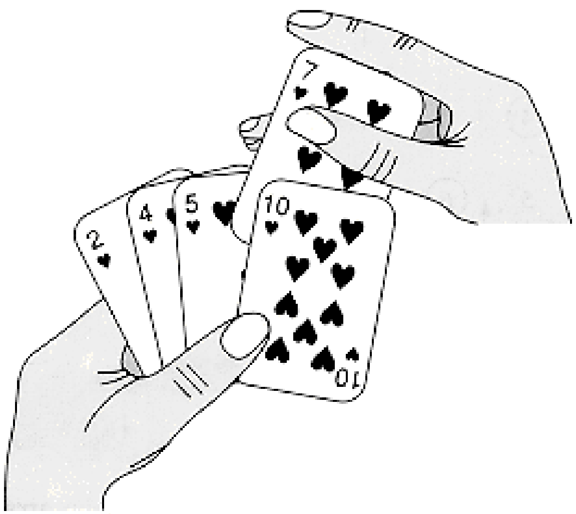
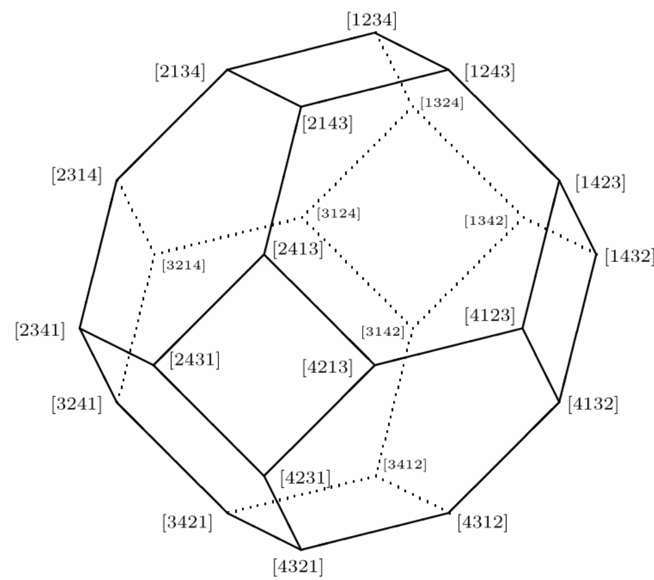
CS3334 Data Structures

Lecture 3: Arrays, Linked Lists, Stacks & Queues





Chee Wei Tan

Introduction (1/2)

- Learn how to store and organize data in a computer so that the data can be managed efficiently
- You will learn
  - Representation of data
  - Algorithms (methods) for managing data (usually include search, insert, delete, and update)

Introduction (2/2)

- Efficiency is important
  - You can almost always use a dumb structure if you do not care about the efficiency
- Focus on main memory data storage; data storage in secondary storage (e.g., hard disks and databases) is usually called indexing structures

Arrays (1/2)

- Array is a data structure that arranges items at equally spaced addresses in computer memory.
- E.g., The foo array, with five elements of type int, can be declared as:

  int foo [7];

10	14	8	5	2		
[0]	[1]	[2]	[3]	[4]	[5]	[6]

- (+ve) Array elements can be accessed by specifying the array name followed by the index in square brackets, e.g., the value of foo[2] is 8.

Arrays (2/2)

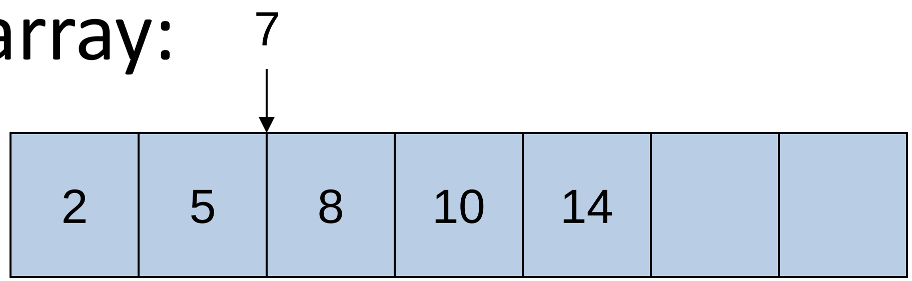
- Efficient for insertion by appending a new element, e.g., insert 7

10	14	8	5	2	7	

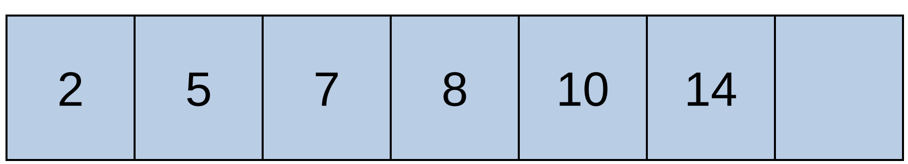
- (-ve) However, searching an item in an unsorted list:
  - Have to scan through the whole array to determine for sure if an item is not there
  - E.g., search for 15

