Objectives

- Finish survey of common running times
- More on Data structures

- Checking in on journal
- Problem Set
  - Solved exercises in text book

Continuing from Friday, Monday

A SURVEY OF COMMON RUNNING TIMES

Polynomial Time: $O(n^k)$ Time

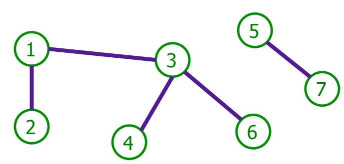
- To get all pairs, the algorithm is $O(n^2)$
- To get all triplets, the algorithm is $O(n^3)$

What is an example of an $O(n^k)$ algorithm?

All subsets of size *k*

Polynomial Time: $O(n^k)$ Time

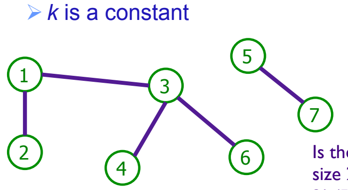
- Independent set of size *k*. Given a graph, are there *k* nodes such that no two are joined by an edge?
  - *k* is a constant



Is there an independent set of size 2? 3? 4? 5?

Polynomial Time: $O(n^k)$ Time

- Independent set of size *k*. Given a graph, are there *k* nodes such that no two are joined by an edge?
  - *k* is a constant



Is there an independent set of size 2? Yes (2-3; 1-5; 6-7; …)
3? (5-6-7; 2-3-5; …)
4? (2-4-5-7; 1-4-6-7; …)
But not 5

Polynomial Time: $O(n^k)$ Time

- Independent set of size *k*. Given a graph, are there *k* nodes such that no two are joined by an edge?
  - *k* is a constant

```
foreach subset S of k nodes
   check whether S in an independent set
   if (S is an independent set)
      report S is an independent set
```

- $O(n^k)$ solution
  1. Enumerate all subsets of k nodes
     $$\begin{bmatrix} n \\ k \end{bmatrix} = \frac{n!}{k!\,(n-k)!} = \frac{n\,(n-1)\,(n-2)\ldots(n-k+1)}{k\,(k-1)\,(k-2)\ldots(2)\,(1)} \le \frac{n^k}{k!}$$
  2. Check whether S is an independent set = $O(k^2)$.

$\boxed{O(k^2\, n^k / k!) = O(n^k)}$

poly-time for k=17
but not practical

Exponential Time

- Independent set. Given a graph, what is the *maximum size* of an independent set?
- $O(n^2 2^n)$ solution. Enumerate all subsets

```
S* = φ
foreach subset S of nodes
   check whether S in an independent set
   if (S is largest independent set seen so far)
      S* = S
```


O(log n) Time

- ***Sublinear*** time
- Know any algorithms that take O(log n) time?

O(log n) Time

- Example: Binary search
- Often requires some pre-processing or data structure that allows cheaper "querying" than *n* time

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

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
  - API
    - 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

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

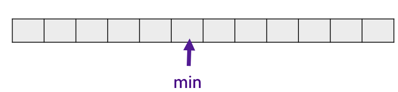
Sorting *n* numbers takes O(n logn) time

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

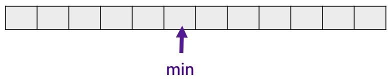
- Consider an *unordered* list, where there is a pointer to minimum



- 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



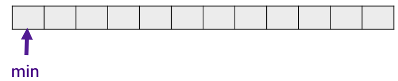
- How difficult (i.e., expensive) is
  - Adding new elements? *easy (O(1))*
  - Extraction? *difficult*
    - Need to find "new" minimum: O(n)

What is the running time for sorting using the PQ in this case?

$O(n^2)$

Implementing a Priority Queue

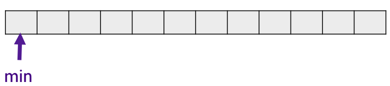
- Consider a *sorted* list where min is at the beginning



- 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?

$O(n^2)$

Reflection

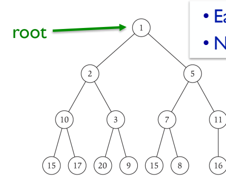
- All of "known" data structures has one operation that takes O(n) time
- Cannot implement PQs with "known" data structures arrays and lists to meet desired O(n log n) runtime

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

HEAPS

Heap Defined

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

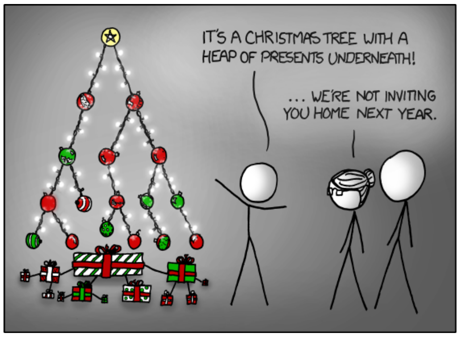


- Each node has *at most* 2 children
- Node value is its *key*

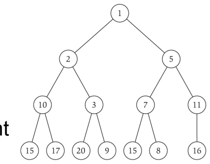
Heap order: each node's key is at least as large as its parent's

Note: **not** a binary search tree

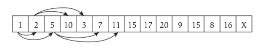
Heaps



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*



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

Implementing a Heap: Operations

- Finding the minimal element?

Implementing a Heap: Operations

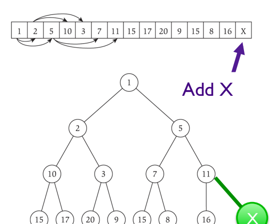
- Finding the minimal element
  - First element
  - O(1)

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?



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

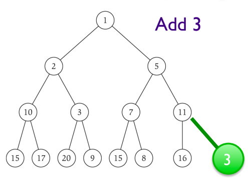
Heap   Position where node added

```
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?

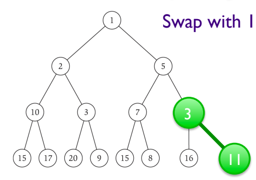
Practice: Heapify-Up

Add 3



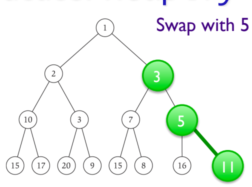
Practice: Heapify-Up

Swap with 11



Practice: Heapify-Up

Swap with 5



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 …

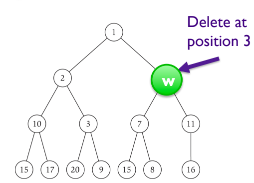
Heapify-Up

- 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

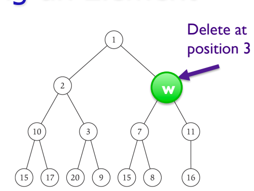
Deleting an Element



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?
  - Give an example for each possibility
  - Consider other deletion spots, # elements in heap

Assignment

- Problem Set Due Friday
- Finish reading, summarizing Chapter 2