Objectives

- Data structure: Heaps
- Implementing a Priority Queue

Review

Summary of Running Times

Running Time	Example
O(log n)	
O(n)	
O(n log n)	
$O(n^2)$	
O(n!)	

Common runtimes: Chapter 2.4

Summary of Running Times

Running Time	Example
O(log n)	Generally dividing problem in half on each iteration
O(n)	Operate on each input value
O(n log n)	Divide and conquer
$O(n^2)$	Operate on each pair of inputs
O(n!)	Operate on each permutation of inputs

Moving from integers, lists, arrays

MORE COMPLEX DATA STRUCTURES

Improving Running Times

After overcoming higher-level obstacles, lower-level **implementation details** can **improve runtime**.

PRIORITY QUEUES

Priority Queues

- Elements have a ***priority*** or *key*
- Each time select an element from the priority queue, want the one with *highest* priority
- More formally…
  - Maintains a set of elements *S*
    - Each element $v \in S$ has a key(v) for its priority
      - Smaller keys represent higher priorities
  - Example methods:
    - Add, delete elements
    - Select element with smallest key

Key	2	4	5	6	9	20
Value	3542	5143	8712	1264	9123	5954

Priority
Process id

 Not implementation, just how to envision

Motivating Example: Scheduling Processes

Key	2	4	5	6	9	20
Value	3542	5143	8712	1264	9123	5954

Priority
Process id

- Each process has a priority or urgency
- Processes do not arrive in priority order
- **Goal**: run process with highest priority

Using a Priority Queue

- Given API:
  - Add an element with a given key (i.e., priority)
  - Delete an element with a given priority
  - Select element with smallest key/highest priority
  - Get the number of elements in PQ

How could we use a PQ to sort a list of numbers?

Priority Queues for Sorting

1. Add elements into PQ with the number's value as its priority
2. Then extract the smallest number *until* done
   - Come out in sorted order

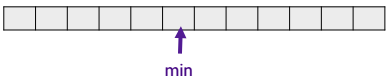
Sorting *n* numbers takes O(n logn) time

What is the goal running time for our PQ's operations? **O(logn)**

Already know our "loops" will be O(n)

Implementing a Priority Queue

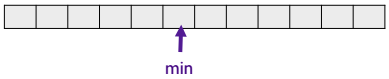
- Consider an *unordered* list, where there is a pointer to minimum



- How difficult (i.e., expensive) is
  - Adding new elements?
  - Extraction?

Implementing a Priority Queue

- Consider an *unordered* list, where there is a pointer to minimum

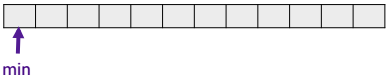


- How difficult (i.e., expensive) is
  - Adding new elements? *easy (O(1))*
  - Extraction? *difficult*
    - Need to find "new" minimum: O(n)

What is the running time for sorting using the PQ in this case? $O(n^2)$

Implementing a Priority Queue

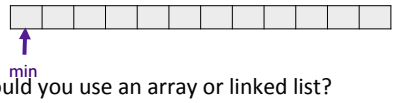
- Consider a *sorted* list where min is at the beginning



- Should you use an array or linked list?
- How difficult is
  - Adding new elements?
  - Extraction?

Implementing a Priority Queue

- Consider a sorted list where min is at the beginning



- Should you use an array or linked list?
- How difficult is
  - Adding new elements? *difficult (insertion)*
  - Extraction? *Easy*

What is the running time for sorting using the PQ in this case? $O(n^2)$

Comparing Data Structures

Operation	Unsorted List	Sorted List
Start(N)		
Insert(v)		
FindMin()		
Delete(i)		
ExtractMin()		

Comparing Data Structures

Operation	Unsorted List	Sorted List
Start(N)	O(1)	O(1)
Insert(v)	O(1)	O(n)
FindMin()	O(1)	O(1)
Delete(i)	O(n)	O(1)
ExtractMin()	O(n)	O(1)

Assuming deleting the first element. If deleting another element, O(i)

Reflection

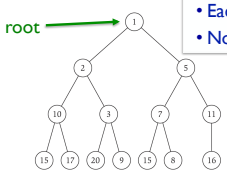
- All of "known" data structures has one operation that takes O(n) time
- Cannot implement PQs with "known" data structures arrays and lists to meet desired O(n log n) runtime

⇨ Motivates use of a new data structure (***heap***) to implement PQ

HEAPS

Heap Defined

- Combines benefits of sorted array and list
- Balanced binary tree

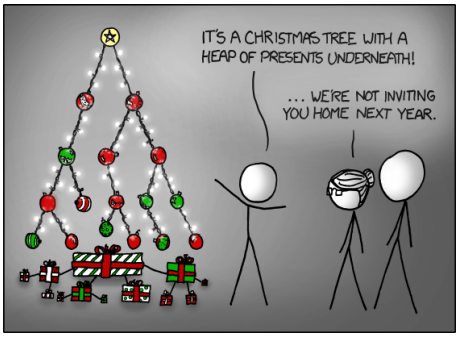


- Each node has *at most* 2 children
- Node value is its *key*

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

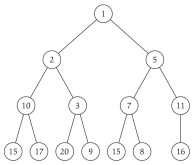
Note: **not** a binary search tree

Heaps



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*



Where does the index in the array start?
If know child's position, what is the position of parent?

