23 Revision

Instructor : Ke Wei（柯韋）

A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP122/19/

Bachelor of Science in Computing, School of Public Administration, Macao Polytechnic Institute

April 24, 2019

Outline

Asymptotic Algorithm Analysis

- The asymptotic analysis of an algorithm determines the running time in the big-Oh notation.
- We find the *maximum possible* number of primitive operations executed, as a function of the input size.
- We express this function with the Big-Oh notation, dropping *constant factors* and *lower-order terms*.
- Example: an algorithm that executes at most $\frac{1}{7}n\log n + 5n - 2$ primitive operations is said "runs in $\mathcal{O}(n\log n)$ time".
- We focus on counting *repeated operations*.

Time Complexities of Typical Loops

$\mathcal{O}(n)$

```
def lin(n):
    for i in range(n):
        ...
```

$\mathcal{O}(n^2)$

```
def tri(n):
    for i in range(n):
        for j in range(i, n):
            ...
```

$\mathcal{O}(n \log n)$

```
def mrgs(n):
    mrgs(n//2)
    mrgs(n-n//2)
    for i in range(n):
        ...
```

$\mathcal{O}(\text{tree size})$

```
def preord(root):
    if root:
        ...
        preord(root.left)
        preord(root.right)
```

Linear Structures

- A linear structure is a structure in which each element (except the last) has a *unique successor*.
- If y is a successor of x, then x is called a *predecessor* of y.
- Arrays and linked lists are examples of linear structures.
- The successor of an element in an array is determined by the *index* of the element.
- The successor of an element in a linked list is determined by the explicit *link field* (reference field) of the element.
- To find an element in a linear structure, you may need to scan over all the elements.

Linked List Operations

- Push a node p to the beginning of a singly linked list h.

  $$p.nxt,\ h = h,\ p$$

- Insert a node p after a node q in a singly lined list.

  $$p.nxt,\ q.nxt = q.nxt,\ p$$

- Push a node p before a node q in a circular doubly linked list.

  $$p.prv,\ p.nxt = q.prv,\ q$$
  $$p.prv.nxt = p.nxt.prv = p$$

- Construct an empty circular doubly linked list with dummy node *dummy*.

  $$dummy.prv = dummy.nxt = dummy$$

Filtering a Singly Linked List Recursively

- A singly linked list can be reduced to a smaller list by splitting it into the head node and the tail list, the tail list can be empty.
- Usually, the base case is when the list is empty, thus cannot be split.
- The *filter_even* function copies the nodes containing even numbers and form a new list.

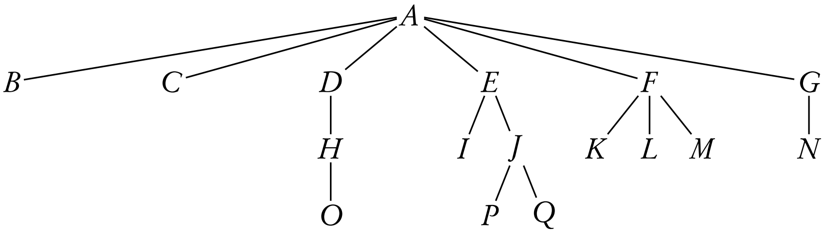
```
def filter_even(h):
    if not h:
        return None
    elif h.elm%2 != 0:
        return filter_even(h.nxt)
    else:
        return Node(h.elm, filter_even(h.nxt))
```

Abstract Data Types

- Stacks
  - LIFO, elements are taken out in their reverse insertion order.
  - Operations: *push*, *pop*, *top* and __*bool*__.
- Queues
  - FIFO, elements are taken out in exactly their insertion order.
  - Operations: *push_back*, *pop*, *top* and __*bool*__.
- Priority queues
  - The minimum element is the first element to take out. The __*le*__ method must be defined between two elements.
  - Operations: *push*, *pop_min*, *get_min* and __*bool*__.
- Associative arrays
  - An associative array is a map from keys to values, where each of the keys is unique.
  - The binary search tree implementation of associative arrays requires that the __*lt*__ method must be defined between two elements.
  - Operations: __*getitem*__, __*setitem*__, __*contains*__ (a key) and __*iter*__ (over keys).

