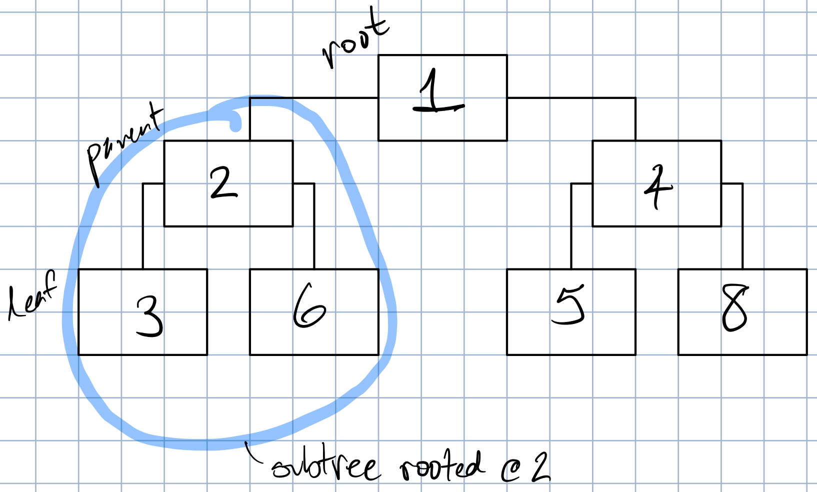
Part 1

arrays & linked lists → order

trees → hierarchy

root: top, no parent
child
parent
leaf
binary tree
↓
up to 2 children

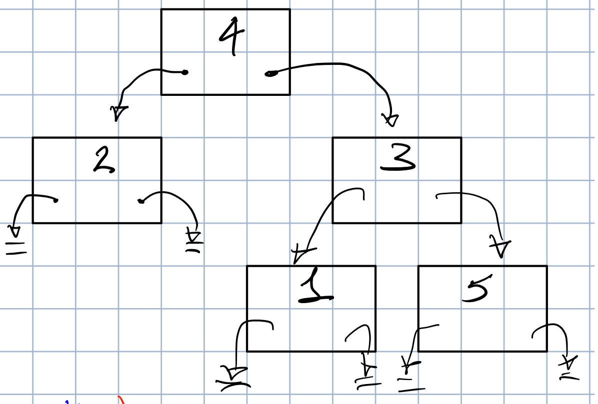


trees are recursive

each node is a tree

pointer-based implementation

```
typedef struct intree intree_t

struct intree
    int val
    intree_t *left
    intree_t *right


intree_t *make_node(int val, intree_t *left,
                             intree_t *right)

    intree_t *t = (intree_t*)malloc(sizeof(intree_t))

    t -> val   = val
    t -> left  = left
    t -> right = right

    return t
```



```
intree_t *t = make_node( 4 ,
                  make_node( 2, Null, null)
                  make_node( 3 ,
                        make_node( 1, null, null )
                        make_node( 5, null, null )   )
```

empty?

```
bool  is_empty( intree_t  *t)
      return   t == null
```

leaf?

```
bool   is_leaf( intree_t  *t)
      return  t -> left == null   && t -> right == null
```

get value

```
int    getval( intree_t  *t)
       assert(!isempty(t))
       return   t -> val
```

```
intree_t  *left(intree_t *t  )
      assert( !is_empty( t ) )
      return   t -> left
```

```
intree_t  *right(intree_t *t  )
      assert( !is_empty( t ) )
      return   t -> right
```

```
bool   is_member( intree_t *t ,  int val)

      if(is_empty( t ))
            return false

      return     t -> val == val   || is_member( t -> left , val)
                                   || is_member( t -> right , val)
```

```
int   count( intree_t *t)

      if(is_empty( t ))
            return 0

      int  left   =  count( t -> left)
      int  right  =  count( t -> left)

      return   left + right + 1
```

```
void free_tree(intree_t *t )

    if(is_empty( t ))
        return

    free_tree(t -> left )
    free_tree(t -> right )

    free( t )
```

avoid reaching
into children tree

```
void get_depth(intree_t *t )

    assert(! is_empty( t ))

    if ( is_empty( t ))
        return 0

    int left_depth = get_depth ( t->left)
    int right_depth = get_depth ( t->right)

    if (left>right )
        return left +1
    else
        return right +1
```