Implementing a Heap: Operations

- Finding the minimal element?

Implementing a Heap: Operations

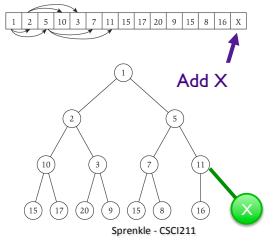
- Finding the minimal element
  - First element
  - O(1)

Implementing a Heap: Operations

- Adding an element?
  - Assume heap has less than N elements

Implementing a Heap: Operations

- Adding an element?
  - Could add element to last position
    - What are possible scenarios?



Assignments

- Journals: Chapter 2.3, 2.4 for Monday

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 `Heapify-up` procedure to fix our heap

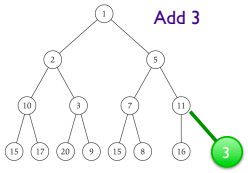
Heapify-Up

Heap — Position where node added

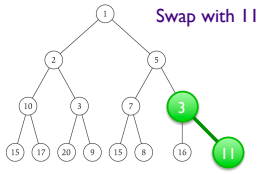
```
Heapify-up(H, i):
   if i > 1 then
       j=parent(i)=floor(i/2)
       if key[H[i]] < key[H[j]] then
          swap array entries H[i] and H[j]
          Heapify-up(H, j)
```

- Why does this algorithm work?
- What is the intuition?

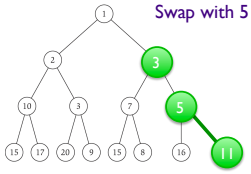
Practice: Heapify-Up



Practice: Heapify-Up



Practice: Heapify-Up



Heapify-Up

- Claim. Assuming array H is almost a heap with key of H[i] too small, Heapify-Up fixes the heap property in O(log i) time
  - Can insert a new element in a heap of *n* elements in O(log n) time

Heapify-Up

- Claim. Assuming array H is almost a heap with key of H[i] too small, Heapify-Up fixes the heap property in O(log i) time
  - Can insert a new element in a heap of *n* elements in O(log n) time
- Proof. By induction
  - If i=1 …

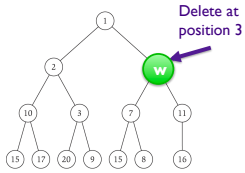
Heapify-Up

- Claim. Assuming array H is almost a heap with key of H[i] too small, Heapify-Up fixes the heap property in O(log i) time
  - Can insert a new element in a heap of *n* elements in O(log n) time
- Proof. By induction
  - If i=1, is already a heap → O(1)
  - If i>1, …

Heapify-Up

- Claim. Assuming array H is almost a heap with key of H[i] too small, Heapify-Up fixes the heap property in O(log i) time
  - Can insert a new element in a heap of *n* elements in O(log n) time
- Proof. By induction
  - If i=1, is already a heap → O(1)
  - If i>1,
    - Swaps are O(1)
    - Swaps continue up to root (max) → log i

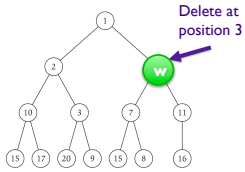
Deleting an Element



Deleting an Element

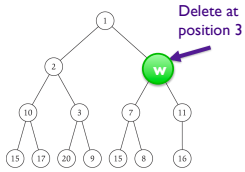
- Delete at position *i*
- Removing an element:
  - Messes up heap order
  - Leaves a "hole" in the heap
- Not as straightforward as `Heapify-Up`
- Algorithm
  1. Fill in element where hole was
     - Patch hole: move n^{th} element into i^{th} spot
  2. Adjust heap to be in order
     - At position *i* because moved n^{th} item up to *i*

Deleting an Element



- What are the possibilities when we move n^{th} element (*w*) into spot where element was removed?

Deleting an Element

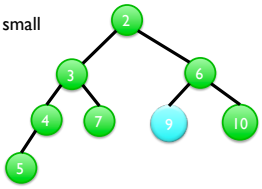


Example of OK:
11 deleted, replaced by 16

- Two "bad" possibilities: element *w* is
  - Too small: violation is between it and parent → `Heapify-Up`
  - Too big: with one or both children → `Heapify-Down` (example: w becomes 12)

Deleting an Element

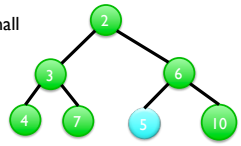
Example where new key is too small



- Delete 9
- Replace with 5

Deleting an Element

Example where new key is too small



- Delete 9
- Replace with 5
- But 5 < 6, so need to `Heapify-Up`

Heapify-Down

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

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

Why can we stop?

