By Mohannad
for any Suggestion
@7xxP7

CSC 212

Data Structure

IMPORTANT

ملخص بسيط للمادة لتثبيت المفاهيم
المراجعة بعد المذاكرة فقط

"و الله ولي التوفيق"

Big-Oh from smallest to largest :-

$O(1)$
↳ $O(\log n)$
↳ $O(n)$
↳ $O(n \log n)$
↳ $O(n^c)$
↳ $O(c^n)$
↳ $O(n!)$
↳ $O(n^n)$

SOME RULES :-

- for/while , < , ++ loops
  freq = max - initial +1
- for/while , <= , ++ loops
  freq = max - initial + 1 +1
- for/while , > , -- loops
  freq = initial - max +1
- for/while , >= , -- loops
  freq = initial - max + 1 +1

Properties of log

$\log A^N = N \log A$

$\log(A \cdot B) = \log A + \log B$

$\log\left(\frac{A}{B}\right) = \log A - \log B$

NOTE

$y = \log_a x \iff a^y = x$

Ex : $y = \log_2 8 \iff 2^y = 8$

$\therefore \underline{y = 3}$

ADT list

This list has 9 methods :-

1- findFirst() 2- findNext() 3- retrieve (Type e) 4- update (Type e)

5- insert(Type e) 6- remov() 7- full (boolean flag) 8- empty(boolean flag) 9- last(boolean flag)

```
Public class Node <T> {
public T data;
Public Node <T> next ,.

public Node (){
      data = null ;
       next = null ;  }
public Node (T val) {
          data = val ;
          next = null ;}
   // Setters/ Getters }

Public class LinkedList <T> implements List<T> {

private Node<T> head ;
private Node<T> Current ;

public LinkedList (){ head = current = null ;}

public boolean empty() {                    1
          return head == null }

Public boolean last (){                     2
      return current.next == null ; }

Public boolean full () {                    3
          return false ; }

public void findFirst(){                    4
          current = head ;}

public void findnext () {                   5
          current = current. Next ; }

public T retrieve () {                      6
     return current . data ; }

public void update (T val) {                7
          current. data = val ; }

Public void insert (T Val){                 8
     Node<T>  tmp ;
     if( empty()){
          head = current = new node<T>(val) ; }
     else {
          tmp = Current. next ;
          current. next = new node<T>(val) ;
          current = current. Next ;
          current. next = tmp ;   }}
```

```
public void remove() {
    if(current == head){
        head = head.next; }

    else {
        Node<T> tmp = head;
        while( tmp.next != current)
            tmp=temp.next;
        tmp.next = current.next ; }

if (current.next == null)
        current = head
else
        current = current.next ; } }
```

9

ADT Array

```
public class Arraylist<T> implements List<T>{

Private int maxSize ;
Private int size ;
Private int current;
Private T[] nodes ;

public Arraylist(int n) {
        maxSize = n ;
        size = 0 ;
        current = -1 ;
        nodes = (T[]) new object[n] ;}
```

```
public boolean empty() {
        return size ==0 }
```

1

```
Public boolean last () {
        return current = size - 1 ;}
```

2

```
Public boolean full () {
        return size == maxSize ; }
```

3

```
public void findFirst(){
        current = 0 ;}
```

4

```
public void findnext () {
        current ++ ; }
```

5

```
public T retrieve () {
    return nodes [current] ; }
```

6

```
public void update (T val) {
    nodes [current] = val ; }
```

7

```
public void insert ( T val) {
    for( int i = size-1 ; i > current ; --i){
        nodes [i+1] = nodes[i] ; }

    current ++ ;
    nodes [current] = val ;
    size ++ ;  }
```

8

```
public void remove () {
for ( int i = current +1 ; i < size ; i++)
        nodes [ i-1] = nodes[i];
 size -- ;
if ( size == 0){
        current = -1 ; }
else if (current == size){
        current = 0 ; } }
```

