Data Structures (I)

Susanna Chan

2021-3-20

香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

Contents

- An introduction
- Queue
- Stack
- Deque
- Monotonic queue / stack
- Linked List

Data Types, Data Structures and Abstract Data Types

The data structure implements the physical form of the data type.

The ADT defines the logical form of the data type.

What are data structures?

A particular way of organizing data so that they can be used more efficiently.

Major operations on ADTs

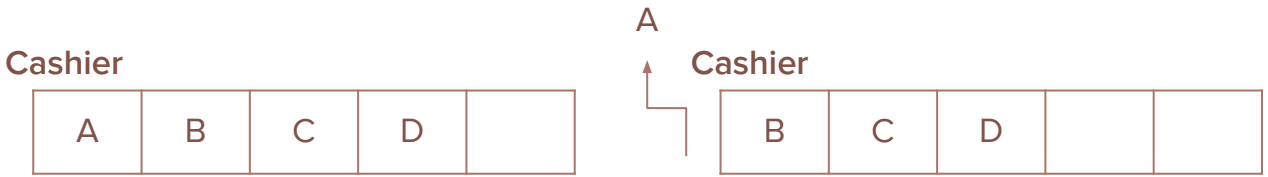
Insertion | Deletion | Modification | Query

Queue

Idea

Just like how we line up to buy things in the supermarket:

- Stand at the back of the queue to line up (push)
- Process the front of the queue when the cashier is ready (query)
- Served customers leave the queue, and the remaining ones move forward in the queue (pop)



Queue

First-In-First-Out (FIFO)

✓ Push (Enqueue) to the back

✓ Pop (Dequeue) from the front

X Push to the front / in between two elements

X Pop from the back / any element between two elements

Implementation - C++ STL queue

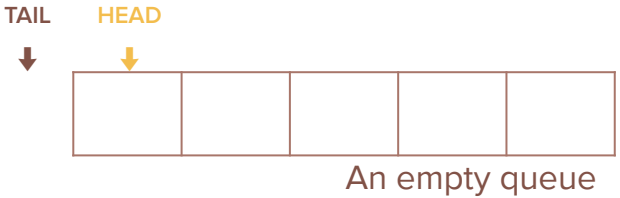
read more

```cpp
#include<queue>

queue<int> q;

q.push(10);
q.pop();
int x = q.front();
if (q.size() != 0) ...
if (!q.empty()) ...
```

Implementation - Array

You need:

- An array
- HEAD
- TAIL



An empty queue

```
const int N = 5;
string queue[N];
int head = 0, tail = -1;
```

Implementation - Array

push operation (if TAIL + 1 **does not exceed** size of the array)

Before:


```
void push(string s) {
  if (queue_full()) {
    cout << "The queue is full!\n";
    return;
  }
  tail++;
  queue[tail] = s;
}

...
push(“Amy”);
```

Implementation - Array

push operation (if TAIL + 1 **does not exceed** size of the array)

After:


```
void push(string s) {
  if (queue_full()) {
    cout << "The queue is full!\n";
    return;
  }
  tail++;
  queue[tail] = s;
}

...
push("Amy");
```

Implementation - Array

`queue_full` checking - useful for push operations

```
bool queue_full() {
  return tail + 1 >= N;
}
```

Implementation - Array

pop operation (If the queue is **not empty**)

Before:


```
void pop() {
  if (!queue_empty()) {
    cout << "The queue is empty!\n";
    return;
  }
  head++;
}

...
pop();
```

Implementation - Array

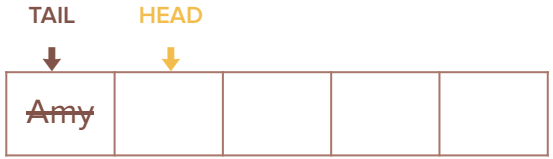
pop operation (If the queue is **not empty**)

After:


```
void pop() {
  if (!queue_empty()) {
    cout << "The queue is empty!\n";
    return;
  }
  head++;
}

...
pop();
```

Implementation - Array

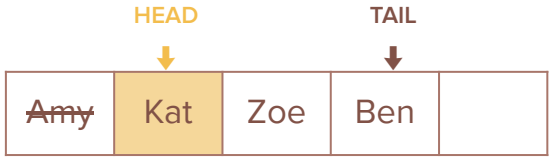
`queue_empty` checking - useful for pop and front operations




```
bool queue_empty() {
  return tail < head;
}
```

Implementation - Array

`front` operation (If the queue is **not empty**)

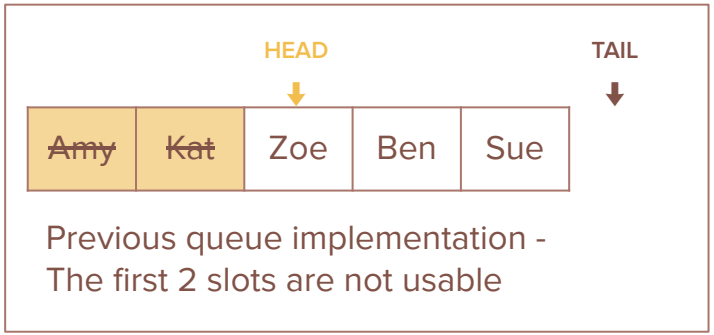



```
string front() {
  if (!queue_empty()) {
    cout << "The queue is empty!\n";
    return;
  }
  return queue[head];
}

...
front();
```

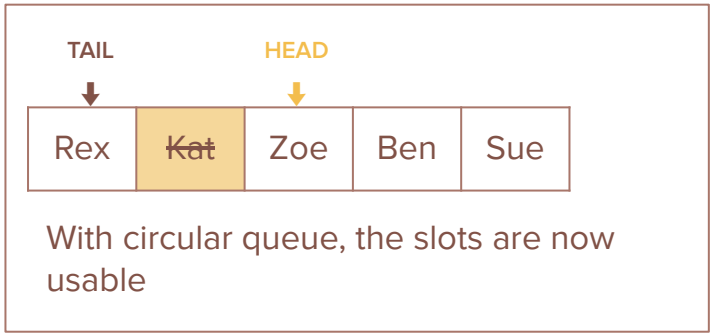
香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

Circular Queue

When TAIL / HEAD exceeds the size of the array, use the slots at the beginning of the array so that space is not wasted.



Previous queue implementation -
The first 2 slots are not usable



With circular queue, the slots are now usable

Circular Queue

To shift the HEAD/ TAIL to the beginning of the array, we can write a if-statement to check if the array boundary is reached.

