Objectives

Data structures: Heaps & Graphs

Review: Priority Queues

Can use priority queues to sort

Sort runtime should be O(n log n)

However, cannot implement PQs with “known” data structures arrays and lists to meet desired runtime

→Motivates use of Heap to implement PQ

→**Goal**: show results in O(n log n) time

Implementing Priority Queues

Operation	Unsorted List	Sorted List	Sorted Array
StartHeap(N)			
Insert(H, v)			
FindMin(H)			
Delete(H, i)			
ExtractMin(H)			

Implementing Priority Queues

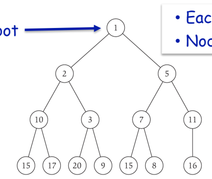
Operation	Unsorted List	Sorted List	Sorted Array
StartHeap(N)	O(1)	O(1)	O(N)
Insert(H, v)	O(1)	O(n)	O(n)
FindMin(H)	O(n)	O(1)	O(1)
Delete(H, i)	O(n)	O(n)	O(n)
ExtractMin(H)	O(n)	O(1)	O(n)

Review: Heap

Combines benefits of sorted array and list

Balanced binary tree

root



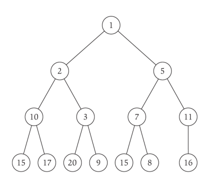
- Each node has *at most* 2 children
- Node value is its key

Heap order: each node's key is at least as large as its parent's

Note: not a binary search tree

Implementing a Heap

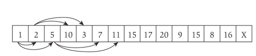
Option 1: Use pointers



- Each node keeps
  - Element it stores, key
  - 3 pointers: 2 children, parent

Option 2: No pointers

- Requires knowing upper bound on *n*
- For node at position *i*
  - left child is at *2i*
  - right child is at *2i+1*



If know child's position, what is the position of parent?

Implementing a Heap: Operations

Adding an element?

- Could add element to last position
  - What are possible scenarios?
    - Heap is no longer balanced
    - Something that is almost a heap but a little off
    - Need a `Heapify-up` procedure to fix our heap

Heapify-Up

Heap

Position where node added

```
Heapify-up(H, i):
   if i > 1 then
     let j=parent(i)=floor(i/2)
     if key[H[i]] < key[H[j]] then
         swap array entries H[i] and H[j]
         Heapify-up(H, j)
```

Can insert a new element in a heap of *n* elements in O(log n) time

Deleting an Element

Delete at position *i*

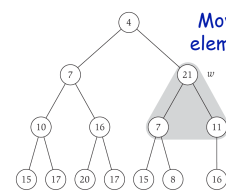
Not only removes an element

- Messes up heap order
- Leaves a "hole" in the heap

Not as straightforward as `Heapify-Up`

- Need to fill-in element where hole was
  - Patch hole: move n^{th} element into i^{th} spot
- Then adjust heap to be in order
  - At position *i* because moved n^{th} item up to *i*

Deleting an Element



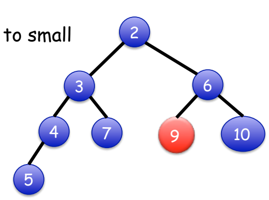
Moved 21 to where element was removed

Two possibilities: element *w* is

- Too small: violation is between it and parent → `Heapify-Up`
- Too big: with one or both children → `Heapify-Down`

Deleting an Element

Example where new key is to small

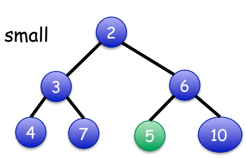


Delete 9

Replace with 5

Deleting an Element

Example where new key is to small



Delete 9

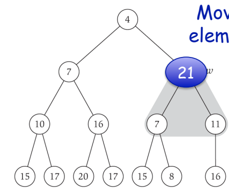
Replace with 5

But 5 < 6, so need to `Heapify-Up`

Heapify-Down

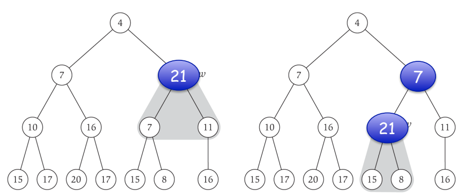
```
Heapify-down(H, i):
   Let n = length(H)
   if 2i > n then
     Terminate with H unchanged
   else if 2i < n then
     let left=2i and right=2i+1
     let j be index that minimizes
              key[H[left]] and key[[H[right]]
   else if 2i = n then
     Let j=2i

   if key[H[j]] < key[H[i]] then
     swap array entries H[i] and H[j]
     Heapify-down(H, j)
```


