## ADT

Priority Queue

~~insert (x, k)~~

enqueue ~~push~~(x, k)

dequeue ~~pop~~()

getMax ~~peek~~

## Implement with BST

enqueue ~~push~~: normal insert : log n

dequeue ~~pop~~() log n

getMax ~~peek~~() : log n

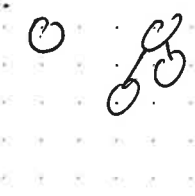
create : n log n

## ~~Both~~ Implementation

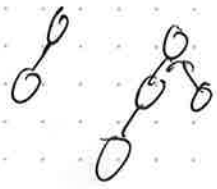
Heap

Nearly complete binary tree

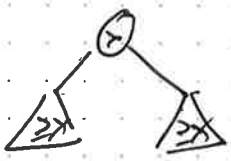
complete:



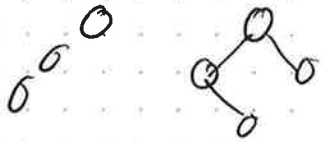
nearly complete



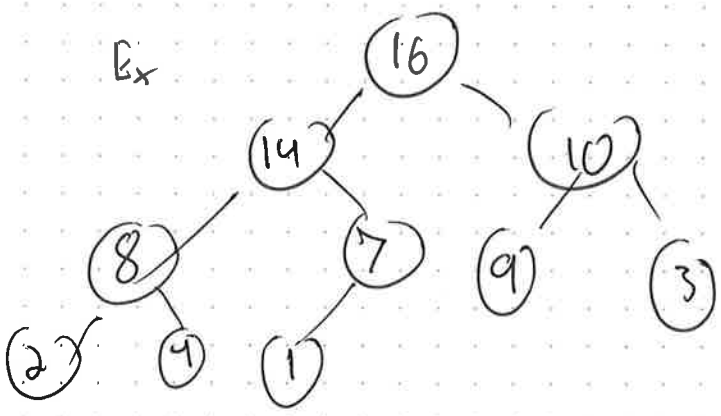
Max-Heap property: key of a node is $\geq$ keys of its children



not nearly complete



Ex




→ 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1

root : first element a[0] i=1

parent(i) = $\frac{i}{2}$ (round down)

left(i) = 2i

right(i) = 2i+1

BuildHeap

for (i = index of heaping Down; i >= 1; i--)

Running Time: $O(n)$

(1.) Heapify Down takes $O(h)$ for a value his height of node i

(2.) There are $\frac{n}{2^h}$ nodes at level h

$$\sum_{i=0}^{\log(n)} \frac{n}{2^h} \cdot h = n \sum_{i=1}^{\log n} \frac{h}{2^h}$$

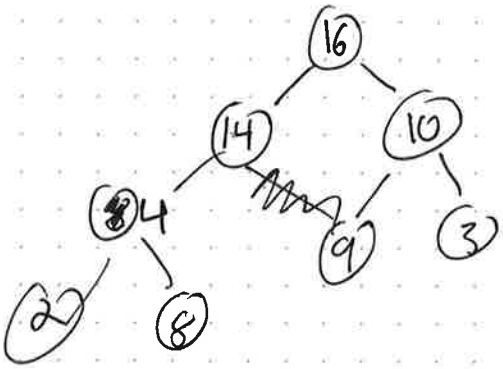
converges to a constant as n gets large

Heap Operations

buildMaxHeap : produces a max heap from an unordered array

heapify : correct a ~~single~~ violation of heap property

~~Exchange~~




```
heapifyUp(i)
  if i != 1 && a[i] > a[parent(i)]
    swap(i, parent(i))
    heapifyUp(parent(i))
```

~~Push~~ ~~Insert~~ enqueue

Increase ~~count~~ index
Place new data at ~~count~~ index
heapify starting at index

```
heapifyDown(i)
    largest = i
    L = leftChild(i)
    R = rightChild(i)
    if A[L] > A[largest]
        largest = L
    if A[R] > A[largest]
        largest = R
    if largest != i
        swap(i, largest)
        heapifyDown(largest)
```