Sorted Arrays

- (+ve) Efficient for searching
- However, inserting an item (i.e., 7) into a sorted array:

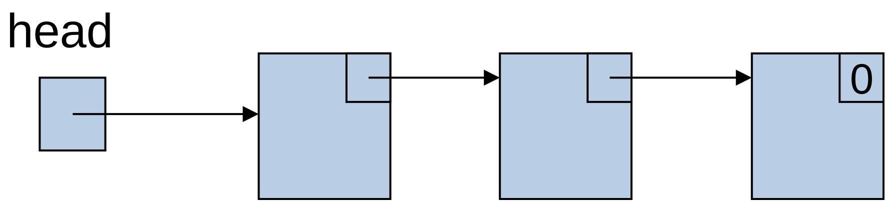


  – (-ve) Have to move all items behind the point of insertion (i.e., 8, 10, and 14) to make room for the new one (i.e., 7)



Linked Lists

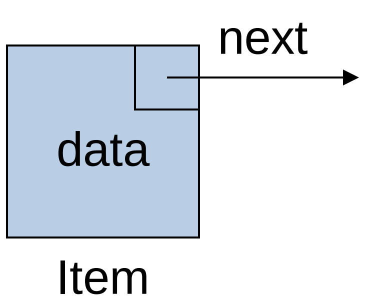
- A linked list is a data structure that allows both efficient searching and insertion/deletion.
- A collection of items linked in a sequence:



- (+ve) Easy to insert/delete items in the middle, provided we know where to insert/delete (a sorted linked list can easily be maintained.)
- (-ve) Difficult to access the *i*-th item, given an arbitrary *i*

Linked Lists: Node

```
struct Node
{
   Node(): next(NULL) {}
   Node(int newData): data(newData), next(NULL) {}
   Item data;    // Item is a generic data type
   Node* next;
};
```



Linked Lists: Traversing (non-circular)

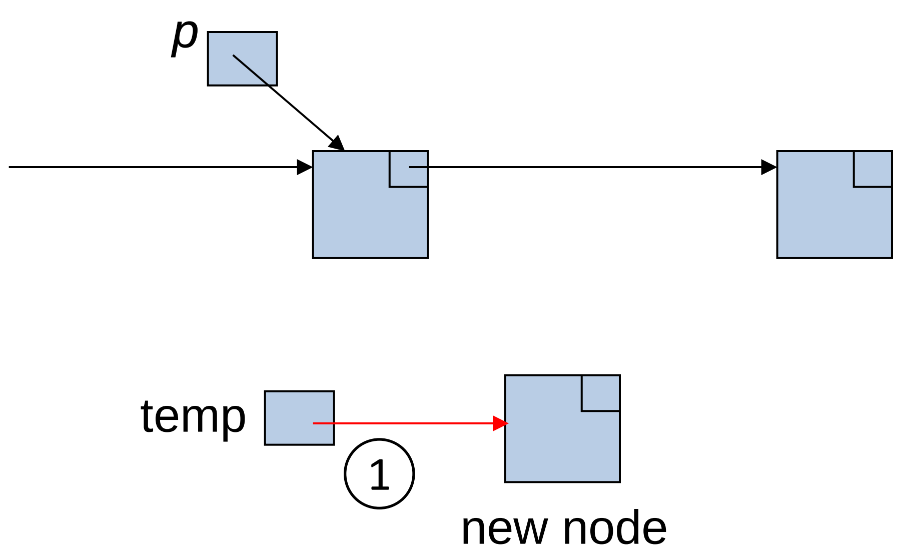
```
//  Suppose head points to the 1st node of a list
Node* p = head;
while (p!=0)
{
   // process p->data
   p = p->next;
}
```

- Compare with a loop that scans through an array:

```
//  Suppose a[0..n-1] is an array
int i = 0;
while (i<n)
{
   // process a[i]
   i++;
}
```