NOTE this is Search Item method

```
public static <T> boolean find (ArrayList l, T key){

    if(l.empty == false) {
        l.findfirst();
        while (l.last == false){
            if(l.retrive().equals (key))
                return true;
            l.find.next ; }
        if(l.retrive().equals (key)
            return true; }
return false ; }
```

ADT DLL

here we have 11 method only 2 methods are new but !! remove() and insert(Tval) are different

```
Public class Node <T> {
public T data;
Public Node <T> next ;
Public Node <T> Previous ;

public Node () {
        data = null ;
        next = null ; }
public Node (T val) {
        data = val ;
        next = null ;
        Previous = null ; }
    // Setters / Getters }

Public class DoubleLinkedList <T> {
private Node <T> head ;
private Node <T> current ;
```

1

```
public boolean first() {
        return current.Previous == null ; }
```

2

```
Public void find Previous () {
        current = current.Previous ; }
```

3

```
public void insert (T val) {

Node <T> tmp = new node <T> (val) ;

    if (empty()) {
        current = head = tmp ; }
    else {
        tmp.next = current.next ;
        tmp.Previous = current ;
        if ( current.next != null)
            current.next.Previous = tmp ;
        current.next = tmp ;
        current = tmp ; }}
```

4

```
public void remove () {
    if (current == head) {
        head = head.next
        if (head != null)
            head.Previous = null ; }
    else {
        current.Previous.next = current.next
        if ( current.next != null)
            current.next.Previous = current.Previous ;
    }
    if ( current.next == null)
        current = head ;
    else current = current.next
```

ADT Queue FIFO

There are Two kind of Queue
1- Queue
2- Priority Queue

and 4 method;
1- Serve();
2- enqueue(Te);
3- length();
4- full();

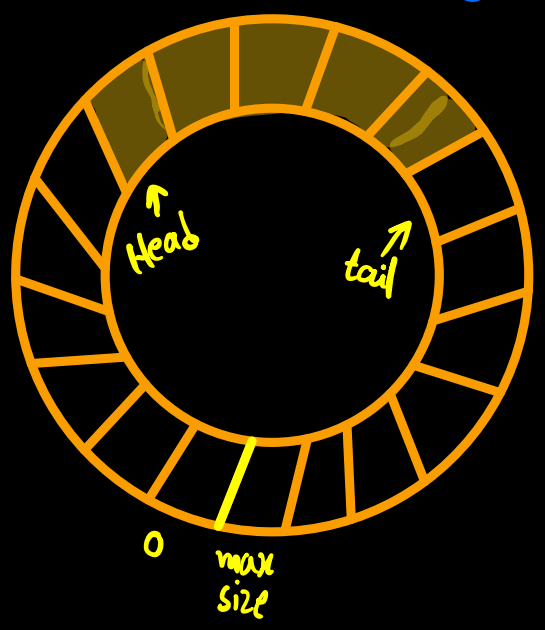
The only linked have length

```
public class Node<T> { etc .... }
```

```
Public class LinkedQueue<T> implements Queue<L>{
    Private Node<T> head, tail ;
    Private int Size ;

    Public LinkedQueue (){
        head = tail = null ;
        Size = 0 ; }

    public boolean full () {
        return false; }

    Public int length () {
        return Size; }

    public void enqueue (T e) {
        if( tail == null) ;
            head = tail = new node<T>(e) ; }

        else {
            tail.Next = new node<T>(e) ;
            tail = tail. Next ; }
        Size ++ ; }
```

```
Public T Serve() {
    T x = head.data ;
    head = head.next ;
    Size -- ;
    if (Size == 0)
        tail = null ;
    return x ;
```