longest path from root to subtree

depth can be edge or nodes

Part2

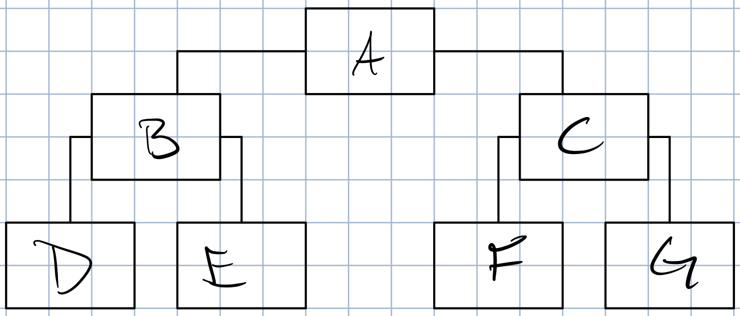
Tree traversal

Depth first: starting from root, visit all nodes in 1 branch

Breadth first: starting from root visit all nodes on current level

Depth

preorder A BDECFG
in order D B E A F C G
post order D E B F G C A



```
void preorder_print ( intree_t *t)

    if( is_empty( t ) )
        return
    printf( t -> val )
```

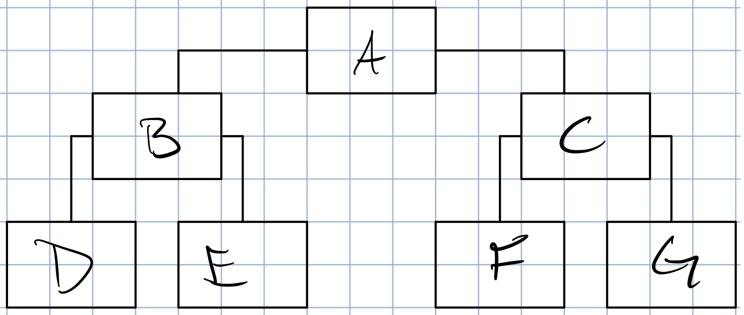
```
    preorder_print( t -> left )

    preorder_print( t -> right)
```

right type?
closer to base case?

```
void  inorder_print ( intree_t  *t)
    if( is_empty( t ) )
          return

    preorder_print( t -> left )

    printf(  t -> val )

    preorder_print( t -> right)
```

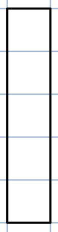
```
void  postorder_print ( intree_t  *t)
    if( is_empty( t ) )
          return

    preorder_print( t -> left )

    preorder_print( t -> right)

    printf(  t -> val )
```

Breadth
    level order: A BC DE FG



lets make a queue!
    collection of elements w/ limited operations

① create empty queue
② Enqueue element at back of queue
③ Dequeue element at front of queue



```
typedef struct  queue queue_t

struct  queue {
     intlist_t  *front
     intlist_t  *back
}

intlist_t  *new_node
queue_t  *create_queue
int       dequeue
```

now to implement

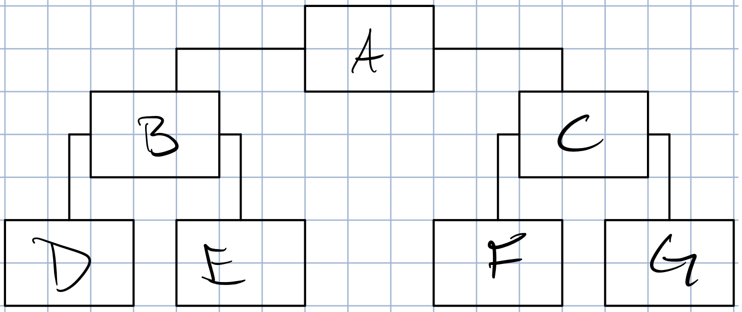
instead of a queue w/ intlists, we need one w/ inttrees