Trees — Concepts and Terminologies

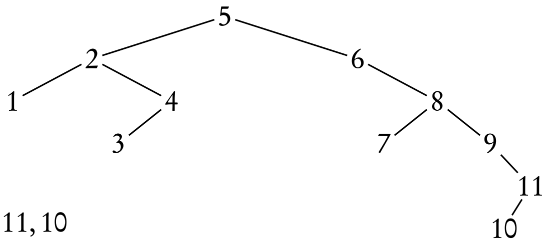
- The root and subtrees
- Parents, children and siblings
- Internal nodes and leaf nodes (external nodes)
- Paths, depths and heights
- The height and depth of a tree
- Ancestors and descendants, proper
- Pre-order and post-order traversals, the Euler tour



Reconstructing a Binary Search Tree

Given the ordering and the pre-order traversal sequence, we can reconstruct the binary search tree that admits the ordering and generates the pre-order sequence, by using the simple insertion procedure:

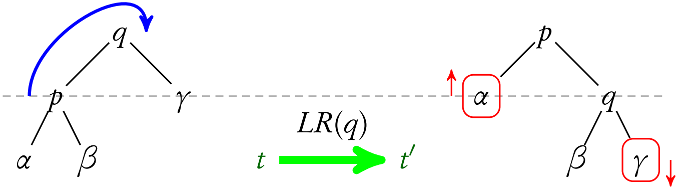
- If the tree is empty, make the new node the tree.
- If the new node is less than the root, insert it to the left sub-tree, otherwise insert it to the right sub-tree.



Pre-order: $5, 2, 1, 4, 3, 6, 8, 7, 9, 11, 10$

Binary Search Tree Rotations

- Let p be the left child of q, α and β the left and right subtrees of p, γ the right subtree of q.
- A *left-to-right rotation* on (sub)tree t rooted at node q is that
  - let q be the right child of p;
  - let β be the left subtree of q;
  - let p be the new root of the resulted tree t'.
- After the rotation, α is shallowed and γ is deepened (by one level), and the depth of β is unchanged.
- It's symmetric for right-to-left rotations.
- *Tree rotations do not change the in-order traversal sequence of a binary tree.*



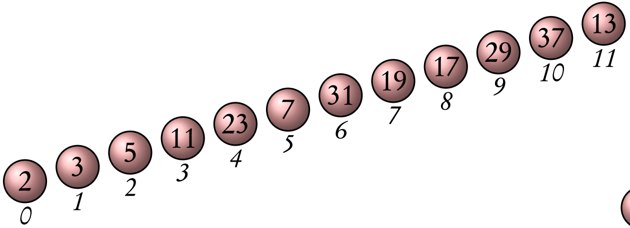
Iterating over the Keys of a Binary Search Tree in a Range

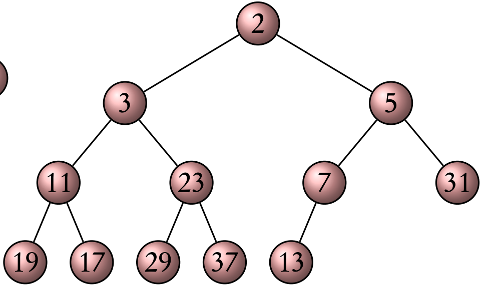
- First, we check if the tree is empty.
- Next, we compare the root (*root.key*) with the lower and upper bounds (*lower*, *upper*).
- Only if $lower \leqslant root.key$, we need to recur to the left subtree.
- Only if $root.key \leqslant upper$, we need to recur to the right subtree.

```
def iter_between(root, lower, upper):
    if root:
        if lower <= root.key:
            yield from iter_between(root.left, lower, upper)
            if root.key <= upper:
                yield root.key
        if root.key <= upper:
            yield from iter_between(root.right, lower, upper)
```