Queue Array



```
public class ArrayQueue<T> implements Queue<T>{
    private int maxSize;
    private int size;
    private int head, tail;
    Privat T[] nodes;

    Public ArrayQueue(int n){
        maxSize = n;
        size = 0;
        head = tail = 0;
        nodes = (T[])new object[n]; }

    public boolean full(){
        return size==maxSize ; }

    Public int length(){
        return size; }

    public void enqueue(T e){
        nodes[tail] = e;
        tail = (1+tail)% maxSize;
        size ++ ; }

    Public T serve(){
        T e = nodes[head]
        head = (1+head) % maxSize;
        size --;
        return e; }
```

ADT Priority Queue

```
public class PQNode<T>{
    Public T data;                  don't have tail
    Public Priority priority;       or int
    public PQNode<T> next;

    Public PQNode(){
        next == null; }

    Public PQNode(T e, Priority p){      or int
        data = e;
        priority = p; }
    // setters/Getters }
```

```
public class LinkedPQ<T>{
Private int size
Private PQNode<T> head ;

  public LinkedPQ(){
      head = null;
      size = 0; }

public boolean full (){
        return false; }

public int length(){
        return size; }

public PQElement<T> Serve(){

PQNode<T> node = head ;
PQElement pqe= new PQElement<T>(node.data, node.p);
head= head.next
Size -- ;
return pqe; }
```

```
public void enqueue(T e, int pty){
   PQNode<T> temp= new PQNode<T>(e,pty);
if( (Size ==0) || ( pty> head.Priority){
            temp.next= head ;
            head = temp ; }
else{
            PQNode<T> p= head;
            PQNode<T> q = null;
  while ( (p!=null) && ( pty <= p.Priority)){
            q= p;
            p= p.next; }
        temp.next = p
        q.next = temp }
Size ++; }
```

ADT Stack LIFO

There are 4 methods

1- Pop(); 2- push(T e);

3- empty(); 4- full();

```
Public class LinkedStack<T> implement, Stack<T>{
    private Node<T> top;

    Public LinkedStack(){
        top = null;  }

    Public boolean empty(){
        return top == null; }

    Public boolean full(){
        return false; }

    public void push(T e){
        Node<T> tmp = new Node<T>(e);
        tmp.next = top;
        top = tmp; }

    Public T pop(){
        T e = top.data;
        top = top.next;
        return e; }
```

. don't have tail

```
Public class ArrayStack<T> implemets Stack<b>{
    Private int maxSize;
    Private int top;
    Private T[] nodes;

    Public ArrayStack(int n){
        maxSize = n;
        top = -1;
        nodes = (T[]) new object[n]; }

    Public boolean empty(){
        return top == -1; }

    Public boolean full(){
        return top == maxsize-1; }

    public void push(T e){
        nodes[++top] = e; }

    public T pop(){
        return nodes[top--];
```

Sorting

Selection Sort :-

```
Public static void SelectionSort(int[] A , int n){
    for(int=0 ; i<n-1 ; i++){
        int min = i ;
        for( int j= i+1 ; j<n ; j++){
            if( A[j] < A[min])
                min = j ; }
        int tmp = A[i]
        A[i] = A[min]
        A[min] = tmp ; }}
```

The most important

- worst case time complexity: $O(n^2)$
- Average case time complexity: $O(n^2)$
- Space complexity: $O(1)$
- In place : Yes
- Stable : NO

Bubble Sort :