dequeue

1. grab root
2. swap last element to root
3. heapifyDown starting at root

Do example

```java
public class QuickSort {

    static void quicksort(int[] arr) {
        quicksort(arr, 0, arr.length);
    }

    static void quicksort(int[] arr, int lo, int hi) {
        if (lo >= hi) {
            return;
        }
        else {
            int pivot = partition(arr, lo, hi);
            quicksort(arr, lo, pivot );
            quicksort(arr, pivot+ 1 , hi);
        }

    }
    static int partition(int[] arr, int lo, int hi) {
        int p = selectPivot(arr, lo, hi);
        int pivotValue = arr[p];
        swap(arr, lo, p);
        int boundary = lo + 1;
        for(int i = lo+1; i < hi; i++) {
            if(arr[i] < pivotValue) {
                swap(arr, i, boundary);
                boundary++;
            }
        }

        swap(arr, lo, boundary - 1 );
        return boundary - 1 ;
    }

    int selectPivot(int[] arr, int lo, int hi) {
        return lo;
    }


    static void swap(int[] arr, int a, int b) {
        int t = arr[a];
        arr[a] = arr[b];
        arr[b] = t;
    }
}
```

Quicksort

Recap: 3 functions

- Quicksort (
  1. partition around a pivot
  2. Quicksort left side
  3. Quicksort right side

Partition

- move pivot to first element in subsection
- keep track of three sections using two indices
  - # < p section } seen section
  - # > p section
  - unseen section
- expand seen section
  - if # encountered is < p, swap with the first number in > p section and move < p boundary
  - else
    expand > p boundary

finally swap pivot with last # in < p section

Select pivot

- lots of options, often use first element or random element

Linear - Time (integer) Sorting

Sort n keys, integers between 0 and k-1

Can do more than comparisons

for k polynomial in n, sort in O(n) time

Counting Sort

Sort

- count integers b/w 0 and 100

use counters just sorts the integers

use a list instead

```
countingsort(a)
- L = array of k empty lists
- for i in element in a:
      L(key(element)).append(element)

output = []
for sublist in L
    output.extend(sublist)
```

Time O(n+k)

if k is much bigger than n, than the bound (k=n²) -> O(n²)

Radix sort

- Imagine each integer in base b
- d = $\log_b k$ digits each of which is b/w 0 and b-1
- Sort by least significant digit
- sort by most significant digit

Req. requires a stable sort

↳ preserve relative order of elements w/ same key

example ~~329~~, 457, 657, 839

| | | | |
|---|---|---|---|
| 329 | 720 | 720 | 329 |
| 457 | 355 | 329 | 355 |
| 657 | 436 | 436 | 436 |
| 839 | 457 | 839 | 457 |
| 436 | 657 | 355 | 657 |
| 720 | 329 | 457 | 720 |
| 355 | 839 | 657 | 839 |
| | sorted | sorted | sorted |

- use counting sort for digit sort
  - ↳ $O(n+k) \Rightarrow O(n+b)$

total time = $O((n+b)d) \Rightarrow O((n+b)\log_b k)$ time

minimized for $b = n$

$O(n \log_n k)$

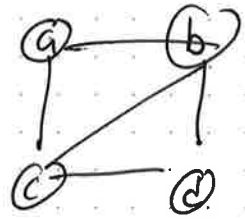
if $k$ is $O(n^c) \Rightarrow O(n \log_n n^c) = O(nc) = O(n)$

# Graphs

graph $G = (V, E)$

$V$ = set of vertices
$E$ = set of edges i.e. vertex pairs $(v_i, v_j)$

- unordered pair $\{v_i, v_j\}$ → undirected graph
- Ordered pair $(v_i, v_j)$ directed graph




Undirected

$V = \{a, b, c, d\}$

$E = \{\{a,b\}, \{a,c\}, \{b,c\}, \{b,d\}, \{c,d\}\}$


Directed

$V = \{a, b, c\}$

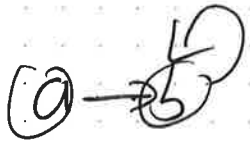
$E = \{(a,c), (b,c), (c,b), (b,a)\}$

has a cycle

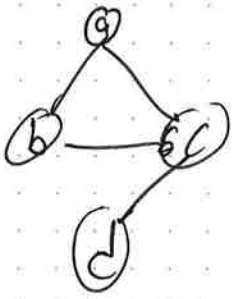
More examples

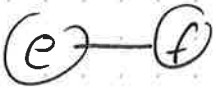



$V = \{a\}$
$E = \{\}$

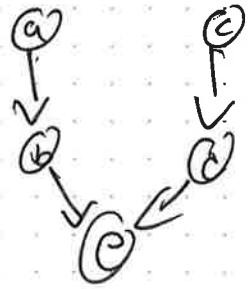



$V = \{a, b\}$
$E = \{(a,b), (b,b)\}$




disconnected graph with two connected components




weakly connected

Conly connected if edges in undirected

Degree: # of edges from node

Some facts $|E| = O(V^2)$

for a connected graph, $|E| > |V|$

Handshaking Lemma (undirected graph)