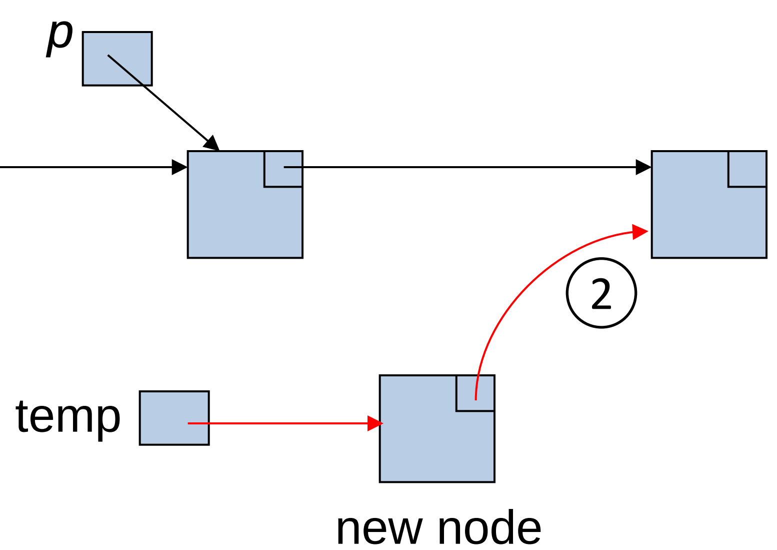
Linked Lists: Insert a Node (1/3)

```
// To insert a node after the node pointed to by p.
// newData is a variable of type Item
1 Node *temp = new Node(newData);
2 temp->next = p->next;
3 p->next = temp;
```



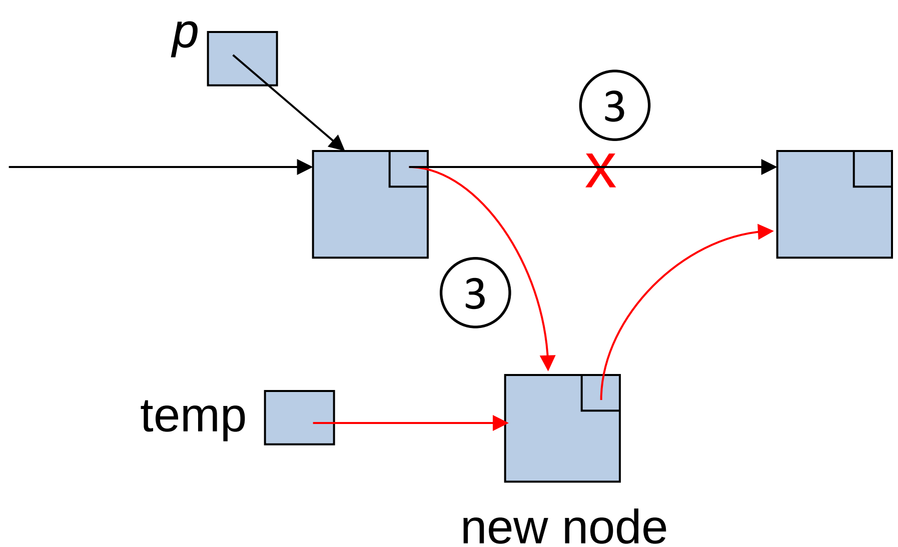
Linked Lists: Insert a Node (2/3)

```
// To insert a node after the node pointed to by p.
// newData is a variable of type Item
① Node *temp = new Node(newData);
② temp->next = p->next;
③ p->next = temp;
```



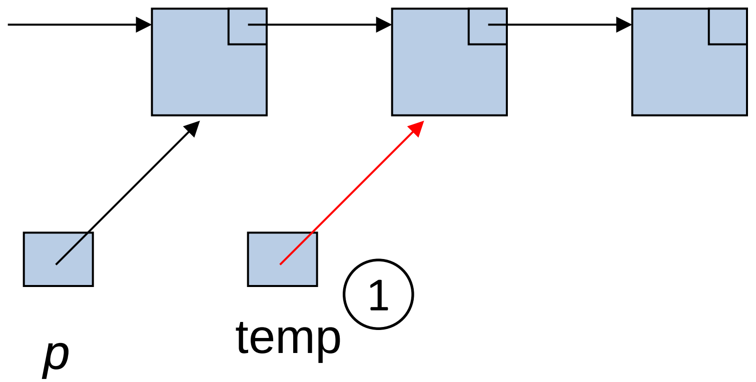
Linked Lists: Insert a Node (3/3)

```
// To insert a node after the node pointed to by p.
// newData is a variable of type Item
1 Node *temp = new Node(newData);
2 temp->next = p->next;
3 p->next = temp;
```