It is more common to write this in one line.
0-based array: `tail = (tail + 1) % N`
1-based array: `tail = tail % N + 1`
To learn more about modular arithmetic, read Math in OI (I)

Circular Queue

push operation (If the queue is **not full**)

Before:


```
void push(string s) {
  if (queue_full()) {
    cout << "The queue is full!\n";
    return;
  }
  queue[tail] = s;
  tail = (tail + 1) % N;
}

...
push("Sue");
push("Rex");
```

香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

Circular Queue

push operation (If the queue is **not full**)

After push("Sue"):

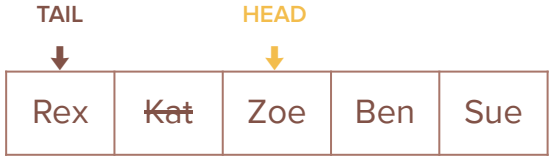

```
void push(string s) {
  if (queue_full()) {
    cout << "The queue is full!\n";
    return;
  }
  queue[tail] = s;
  tail = (tail + 1) % N;
}

...
push("Sue");
push("Rex");
```

香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

Circular Queue

push operation (If the queue is **not full**)

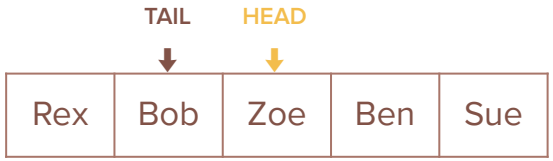
After push("Rex"):




```
void push(string s) {
  if (queue_full()) {
    cout << "The queue is full!\n";
    return;
  }
  queue[tail] = s;
  tail = (tail + 1) % N;
}

...
push("Sue");
push("Rex");
```

Circular Queue

`queue_full` checking for circular queues is not simply checking the bound of the array




```
bool queue_full() {
  return (tail + 1) % N == head;
}
```

Circular Queue

pop operation (If the queue is **not empty**)

Before:


```
void pop() {
  if (!queue_empty()) {
    cout << "The queue is empty!\n";
    return;
  }
  head = (head + 1) % N;
}

...
pop();
```

香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

Circular Queue

pop operation (If the queue is **not empty**)

After:


```
void pop() {
  if (!queue_empty()) {
    cout << "The queue is empty!\n";
    return;
  }
  head = (head + 1) % N;
}

...
pop();
```

香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

Circular Queue

`queue_empty` checking for circular queues is not simply comparing head and tail

Implementation is left as an exercise - you might need to change the implementation for other operations

```
bool queue_empty() {
  ------------------
}
```

Circular queue

:o) Save memory

:o(When the queue is full ➔ error / data loss (depends on implementation)

Time Complexity

Push operation: $O(1)$

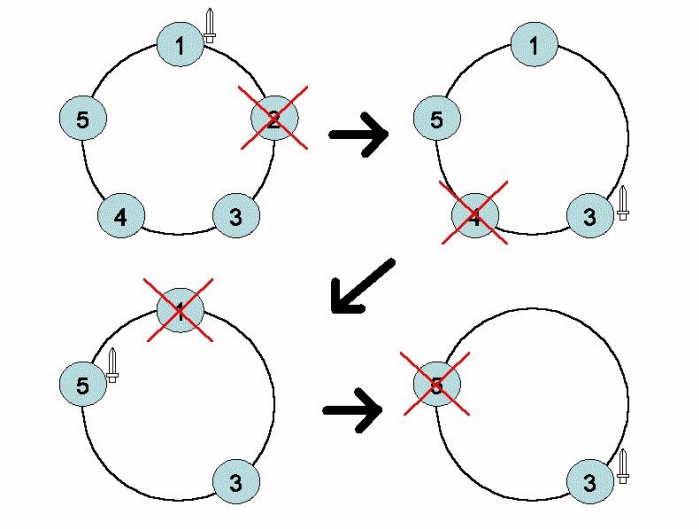
Pop operation: $O(1)$

Front operation: $O(1)$

Application - The Josephus Problem

- $1 \leq N \leq 1000$ soldiers arranged in a circle
- Soldier 1 is holding a sword initially
- The one holding a sword will:
  a. kill the survivor on his left
  b. pass the sword to the survivor on his left

Who is the final survivor?



Idea

Since the processed soldiers are always adjacent and in the same order, we can imagine the process to be done in a line.

Soldiers are waiting to be ‘processed’ one by one, starting from the one with index 1. Repeatedly, we make the soldier at the front call the next soldier and kill him; after that this surviving soldier needs to be processed again, so he has to ‘line up’ again at the back.

It is obvious that this simulation could be done by a queue.

Application - The Josephus Problem

```
push 1 to N into a queue.
repeat N - 1 times
    record the front element (x)
    pop twice
    push x into the queue
return the remaining element in the queue
```

Application - The Josephus Problem

▨ = HEAD
▨ = TAIL

Let's say N=5. Here we use a queue of fixed size 10.

Initial	1	2	3	4	5					
Kill #1			3	4	5	1				
Kill #2					5	1	3			
Kill #3							3	5		
Kill #4									3	

A lot of memory is wasted :(

Application - The Josephus Problem

Circular queue of fixed size 5.

Initial	1	2	3	4	5
Kill #1	1		3	4	5
Kill #2	1	3			5
Kill #3		3	5		
Kill #4				3	

▨ = HEAD
▨ = TAIL

Better in terms of memory consumption

More Applications

Breadth-First-Search (BFS) ➔ Graph (I)

Shortest Path Faster Algorithm (SPFA) ➔ Graph (II)

Stack

Idea

Analogy: cereals on the selves in the supermarket

- The staff put the cereals starting from the deep end of the shelf (push)
- You only see the lastly placed cereal (query)
- You take the cereal closest to you from the shelf

Stack

Last-In-First-Out (LIFO)

✓ Push to the back

✓ Pop from the back

X Push to the front / in between two elements

X Pop from the front / any element between two elements

Implementation - C++ STL stack

read more

```cpp
#include<stack>
stack<int> s;

s.push(10);

s.pop();

int x = s.top();

int size = s.size();
if (!s.empty()) ...
```

Implementation - Array

You need:

- An array
- TOP

An empty stack

```
const int N = 5;
string stack[N];
int top = -1;
```

Implementation - Array

push operation (if TOP+1 **does not exceed** size of the array)

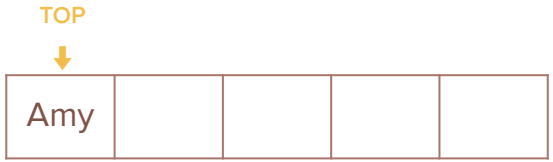
Before:


```
void push(string s) {
  if (top + 1 >= N) {
    cout << "The stack is full!\n";
    return;
  }
  top++;
  stack[top] = s;
}

...
push("Amy");
```

Implementation - Array

push operation (if TOP+1 **does not exceed** size of the array)

