Data Structures (I)

Stack
Queue
Linked List
What are data structures?

A data structure is a particular way of organizing data so that they can be used efficiently.
What are data structures?

A data structure is a particular way of organizing data so that they can be used efficiently.
Abstract Data Types - implemented by data structures

- Stack - LIFO
- Queue - FIFO

Data structures

- Array
- Linked List
  - Singly linked list
  - Doubly linked list
  - Circular linked list
  - XOR linked list
Stack

A stack of books
Stack

We can

● put a book on top
● remove the top book

We cannot

● insert a book in the middle
● remove a book in the middle

A stack of books
Stack

We can

● put an element on top
● remove the top element

We cannot

● insert an element in the middle
● remove an element in the middle

Output order

● Last In First Out (LIFO)
Implementation

bear 0 1 2 3 4 5 6 7

0 1 2 3 4 5 6 7

bear cow 0 1 2 3 4 5 6 7

0 1 2 3 4 5 6 7

bear cow duck 0 1 2 3 4 5 6 7

0 1 2 3 4 5 6 7

bear cow 0 1 2 3 4 5 6 7

0 1 3 4 5 6 7
Why not use array?

- Array is too powerful, easier to make mistakes
- Stack is so simple, you can't make a mistake
- Stack is easier to understand than array
Parentheses balance

Determine if a string of 1 <= N <= 10000 parentheses is balanced.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>{{{}}}</td>
<td>Yes</td>
</tr>
<tr>
<td>{[[]]}</td>
<td>No</td>
</tr>
</tbody>
</table>
Parentheses balance

Maintain a stack,

left parentheses → push stack
right parentheses → pop stack
Parentheses balance

Maintain a stack,

left parentheses → push stack
right parentheses → pop stack

{()[]}
Parentheses balance

Maintain a stack,

left parentheses → push stack
right parentheses → pop stack

{ ( ) [ ] }

{ ( ) [ ] }

{ ( ) [ ] }

{ ( ) [ ] }
Parentheses balance

Maintain a stack,

left parentheses → push stack
right parentheses → pop stack
Parentheses balance

Maintain a stack,

left parentheses → push stack
geright parentheses → pop stack
Parentheses balance

How many 'push' were performed?

3

You tell me
## Parentheses balance

How many 'push' were performed?

<table>
<thead>
<tr>
<th>Expression</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{()[]}</code></td>
<td>3</td>
</tr>
<tr>
<td><code>{()[]}{}</code></td>
<td>4</td>
</tr>
<tr>
<td><code>()[()]{}{()&gt;&gt;()</code></td>
<td>You tell me</td>
</tr>
</tbody>
</table>
Parentheses balance

How many 'push' were performed?

- \{()[]\}\ 3
- \{()[]{}\}\ 4
- ()[(()){(())()]} 3
Parentheses balance

Time complexity: $O(N)$

Space complexity: $O(N)$
Expression evaluation

input: \(1+2\times3-4\)

output: 3
Expression

Infix notation
Operator *between* operands

\[ 1 + 2 \]
Expression

Prefix notation
(Polish Notation)
Operator before operands

Infix notation
Operator between operands

Postfix notation
(Reverse Polish Notation)
Operator after operands

Note that we will only deal with single digits in the following slides, so "12" should be interpreted as "one two" rather than "twelve"
Expression

Prefix notation
(Polish Notation)
Operator before operands

Infix notation
Operator between operands

Postfix notation
(Reverse Polish Notation)
Operator after operands

\[-+1*234\]

\[1+2*3-4\]

You tell me
Expression

Prefix notation
(Polish Notation)
Operator before operands

Infix notation
Operator between operands

Postfix notation
(Reverse Polish Notation)
Operator after operands

- + 1 * 2 3 4

1 + 2 * 3 - 4

1 2 3 * + 4 -
Expression

Prefix notation
(Polish Notation)
Operator before operands

Infix notation
Operator between operands

Postfix notation
(Reverse Polish Notation)
Operator after operands

You tell me
(1+2)*3-4
Expression

Prefix notation
(Polish Notation)
Operator before operands

Infix notation
Operator between operands

Postfix notation
(Reverse Polish Notation)
Operator after operands

- * + 1234

(1+2)*3-4

You tell me
Expression

Prefix notation
(Polish Notation)
Operator before operands

Infix notation
Operator between operands

Postfix notation
(Reverse Polish Notation)
Operator after operands

No parentheses are needed for both Polish notations
Prefix notation evaluation

input: \(+*23-41\)
output: You tell me
Prefix notation evaluation

input: $+*23-41$

output: 9
Prefix notation evaluation

Push to stack one by one, evaluate the top three elements whenever possible.

