Objectives

- Finish survey of common running times
- Data structures

Book Notes in Sakai Wiki

- Notes for wiki syntax are in sidebar
- New page per chapter
  - Could go by section
- Include a page on the Preface too (up to Overview)
- **What to Write in Your Notes**
  - Brief summary of what the chapter/section covers (~1 paragraph of about 5 sentences/section; feel free to write more if that will help you)
  - Include motivations for the given problem, as appropriate
  - Questions you have about motivation/solution/proofs/analysis
  - Discuss anything that makes more sense after reading it again, after it was presented in class (or vice versa)
  - Anything that you want to remember, anything that will help you

Cubic Time: $O(n^3)$

- Enumerate all triples of elements
- Set disjointness. Given *n* sets $S_1, \ldots, S_n$ each of which is a subset of 1, 2, …, *n*, is there some pair of these which are disjoint?
- $O(n^3)$ solution. For each pair of sets, determine if they are disjoint

```
foreach set Si
   foreach other set Sj
      foreach element p of Si
         determine whether p also belongs to Sj

      if (no element of Si belongs to Sj)
         report that Si and Sj are disjoint
```


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

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

- If the algorithm to find all pairs is $O(n^2)$, what is an example of an $O(n^k)$ algorithm?

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

- If the algorithm to find all pairs is $O(n^2)$, 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

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
   if (S is an independent set)
      report S is an independent set
```

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

$O(k^2 n^k / k!) = O(n^k)$

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

DATA STRUCTURES

Stable Matching Implementation

- What do we need to represent?
- How should we represent them?

Stable Matching Implementation

- What do we need to represent? How should we represent them?

Data	How represented
Preference lists	Array of arrays
Unmatched men	List
Who men proposed to	Integer
Engagements	Array

- What's the difference between an array and a list?

Arrays

- *Fixed* number of elements
- What is the runtime of
  - Determining the value of the i^{th} item in the array?
  - Determining if a value *e* is in the array?
  - Determining if a value *e* is in the array if the array is sorted?

Array Operations' Running Times

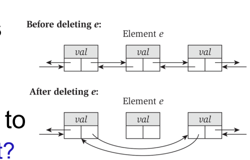
Operation	Running Time
Value of i^{th} item	O(1) → direct access
If *e* is in the array	O(n) → look through all the elements
If *e* is in the array if sorted	O(log n) → binary search

Limitation of arrays?

Fixed size, so can't add/delete elements

Lists

- Dynamic set of elements
  - Linked list
  - Doubly linked list
- What is the running time to
  - Add an element to the list?
  - Delete an element from the list?
  - Find an element *e* in the list?
  - Find the i^{th} element in the list?



List Operations' Running Time

Operation	Running Time
Add element	O(1)
Delete element	O(1)
Find element	O(n)
Find i^{th} element	O(i)

Disadvantage of list instead of array?

Finding i^{th} element is slower

Converting between Lists and Arrays (and Vice Versa)

- What is the running time of converting a list to an array?
- An array to a list?

O(n)

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
  - Supported operations
    - Add, delete elements
    - Select element with smallest key

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

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

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

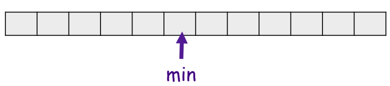
Sorting *n* numbers takes at least O(n logn) time

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

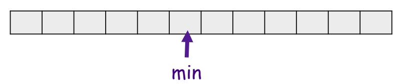
- Consider an unordered list, where there is a pointer to minimum



- 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

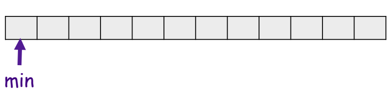


- How difficult (i.e., expensive) is
  - Adding new elements? *easy*
  - Extraction? *difficult*
    - Need to find "new" minimum: O(n)

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

Implementing a Priority Queue

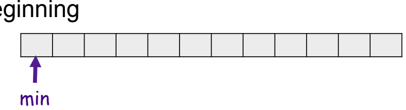
- Consider a sorted list where min is at the beginning



- 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? *more difficult (insertion)*
  - Extraction? *Easy*

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

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 runtime: O(n log n)

→Motivates use of ***heap*** to implement PQ

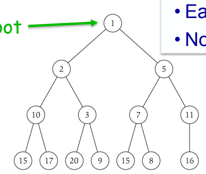
Goal: show results in O(n log n) time

HEAPS

Heap Defined

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

root

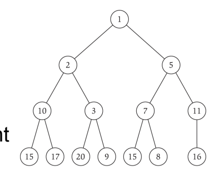


- Each node has *at most* 2 children
- Node value is its *key*

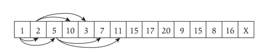
Heap order: each node's key is at least as large as its parent's

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

- Finding the minimal element?

Implementing a Heap: Operations

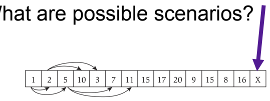
- Finding the minimal element
  - First element
  - O(1)

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

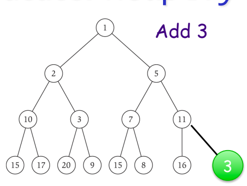
- Adding an element?
  - Could add element to last position
    - What are possible scenarios?
      - Heap is no longer balanced
      - Something that is almost a heap but a little off
      - Need Heapify-up procedure to fix our heap

Heapify-Up

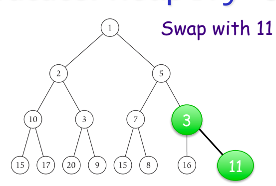
Heap Position where node added

```
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)
```

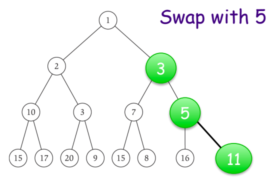

Practice: Heapify-Up



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
  - If i>1, …