After:




```
void push(string s) {
  if (top + 1 >= N) {
    cout << "The stack is full!\n";
    return;
  }
  top++;
  stack[top] = s;
}

...
push("Amy");
```

Implementation - Array

pop operation (If the stack is **not empty**)

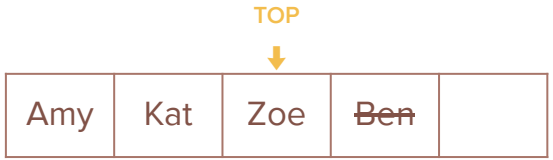
Before:


```
void pop() {
 if (stack_empty()) {
    cout << "The stack is empty!\n";
    return;
  }
  top--;
}

...
pop();
```

Implementation - Array

pop operation (If the stack is **not empty**)

After:




```
void pop() {
 if (stack_empty()) {
    cout << "The stack is empty!\n";
    return;
  }
  top--;
}

...
pop();
```

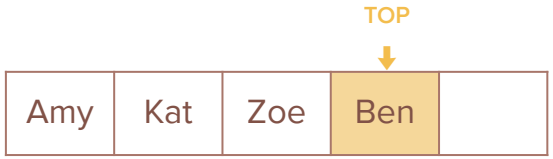
Implementation - Array

`stack_empty` checking - useful for pop and top operations


```
void stack_empty() {
  return top == -1;
}
```

Implementation - Array

`top` operation (If the stack is **not empty**)




```
int get_top() {
  if (stack_empty()) {
    cout << "The stack is empty!\n";
    return -1;
  }
  return stack[top];
}
```

Application - Parentheses Balance

Determine if a string (length = N, $1 \leq N \leq 10000$) parentheses is balanced.

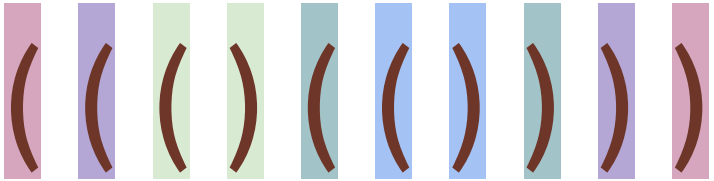
For example,

{ ( ) [ ] } ➔ True

{ ( ] [ ) } ➔ False

Idea

Given ((( ) (( )))), how do you pair up the parentheses to check if it is balanced?



If we scan the sequence from left to right, we may observe that the latest open bracket (last in) always pairs with the closest closed bracket (first out) ➔ stack

Application - Parentheses Balance

maintain a stack

iterate the sequence of parentheses

- open parentheses (/ [/ { ➔ push
- matching close parentheses) /] / } ➔ pop
- otherwise ➔ return false

(if the current element =], the top element in the stack has to be [to match)

if the stack is empty ➔ return true

Time complexity: O(N) Space complexity: O(N)

Application - Parentheses Balance

{()[]}	{				push {
{()[]}	{	(			push (
{()[]}	{				match + pop (
{()[]}	{	[			push [
{()[]}	{				match + pop]
{()[]}					match + pop }

Final stack is empty ➔ balanced!

Application - Parentheses Balance

Another example:

{(][)}	{				push {
{(][)}	{	(			push (
{(][)}	{	(			**unmatch!**

Unbalanced...

Application - Arithmetic Expressions

Prefix

Operators before operands; e.g. + 1 * 2 3 = 7 ; - + * 2 3 / 5 4 9 = -1.75

Infix

What we usually see; might need parenthesis; e.g. 1 + 2 * 3 = 7 ; 2 * 3 + 5 / 4 - 9 = 6 + 1.25 - 9 = -1.75

Postfix

Operands before operators; e.g. 1 2 3 * + = 7, 2 3 * 5 4 / + 9 - = -1.75

Here we assume all operands are single-digit numbers (0 - 9)

Application - Arithmetic Expressions

How do we use stack to handle the expressions?

Prefix:

Push all elements to the stack one by one, perform arithmetic on the first 3 elements whenever it's possible to do so

Postfix:

Similar approach could be used

Infix:

Maintain 2 stacks (1) operand (2) operator

Application - Arithmetic Expressions

+ * 2 3 - 4 1	+				push(+)
+ * 2 3 - 4 1	+	*			push(*)
+ * 2 3 - 4 1	+	*	2		push(2)
+ * 2 3 - 4 1	+	6			num1=pop()=2, num2=3, op=pop()=* push(3*2)
+ * 2 3 - 4 1	+	6	-		push(-)
+ * 2 3 - 4 1	+	6	-	4	push(4)
+ * 2 3 - 4 1	9				num1=pop()=4, num2=1, op=pop()=- push(4-1) num1=pop()=3, num2=6, op=pop()=+ push(3+6)

More Applications

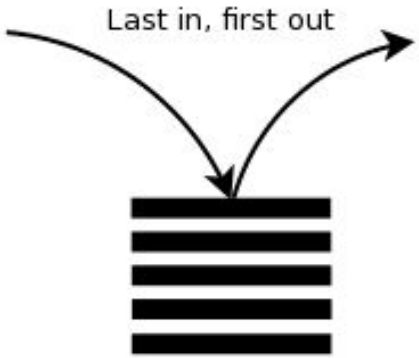
Recursion ➔ Recursion, Divide and Conquer

- Exhaust permutations
- Exhaust combinations

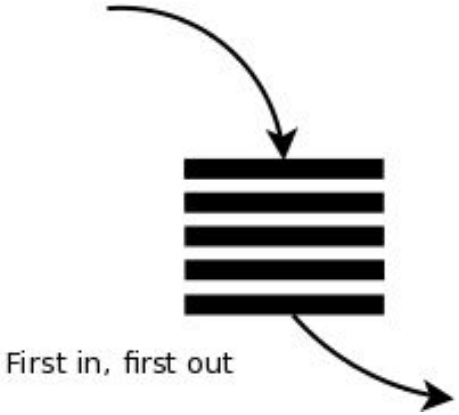
Depth-First-Search ➔ Graph (I)

Stack vs Queue

Stack:



Queue:



Deque

Deque

Double-Ended Queue

Something like stack + queue

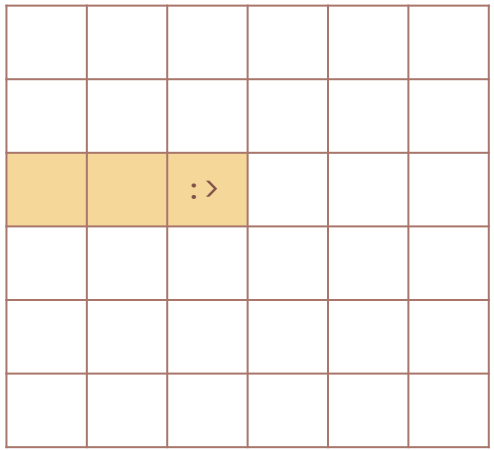
Can push / pop elements at the front / back efficiently

