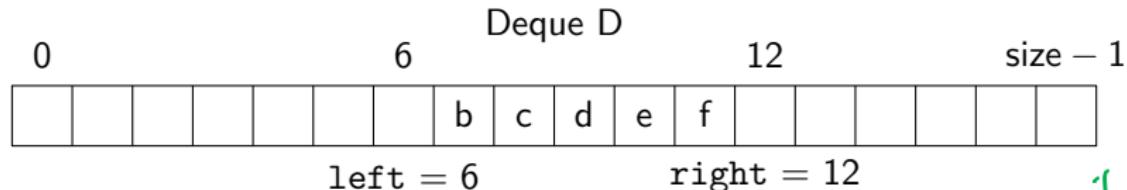


## Deque property

Deque maintains a list of items.

push/pop adds to/removes from front(L)/back(R) of list.

# Circular Array Deque Data Structure



```

void pushL(Object x) {
    assert(!is_full());
    D[left] = x;
    left = (left - 1) % size;
}

Object popR() {
    assert(!is_empty());
    right = (right - 1) % size;
    return D[right];
}

...

```

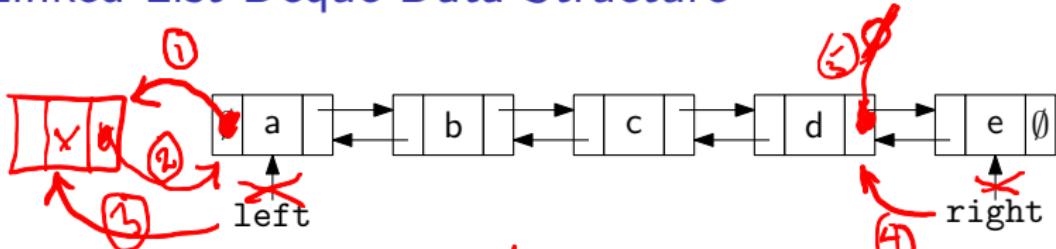
```

bool is_empty() {           is size <
    return( left ==          right - 1 ) % size);
}
bool is_full() {
    return( left ==          right + 1 ) % size);
}

```

is empty  
Degree

# Linked List Deque Data Structure



*Node ≠ left, ≠ right;*

```
void pushL(Object x) {  
    if( is_empty() )  
        left = right = new Node(x);  
    else {  
        ⑤ left->prev = new Node(x);  
        ② left->prev->next = left;  
        ③ left = left->prev;  
    }  
}
```

```
bool is_empty() {return left==NULL;}
```

```
Object popR() {  
    assert(!is_empty());  
    Object ret = right->data;  
    Node *temp = right;  
    right = right->prev; ④  
    ⑤ if( right ) right->next = NULL;  
    else left = NULL;  
    delete temp;  
    return ret;  
}
```

*right  
 $\neq$  NULL*

## Data structures you should already know (a bit)

- ▶ Arrays
- ▶ Linked lists
- ▶ Trees
- ▶ Queues
- ▶ Stacks