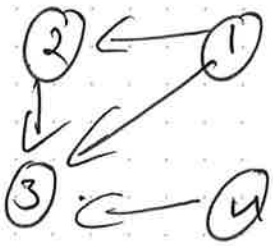
$$\sum_{v \in V} \text{degree}(v) = 2|E|$$

Graph Representations

Adjacency Matrix of $G = (V, E)$

where $V = \{1, 2, \ldots n\}$ is the $n \times n$ matrix $A$ given by $A[i,j] = 1$ if $(i,j) \in E$, 0 otherwise

space




| A | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 |
| 2 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 1 | 0 |

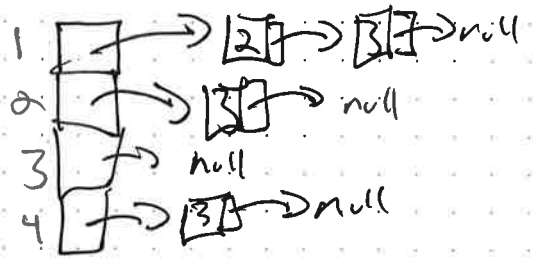
storage $\Theta(V^2)$

dense representation

Adjacency List of $G = (V, E)$

Array Adj of $|V|$ linked list for each vertex $v$ in $V$, Adj[v] stores the neighbors of $v$

Eg




Normally use a hash map (dictionary in python)

Space is $O(V+E)$, sparse representation

Implicit graphs: Adj(v) is a function

- Compute neighbors on the fly, i.e. class

Zero space

Object Oriented

- Object for each vertex
- v.neighbors = list of neighbors (Adj v)

Disadvantage: can't store multiple graphs on same vertex space

~~Applications~~

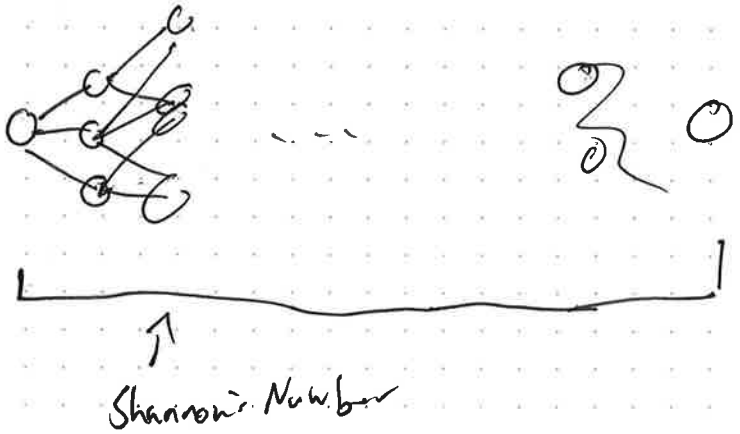
Graph Search: "explore a graph"

- find a path from start vertex $s$ to a desired vertex
- find all vertices reachable from some vertex $s$

Applications
- Web Crawling
- Social Networks
- Network Routing
- Garbage Collection
- Solving Puzzles

Class

- Position Graph
  - Vertex for each possible state
  - edge for each move
  - directed, moves are not always reversible




| # of men | # of games |
|---|---|
| 1 | 20 |
| 2 | 400 |
| 3 | 8902 |
| 4 | 197281 |

estimated number of positions: $5 \times 10^{52}$ ~~of 2~~ $\rightarrow 2^{165}$

# of games: $10^{\frac{123}{4}}$

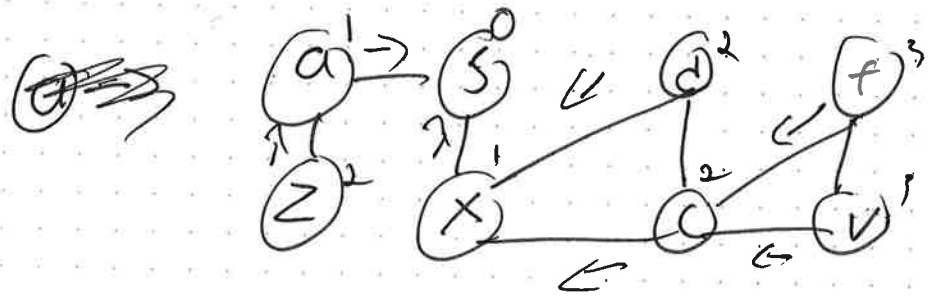
# Breadth First Search (BFS)

explore graph level by level from $s$

- level $0 = \{s\}$
- level $i$ = vertices reachable by path of $i$ edges but not fewer