Practice: Heapify-Down

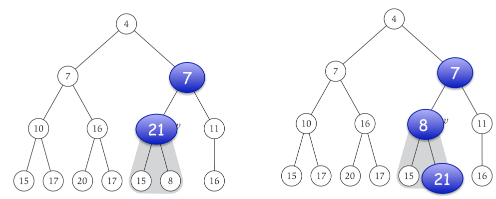


Moved 21 to where element was removed

Practice: Heapify-Down



Practice: Heapify-Down



Runtime of Heapify-Down?

```
Heapify-down(H, i):
   Let n = length(H)
   if 2i > n then
     Terminate with H unchanged
   else if 2i < n then
     let left=2i and right=2i+1
     let j be index that minimizes
              key[H[left]] and key[[H[right]]
   else if 2i = n then
     Let j=2i

   if key[H[j]] < key[H[i]] then
     swap array entries H[i] and H[j]
     Heapify-down(H, j)
```


Runtime of Heapify-Down: O(log n)

Computation of j: O(1)

Swap: O(1)

How many swaps: O(log n)

Implementing Priority Queues with Heaps

Operation	Description	Run Time
StartHeap(N)	Creates an empty heap that can hold N elements	
Insert(H, v)	Inserts item *v* into heap *H*	
FindMin(H)	Identifies minimum element in heap *H* but does not remove it	
Delete(H, i)	Deletes element in heap position *i*	
ExtractMin(H)	Identifies and deletes an element with minimum key from heap	

Implementing Priority Queues with Heaps

Operation	Description	Run Time
StartHeap(N)	Creates an empty heap that can hold N elements	O(N)
Insert(H, v)	Inserts item *v* into heap *H*	O(log n)
FindMin(H)	Identifies minimum element in heap *H* but does not remove it	O(1)
Delete(H, i)	Deletes element in heap position *i*	O(log n)
ExtractMin(H)	Identifies and deletes an element with minimum key from heap	O(log n)

Implementing Priority Queues

Operation	Heap	Unsorted List	Sorted List
StartHeap(N)	O(N)		
Insert(H, v)	O(log n)		
FindMin(H)	O(1)		
Delete(H, i)	O(log n)		
ExtractMin(H)	O(log n)		

Implementing Priority Queues

Operation	Heap	Unsorted List	Sorted List
StartHeap(N)	O(N)	O(1)	O(1)
Insert(H, v)	O(log n)	O(1)	O(n)
FindMin(H)	O(1)	O(n)	O(1)
Delete(H, i)	O(log n)	O(n)	O(n)
ExtractMin(H)	O(log n)	O(n)	O(1)

Additional Heap Operations

Access given element of PQ

- Maintain additional array `Position` that stores current position of each element in heap

Operations:

- Delete(H, Position[v])
  - Does not increase overall running time
- ChangeKey(H, v, a)
  - Changes key value of element v to key(v) = a
  - Identify position of element v in array (`Position` array)
  - Change key, heapify

GRAPHS

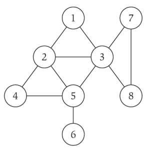
Undirected Graphs G = (V, E)

V = nodes (vertices)

E = edges between pairs of nodes

Captures pairwise relationship between objects

Graph size parameters: n = |V|, m = |E|

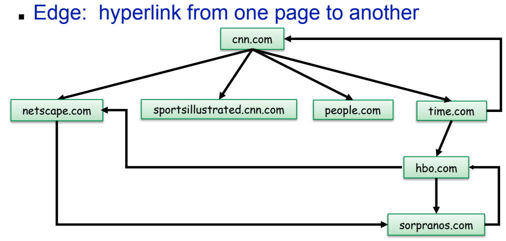


V = { 1, 2, 3, 4, 5, 6, 7, 8 }
E = { 1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6 }
n = 8
m = 11