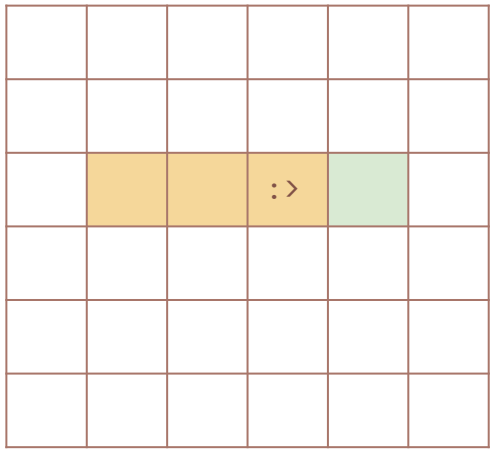
Implementation - C++ STL deque

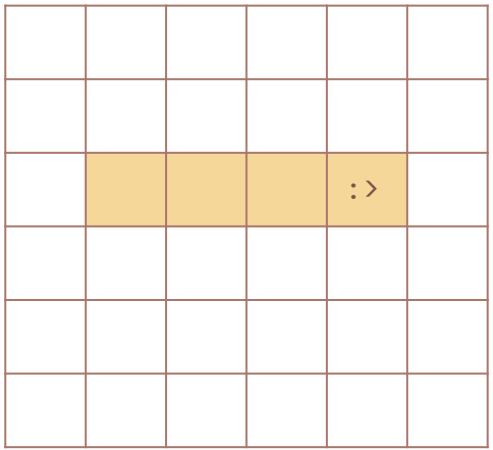
read more

```cpp
#include<deque>
deque<int> dq;

dq.push_front(10);
dq.push_back(20);

dq.pop_front();
dq.pop_back();

int x = dq.front();
int y = dq.back();

int size = dq.size();
if (!dq.empty()) ...
```

Application: Snake Game (search ‘snake game’ on Google to play it)







Application: Snake Game

Suppose we store a list of coordinates of the snake in this way `[(2, 2), (2, 1), (2, 0)]`

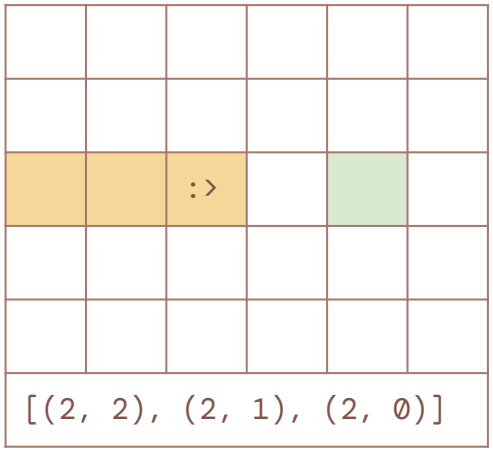
Problem: How do we update the coordinates efficiently as the snake moves / grows?

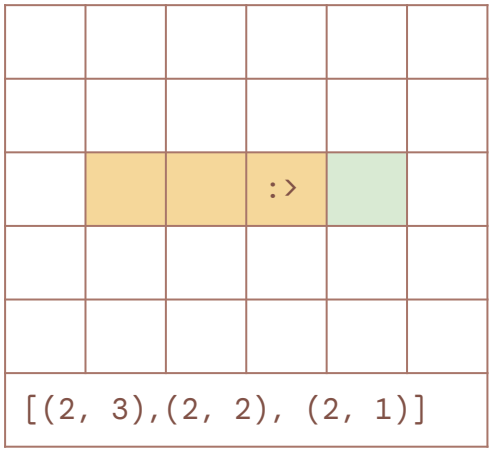
Move rule:

- The snake's head is moved to the new position, adjacent to the old head position
- The snake's tail is shifted to its previous second last position

Application: Snake Game

Move rule illustration :





Application: Snake Game

Suppose we store a list of coordinates of the snake in this way `[(2, 2), (2, 1), (2, 0)]`

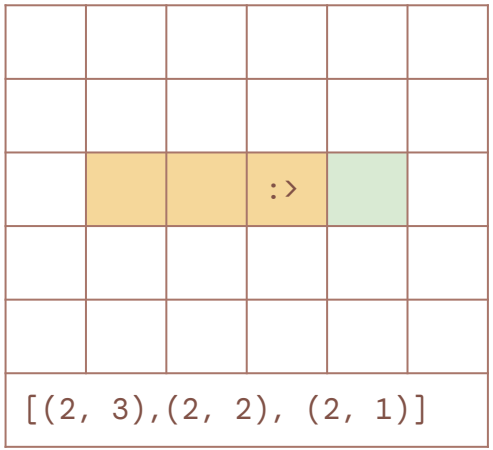
Problem: How do we update the coordinates efficiently as the snake moves / grows?

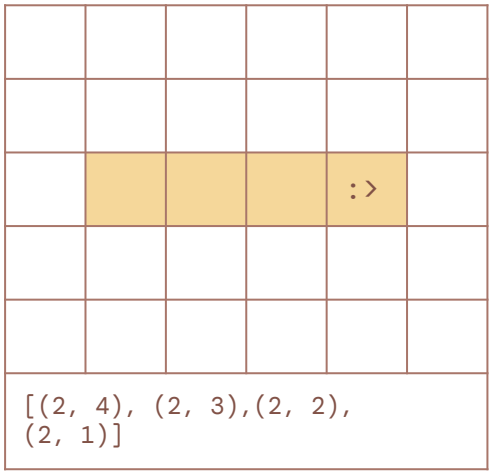
Grow rule:

- The snake's head is moved to the food's position, so the snake's length is grown by 1 unit

Application: Snake Game

Grow rule illustration:





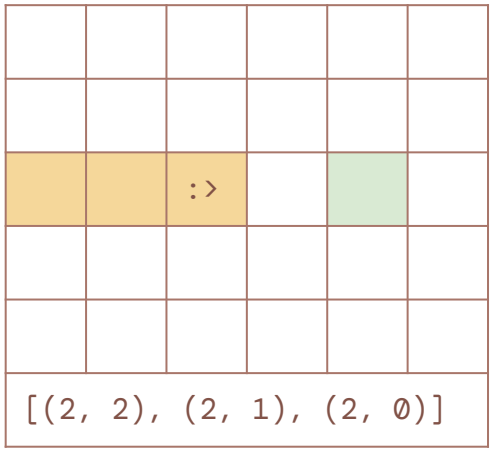
Application: Snake Game

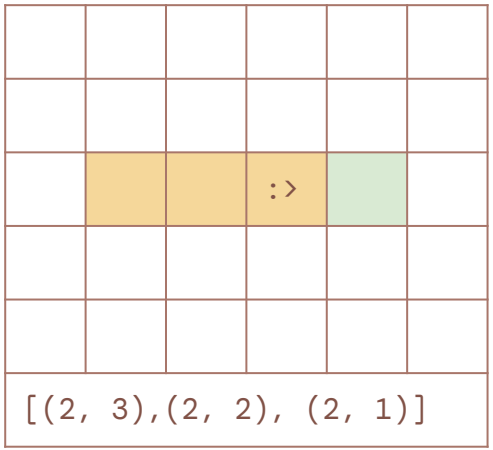
Store it in a deque

- Coordinates of the head of the snake as head of the deque, followed by the coordinates of the body parts
- Push the new head to the front / pop the old tail according to the rules

Application: Snake Game

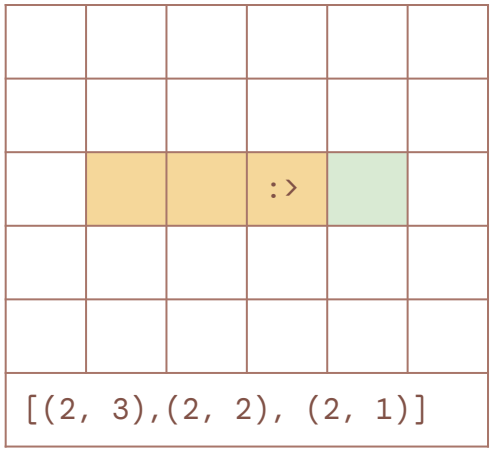
```
snake_move()
    some checking and some calculations on the next coordinates...
    (nx, ny) ← next coordinate of the snake head
    deque.push_front((nx, ny))
    deque.pop_back()
```

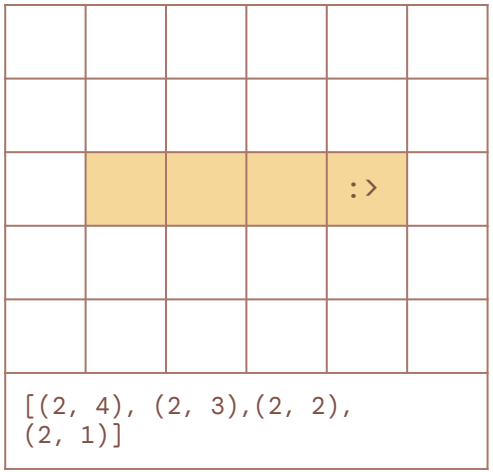




Application: Snake Game

```
snake_grow()
    (nx, ny) ← coordinate of the food eaten
    deque.push_front((nx, ny))
```

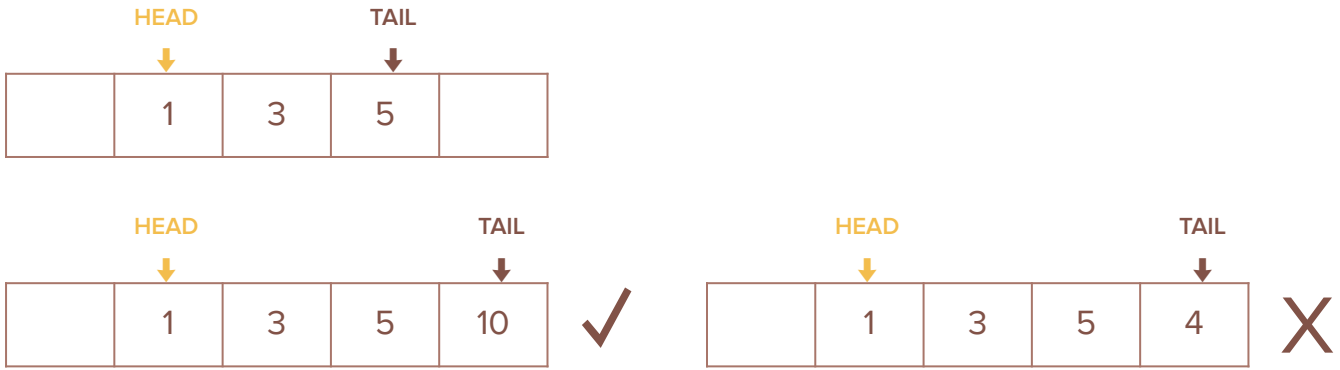




Monotonic Queue / Stack

Monotonic queue (deque)

A deque with all elements following a certain order (increasing/ decreasing)

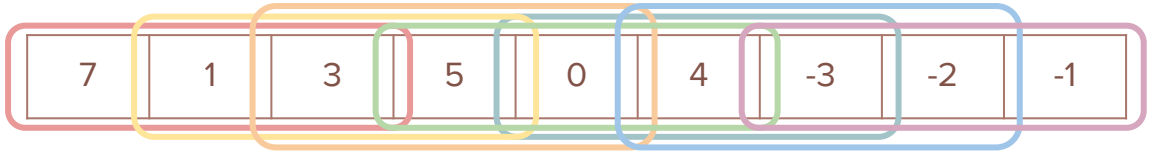


Application: Sliding window maximum

Problem: Given a integer array A, there is a sliding window of size k that slides from the beginning to the end of the array.

For simplicity, assume the elements in the array are distinct.

Find the maximum element in the sliding window for every window in A.



Assume k = 3, the results should be: 7, 5, 5, 5, 4, 4, -1

Application: Sliding window maximum

Queue / two-pointer implementation:

- The queue keeps track of the elements in the sliding window.
- For every window, finding the max element using linear scan takes $O(k)$ ➔ not good enough
- What if I use a heap? (Data Structures II) ➔ $O(\log k)$ ➔ still not good enough

Monotonic queue (deque) implementation:

- The deque keeps track of some of the elements in the sliding window following a decreasing order.
- We can find the maximum element of a window in constant time ($O(1)$)

Idea

For sure we want to keep the maximum element within the window, but does that mean we should forget the remaining elements in the same window?

Example: A = [10, 1, 3, 2] and k = 3

The max element in the first window is 10. If we ignore the other elements, we would not get 3 as the max unless we iterate through the entire window.

Idea

If we focus in the first window [10, 1, 3], we can already confirm that 1 would not be a candidate for the following windows.

Why?

In any window containing 1, it must either contain 10 or 3, which are better candidates than 1 for a window's maximum.

Maybe we can keep track of the possible candidates in a window to find the maximum value.