Array-Based Heaps

- If a complete binary tree also has the heap property, then such a heap can be stored in an array-based list a.
- Obviously, the root $a[0]$ contains the minimum element.
- The parent-child relation can be computed by the indices — the left child of $a[i]$ is $a[2i+1]$, the right child of $a[i]$ is $a[2i+2]$, and the parent of $a[i]$ is $a\left[\lfloor \frac{i-1}{2} \rfloor\right]$.

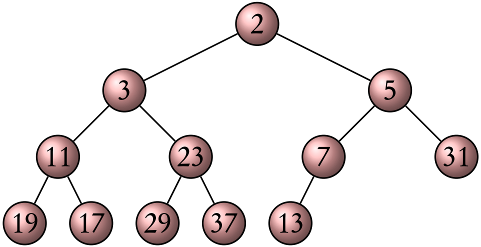




Recovering the Heap Property by Sifting Down

If we want to remove the root, we need to relocate a node in the tree to the root, and we must recover the heap property.

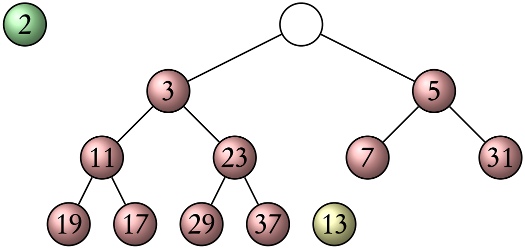
- We can only detach the bottom-right most node x, in order to maintain the complete binary tree. This is the last element in the array.
- We put x to the root, and sift it down to a proper location where the children are no less, maintaining the heap property.
- We must choose the least node among x and its two children at each step. This is in fact a rotation along some path.



Recovering the Heap Property by Sifting Down

If we want to remove the root, we need to relocate a node in the tree to the root, and we must recover the heap property.

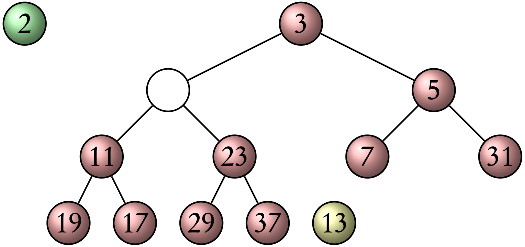
- We can only detach the bottom-right most node x, in order to maintain the complete binary tree. This is the last element in the array.
- We put x to the root, and sift it down to a proper location where the children are no less, maintaining the heap property.
- We must choose the least node among x and its two children at each step. This is in fact a rotation along some path.



Recovering the Heap Property by Sifting Down

If we want to remove the root, we need to relocate a node in the tree to the root, and we must recover the heap property.

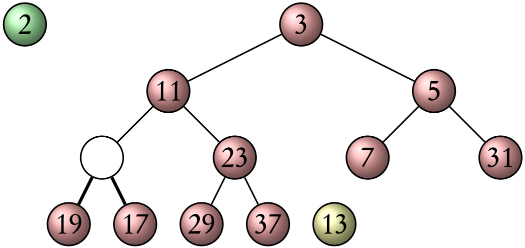
- We can only detach the bottom-right most node x, in order to maintain the complete binary tree. This is the last element in the array.
- We put x to the root, and sift it down to a proper location where the children are no less, maintaining the heap property.
- We must choose the least node among x and its two children at each step. This is in fact a rotation along some path.



Recovering the Heap Property by Sifting Down

If we want to remove the root, we need to relocate a node in the tree to the root, and we must recover the heap property.