after printing a tree, we enqueue its children, dequeue that tree

t
print t

Ⓐ B C
print B

Ⓐ Ⓑ C DE

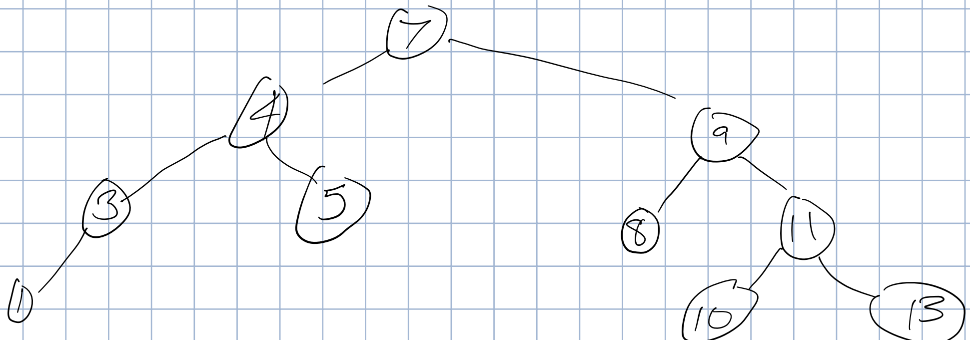


```
queue_t *q = create_queue( )
inttree_t *curr = t

if ( is_empty( t ) )
    return

printf ( curr->val )

if ( ! is_empt(curr->left) )
    enqueue( q , curr->left)
if ( ! is_empt(curr->right))
    enqueue( q , curr->right)


curr = dequeue( q )
```

Part?

Binary Search Tree (BST)
values are unique
values in left subtree are strictly less than root
values in right subtree are strictly more than root

left & right are BSTs



how do we find if something is in tree?
can now pick where to go if node != value.

can ask node ≥ value or nod ≤ value

```
bool    bst_lookup (intree_t *t, int val)

    if( is_empty( t ))
        return false

    if ( t -> val == val)
        return true

    if ( t -> val > val )
        return bst_lookup( t->left, val)
    if ( t -> val < val )                       —▷ unnecessary
        return bst_lookup( t->right, val)
```

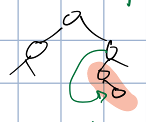
```
inttree_t  *insert ( intree_t *t, int val)

    if( is_empty( t ))
        return makenode(val, null, null )

    if ( t -> val > val )
        t -> left = insert( t->left, val)

    else  if ( t -> val < val )
        t ->right = insert( t->left, val)
    else
        ayo, this shit already there dawg
    return t
```

remove value
- leaf? —▷ remove it
- if value has only 1 child —▷ copy value from child & remove child
- if value has 2 children —▷ copy value from in-order successor & delete in order successor

```
inttree_t  *remove_node( inttree *t, int val)

    if( is_empty( t ))
        return t

    if( t->val > val)
        t -> left = remove_node( t->left, val)
```

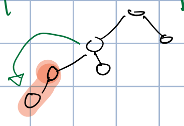
```
else if (t->val < val      )
     t ->right = remove_node ( t->left, val)

else {

     if ( is_empty(t->left))
          inttreet *temp = t->right
          free( t )
          return temp
```

only 1 child , right . left is empty



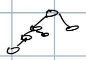
```
     if ( is_empty(t->right))
          inttreet *temp = t->left
          free( t )
          return temp
```

only 1 child, left . right is empty

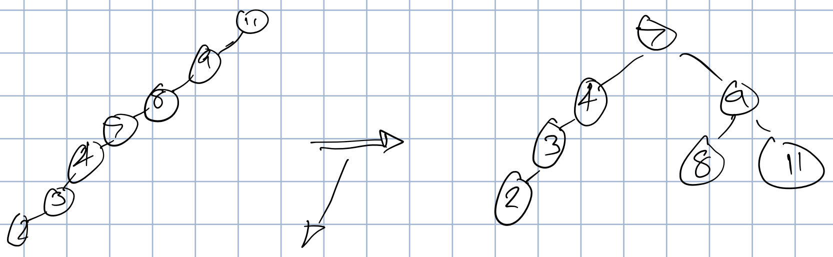


```
     inttreet  *temp = min_value(t-> right)
     t->val = temp->val
     t-> right = remove_node( t->right, temp->val)

}

return t
```

more

binary search tree (BST) can be unbalanced  vs balanced 

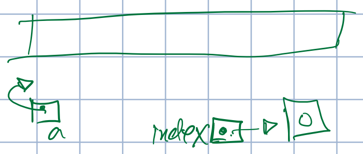
↳ bad lookup times



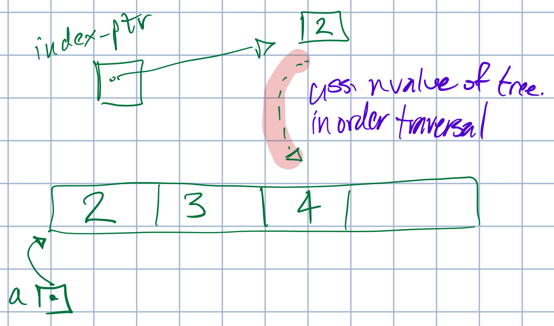
Inorder traversal : 2 3 4 7 8 9 11

```
int count_nodes (inttree_t *t)

    if( is-empty( t ))
          return 0

    int left  = count_nodes ( t->left )
    int right = count_nodes ( t->right )

    return  1+left +right
```