\[ + \times 23 - 41 \]

[Blank space for evaluation steps]
Prefix notation evaluation

Push to stack one by one, evaluate the top three elements whenever possible.

\[
\begin{array}{c}
+*23-41 \\
+*23-41
\end{array}
\]
Prefix notation evaluation

Push to stack one by one, evaluate the top three elements whenever possible.

\[
\begin{array}{c}
+ \times 23 - 41 \\
+ \times 23 - 41 \\
+ \times 23 - 41 \\
\end{array}
\]
Prefix notation evaluation

Push to stack one by one, evaluate the top three elements whenever possible.

```
+ * 2 3
+ 6
+ * 2 3
+ * 2
+ *
+ *
+ *
+ *
+ *
+ *
+ *
+ *
+ *
```

```
+ * 2 3
+ 6
+ * 2 3
+ *
+ *
+ *
+ *
+ *
+ *
+ *
+ *
```
Prefix notation evaluation

Push to stack one by one, evaluate the top three elements whenever possible.
Prefix notation evaluation

Push to stack one by one, evaluate the top three elements whenever possible.

\[
* + 23 - 4 1
\]

\[
+ \quad 6 \quad - \quad 4
\]
Prefix notation evaluation

Push to stack one by one, evaluate the top three elements whenever possible.

\[ + \times 23 - 41 \]

\[ + \times 23 - 41 \]

\[ + 6 - 4 \]

\[ + 6 - 4 1 \]

\[ + 6 3 \]

\[ 9 \]
Postfix notation evaluation

We can use the same approach
Infix notation evaluation

input: \( 4 + 2 \ast (3 - 1) \)

We have to handle

- Operator precedence: \( \ast / + - \)
- Parentheses

Maintain 2 stacks, one for operators and one for operands
Other applications

- Recursion
  - Exhaust permutations
  - Exhaust combinations
  - taught in *Recursion, Divide and Conquer*

- Depth-first search (DFS)
  - solve a maze
  - will be taught in *Graph (I)*
Practice time

- HKOJ 01015 Parentheses Balance
- HKOJ 01033 Simple Arithmetic
Queue

We can

- put a new comer at the back
- serve the customer in the front

We cannot

- insert a new comer in the middle
- serve a customer in the middle

A queue of human
Queue

We can

- put an element at the back
- remove the front element

We cannot

- insert an element in the middle
- remove an element in the middle

Output order

- First In First Out (FIFO)
<table>
<thead>
<tr>
<th>Bear</th>
<th>Cow</th>
<th>Duck</th>
<th>Eagle</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Implementenion
## Implementation - circular buffer

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>Bear</td>
<td>Cow</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>Duck</td>
<td>Eagle</td>
</tr>
<tr>
<td>Fox</td>
<td>Bear</td>
<td>Cow</td>
<td>Duck</td>
<td>Eagle</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Fox</td>
<td>Bear</td>
<td>Cow</td>
<td>Duck</td>
<td>Eagle</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Fox</td>
<td>Goat</td>
<td>Cow</td>
<td>Duck</td>
<td>Eagle</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>
Why not array?

Again,

- Queue is much simpler to understand
- Queue avoid mistakes
- Problems that can be solved by queue is definitely easier than problems that can only be solved by array
The Josephus Problem

Given $1 \leq N \leq 1000$ people arranged in a circle and 1 sword. The one holding a sword will kill the one on his left and pass the sword to the left. Who will survive?
The Josephus Problem

Enqueue 1 to N, repeat {Enqueue(Dequeue()); Dequeue();}
# The Josephus Problem

Using circular queue

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Breath-first search (BFS)

Grey nodes are in queue

Will be taught in Graph (I)
Practice time

- HKOJ P005 Rails
- HKOJ 01017 Car Sorter
- HKOJ 01030 The Josephus Problem
Linked list