Application: Sliding window maximum

Maintain a deque D

for each element A_i in array A do

    if D.front() == A_{i-k} then D.pop_front()

    while D.back() < A_i do D.pop_back()

    D.push_back(A_i)

    if i + 1 >= k then output D.front()

Application: Sliding window maximum

7	3	1	5	0	4	-3	-2	-1

k = 3

▨ = sliding window max

(1)	7			push back (7)
(2)	7	3		push back (7)
(3)	7	3	1	push back (1)
(4a)	3	1		pop front (7 not in the window)
(4b)	3			pop back (1 impossible to be max)

(4c)				Pop back (3 impossible to be max)
(4d)	5			push back (5)
(5)	5	0		push back (0)
(6a)	5			pop back (0 impossible to be max)
(6b)	5	4		push back (4)

Application: Sliding window maximum

7	3	1	5	0	4	-3	-2	-1

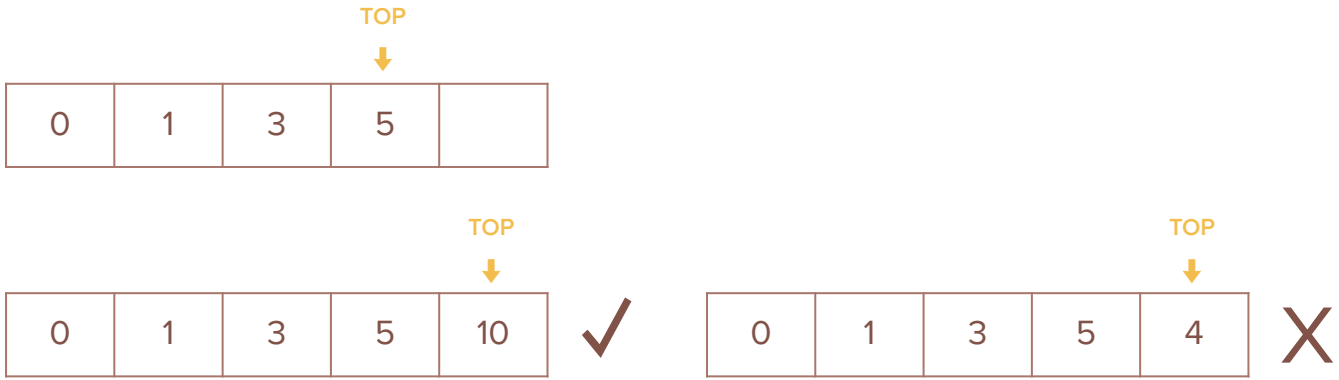
k = 3

(7a)	4			pop front (5 not in the window)
(7b)	4	-3		push back (-3)
(8a)	4			pop back (-3 impossible to be max)
(8b)	4	-2		push back (-2)
(9a)	-2			pop front (4 not in the window)

(9b)				pop back (-2 impossible to be max)
(9c)	-1			push back (-1)

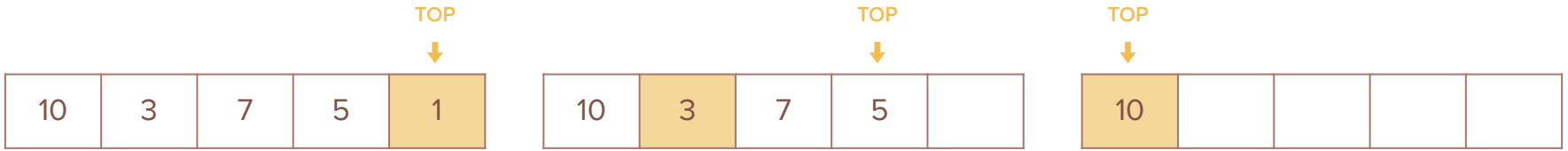
Monotonic Stack

A stack with all elements following a certain order (increasing/ decreasing)



Application: Min Stack

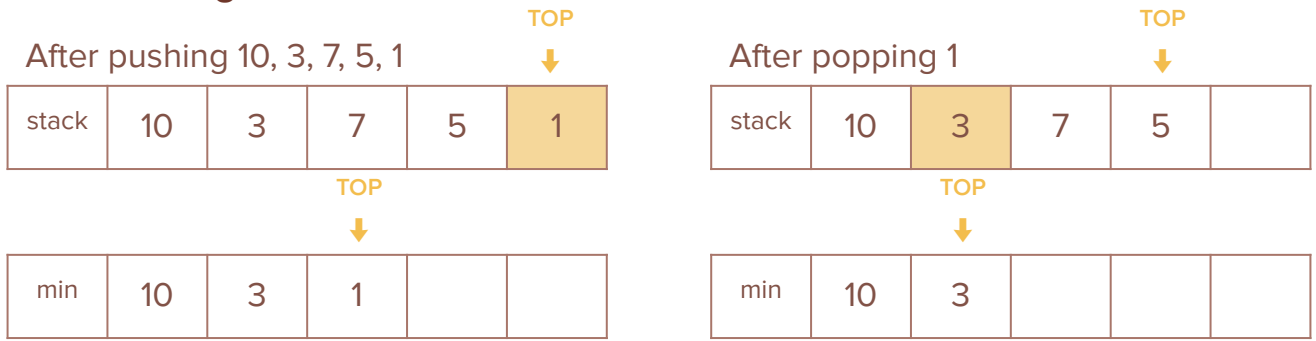
Implement a stack that supports an additional type of query: `min()` which returns the minimum element in the stack (for simplicity we assume the elements are unique)



Application: Min Stack

Similar idea as the sliding window maximum problem - use an extra stack to maintain the minimum element + the candidate min elements after popping the current minimum.

In this way, a decreasing stack is maintained.



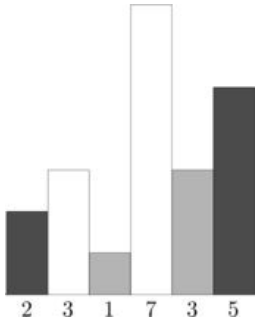
Application: Min Stack

```
maintain stack S and stack Min
push(x):
    S.push(x)
    if Min.top() > x do Min.push(x)
pop():
    x = S.top()
    S.pop()
    if Min.top() == x do Min.pop()
min():
    return Min.top()
```

Application - Largest Rectangle in Histogram

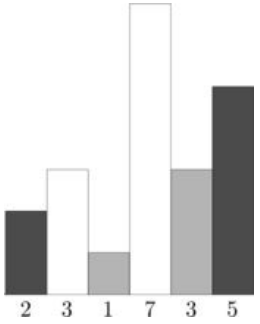
Given a histogram with n bars, each of height h_i.

Find the area of the largest rectangle in the histogram.