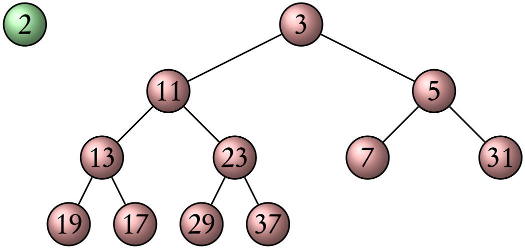
- We can only detach the bottom-right most node x, in order to maintain the complete binary tree. This is the last element in the array.
- We put x to the root, and sift it down to a proper location where the children are no less, maintaining the heap property.
- We must choose the least node among x and its two children at each step. This is in fact a rotation along some path.



Recovering the Heap Property by Sifting Down

If we want to remove the root, we need to relocate a node in the tree to the root, and we must recover the heap property.

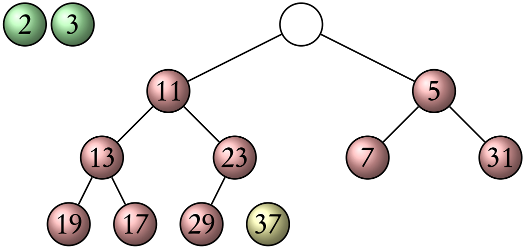
- We can only detach the bottom-right most node x, in order to maintain the complete binary tree. This is the last element in the array.
- We put x to the root, and sift it down to a proper location where the children are no less, maintaining the heap property.
- We must choose the least node among x and its two children at each step. This is in fact a rotation along some path.



Recovering the Heap Property by Sifting Down

If we want to remove the root, we need to relocate a node in the tree to the root, and we must recover the heap property.

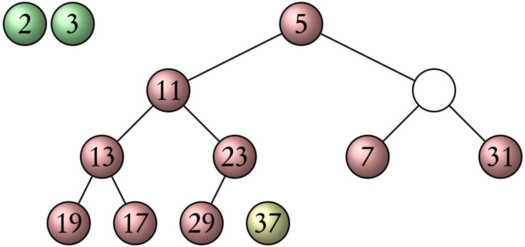
- We can only detach the bottom-right most node x, in order to maintain the complete binary tree. This is the last element in the array.
- We put x to the root, and sift it down to a proper location where the children are no less, maintaining the heap property.
- We must choose the least node among x and its two children at each step. This is in fact a rotation along some path.



Recovering the Heap Property by Sifting Down

If we want to remove the root, we need to relocate a node in the tree to the root, and we must recover the heap property.

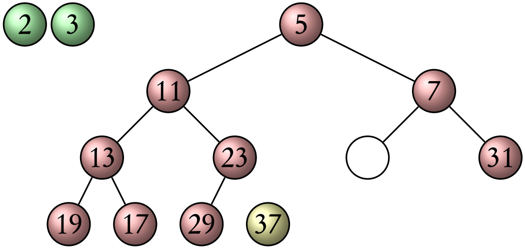
- We can only detach the bottom-right most node x, in order to maintain the complete binary tree. This is the last element in the array.
- We put x to the root, and sift it down to a proper location where the children are no less, maintaining the heap property.
- We must choose the least node among x and its two children at each step. This is in fact a rotation along some path.



Recovering the Heap Property by Sifting Down

If we want to remove the root, we need to relocate a node in the tree to the root, and we must recover the heap property.

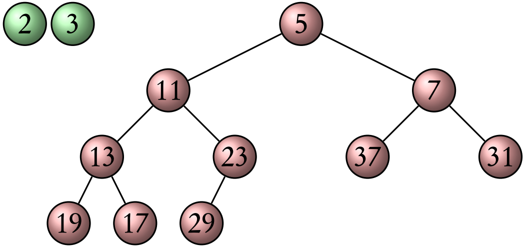
- We can only detach the bottom-right most node x, in order to maintain the complete binary tree. This is the last element in the array.
- We put x to the root, and sift it down to a proper location where the children are no less, maintaining the heap property.
- We must choose the least node among x and its two children at each step. This is in fact a rotation along some path.