We can

- insert a paper clip anywhere
- remove a paper clip anywhere
Linked list

We can

- insert a paper clip anywhere
- remove a paper clip anywhere

We cannot

- locate the $n^{th}$ paper clip quickly
Linked list

12 → 99 → 37 →  


Linked list

Insertion

12 → 99 → 37

newNode

37

node
node.next

newNode

37

node
node.next
Linked list

**Insertion**

12 → 99 → 37 → 

12 node → 99 node.next → 

newNode 37 → 

12 node → 99 node.next → 

newNode 37 → 

**Deletion**

12 → 99 → 37 → 

12 node → 99 node.next → 37 node.next.next → 

12 node → 99 node.next → 37 node.next.next → 

12 node → 99 node.next → 37 node.next.next →

12 node → 37 node.next.next →
### Implementation

<table>
<thead>
<tr>
<th>value</th>
<th>Bear</th>
<th>Cow</th>
<th>Duck</th>
<th>Eagle</th>
<th>Fox</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Here, `next` indicates the next value in the sequence, and each value corresponds to a specific animal as follows:
- Value 0: Bear
- Value 1: Cow
- Value 2: Duck
- Value 3: Eagle
- Value 4: Fox

The table shows the mapping between the value and the next value in the sequence.
Implementation

<table>
<thead>
<tr>
<th>value</th>
<th>Bear</th>
<th>Cow</th>
<th>Duck</th>
<th>Eagle</th>
<th>Fox</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>-1</td>
<td>1</td>
</tr>
</tbody>
</table>

- head → Cow → Eagle → Bear → Fox → Duck
Implementation

<table>
<thead>
<tr>
<th>next</th>
<th>value</th>
<th>Bear</th>
<th>Cow</th>
<th>Duck</th>
<th>Eagle</th>
<th>Fox</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>4</td>
<td></td>
<td>-1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

head → Cow → Eagle → Bear → Fox → Duck

<table>
<thead>
<tr>
<th>next</th>
<th>value</th>
<th>Bear</th>
<th>Cow</th>
<th>Duck</th>
<th>Eagle</th>
<th>Fox</th>
<th>Goat</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>-1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>4</td>
<td></td>
<td>-1</td>
<td>6</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

head → Cow → Eagle → Bear → Fox → Duck
# Linked list vs Array

<table>
<thead>
<tr>
<th>Time Complexities</th>
<th>Linked list</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert at the front</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Insert in the middle</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Insert at the back</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Find an element</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Find the $k^{th}$ element</td>
<td>$O(k)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
Linked lists

Singly linked list

Circularly linked list

Doubly linked list

XOR linked list

Cow  Eagle  Bear  Fox
### XOR linked list

<table>
<thead>
<tr>
<th></th>
<th>Singly linked list</th>
<th>Doubly linked list</th>
<th>XOR linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space complexity</td>
<td>(O(2N) = O(N))</td>
<td>(O(3N) = O(N))</td>
<td>(O(2N) = O(N))</td>
</tr>
<tr>
<td>Backward traversal</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Not necessary in HKOI, I won't teach this today.

Read the slides below by yourself.
XOR linked list

Recall:
'⊕' stands for XOR
0 ⊕ 0 = 0
0 ⊕ 1 = 1
1 ⊕ 0 = 1
1 ⊕ 1 = 0
A ⊕ B = B ⊕ A
A ⊕ (B ⊕ A) = B

A

value

<table>
<thead>
<tr>
<th></th>
<th>Bear</th>
<th>Cow</th>
<th>Duck</th>
<th>Eagle</th>
<th>Fox</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3 ⊕ 4</td>
<td>2 ⊕ 3</td>
<td>1 ⊕ 4</td>
<td>0 ⊕ 1</td>
<td>2 ⊕ 0</td>
</tr>
</tbody>
</table>

next

|   | 4 ⊕ 3| 3 ⊕ 2| 1 ⊕ 4| 0 ⊕ 1| 2 ⊕ 0|

prev

|   | 4 ⊕ 3| 3 ⊕ 2| 1 ⊕ 4| 0 ⊕ 1| 2 ⊕ 0|

Cow — Eagle — Bear — Fox — Duck

Cow — Eagle — Bear — Fox — Duck
XOR linked list - traverse forward

<table>
<thead>
<tr>
<th>value</th>
<th>Bear</th>
<th>Cow</th>
<th>Duck</th>
<th>Eagle</th>
<th>Fox</th>
</tr>
</thead>
<tbody>
<tr>
<td>next⊕prev</td>
<td>4 ⊕ 3</td>
<td>3 ⊕ 2</td>
<td>1 ⊕ 4</td>
<td>0 ⊕ 1</td>
<td>2 ⊕ 0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Cow ➔ Eagle ➔ Bear ➔ Fox ➔ Duck

1 ⊕ (0 ⊕ 1) = 0
XOR linked list - traverse forward

```
<table>
<thead>
<tr>
<th></th>
<th>Bear</th>
<th>Cow</th>
<th>Duck</th>
<th>Eagle</th>
<th>Fox</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
```