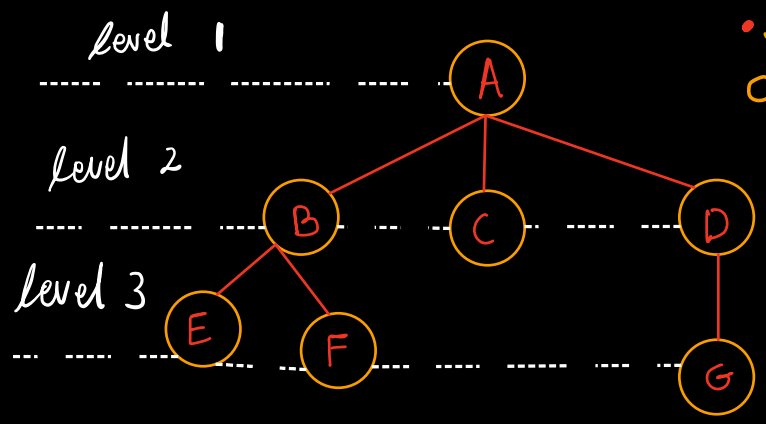
```
Public static void bubbleSort(int A[], int n){
    for(int i=0 ; i<n-1 ; i++){
        for( int j=0 ; j<n-i-1 ; i++){
            if ( A[j] > A[j+1]){
                int tmp= A[j] ;
                A[j] = A[j+1] ;
                A[j+1]= tmp ; }}}}
```

- worst case time complexity: $O(n^2)$
- Average case time complexity: $O(n^2)$
- Space complexity: $O(1)$
- In place : yes
- Stable : Yes

Merge Sort :

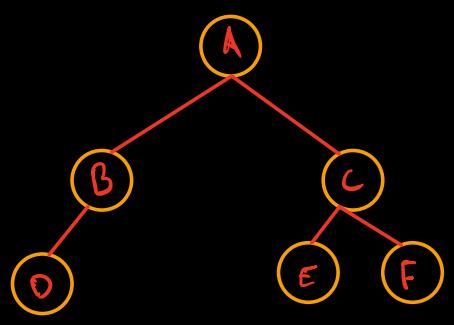
- worst case time complexity: $O(n \log n)$
- Average case time complexity: $O(n \log n)$
- Space complexity: $O(n)$
- In place : NO
- Stable : Yes

General Tree



- for Each node can have 0 – * child

Binary Tree



- only at most 2 child for Each node

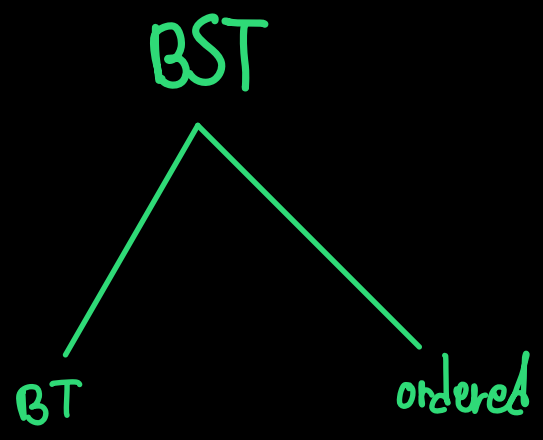
We have three kind of Traverse

1. PreOrder NLR | N : root node
2. InOrder LNR | R : Right node
3. PostOrder LRN | L: Left node

ADT BT

1. Traverse(order ord)
2. retrive ()
3. update (T val)
4. boolean insert (Relative rel, T val)
5. deleteSubtree()
6. boolean find (Relative rel)
8. Empty ()
7. BTNode<T> findParent(BTNode<T> p, BTNode<T> t)

```
Public class BTNode <T>{

 Public T data
 Public BTNode<T> left, right;
   //      //    etc }
```

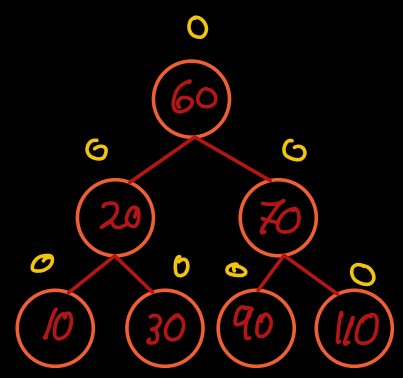


Best case : $O(1)$
Avg case : $O(\log(n))$
worst case : $O(n)$

1. Traverse(order ord)
2. retrive ()
3. update (int k, T val)
4. boolean insert (int k , T val)
5. deleteSubtree()
6. boolean findKey(int k)
8. Empty ()
7. BTNode<T> findParent(BTNode<T> p, BTNode<T> t)
9. remove key(int k)
10. full();