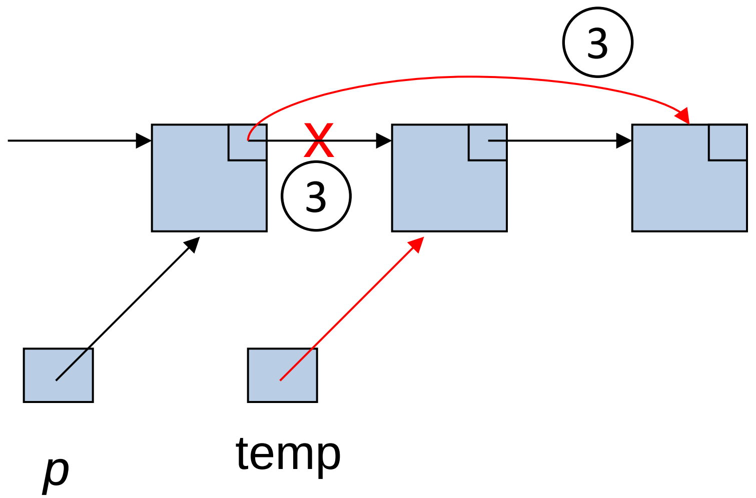
Linked Lists: Delete a Node (1/2)

```
// To delete a node after the node pointed to by p
// var. retData to retrieve content of deleted node
1 Node *temp = p->next;
2 retData = temp->data;
3 p->next = temp->next;
4 delete temp;
```



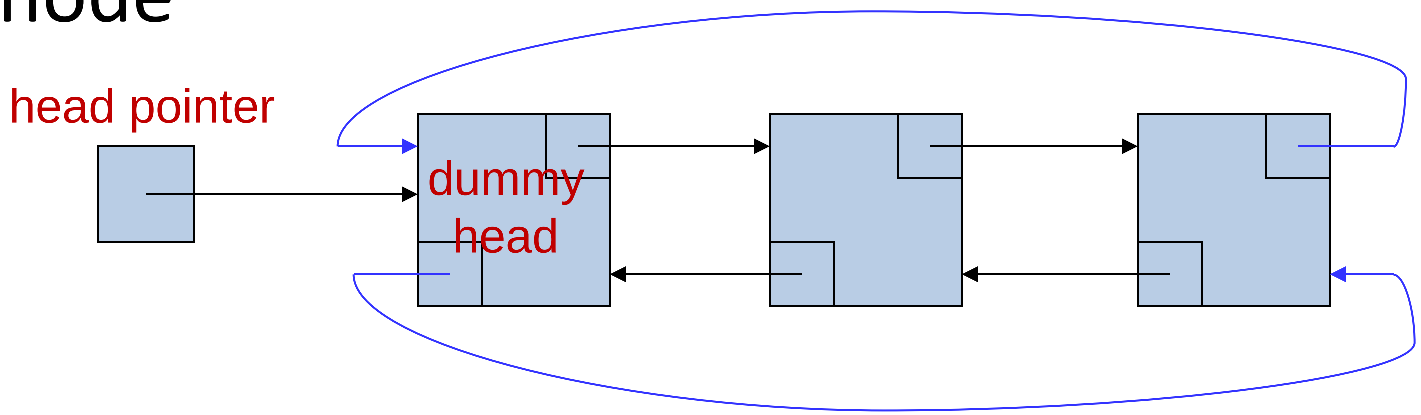
Linked Lists: Delete a Node (2/2)

```
// To delete a node after the node pointed to by p
// var. retData to retrieve content of deleted node
① Node *temp = p->next;
② retData = temp->data;
③ p->next = temp->next;
④ delete temp;
```



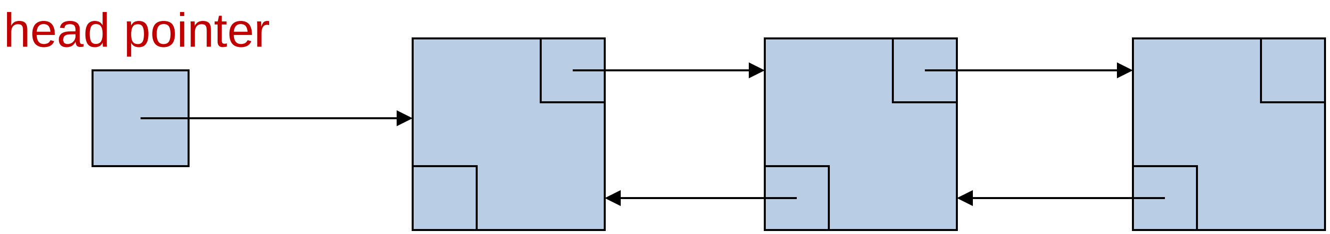
Variations of Linked Lists (1/2)

- Some variations of linked list
  - Singly- / doubly-linked
  - With / without dummy head node
  - Circular / non-circular
- E.g., a circular doubly-linked list with a dummy head node