World Wide Web

Web graph

- Node: web page
- Edge: hyperlink from one page to another



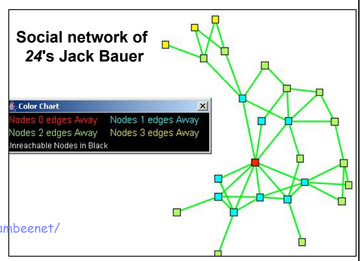
Directed Graph

Social Networks

Node: people; Edge: relationship between 2 people

Everything Bad Is Good for You: How Today's Popular Culture Is Actually Making Us Smarter

- Television shows have complex plots, complex social networks

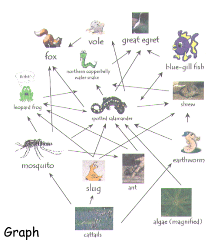


http://www.cs.duke.edu/csed/harambeenet/modules.html

Ecological Food Web

Food web graph

- Node = species
- Edge = from prey to predator



Directed Graph

Reference: http://www.twingroves.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.giff

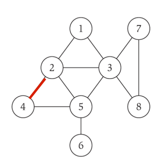
Graph Applications

Graph	*Nodes*	*Edges*
transportation	street intersections	highways
communication	computers	fiber optic cables
World Wide Web	web pages	hyperlinks
social	people	relationships
food web	species	predator-prey
software systems	functions	function calls
scheduling	tasks	precedence constraints
circuits	gates	wires

Graph Representation: Adjacency Matrix

n×n matrix with A_{uv} = 1 if (u, v) is an edge

- Two representations of each edge (symmetric matrix)
- Space?
- Checking if (u, v) is an edge?
- Identifying all edges?

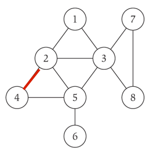


	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	1	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Graph Representation: Adjacency Matrix

$n \times n$ matrix with $A_{uv} = 1$ if (u, v) is an edge

- Two representations of each edge (symmetric matrix)
- Space proportional to n^2
- Checking if (u, v) is an edge takes $\Theta(1)$ time
- Identifying all edges takes $\Theta(n^2)$ time



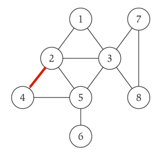
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	1	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

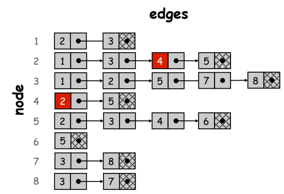
31

Graph Representation: Adjacency List

Node indexed array of lists

- Two representations of each edge
- Space?
- Checking if (u, v) is an edge?
- Identifying all edges?





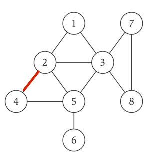
32

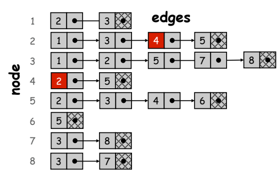
Graph Representation: Adjacency List

Node indexed array of lists

- Two representations of each edge
- Space = $2m + n = O(m + n)$
- Checking if (u, v) is an edge takes $O(\deg(u))$ time
- Identifying all edges takes $\Theta(m + n)$ time

degree = number of neighbors of u





33

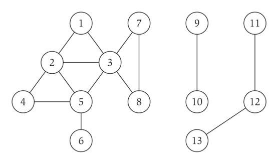
Paths and Connectivity

Def. A *path* in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k$

- each consecutive pair v_i, v_{i+1} is joined by an edge in E

Def. A path is *simple* if all nodes are distinct

Def. An undirected graph is *connected* if $\forall$ pair of nodes u and v, there is a path between u and v

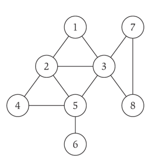


- Short path
- Distance

34

Cycles

Def. A *cycle* is a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first $k-1$ nodes are all distinct



cycle C = 1-2-4-5-3-1

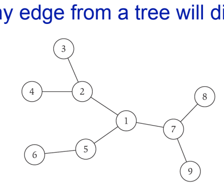
35

Trees

Def. An undirected graph is a *tree* if it is connected and does not contain a cycle

Simplest connected graph

- Deleting any edge from a tree will disconnect it

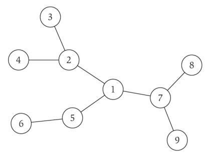


36

Trees

Theorem. Let G be an undirected graph on *n* nodes. Any two of the following statements imply the third.