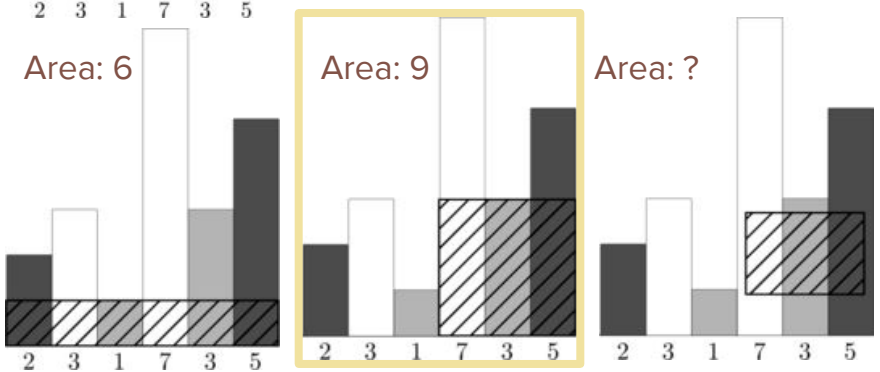


Application - Largest Rectangle in Histogram

For example we have this histogram:



Here are some possible rectangles:



Idea

Key observation: the height of the largest rectangle must be equal to one the bars in the histogram.

The question is: what is the width of the rectangle?

When we fix one bar and treat it as the height of the rectangle (h), we can expand our rectangle if the height of the bars adjacent to the rectangle >= h

The question now becomes: for each bar, find the number of bars with height >= its height so that they can stick together as a rectangle.

Idea

For the i-th bar, we want to find the leftmost bar and the rightmost bar that are lower than h_i

For the rightmost bar - as we scan the bars from left to right, we can include all bars after bar i and stop once we see a bar of height < h_i on the right

For the leftmost bar - we use a monotonic stack to store the bars shorter than bar i

Since we are scanning the bars from the left, the indices are obviously in increasing order. What we want is to store indices of the bars in increasing heights (indices at the top of the stack have higher bars), while all the heights in the stack are shorter than bar i.

Application - Largest Rectangle in Histogram

```
Maintain a stack S and maxArea = 0
S.push(0) // dummy bar
for every bar i do
  while S not empty AND h[i] < h[S.top()] do
    height = h[S.top()] // calculate max area including bar of index S.top()
    S.pop()
    left = S.empty() ? 0 : S.top() + 1
    maxArea = max(maxArea, height * (i - left))
  S.push(i)
return maxArea
```

More Applications

Dynamic programming optimizations (Dynamic Programming (III))

Linked List

Singly-linked List

✓ Insert an element in any position

✓ Erase an element in any position

✓ Access the first element directly

X Access the nth element directly (except the first element)

Implementation - C++ STL list

Read more

```cpp
#include<list>
list<int> li;
li.push_back(2);
li.push_front(4);
li.push_front(9);
li.insert(++li.begin(), 3);

auto it = li.begin();
advance(it, 2);
li.erase(it);
li.pop_back();
li.pop_front();
li.remove(450);

int x = li.size();
if (li.empty()) ...
```

Implementation - Array

You need:

- Two arrays

  One stores the data (A), another one stores index of the next node in A (NEXT)

- HEAD
- SIZE

A					
NEXT					

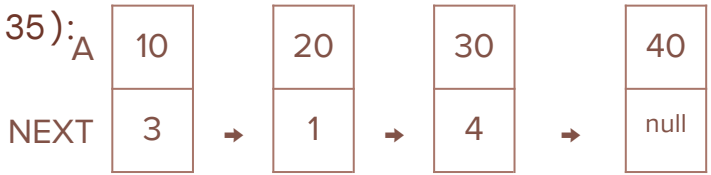
```
const int N = 5;
int A[N], Next[N];
int head = 0, size = 0;
```

Implementation - Array

Insert new node after x^{th} node (`x < size`)

Connect the previous node and the next node with the new node

Before `insert(3, 35)`:



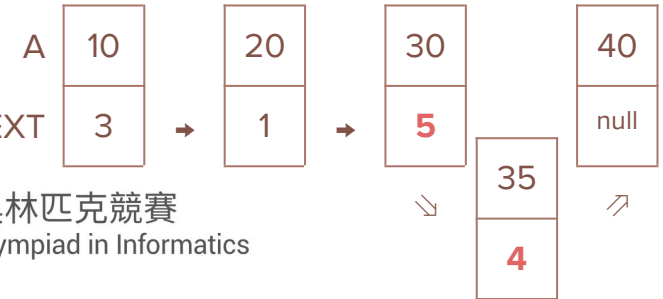

```
void insert(int x, int val) {
  if (x >= size) { // invalid insertion
    return;
  }
  size++;
  // get unoccupied slot in the array
  int slot = get_available_slot();
  A[slot] = val;
  if (x == -1) { // insert to the front
    Next[slot] = head;
    head = slot;
  }
  else { // insert after x-th node
    int cur = head;
    for (int i = 0; i < x; i++) {
      cur = Next[cur];
    }
    Next[slot] = Next[cur];
    Next[cur] = slot;
  }
}
```

Implementation - Array

Insert new node after x^{th} node (`x < size`)

Connect the previous node and the next node with the new node

After `insert(3, 35)`:



香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

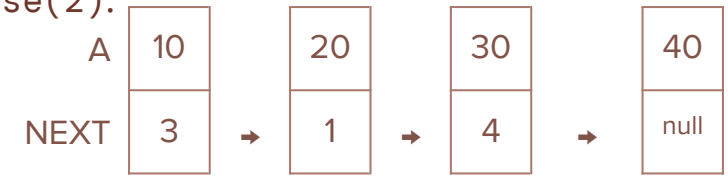
```
void insert(int x, int val) {
  if (x >= size) { // invalid insertion
    return;
  }
  size++;
  // get unoccupied slot in the array
  int slot = get_available_slot();
  A[slot] = val;
  if (x == -1) { // insert to the front
    Next[slot] = head;
    head = slot;
  }
  else { // insert after x-th node
    int cur = head;
    for (int i = 0; i < x; i++) {
      cur = Next[cur];
    }
    Next[slot] = Next[cur];
    Next[cur] = slot;
  }
}
```

Implementation - Array

`Erase` operation (erase the x^{th} node)

Connect the previous node and the next node

Before `erase(2)`:



香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

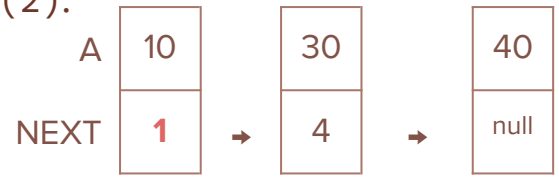
```
void erase(int x) {
  if (x >= size) {
    return;
  }
  size--;
  int xth_node;
  if (x == 0) { // delete head
    xth_node = head;
    head = Next[head];
  }
  else { // delete x-th node
    int prev = head;
    for (int i = 0; i < x - 1; i++) {
      prev = Next[prev];
    }
    xth_node = Next[prev];
    Next[prev] = Next[xth_node];

  }
  // make slot available for insertion
  make_slot_available(xth_node);
}
```