- build level $i$ from level $i-1$ by trying all outgoing edges but ignoring vertices from previous levels.

```
BFS(s, Adj)
    level = {s: 0}        ← maps
    parent = {s: None}
    i = 1
    queue = []
    queue.push(s)
    while queue is not empty:
        for q = queue.dequeue()
            for b in Adj[a]
                if b not in level:
                    level[b] = level[a] + 1
                    parent[b] = a
                    queue.enqueue(b)
```




Queue = [s]
[a, x]
[x, z]
[z, c]
[c, d]
[d, f, v]
[f]
[v]

Analysis

- vertex $V$ enters queue only once
- $Adj[v]$ looped through only once
  - time $\sum_{v \in V} |Adj[v]| = \sum_{v \in V} deg(v) = |E|$ for directed graph
    $2|E|$ for undirected graph

$\Rightarrow O(E)$

If connected

$O(V+E)$ to also list vertices not reachable from $s$

Shortest Path:

- for every vertex $V$, fewest edges to get from $S$ to $V$ is $\begin{cases} level[v] & \text{if } v \text{ in bfs} \\ \infty & \text{otherwise (not reachable)} \end{cases}$
- parent pointers form the shortest path
  - to find shortest path take $V$, parent[V], parent[parent[V]] etc. until back to $S$

Depth First Search

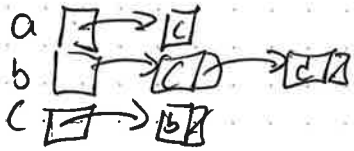
Edge Classification
Cycle Detection
Topological Sort

Recap Graph Search: explore a graph

Adjacency Lists: Array Adj of |V| linked lists

for each vertex a in V, Adj[a] stores as neighbors

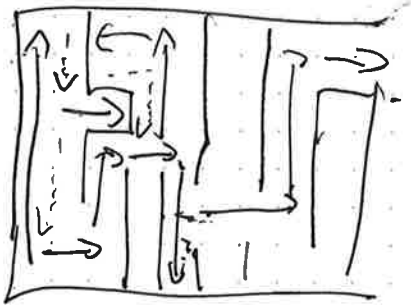

a
b
c


a
b
c

BFS: Explore level by level
- finds shortest path

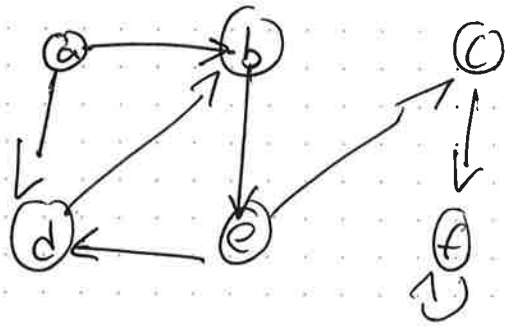
Depth First Search (DFS): exploring a maze

- follow a path until you get stuck
- backtrack along breadcrumbs until you reach a new section (i.e. unexplored neighbor)
- recursively explore
- don't repeat vertices



```
DFS(s, Adj)
  parent = {s: None}
```

```
DFS_visit(s, Adj, parent)
  for v in Adj[s]:
    if v not in parent:
      parent[v] = s
      DFS_visit(v, Adj, parent)
```

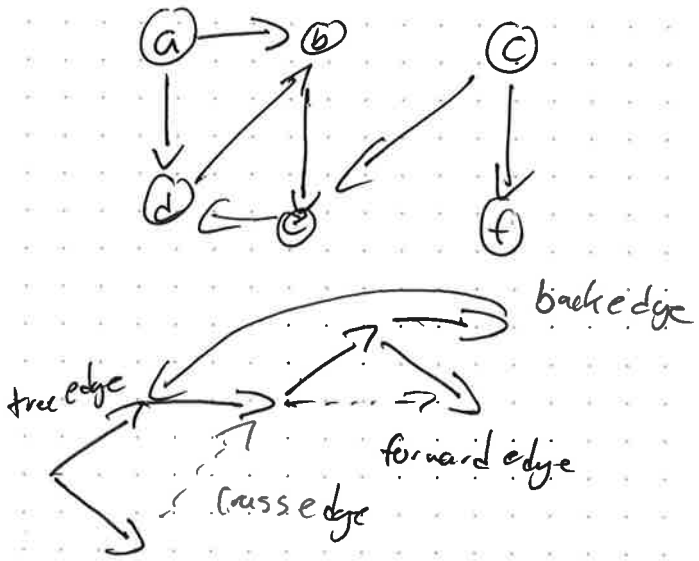



BFS/DFS

```
DFS(s, Adj)
  parent = {s: None}
  stack = []
  stack.push(s)
  while stack is not empty:
    a = stack.pop()
    for b in Adj[a]:
      if b not in parent:
        parent[b] = a
        stack.push(b)
```

order among neighbors slightly different

# Edge Classification




Tree Edge : formed by parent
Back Edge : to ancestor
Forward Edge : to descendant
Cross Edge : to subtree
other

To calculate use white vertices are in stack or not.