Recovering the Heap Property by Sifting Down

If we want to remove the root, we need to relocate a node in the tree to the root, and we must recover the heap property.

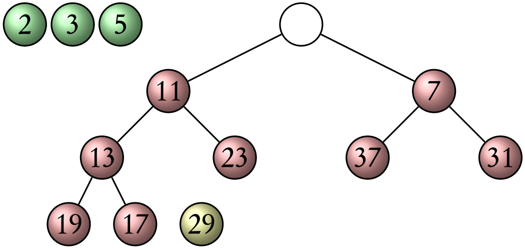
- We can only detach the bottom-right most node x, in order to maintain the complete binary tree. This is the last element in the array.
- We put x to the root, and sift it down to a proper location where the children are no less, maintaining the heap property.
- We must choose the least node among x and its two children at each step. This is in fact a rotation along some path.



Recovering the Heap Property by Sifting Down

If we want to remove the root, we need to relocate a node in the tree to the root, and we must recover the heap property.

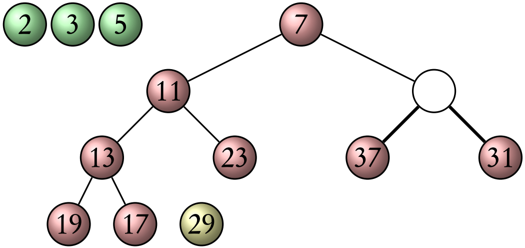
- We can only detach the bottom-right most node x, in order to maintain the complete binary tree. This is the last element in the array.
- We put x to the root, and sift it down to a proper location where the children are no less, maintaining the heap property.
- We must choose the least node among x and its two children at each step. This is in fact a rotation along some path.



Recovering the Heap Property by Sifting Down

If we want to remove the root, we need to relocate a node in the tree to the root, and we must recover the heap property.

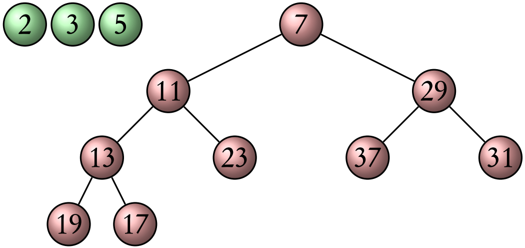
- We can only detach the bottom-right most node x, in order to maintain the complete binary tree. This is the last element in the array.
- We put x to the root, and sift it down to a proper location where the children are no less, maintaining the heap property.
- We must choose the least node among x and its two children at each step. This is in fact a rotation along some path.



Recovering the Heap Property by Sifting Down

If we want to remove the root, we need to relocate a node in the tree to the root, and we must recover the heap property.

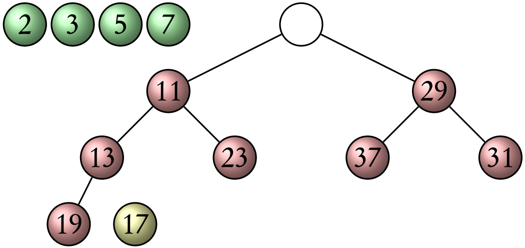
- We can only detach the bottom-right most node x, in order to maintain the complete binary tree. This is the last element in the array.
- We put x to the root, and sift it down to a proper location where the children are no less, maintaining the heap property.
- We must choose the least node among x and its two children at each step. This is in fact a rotation along some path.



Recovering the Heap Property by Sifting Down

If we want to remove the root, we need to relocate a node in the tree to the root, and we must recover the heap property.