AVL

must be BST and Balanced



Balance = Right Subtree Height - Left Subtree Height

it must be $\{-1, 0, 1\}$

- $-1, -2 \Rightarrow$ left rotation , $1, 2$ Right Rotation
- $-1, 2$ or $1, -2$ double rotation

B-Tree

	min	max
non-leaf	$\lceil \frac{k}{2} \rceil - 1$	$k-1$
leaf	$\lceil \frac{k}{2} \rceil$	k

insert ① Transfer , ② Split

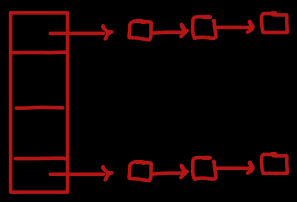
delete ① borrow , ③ merge

Hash

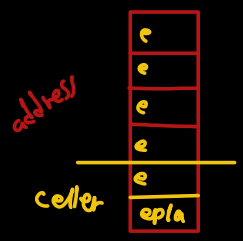
Open address method

- linear rehashing
- quadratic "
- random, double

External chaining



Coalesced chaining



Heap

Heap is a complete binary tree

Heap , full كل level مرتبة في

index 0

$h = \lfloor \log n \rfloor$

max $n = 2^{h+1} - 1$

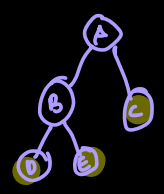
min $n = 2^{h}$

index 1

$h = \lceil \log(n+1) \rceil$

max $n = 2^{h} - 1$

min $n = 2^{h-1}$



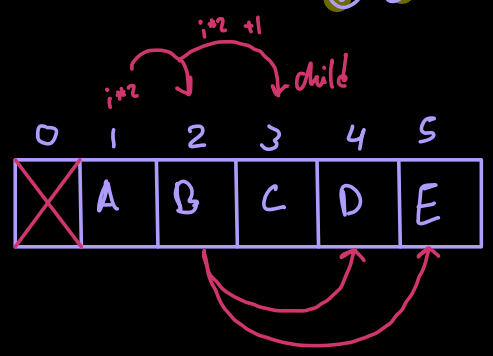
Heap sort $O(n \log n)$

enqueue $O(\log n)$, sift up

serve $O(\log n)$ sift down

full, length $O(1)$

height of the heap $O(\log n)$



$\left\lceil \frac{size}{2} \right\rceil$ = leaf => $\frac{5}{2} = \lceil 2.5 \rceil = 3$

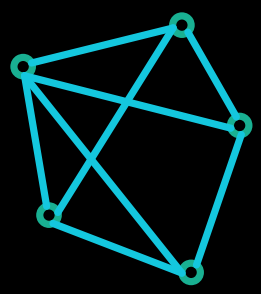
Heap Sort :-

1) إدخال الأعداد المطلوبة عصفوفة
2) تحويلها لـ Heap
3) نقوم بالإلغاء الـ root
4) نقوم بوضع الـ root في المصفوفة الجديدة

min heap ⟶ تنازلي (descending) 3,2,1

max heap ⟶ تصاعدي (Ascending) 1,2,1

Graphs



Path : ممر يسمح فيه التكرار

Simple path : لا يسمح فيه التكرار

cycle : نقطة البداية هي النهاية

- Tree is a graph that :
  - is connected
  - has no cycle

- spanning tree doesn't have root
- oriented tree has root

There are Two approaches to represent graphs :-

1. Adjacency Matrix space requirement is $O(n^2)$
2. Adjacency List space requirement is $O(n+e)$

There are two method for graph traversal

1. Breadth First Search using Queue
2. Depth First Search using Stack