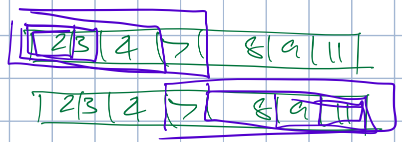
```
inttree_t *balance(inttree_t *t)

    int count = count_nodes(t)
    int *a = (int*) malloc(sizeof(int) * count)
    int index = 0
    make_array(t, a, &index)

    return make_bst(a, 0, count-1)
```

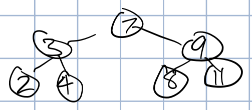


```
void make_array(inttree_t *t, int *a, int *index_ptr)

    if( is_empty( t ))
        return;

    make_array(t->left, a, index_ptr)
    a[*index_ptr] = t->val
    *index_ptr += 1
    make_array(t->right, a, index_ptr)
```



```
                                   0              count-1
inttree_t  make_bst(int *a, int start, int end)

    if(start > end)
        return null

    int middle = (start+end)/2
    inttree_t *t = make_node( a[middle], null, null)

    t->left = make_bst( a, start, middle-1)
    t->right = make_bst( a, middle+1, end)

    return t
```

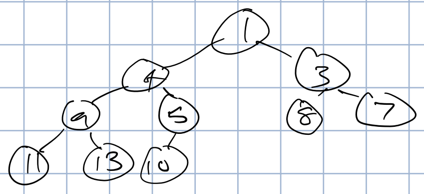




full binary tree: all nodes not a leaf has 2 children

complete binary tree: all levels (except maybe last) filled & all nodes in last level are left justified

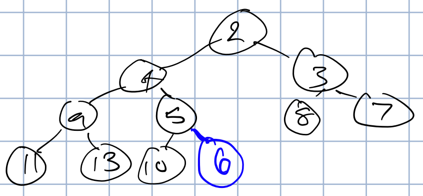
min heaps: binary tree
- complete
- unique values
- values in left & right subtrees strictly greater than root
- left & right subtrees are min heaps



max heap
complete
unique values
values in left & right subtrees strictly less than root
left & right subtrees are min heaps

cool operations: add min heap, remove min, is empty

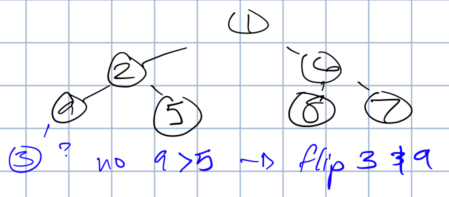
let's add 6

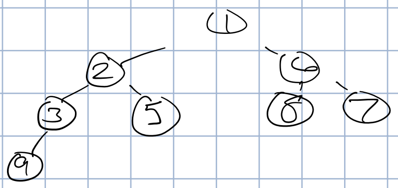


```
bool  *is_heap(intreect  *t )

    if( isleaf( t ))
        return  true

    if( is_empty( t->right))
        return  (t->left->val)  >  (t->val)

    if( (t->left->val) > (t->val)  &&  (t->right->val) > (t->val))
        return   is_heap( t->left)  &&  is_heap(t->right)

    return false
```

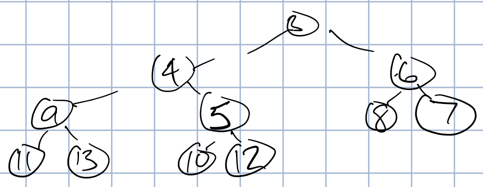
no right kids, only need to check left value

add 3

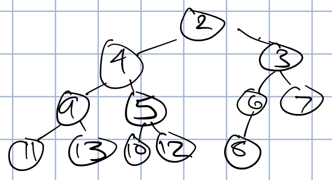




add 2



sift up until we have a heap



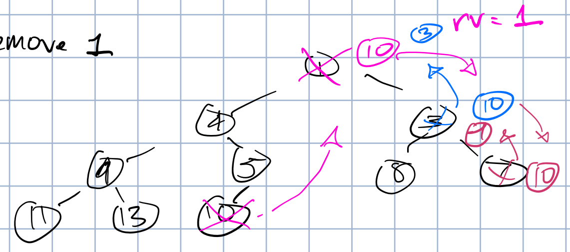
put into next open slot

Heaps & stuff

min heap is a BST. complete & children greater than node

① ② ③

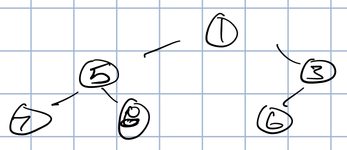
remove 1



siftdown to smallest value for 10 & 3
again for 10 & 7

Cool! Alert!