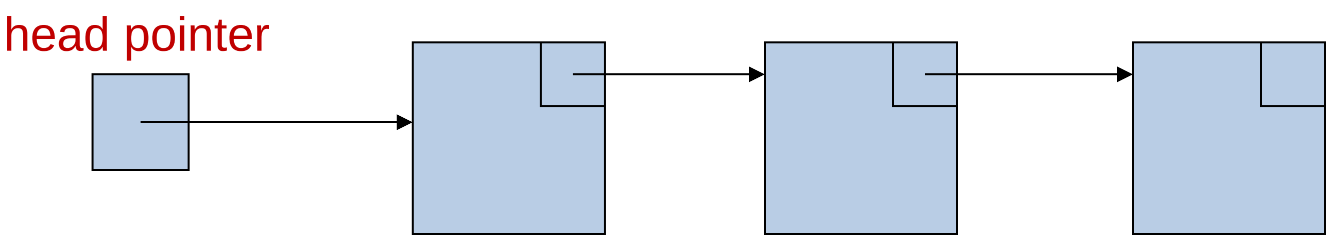


Variations of Linked List (2/2)

- E.g., a non-circular doubly-linked list without a dummy head node



- E.g., a non-circular singly-linked list without a dummy head node



Stacks

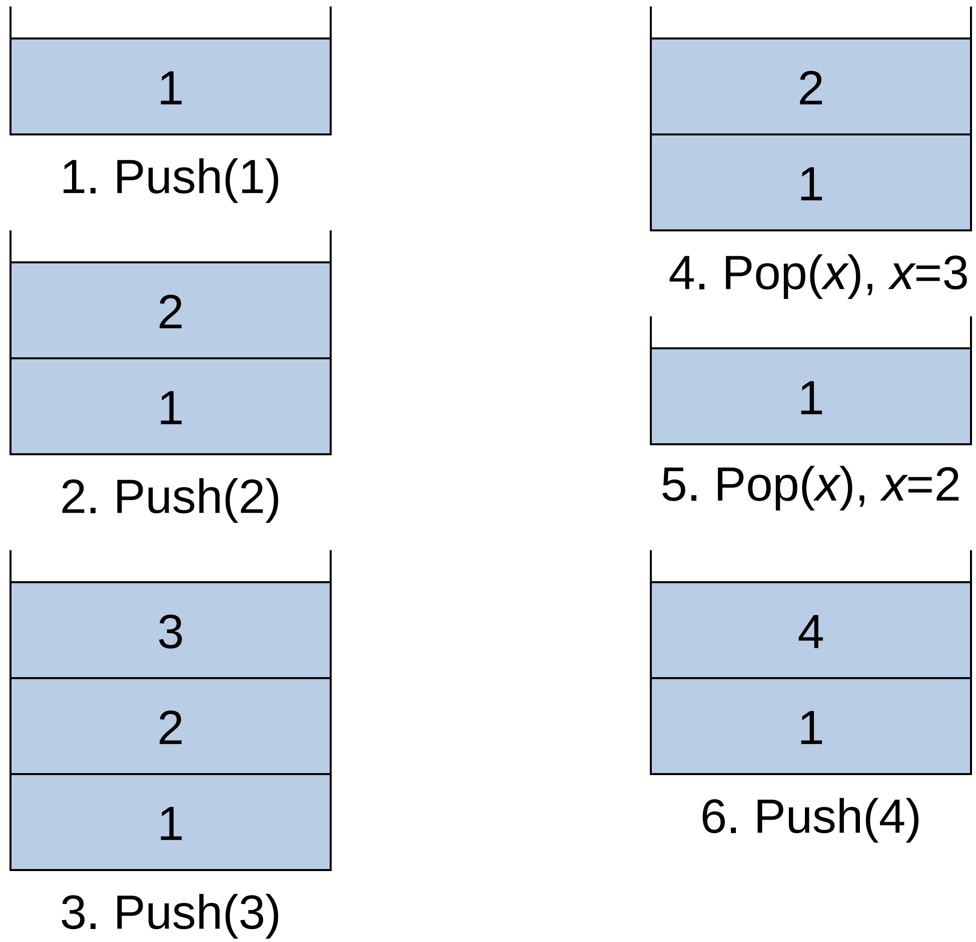
- A stack is a sequence of elements in which update can only happen at one end of the sequence, called the top.
- Operations supported:
  - push(x): add an element x to the top
  - pop(x): remove the top element and return it in x, i.e., first-in-last-out (FILO)

Stacks: Example

Actions

1. Push(1)
2. Push(2)
3. Push(3)
4. Pop(x)
5. Pop(x)
6. Push(4)

Stack | **Stack**



Stacks: Use Array (1/2)

- Array implementation of stack :
  - Maintain `size`, i.e., the number of elements in stack
  - Elements stored in `A[0..size-1]`
    - The oldest one at `A[0]` is called *bottom of stack*
    - The newest one at `A[size-1]` is called *top of stack*
  - push(*x*):
    - Store *x* at `A[size]`; then increase `size` by 1
  - pop(*x*):
    - If `size` = 0, return "Empty Stack", otherwise decrease `size` by 1 and store `A[size]` in *x*

Stacks: Use Array (2/2)

- How to choose the size of array `A[]`?
- As we insert more and more, eventually the array will be full
- Solution: Use a dynamic array
  - Maintain `capacity` of `A[]`
  - Double `capacity` when `capacity=size` (i.e. full)
  - Half `capacity` when `size`≤`capacity/4`
- Question: What if we change `capacity/4` to `capacity/2`?
  - Avoid repeatedly growing and shrinking, e.g., initial cap is 4; I, I, I, I, I (expand; cap=8, size=5), D (shrink; cap=4, size=4), I (expand; cap=8, size=5), D (shrink; cap=4, size=4), I (expand), D (shrink), ….