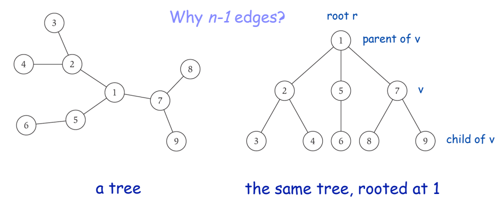
- G is connected
- G does not contain a cycle
- G has n-1 edges



Rooted Trees

Given a tree T, choose a root node *r* and orient each edge away from *r*

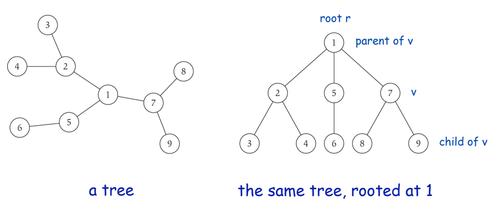
Models hierarchical structure



Rooted Trees

Why *n-1* edges?

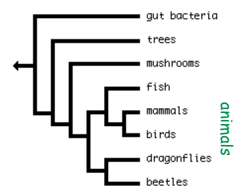
- Each node except for root has an edge to its parent



Phylogeny Trees

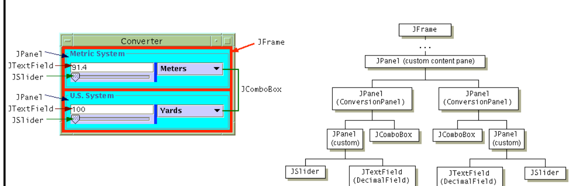
Describe evolutionary history of species

- ancestral species to mammals and birds
  - not to other species shown
- mammals and birds share a common ancestor that they do not share with other species
- all animals are descended from an ancestor not shared with mushrooms, trees, and bacteria



GUI Containment Hierarchy

Describe organization of GUI widgets



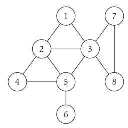
Reference: http://java.sun.com/docs/books/tutorial/uiswing/overview/anatomy.html

GRAPH TRAVERSAL

Connectivity

s-t connectivity problem. Given two node *s* and *t*, is there a path between *s* and *t*?

s-t shortest path problem. Given two node *s* and *t*, what is the length of the shortest path between s and t?



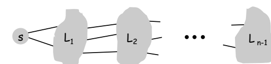
Applications

- Facebook
- Maze traversal
- Kevin Bacon number
- Fewest number of hops in a communication network

Breadth First Search

Intuition. Explore outward from *s* in all possible directions, adding nodes one "layer" at a time

Algorithm



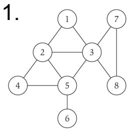
- L_0 = { s }
- L_1 = all neighbors of L_0
- L_2 = all nodes that do not belong to L_0 or L_1, and that have an edge to a node in L_1
- L_{i+1} = all nodes that do not belong to an earlier layer, and that have an edge to a node in L_i

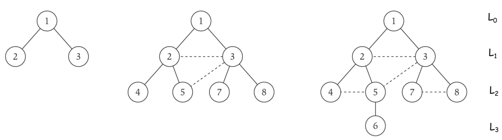
Theorem. For each *i*, L_i consists of all nodes at distance exactly *i* from *s*. There is a path from *s* to *t* iff *t* appears in some layer.

Breadth First Search

Property. Let T be a BFS tree of G = (V, E), and let (x, y) be an edge of G. Then the level of x and y differ by at most 1.

G:





(a) (b) (c)

Breadth First Search: Analysis

Theorem. The BFS implementation runs in O(m + n) time if graph is given by its adjacency representation

Pf.

- Easy to prove $O(n^2)$ running time:
  - at most n lists L[i]
  - each node occurs on at most one list; for loop runs ≤ n times
  - when we consider node u, there are ≤ n incident edges (u, v), and we spend O(1) processing each edge

Breadth First Search: Analysis

Theorem. The BFS implementation runs in O(m + n) time if graph is given by its adjacency representation

Pf.

- Actually runs in O(m + n) time:
  - when we consider node u, there are deg(u) incident edges (u, v)
  - total time processing edges is $\Sigma_{u \in V}$ deg(u) = 2m ▪

each edge (u, v) is counted exactly twice in sum: once in deg(u) and once in deg(v)