heaps stored in arrays
level ordered



1	5	3	7	8	6

```
int val    = a[i]
int parent = a[(i-1) / 2 ]
int lchild = a[ 2·i +1 ]
int rchild = a[ 2·i +2 ]
```

truncation helps

val 8 w/ index 4: parent (4-1)/2 = 3/2 = 1
a[1] or 5 trooo

some helper fn's

```
void swap( int *a, int *b )
    int temp = *a
    *a = *b
    *b = temp
```

```
void print_heap( int heap[] , int size)
    for( int i=0 ; i<size; i++)
        printf( "%d" , heap[i])
```

```
void sift_up (int heap[] , int size, int index)

    if( index==0 ) {
        return;
    }

    int parent = (index -1) /2
    if( heap[index] < heap[parent])
        swap( & heap[index] , & heap[parent])
        sift_up( heap, parent)
```

```
void insert(int heap[], int size, int index)

    *size = *size+1
    heap[*size-1] = val              insert @ end of array
    sift_up( heap, *size-1)          sift that mf up
```

make sure we got space in heap

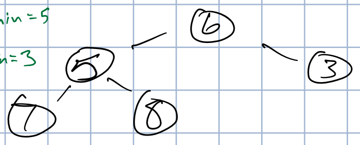
```
void sift_down(int heap[], int size, int index)

    int min_index = index
    int left  = 2*index+1
    int right = 2*index+2

    if( left < size)
        if( heap[left] < heap[min_index])
            min_index = left

    if( right < size)
        if(  heap[right] < heap[min_index] )
            min_index = right

    if ( min_index != index)
```

② in earlier example

5<6 ✓ min=5

3<5 ✓ min=3

swap ③ & ⑥



```
void remove_min( int heap[], int*size)

    int last = heap[*size -1]
    heap[0] = last
    *size = *size -1

    sift_down( heap, *size, 0 )


void build_heap

    for ( int i = size/2 ; i >=0 ; i--)
        sift_down( a, size, i)
```

more heaps

```
void heap_sort(int heap[], int size)

    for( int i = size-1 ; i>0 ; i--)
        swap( &heap[0] , &heap[i])
        sift_down( heap, i , 0         )
```

priority queue : queue with...
- every element has priority
- elements w/ high priority dequeued before elements w/ lower priority
- ...

dictionaries : maps key to values
key → unique

set: store collection of unique values

how to implement?

linked lists!
- easy
- but lookup is time proportional to length of list
- but adding key-value pairs takes time proportional length of list

binary search tree!
- easy
- adding values is pretty good
- but lookup time is proportional to height of tree

hashtables!

hash tables!
- goal: add, remove, query in near constant time
- idea: trade space for time

A[key] = item

space of array?
need slot for each possible value key could take

what if keys are integers? strings?

may need array larger than memory of computer

only need array large enough to hold max number of items stored @ same time

A[hash(key)] = item
hash(key1) != hash(key2) if key1 != key2
$0 <= hash(key) < N$ → array length

collisions can happen w/ rare occurance

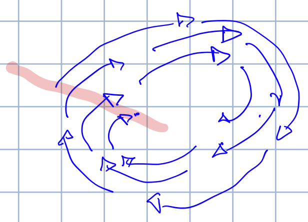
Suppose we have 1000 spots in A

hash(key) = key % 1000

nice if uniformly distributed

string keys w/ 1000 spots

hash code should depend on entire input key
output should look "random"



```
int hash( char *s )
    int h=0
    while( *s )          →prime
        h = h*37 + int( * s++ )
        h = h % HASH_SIZE
```

2 handles of collisions

linear probing: put item in next available slot if original slot is occupied

seperate chaining: keep list of items in each slot

0	1	2	3	4	5	6	7	8	9
B	C	X	W	Y	Z	D	E	F	A

insert W : hash(W) → 3
hash(X) → 2
hash(Y) → 4
hash(Z) → 2   3? 4? 5? ✓

lookup Z : hash(Z) → 2   ö what abt. 3? 4? 5? ✓

lookup H : hash(H) → 2   3? 4? 5? 6? X → H is not in hashtable

insert A: hash(A) → 9
hash(B) → 9   ö add 1 & mod(size) → 0? ✓

insert rest:

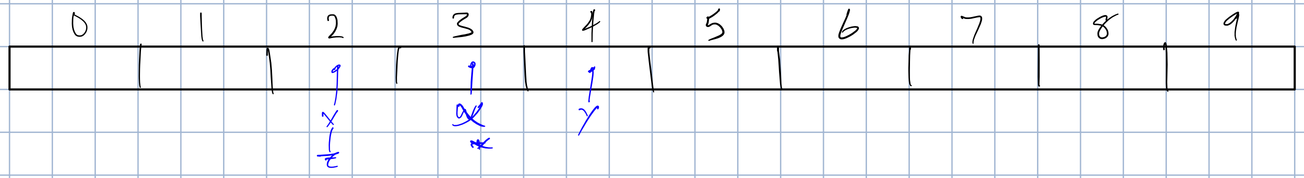
insert G: hash(G) → 6   infinite loop!   grow array or if we get back to 6, say not there

remove W! hash(W) → 3   but now can't find Z!
don't remove, mark is dead !

limit collisions
move as lookup
re-hash

More hash tables

separate chaining → buckets



insert W: hash(W) → 3
hash(Y) → 4
hash(X) → 2
hash(Z) → 2

lookup F: lookup(F) → 2   F is not X or Z, either in bucket 2 or not

remove W: hash(W) → 3   take out W & put asterisk. bucket. don't need tombstone

how to implement bucket? linked list

long bucket? use good hash f^n

rehashing:
define load factor as avg num of entries per slot
if it reaches specified threshold, create larger table, hash all data into it, replace current

bonus option: keep old table for while, gradually move items to new table

Graphs!

graphs have vertices, nodes representing entities

graphs have edges to represent relationships between 2 vertices types

directed - links go one way, not reciprocated

source → sink

undirected - all edges go both ways

weighted - each edge has weight signaling strength of relationship

unweighted - relationship strengths are equal

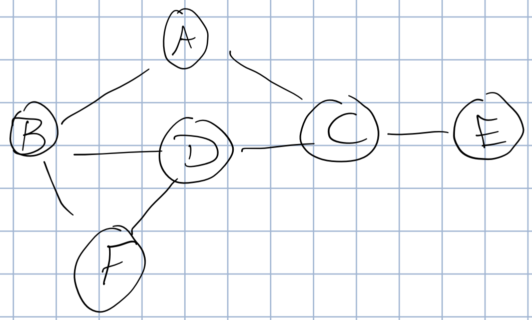
researchers co-authored at least 1 paper

undirected

can be weighted by # of papers

adjacency matrix

let's have an N × N matrix

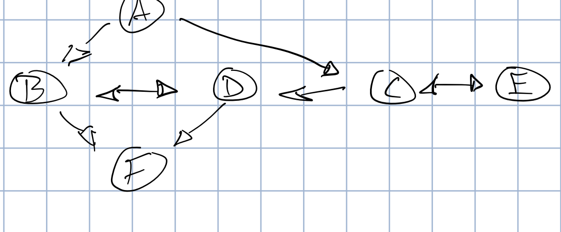


$$\begin{array}{c|cccccc} & A & B & C & D & E & F \\ \hline A & 0 & 1 & 1 & 0 & 0 & 0 \\ B & 1 & 0 & 0 & 1 & 0 & 1 \\ C & 1 & 0 & 0 & 1 & 1 & 0 \\ D & 0 & 1 & 1 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 0 & 0 & 0 \\ F & 0 & 1 & 0 & 1 & 0 & 0 \end{array}$$

table is symmetric

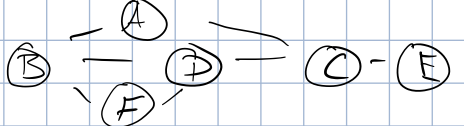
if it weighted, put value w

if directed, it is not symmetric



$$\begin{array}{c|cccccc} & A & B & C & D & E & F \\ \hline A & & & & & & \\ B & & & & & & \\ C & & & & 1 & & \\ D & 0 & 1 & 0 & 0 & 0 & 1 \\ E & & & & & & \\ F & & & & & & \end{array}$$

Adjacency list



list

A: B, C

B: A, D, F

C: A, D, E

D: B, C, F

E : C
F : B, D

for weighted, have tuples?

A: (B, 2) , (C, 1)