Stacks: C++ Code (1/3)

```
class Stack
{
   public:
     Stack(int initCap=100);
     Stack(const Stack& rhs);
     ~Stack();

     void push(Item x);
     void pop(Item& x);

   private:
     void realloc(int newCap);
     Item* array;
     int size;
     int cap;
};
```

Stacks: C++ Code (2/3)

```
void Stack::push(Item x)
{
   if (size==cap) realloc(2*cap);
   array[size++]=x;
}

// An internal func. to support resizing of array
void Stack::realloc(int newCap)
{
   if (newCap < size) return;
   Item *oldarray = array; //oldarray is “point to” array
   array = new Item[newCap]; //create new space for array
                             //with a size of newCap
   for (int i=0; i<size; i++)
      array[i] = oldarray[i];
   cap = newCap;
   delete [] oldarray;
}
```

Stacks: C++ Code (3/3)

```
void Stack::pop(Item& x)
{
   if (size==0)
     x=EmptyStack;
     // assume EmptyStack is a special value
   else
   {
     x=array[--size];
     if (size <= cap/4)
        realloc(cap/2);
   }
}
```

Stacks: Time Complexity

- Let n = current number of elements in the stack; n changes as pushes/pops are performed
- Excluding the time for expanding/shrinking the array, push() and pop() need O(1) time in the worst case (the best one can hope for)
- Array expanding/shrinking is costly (O(n) time) but is needed once after at least n/2 operations
- E.g., initial cap is 4, after 5 push operations
  - *size* = n = 5, *cap* = 8, n/2 = 2.5
  - Need $\geq$ 4 more push operations to trigger array expanding
  - Need $\geq$ 3 more pop operations to trigger array shrinking

Stacks: Space Complexity

- When there are n elements in a stack, the largest required capacity of the stack is $4n$. E.g., when $size=n=2$, the largest required capacity $cap=8$ (just before shrinking).

- Space allocated for array `A[0..cap-1]` is:

  $$\leq \max\{100, 4n\}$$

  Therefore, the space complexity is $S(n) = O(n)$.

Stacks: Use Linked List

- Linked-list implementation of stack
  - May use a circular doubly-linked list
  - push(x): call insertFront(x)
  - pop(x): if stack not empty, call deleteFront(x)
- Both array and linked-list implementations are efficient:
  - Excluding array expansion/shrinking, all operations take O(1) time in the worst case

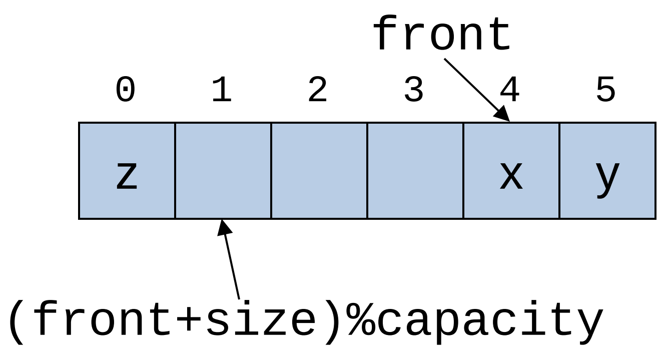
Queues

- A queue is a list of elements in which insertion can only be done at one end and deletion at the other.
- Operations supported:
  - EnQueue(x): insert element x at the end
  - DeQueue(x): delete the front element and return it in x, i.e., first-in-first-out (FIFO)

Queues: Use Circular Array (1/2)

- Circular array implementation of queue:
  - Maintain 2 variables: `front` and `size`
  - Elements stored in `A[front..front+size-1]`, the oldest element at `A[front]`, the newest element at `A[front+size-1]`
  - Wrap around the array, if necessary, using "`%capacity`".