To Do

- Read/summarize for journal
- Problem Set 2 due next Friday
- Class is at 9:50 a.m.

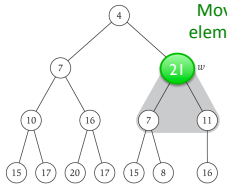
Heapify-Down

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

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

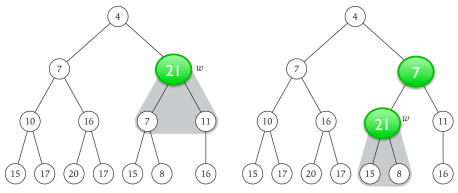
i is a leaf – nowhere to go

Practice: Heapify-Down

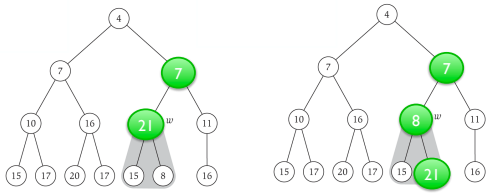


Moved 21 to where element was removed

Practice: Heapify-Down



Practice: Heapify-Down



Runtime of Heapify-Down?

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

   if key[H[j]] < key[H[i]] then
      swap array entries H[i] and H[j] O(1)
      Heapify-down(H, j)
```

Num 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(v)	Inserts item v into heap	
FindMin()	Identifies minimum element in heap but does not remove it	
Delete(i)	Deletes element in heap at position i	
ExtractMin()	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(v)	Inserts item v into heap	O(log n)
FindMin()	Identifies minimum element in heap but does not remove it	O(1)
Delete(i)	Deletes element in heap at position i	O(log n)
ExtractMin()	Identifies and deletes an element with minimum key from heap	O(log n)

Putting It All Together…

1. Add elements into PQ with the number's value as its priority
2. Then extract the smallest number until done
   - Come out in sorted order

What is the running time of sorting numbers using a PQ implemented with a Heap?

O(n log n)

Comparing Data Structures

Operation	Heap	Unsorted List	Sorted List
Start(N)			
Insert(v)			
FindMin()			
Delete(i)			
ExtractMin()			

Comparing Data Structures

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

Comparing Data Structures

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

Additional Heap Operations

- Access elements in PQ by "name"

Key	2	4	5	6	9	20
Value	3542	5143	8712	1264	9123	5954

Priority
Process id

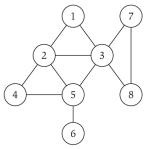
  - Maintain additional array `Position` that stores current position of each element in heap

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

GRAPHS

Undirected Graphs G = (V, E)

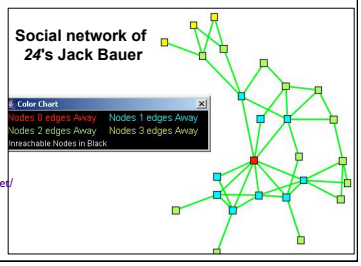
- V = nodes (vertices)
- E = edges between pairs of nodes
- Captures pairwise relationship between objects
- Graph size parameters: n = |V|, m = |E|



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

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

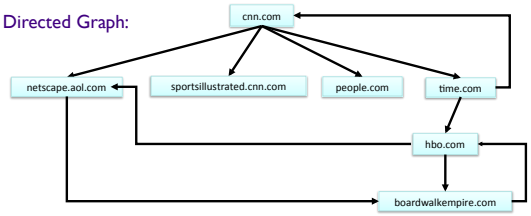
Facebook: Visualizing Friends

http://www.facebook.com/notes/facebook-engineering/visualizing-friendships/469716398919

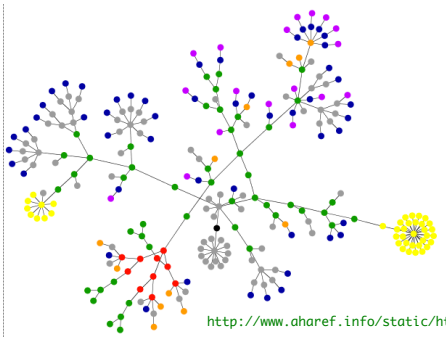
World Wide Web

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

Directed Graph:



Graph of Web Page www.wlu.edu

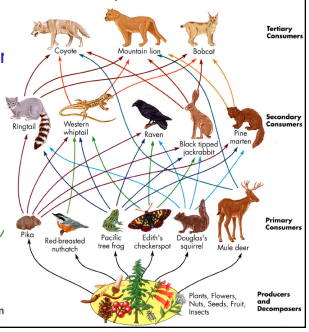


http://www.aharef.info/static/htmlgraph

Ecological Food Web

- Food web graph
  - Node = species
  - Edge = from prey to pr

Directed Graph:



Reference:
https://www.msu.edu/course/isb/202/ebertmay/images/foodweb.jpg

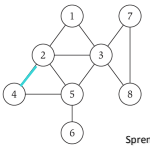
Rock Paper Scissors Lizard Spock

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

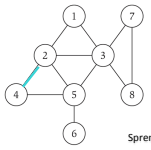
- n×n matrix with A_{uv} = 1 if (u, v) is an edge
  - Two representations of each edge (symmetric matrix)
  - Space?
  - Checking if (u, v) is an edge?
  - Identifying all edges?



	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×n matrix with A_{uv} = 1 if (u, v) is an edge
  - Two representations of each edge (symmetric matrix)
  - Space: $\Theta(n^2)$
  - Checking if (u, v) is an edge: $\Theta(1)$ time
  - Identifying all edges: $\Theta(n^2)$ time

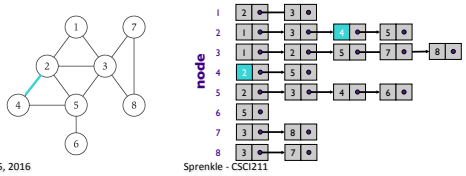


	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	1	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

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?

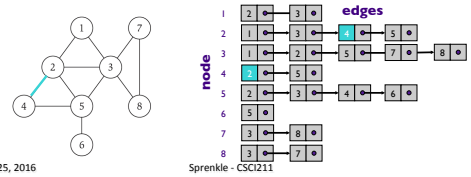
What are the extremes?



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

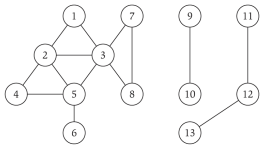


Assignments

- Journals: Finish Chapter 2 for Tuesday
  - Chapter 3 started today but we'll leave it for next week
- Problem Set 2 due Friday

Paths and Connectivity

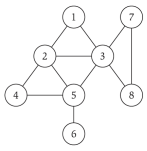
- Def. A ***path*** in an undirected graph G = (V, E) is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k$
  - Each consecutive pair v_i, v_{i+1} is joined by an edge in E
- Def. A path is ***simple*** if all nodes are *distinct*
- Def. An undirected graph is ***connected*** if ∀ pair of nodes u and v, there is a path between u and v



- *Short path*
- *Distance*

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

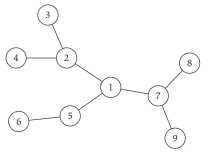


cycle C = 1-2-4-5-3-1

TREES

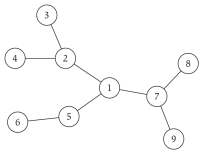
Trees

- 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



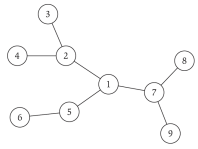
Trees

- 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

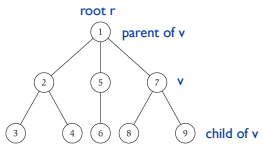


Rooted Trees

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



a tree

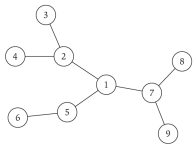


the same tree, rooted at 1

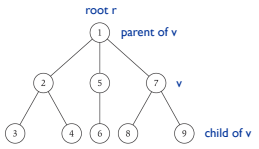
Why *n-1* edges?

Rooted Trees

- Why *n-1* edges?
  - Each node except for root has an edge to its parent



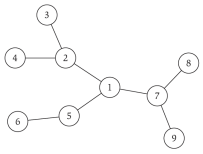
a tree



the same tree, rooted at 1

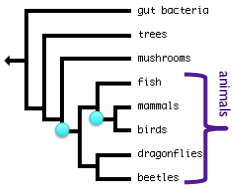
Trees

- 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



Phylogeny Trees

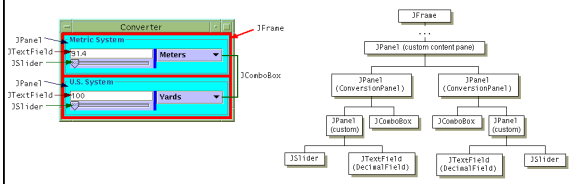
- Describe evolutionary history of species
  - mammals and birds share a common ancestor that they do not share with other species
  - all animals are descended from an ancestor not shared with mushrooms, trees, and bacteria



Tiffani Williams, Texas A&M Computational Biology

GUI Containment Hierarchy

- Describe organization of GUI widgets



Assignments

- Journals: Finish Chapter 2 for Tuesday
- Problem Set 2 due Friday