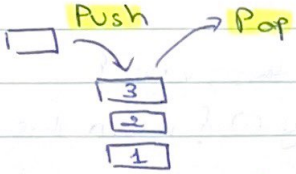
DSA slides

Date: Note:

- **Big O notation**: A theoretical measure of execution of an algorithm, usually the time or memory needed, given the problems size n which is usually the number of items
  - ↳ O(n) is fair; O(1) is the best, O(lg N) is good, $O(n^2)$ bad

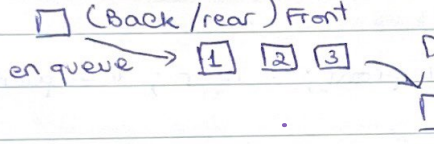
- **Stack** is a container of objects that are inserted and removed according to the last in first out (LIFO) principle.



push: to insert
pop: to remove

⟹ push 1, 2, 3 ; pop 3, 2, 1

- **Queue** is a container of objects (a linear collection) that are inserted and removed according first-in-first-out (FIFO)



enqueue: insert
dequeue: remove

enqueue 1, 2, 3 dequeue 1, 2, 3

- Advantage: Help data in more particular way than array and list
  Array and list are random access. They are flexible and easy to corrupt. To manage data as IFO, FIFO => use stack and queue

```
class Stack <T>
    private stack <T> previous;
    private T value;
    stack () {} # blank constructor
    stack (T value) { this.value = value}
    Stack (Stack <T> previous, T value)
        { this.previous = previous;
          this.value = value; }

// push
    public void push (T value)
    { this.previous = new Stack (this.previous, this.value);
      this.value = value; }
```

```
// pop
public T pop() { if (this. isEmpty)
                    print ("empty");
    T top = this.value;
    this.value = this.previous.value;
    this.previous = this.previous.previous;
    return top; }
// peek
    public T peek { return this.value}
    public boolean isEmpty () { return this.previous = null }
//size
    return this.isEmpty () ? 0 : 1 +
                    this.previous.size;
```

```
class Queue {
  private int [ ] arr; // front; // rear; // capacity; // count;}
  Queue (int size) {
    arr = new int [size]
    capacity = size; front = 0; rear = -1; count = 0; }
public void enqueue (int item){
    if (isfull ()) {
      print ("full"); rear = (rear + 1) % capacity;
      arr [rear] = item; count ++; }
public int peek () {
    if (isEmpty () )
      return -1;
    else return arr [Front];
public int size() { return count;}
public boolean isEmpty () { return (size() == 0);
public boolean isFull () { return (size () = capacity);}
```

Date : Note: KHANG VIET BOOK

Postfix to Prefix

```
class A{
    boolean IsOperator (char x){
        switch (x){
            case '+'
                 '-'
                 '*'
                 '/'
                 return true;}
            return false;}
string PostoPre (String Post_exp)
  { static <String> S = new Stack <String> ();
    int length = post_exp.length ();
for (int i=0; i < length; i++)        → {if (isOperator(
{ String op1 = s.peek ();                  post_exp.
        s.pop ();                          clearAt(i)))
    String op2 = S.peak ();
        s.pop ();
    string (temp) = post_exp.clearAt (i)
                    + op2 + op1
                    s.push (temp);
else s.pus (post_exp.clearAt (i) + " "); } }
   Str ans = " ";
       for (String i : S)
          ans + = i;
       return ans;
```

- infix normal express;
- post fix: An expression is called the post fix expression if the operator appear in the expression after operand simply of the form

ex: (A+B) + (C-D)

AB + CD - R

prefix an expression is called the prefix expression if the operator appears in the expression before operand.