    E.g., `capacity=6, front=4, size=3:`



modulo operation finds the remainder of division of a by n: a mod n = a – (n * $\lfloor a/n \rfloor$)

10 mod 3 = 10 – (3 * $\lfloor 10/3 \rfloor$)

= 10 – 3 * 3 = 1

floor(x) = $\lfloor x \rfloor$ *is the largest integer not greater than x*

Queues: Use Circular Array (2/2)

- EnQueue(*x*):
  - Store x at `A[(front+size)%capacity]` and increase `size` by 1
- DeQueue(*x*):
  - If queue is not empty, copy `A[front]` to *x*, increase `front` by 1 and decrease `size` by 1
- Whenever `front` becomes `>=capacity` of `A[]`, subtract `front` by `capacity`, i.e., `front=front-capacity`.

Queues: Example

0	1	2	3	4	5

capacity=6
front=0
size=0

0	1	2	3	4	5
1					

EnQueue(1)
Insert “1” to (front+size)%capacity =(0+0)%6=0
Then, front=0; size=1

0	1	2	3	4	5
1	2				

EnQueue(2)
Insert “2” to (front+size)%capacity =(0+1)%6=1
Then, front=0; size=2

0	1	2	3	4	5
1	2	3			

EnQueue(3)
front=0; size=3

0	1	2	3	4	5
1	2	3	4		

EnQueue(4)
front=0; size=4

0	1	2	3	4	5
1	2	3	4	5	

EnQueue(5)
front=0; size=5

0	1	2	3	4	5
1	2	3	4	5	6

EnQueue(6)
front=0; size=6

0	1	2	3	4	5
	2	3	4	5	6

DeQueue(x), x=1
front=1; size=5

0	1	2	3	4	5
		3	4	5	6

DeQueue(x), x=2
front=2; size=4

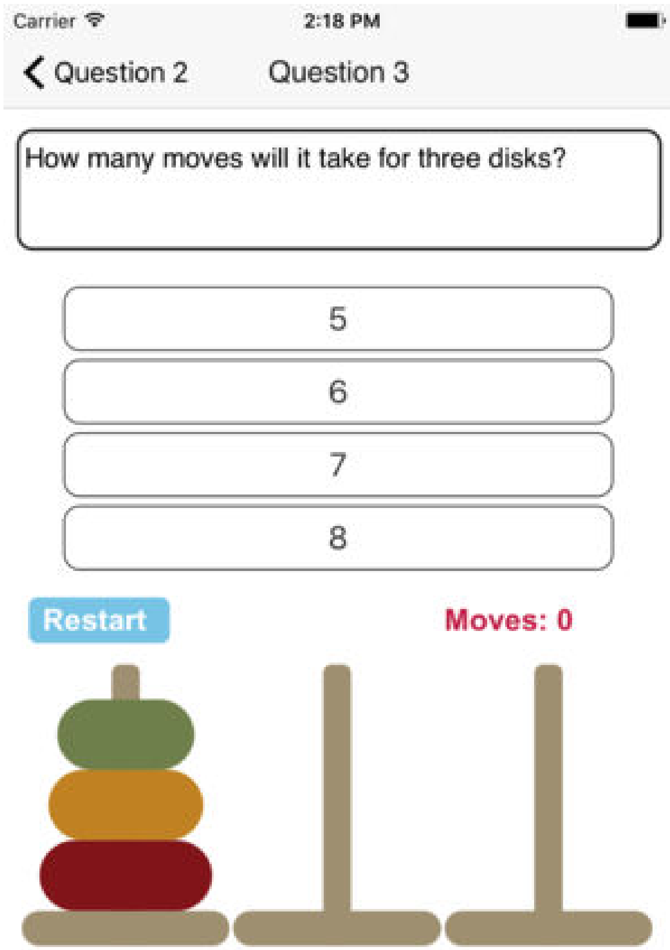
0	1	2	3	4	5
7		3	4	5	6

EnQueue(7)
Insert “7” to (front+size)%capacity =(2+4)%6=0
Then, front=2; size=5

Queues: Use Linked List

- How to check queue emptiness?
  - Check if `size==0`
- Linked list implementation of queue:
  - May use a circular doubly-linked list
  - EnQueue(x): call insertBack(x)
  - DeQueue(x): if queue is not empty, call deleteFront(x)