1 ⊕ (0 ⊕ 1) = 0
XOR linked list - traverse backward

\[ 4 \oplus (4 \oplus 3) = 3 \]
XOR linked list - traverse backward

\[ 4 \oplus (4 \oplus 3) = 3 \]

<table>
<thead>
<tr>
<th>value</th>
<th>Bear</th>
<th>Cow</th>
<th>Duck</th>
<th>Eagle</th>
<th>Fox</th>
</tr>
</thead>
<tbody>
<tr>
<td>next⊕prev</td>
<td>4 ⊕ 3</td>
<td>3 ⊕ 2</td>
<td>1 ⊕ 4</td>
<td>0 ⊕ 1</td>
<td>2 ⊕ 0</td>
</tr>
</tbody>
</table>

Cow → Eagle → Bear → Fox → Duck

Cow → Eagle → Bear → Fox → Duck

\[ 4 \oplus (4 \oplus 3) = 3 \]
XOR linked list - rotate segment

<table>
<thead>
<tr>
<th>value</th>
<th>Bear</th>
<th>Cow</th>
<th>Duck</th>
<th>Eagle</th>
<th>Fox</th>
</tr>
</thead>
<tbody>
<tr>
<td>next⊕prev</td>
<td>4 ⊕ 3</td>
<td>3 ⊕ 2</td>
<td>1 ⊕ 4</td>
<td>0 ⊕ 1</td>
<td>2 ⊕ 0</td>
</tr>
</tbody>
</table>

Cow → Eagle → Bear → Fox → Duck
### XOR linked list - rotate segment

<table>
<thead>
<tr>
<th>value</th>
<th>Bear</th>
<th>Cow</th>
<th>Duck</th>
<th>Eagle</th>
<th>Fox</th>
</tr>
</thead>
<tbody>
<tr>
<td>next⊕prev</td>
<td>4 ⊕ 3</td>
<td>3 ⊕ 2</td>
<td>1 ⊕ 4</td>
<td>0 ⊕ 1</td>
<td>2 ⊕ 0</td>
</tr>
</tbody>
</table>

0 1 2 3 4 5 6 7

Cow ➞ Eagle ➞ Bear ➞ Fox ➞ Duck

Only the two end points were changed!

<table>
<thead>
<tr>
<th>value</th>
<th>Bear</th>
<th>Cow</th>
<th>Duck</th>
<th>Eagle</th>
<th>Fox</th>
</tr>
</thead>
<tbody>
<tr>
<td>next⊕prev</td>
<td>3 ⊕ 4</td>
<td>3 ⊕ 2</td>
<td>1 ⊕ 4</td>
<td>2 ⊕ 0</td>
<td>0 ⊕ 1</td>
</tr>
</tbody>
</table>

0 1 2 3 4 5 6 7

Cow ➞ Fox ➞ Bear ➞ Eagle ➞ Duck

Only the two end points were changed!
# XOR linked list

<table>
<thead>
<tr>
<th></th>
<th>Singly linked list</th>
<th>Doubly linked list</th>
<th>XOR linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space complexity</strong></td>
<td>(O(2N) = O(N))</td>
<td>(O(3N) = O(N))</td>
<td>(O(2N) = O(N))</td>
</tr>
<tr>
<td><strong>Backward traversal</strong></td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Rotate segment</strong></td>
<td>(O(N))</td>
<td>(O(N))</td>
<td>(O(1))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Unrolled linked list</th>
<th>Splay tree</th>
<th>XOR linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Locate segment</strong></td>
<td>(O(N))</td>
<td>(O(\log(N)))</td>
<td>(O(N))</td>
</tr>
<tr>
<td><strong>Rotate segment</strong></td>
<td>(O(\sqrt{N}))</td>
<td>(O(\log(N)))</td>
<td>(O(1))</td>
</tr>
<tr>
<td><strong>Difficulty</strong></td>
<td>medium-hard</td>
<td>hard</td>
<td>easy</td>
</tr>
</tbody>
</table>
Summary

Abstract Data Types - implemented by data structures

- Stack - LIFO
- Queue - FIFO

Data structures

- Array
- Linked List
  - Singly linked list
  - Doubly linked list
  - Circular linked list
  - XOR linked list
More practices

- HKOJ P004 Broken Keyboard
- HKOJ P006 Largest Rectangle in Histogram
- HKOJ M1803 I love you I love you
- HKOJ M1313 Bookstack