Searching O(n)

Linear searching:

```
int LinearSearch (int a[], int N, int x) {
    int i=0;
    while ((i<N) && (a[i] != x))
        i++;
    if (i == N)
        return i;
    else return -1; }
```

Binary searching:

```
int Binarysearching (int a[], int N, int x) {
    int left=0, right=N-1;
    int mid
    while (left <= right){
        mid = (right - left)/2
        if (mid == x) return mid;
        else if (mid <x)  left = mid +1;
        else if (mid >x)  right = mid -1; }
    return -1;
    }
```

Date : Note: KHANG VIET BOOK

Linear	Binary
- Checking every single-element until finding the matching element	- Find target with sorted array
- Complexity : $O(n)$	- Complexity : $O(\log_2 N)$
- Tiest ease: First element	- Test ease: Middle element
Don't have to	Must sort array before searching
Less efficient with big array	More efficient
Less complex	More complex

* Sorting : $O(n^2)$

◦ Selection sort

```
int [] selection Sort (int[] a){
    int min =0;
      for (int i=0 ; i< a.length ; i++){
        min = i ;
        for (int j =i+1 ; j < a. length-1 ; j++)
            if (a[min] < a [j])
                min = j ;
                // <for dese>  for ASC
                // swap
        int temp = a [min];
        a[min] = a [i];
        a [i]  = temp; }
    return a; }
```

⇒ Explain: Choose smallest value the put in the first place for ASC & inverse with DESC order.

Date: Note:

KHANG VIET BOOK

- Insertion Sort: $O(n^2)$

```
int Insertionsort (int a[], int N){
    int pos, i; int x;
    for (int i=1; i<N; i++){
        x = a[i], pos = i-1;
        while ((pos>=0) & (a[pos]>x))      // < for DESC
            {a[pos+1] = a[pos];
             pos--; }
        a[pos+1] = x;   // insert x to the array }
    return a; }
```

↳ Explain: first, assuming the first element is ordered
Looping from a[1] to a[n]: compare current element to other in the left then move current element to left of the element compare (smaller or greater) shift all greater element to the pos the right.

- Bubble sort: $O(n^2)$

```
int [] Bubblesort (int a[], N){
    for (int i=0; i<N; i++)
        for (int j=N-1; j>i; j--)
            if (a[j] < a[j-1])
                // <for ASC, > for DESC
                swap (a[j], a[j-1]);
```

⇒ Explain: Looping to more the smaller or greater element to first or last positive V then ignore the element to the next step.
Keep looping until the last couple of elements.

- **Recursion** is the process which a function call itself directly or indirectly

ex:
```
int factorial (int n){
    if (n <= 1)
        return 1;
    else
        return n* factorial (n-1);
```

- **Linked list**: is a linear data structure, in which the element are not stored at contiguous memory locations. The elements in a linked list are linked

In simple word, a linked list consists of node where each node contains a field and a reference (or link) to the next node in the list

Difference between Array and Linked List

- Array stores elements in contiguous (kế bên) memory location, resulting in easily calculatable addresses for the elements stored and this allows faster access to an element at a specific index.
- Linked lists are less rigid (cứng nhắc) in their storage structure and the elements are usually not stored in contigous location, hence they need to be stored with additional tags giving a reference to the next element.

Singly linked list Class

```
class LinkedList { Node head;
  // Linked list node
      class Node { int data; Node next;
/*Constructor */ Node (int data) { this.data = data; next = null; }
// insert new Node in front of the list
  public push (int new_data)
  { // Allocate the Node and set Data
    new_node = new Node (new_data)
    // Make next of new Node as head
```

```
        new_node.next = head;
        // Move the head to the new Node
        head = new_node }
    public void display () { Node temp = head;
        while (temp ! null)
        { print (temp.data + " ");
            temp = temp.next; }
        print but /n }
    public static void main (String args[]) { ... }
```

- Remove duplicate (Sorted)

```
void remove Duplicate () {
    // Link to the head
    Node curr = head
    // go through the list till the last
    while (curr != null) {
        Node temp;
            // Compare current data to others
        while (temp! null && temp.data == arr.data)
            { temp = temp.next }}
    // Set current Node to the next different element denoted by temp
        curr next = temp;
        curr = curr.next; }}
```