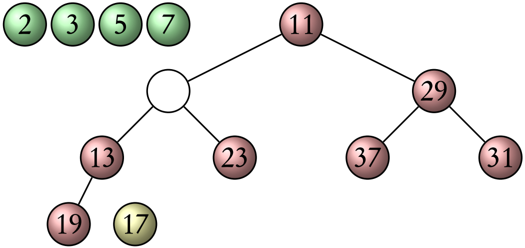
- We can only detach the bottom-right most node x, in order to maintain the complete binary tree. This is the last element in the array.
- We put x to the root, and sift it down to a proper location where the children are no less, maintaining the heap property.
- We must choose the least node among x and its two children at each step. This is in fact a rotation along some path.



Recovering the Heap Property by Sifting Down

If we want to remove the root, we need to relocate a node in the tree to the root, and we must recover the heap property.

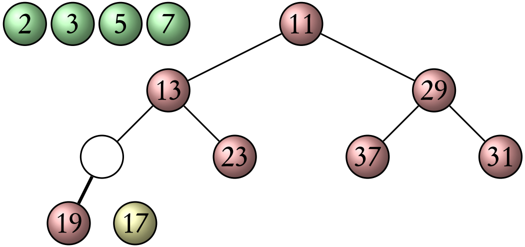
- We can only detach the bottom-right most node x, in order to maintain the complete binary tree. This is the last element in the array.
- We put x to the root, and sift it down to a proper location where the children are no less, maintaining the heap property.
- We must choose the least node among x and its two children at each step. This is in fact a rotation along some path.



Recovering the Heap Property by Sifting Down

If we want to remove the root, we need to relocate a node in the tree to the root, and we must recover the heap property.

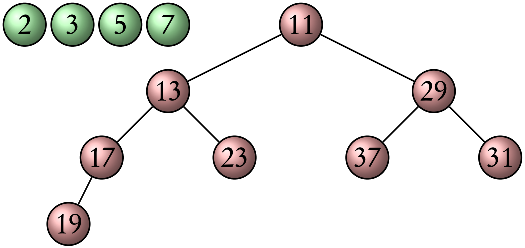
- We can only detach the bottom-right most node x, in order to maintain the complete binary tree. This is the last element in the array.
- We put x to the root, and sift it down to a proper location where the children are no less, maintaining the heap property.
- We must choose the least node among x and its two children at each step. This is in fact a rotation along some path.



Recovering the Heap Property by Sifting Down

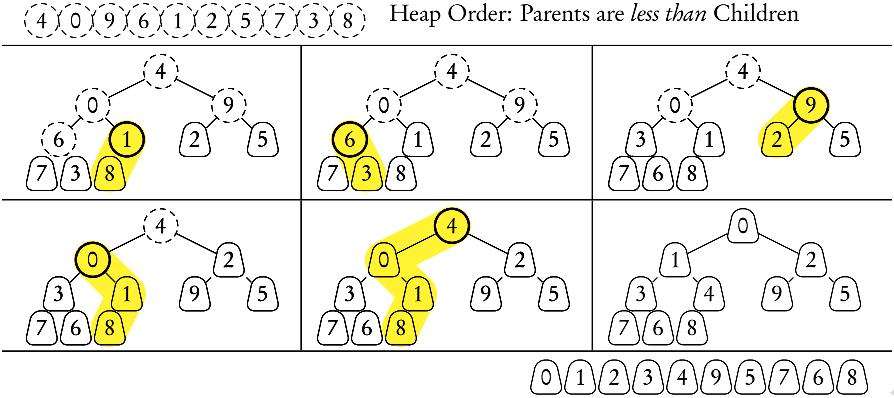
If we want to remove the root, we need to relocate a node in the tree to the root, and we must recover the heap property.

- We can only detach the bottom-right most node x, in order to maintain the complete binary tree. This is the last element in the array.
- We put x to the root, and sift it down to a proper location where the children are no less, maintaining the heap property.
- We must choose the least node among x and its two children at each step. This is in fact a rotation along some path.



Heapifying by Sifting-down

We may build the heap by sifting down, starting from the bottom up to the top. This method takes only linear time. Sifting a node down can be regarded as merging the node with its two sub-heaps into a bigger heap.