Tower of Hanoi: Sorting by Stacks

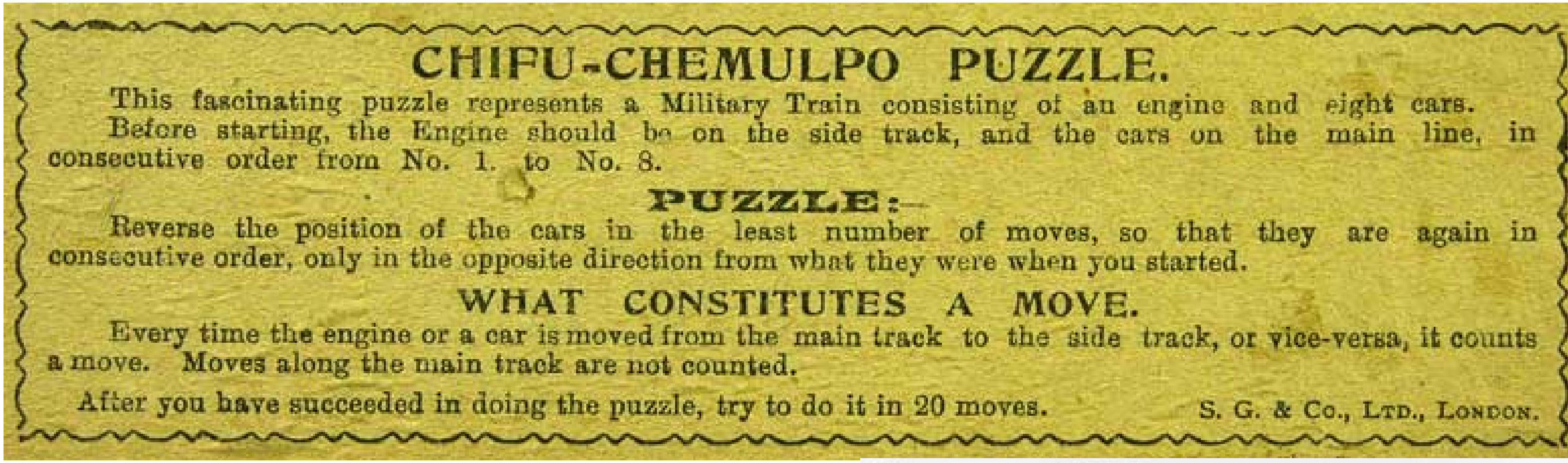


How long will it take for four disks?

How long will it take for *n* disks?

What is its Recurrence Equation?

Trains of Thought: Data Structures at Play



CHIFU-CHEMULPO PUZZLE.

This fascinating puzzle represents a Military Train consisting of an engine and eight cars.

Before starting, the Engine should be on the side track, and the cars on the main line, in consecutive order from No. 1. to No. 8.

PUZZLE:—

Reverse the position of the cars in the least number of moves, so that they are again in consecutive order, only in the opposite direction from what they were when you started.

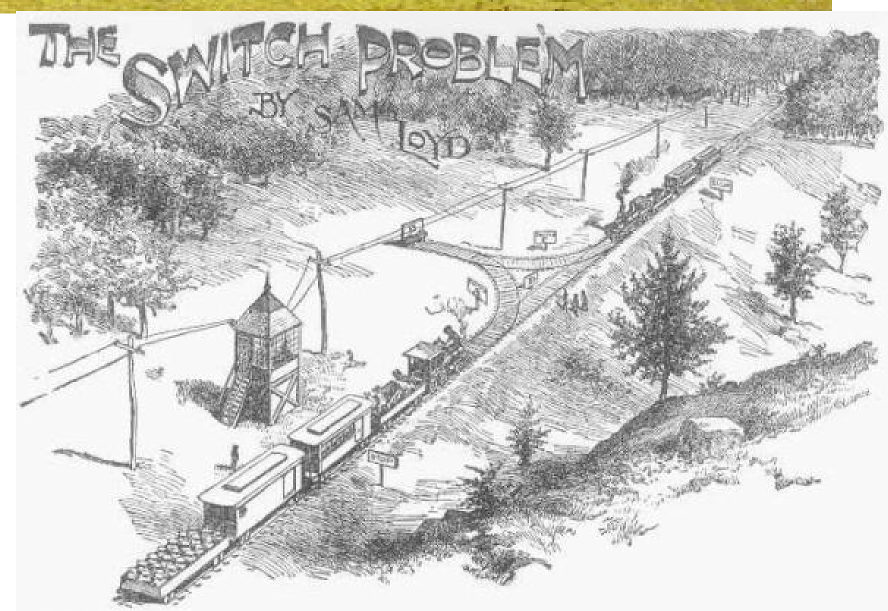
WHAT CONSTITUTES A MOVE.

Every time the engine or a car is moved from the main track to the side track, or vice-versa, it counts a move. Moves along the main track are not counted.

After you have succeeded in doing the puzzle, try to do it in 20 moves. S. G. & Co., Ltd., London.

How to compute the solution with fewest moves for *n* cars ?

1) Wikipedia on Stacks and Queues
https://en.wikibooks.org/wiki/Data_Structures/Stacks_and_Queues
2) A. J. T. Colin, A. D. McGettrick and P. D. Smith, Sorting Trains, The Computer Journal, Volume 23, 1978
3) Trains of Thought, B. Hayes, American Scientist, the magazine of Sigma Xi, The Scientific Research Society, Volume 95, 2007



PROPOSITION—Show how to let the two trains pass.