- Remove duplicate unsorted

```
void remove_dup() {
    Node ptr1 = null;
    Node ptr2 = null;
    Node dup = null;
    ptr1 = head;
// Pick elements one by one
    while (ptr1 != null && ptr1.next != null){
        ptr2 = ptr1;
        // compare the picked element to others
        while ()
// check duplicate/ { if (ptr1.data == ptr2.data){
                ptr2.next = ptr2.next.next;
                System.gc(); }
            else
                ptr2 = ptr2.next }
        ptr1 = ptr1.next; }}
```

- Swap node

```
public void swapNote (int x, int y) {
  // if 2 values is the same
    if (x == y) return;
  // search for x
    Node PrevX = null ; currX = head;
    while (currx != null && currx.data != x){
        prevX = currX;
        currX = currX.next; }
  // search for y, like search for X but y
    while ....
  // there is no x, y
    if (currx == null // curryY == null)
        return;
```

```
// if x is not head of LinkedList
if (prevX != null)
    prevX.next = currY;
else head = currX;
// if y is not head of linked list
if...
// swap pointer
Node temp = currX;
currX = currY;
currY = temp; }
```

Move To Front

```
void move_to_Front(){
// check Empty if (head == null || head.next == null)
    return;
// Initialize second last and last pointers
Node SecLast = null;
Node Last = head;
// After this loop seclast contain the seclast node and so do the last
while (last.next != null)
    { secLast = last;
      last = last.next;
// set the next of second last to null
Seclast.next = null;
// set next of last as head
last.next = head;
// change head to last node
head = last;
```

Intersect 2 Linked List

```
static Node a = null ; b = null ; dummy = null
void sortedIntersect ()
 { Node p = a , q = b
   while (p! = null && q! = null) {
      if (p.data = q.data)
         push (p.data) ;
         p = p.next)
         q = q.next ;
      else if (p.data < q.data)
         p = p.next
      else  q = q.next ; }}
main ()
   list.display (dummy) ;
```

Circular Linked List

is a linked list where all nodes are connected to form a cycle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.

Advantages :

+ Any node can be starting point
+ Use implementation of queue
+ Useful in application to repeatedly go around
+ Circular doubly linked list are used for implementation

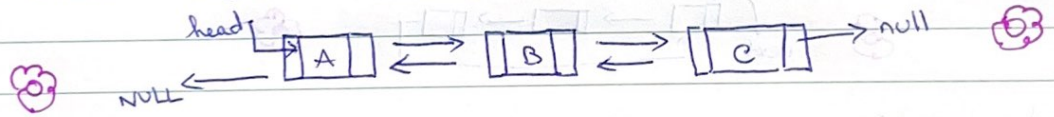
Date : Note: KHANG VIET BOOK

Circular Linked List

```
class CircularLinkedList{
    Static Node {int datat; Node next;}
    Static Node push (Node head_ref, int data)
      { Node ptr1 = new Node();
        Node temp = head_ref;
        ptr1.data = data
        ptr1.next = head_ref
      // if linked list is not null until the next is last node
        if (head_ref != null)
          { while (temp.next != head_ref)
              temp = temp.next; }
        else ptr1.next= ptr1;
          return head_ref; }
        display()
```

like Linked List

Doubly Linked List : contains an extra pointer typically call previous pointer, together with next pointer and data which are there in singly linked list



ADVANTAGES	DISADVANTAGES
- Can be tranversed in both forward and backward directly	- Require extra memory for each node
- The delete operation in DLL is more efficient if pointer to the node to be deleted given.	- Operations require more time to handle extra pointers
- Quickly inserted a new node before a given node.	- No random access of elements

Doubly Circular Linked list has the properties of both doubly linked list and circular linked list, which two consecutive elements are linked or connected by previous and next pointer, and the last node point to first node by next pointer ; and the first node by the previous pointer.

ADVANTAGES	DISADVATAGES
- Tranversed both directly back + foward - Jump from head to tail or inverse constanly (O(1))	- Require more memory - Lots of pointers involved

Application: manage playlist, shopping cart