General Sorting Algorithms

Name	Worst Time	Average Time	Auxiliary Space on Arrays	In Place on Arrays	Linked Lists	Stable on Arrays
Insertion Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Yes	Yes	Yes
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Yes	Yes	No
Heapsort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	Yes	No	No
Mergesort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	No	Yes	Yes
Quicksort	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	Yes	Yes	No

For any sorting algorithm based on comparisons, $\log(n!) \in \Omega(n \log n)$ is the lower bound of the worst-case time complexity.

Insertion Sort on Arrays

The *insertion_sort_a* function sorts an array-based list *a* in place.

```
def insertion_sort_a(a):
    for i in range(1, len(a)):
        t = a[i]
        j = i
        while j > 0 and not a[j-1] <= t:
            a[j] = a[j-1]
            j -= 1
        a[j] = t
```

Three-way Merge of Singly Linked Lists

The *merge_three* function merges three sorted linked lists a, b, c into one linked list.

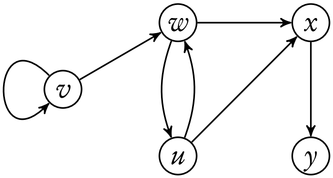
```
def merge_three(a, b, c):
    t = [a, b, c]
    if t == [None, None, None]:
        return None

    m = 0
    for i in range(1, 3):
        if (t[m] is None or
            t[i] is not None and
            t[i].elm < t[m].elm):
            m = i

    s = t[m]
    r = s
    t[m] = t[m].nxt
    while t != [None, None, None]:
        m = 0
        for i in range(1, 3):
            if (t[m] is None or
                t[i] is not None and
                t[i].elm < t[m].elm):
                m = i

        r.nxt = t[m]
        r = r.nxt
        t[m] = t[m].nxt

    r.nxt = None
    return s
```

Graphs and Connectivity

- A graph $G = \langle V, E \rangle$ consists of a set of *vertices* (vertex): V, and a set of *edges*: E. Each edge is a pair (v, w), where $v, w \in V$.
- If the pairs are ordered, then the edges are *directed*, and the graph is called a directed graph. Otherwise the graph is *undirected*.
- A vertex w is *adjacent* to a vertex v if and only if $(v, w) \in E$. In an undirected graph containing edge (v, w), and hence (w, v), v is adjacent to w and w is adjacent to v.
- A path in a graph is a sequence of vertices $v_1, v_2, \ldots, v_n$ such that $(v_i, v_{i+1}) \in E$, for $1 \leqslant i < n$.
- An undirected graph is *connected* if there is a path from every vertex to every other vertex.
- A directed graph with such a property is called *strongly connected*.
- If a directed graph is not strongly connected, but it would be connected by ignoring the direction of edges, then it is called *weakly connected*.

Adjacency Matrix

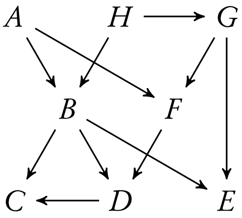
We use a two-dimensional array a to represent a graph. It is known as an adjacency matrix representation. For each edge (u, v), we set $a[u][v] \leftarrow 1$; and all other entries in the array are set to 0.



sources	u	v	w	x	y	destinations
u	0	0	1	1	0	
v	0	1	1	0	0	
w	1	0	0	1	0	
x	0	0	0	0	1	
y	0	0	0	0	0	

Topological Sort

- A topological sort is an ordering of vertices in a directed acyclic graph, such that if there is a path from v to w, then w appears after v in the ordering.
- It has an interpretation that the starting of w is dependent on the completion of v.
- There may be more than one topological orders for a given graph.