Implementation - Array

`Erase` operation (erase the x^{th} node)

Connect the previous node and the next node

After `erase(2)`:



香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

```
void erase(int x) {
  if (x >= size) {
    return;
  }
  size--;
  int xth_node;
  if (x == 0) { // delete head
    xth_node = head;
    head = Next[head];
  }
  else { // delete x-th node
    int prev = head;
    for (int i = 0; i < x - 1; i++) {
      prev = Next[prev];
    }
    xth_node = Next[prev];
    Next[prev] = Next[xth_node];

  }
  // make slot available for insertion
  make_slot_available(xth_node);
}
```

Implementation - Array

`query` the x^{th} node in the list (if `x < size`)

From HEAD, traverse the linked list one by one until the target element is found.

```
int query(int x) {
  if (x >= size) {
    return -1;
  }
  int cur = head;
  for (int i = 0; i < x; i++) {
    cur = Next[cur];
  }
  return A[cur];
}
```

Implementation - Array

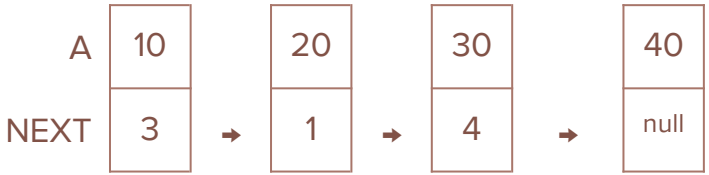
Need to find available slots in the array to store the new values

The list is in a consecutive block of memory - quite restrictive

Pointer approach - achieve dynamic allocation of memory

Implementation - Pointers

Linked list preparation




```
struct Node {
  int data;
  struct Node* next;
  Node(int x): data(x), next(NULL) {}
};

Node* head = NULL;
int li_size = 0;
```

Implementation - Pointers

Insert new node after x^{th} node (`x < size`)

Connect the previous node and the next node with the new node

```
void pointer_insert(int x, int val) {
  if (x >= li_size) {
    return;
  }
  li_size++;
  Node* new_node = new Node(val);
  if (x == -1) {
    new_node->next = head;
    head = new_node;
  }
  else {
    Node* cur = head;
    for (int i = 0; i < x; i++) {
      cur = cur->next;
    }
    new_node->next = cur->next;
    cur->next = new_node;
  }
}
```

Implementation - Pointers

`Erase` operation (erase the x^{th} node)

Connect the previous node and the next node

```
void pointer_erase(int x) {
  if (x >= li_size) {
    return;
  }
  li_size--;
  Node* xth_node;
  if (x == 0) {
    xth_node = head;
    head = xth_node->next;
  }
  else {
    Node* prev = head;
    for (int i = 0; i < x - 1; i++) {
      prev = prev->next;
    }
    xth_node = prev->next;
    prev->next = xth_node->next;
  }
  delete xth_node;
}
```

香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

Implementation - Pointers

`query` the x^{th} node in the list (if `x < size`)

From HEAD, traverse the linked list one by one until the target element is found.

Implementation is left as exercise - similar to array implementation

```
int query(int x) {
  ------------------
}
```

香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

Time complexity

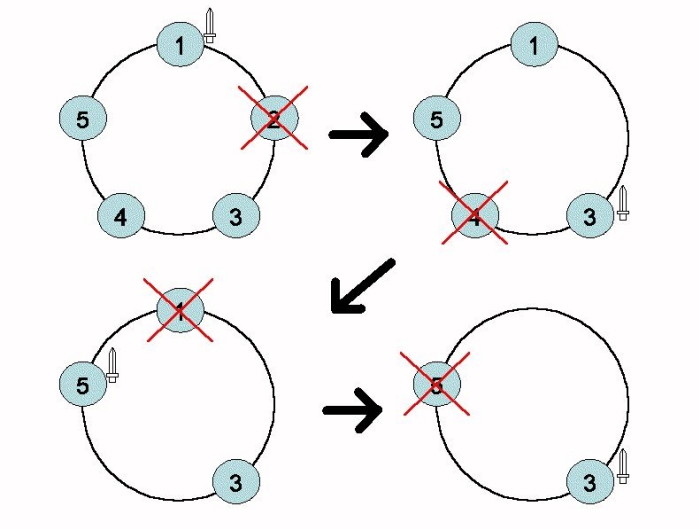
Insertion at the front	$O(1)$
Insertion in general	$O(n)$
Deletion at the front	$O(1)$
Deletion in general	$O(n)$
Find an element	$O(n)$

where n is the size of the linked list

Application - The Josephus Problem

- $1 \leq N \leq 1000$ soldiers arranged in a circle
- Soldier 1 is holding a sword initially
- The one holding a sword will:
  a. kill the survivor on his left
  b. pass the sword to the survivor on his left

Who is the final survivor?



Application - The Josephus Problem

The linked list approach might be more intuitive - every soldier is adjacent to the next surviving soldier.

Here we maintain a linked list NEXT, where NEXT[i] stores index of the soldier standing next to soldier *i*

When we kill one soldier, remove him from the list and pass the sword to the soldier currently standing next to him.

Continue this process for (N - 1) times.

Time complexity: O(N) Space complexity: O(N)

Application - The Josephus Problem

Let's say N=5. Here we use a queue of fixed size 10.

Initial	2	3	4	5	1
Kill #1	3	NULL	4	5	1
Kill #2	3	NULL	5	NULL	1
Kill #3	NULL	NULL	5	NULL	3
Kill #4	NULL	NULL	5	NULL	NULL

We don't need an element array since the element in the i^{th} position is i itself

More linked lists...

Doubly-linked list

Circular-linked list

XOR linked lists

Refer to Training Materials 2018 on Data Structures (I)

More Applications

Adjacency list - Graph (I)

Practice Problems

Stack / Queue

HKOJ P005 Rails

HKOJ 01017 Car Sorter

HKOJ M1721 Bus Fare II

HKOJ 32462 Largest Rectangle in Histogram

HKOJ M1803 I love you I love you

HKOJ 01015 Parentheses Balance

HKOJ 01033 Simple Arithmetic

HKOJ M1313 Bookstack

HKOI NP1712 時間複雜度

Linked Lists

HKOJ 01030 The Josephus Problem

HKOJ 31988 Broken Keyboard

CF 797C Minimal String

More problems on

Codeforces [Data Structures]