Back Edge : $V_i, V_j$ if $V_j$ is on stack and in progress
when $V_i$ is visited

Forward Edge : $V_i, V_j$ if $V_j$ is finished before $V_i$ and $V_i$ is on stack

Cross Edge : $V_i$ $V_j$ if $V_j$ completes before $V_i$ starts and $V_i$ is on stack

Analysis

DFS-visit a

Vertex a is pushed to stack only once

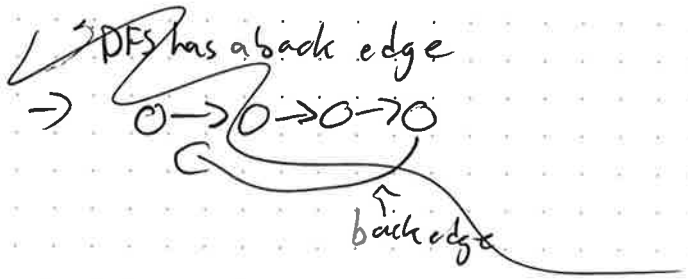
Adj[a] is looped through only once

$\Rightarrow \sum_{s \in V} |Adj[s]| = O(|E|)$ (handshake)

Get everything cost $|V|$ $O(|V|+|E|)$

Cycle Detection

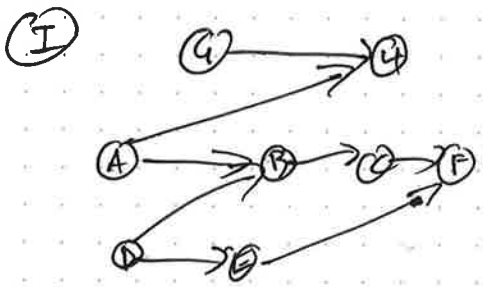
Graph G has a cycle if and only if DFS reveals a back edge




Do DFS, if there is a back edge then is a cycle

Job Scheduling: Given a Directed Acyclic Graph (DAG),

when vertices represent tasks & edges represent dependencies, order tasks w/o violating dependencies



Source : Vertex with in-degree 0

(A, G, D, I)

Try BFS, :

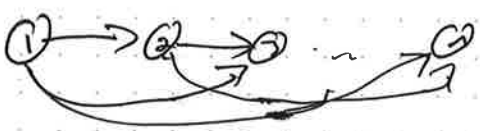
Topological Sort

Do DFS

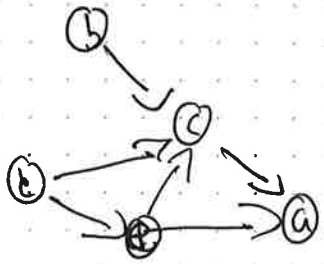
Use the reverse of finishing time

```
TSort(s, Adj, parent, order)
    for v in Adj[s]:
        if v not in parent
            parent[v] = s
            TSort(s, Adj, parent, order)
    order.append(s)
```

One more sorting algorithm

Create graph 

Tsort graph



| A | a | b | c | d | e |
|---|---|---|---|---|---|
| a | 0 | 0 | 0 | 0 | 0 |
| b | 0 | 0 | 1 | 0 | 0 |
| c | 1 | 0 | 0 | 0 | 0 |
| d | 0 | 0 | 1 | 0 | 0 |
| e | 1 | 0 | 1 | 0 | 0 |

b d e c e a

| A | b | d | c | e | a |
|---|---|---|---|---|---|
| b | 0 | 0 | | | |
| d | | | | | |
| c | | | | | |
| e | | | | | |
| a | | | | | |

| | c | e | a | b | d |
|---|---|---|---|---|---|
| a | 0 | 0 | 0 | 0 | 0 |
| c | 1 | 0 | 0 | 0 | 0 |
| e | 1 | 1 | 0 | 0 | 0 |
| d | 0 | 1 | 1 | 0 | 0 |
| b | 0 | 1 | 0 | 0 | 0 |