A, H, G, B, F, D, C, E ✔

H, A, B, G, E, F, D, C ✔

A, H, G, B, F, C, D, E ✘

A, H, G, E, F, B, D, C ✘

Depth First Search

- Depth-first search is a generalization of pre-order traversal.
- Starting from some vertex v, we visit v and then recursively traverse all the vertices adjacent to v.
- We need to be careful to avoid cycles. When we visit a vertex v, we mark it *visited*, and recursively perform depth-first search on all the adjacent vertices that have not been visited.
- Although we must mark the vertex before exploring its adjacent vertices, we may perform the real processing *before* and/or *after* the exploration, according to the application.
- We may use a stack to explicitly express the searching sequence without recursion.
- DFS generates a *spanning tree* if the graph is connected or rooted at v.

Breadth First Search

- Breadth-first search is a by-level search strategy.
- Starting from some vertex v, we visit v and then all the vertices adjacent to v, and then their adjacent vertices, and so on.
- We need the same trick, the *visited* marks, as in DFS to avoid cycles.
- A recursive definition of breadth-first search is not possible. We need a FIFO queue to line up the vertices to visit.
  - We first enqueue vertex v.
  - We repeatedly dequeue a new vertex, visit it, enqueue its adjacent vertices.
  - Until all the reachable vertices have been visited (the queue is empty.)
- BFS also generates a spanning tree if the graph is connected or rooted at v.

Dijkstra's Algorithm

- The distance of a vertex v from a vertex s is the total weight of a path between s and v.
- Dijkstra's algorithm computes the shortest distances of all the vertices from a given starting vertex s.
- Assumptions:
  - The graph is connected. Edges of infinite weight can be introduced to apply the algorithm to a general graph.
  - The edge weights are *non-negative*. A path can not be shortened by appending more edges.
- We grow a set of "known" vertices, beginning with s and eventually containing all the vertices.
- We store with each vertex v a field $dist(v)$, called the *distance* of v, representing the shortest distance of v from s in the *subgraph* consisting of
  - the set of "known" vertices, often called the "cloud", and
  - their adjacent vertices, with only the edges from the "known" vertices (the cloud).

Edge Relaxation

- At each step:
  - we add to the "known" set the vertex u outside the set with the shortest distance field, then
  - we update the distance fields of the vertices adjacent to u, if the fields can be shortened.
- Consider an edge $e = (u, z)$ such that
  - u is the vertex most recently added to the "known" set.
  - z is not in the "known" set.
- The relaxation of the edge e updates $dist(z)$, the distance of z, as follows:

$$dist(z) \leftarrow \min(dist(z), dist(u) + weight(e)).$$

- We also record u as the parent of z in the spanning tree. We can then use the spanning tree to track back the path from s to z.

Reasoning about Recursive Functions using Mathematical Induction

For an integer $n \geqslant 0$ and an arbitrary number s, function f is defined below.

$$f(n,s) = \begin{cases} s & \text{if } n = 0, \\ 2 + f(n-1, f(n-1, f(n-1, s))) & \text{if } n \geqslant 1. \end{cases}$$

Prove by mathematical induction that $f(n,s) = s + 3^n - 1$ for all $n \geqslant 0$.

1. Base case: when $n = 0$, we have $f(0,s) = s = s + 1 - 1 = s + 3^0 - 1$.
2. Induction step: when $n \geqslant 1$,

$$\begin{aligned} f(n,s) &= 2 + f(n-1, f(n-1, f(n-1, s))) && [\text{by } f] \\ &= 2 + f\left(n-1, f\left(n-1, s + 3^{n-1} - 1\right)\right) && [\text{by induction hypothesis}] \\ &= 2 + f\left(n-1, \left(s + 3^{n-1} - 1\right) + 3^{n-1} - 1\right) && [\text{by induction hypothesis}] \\ &= 2 + \left(s + 2\left(3^{n-1} - 1\right)\right) + 3^{n-1} - 1 && [\text{by induction hypothesis}] \\ &= s + 3^n - 1. && [\text{by arithmetic}] \end{aligned}$$