Objectives

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

- The value of the i^{th} item in the list?
- Determining if a value *e* is in the list?
- Determining if a value *e* is in the list if the list is sorted?

Arrays

What is the runtime of

- The value of the i^{th} item in the array?
  - O(1) → direct access
- Determining if a value *e* is in the array?
  - O(n) → look through all the elements
- Determining if a value *e* is in the list if the array is sorted?
  - O(log n) → binary search

Arrays

What is the runtime of

- Adding an element to the array?
- Deleting an element from the array?

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?

Lists

What is the running time to

- Add an element to the list?
  - O(1)
- Delete an element from the list?
  - O(1)
- Find an element *e* in the list?
  - O(n)
- Find the i^{th} element in the list?
  - O(i)

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?

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 ***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 ∈ S has a value key(v) for its priority
    - Smaller keys represent higher priorities
- Supported operations
  - Add, delete elements
  - Selection of element with smallest key

Motivating Example: Scheduling Processes

Each process has priority or urgency

Processes do not arrive in priority order

Goal: run process with highest priority

Priority Queues for Sorting

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

Priority Queues for Sorting

Add elements into PQ with the number's value as its priority

Then extract the smallest number until done

- Come out in sorted order

➢ Any sequence of PQ operations that results in sorting *n* numbers must take at least O(n logn) time

➢Goal running time for our operations?

Priority Queues for Sorting

Add elements into PQ with the number's value as its priority

Then extract the smallest number until done

- Come out in sorted order

➢ Any sequence of PQ operations that results in sorting *n* numbers must take at least O(n logn) time

➢Goal running time for our operations? **O(log n)**

Implementing a Priority Queue

List?

- Keep elements in an unordered list
- Pointer to minimum
- How difficult is
  - Adding new elements
  - Extraction

Implementing a Priority Queue

List?

- Keep elements in an unordered list
- Pointer to minimum
- How difficult is
  - Adding new elements: easy
  - Extraction: difficult
    - Need to find "new" minimum

Implementing a Priority Queue

Sorted List?

- Min is at the beginning
- Array or Linked list?
- How difficult is
  - Adding new elements
  - Extraction

Implementing a Priority Queue

Sorted List?

- Min is at the beginning
- How difficult is
  - Adding new elements: more difficult (insertion)
  - Extraction: easy

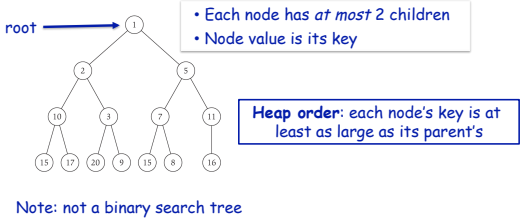
Summary

All of "known" data structures has one operation that takes O(n) time

Data Structure: Heap

Combines benefits of sorted array and list

Balanced binary tree

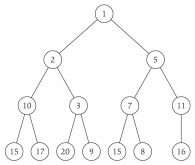


Note: not a binary search tree

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 a `Heapify-up` procedure to fix our heap

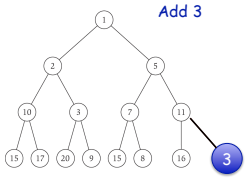
Heapify-Up

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)
```

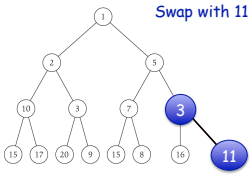

Practice: Heapify-Up

Add 3



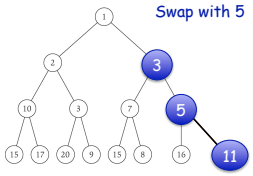
Practice: Heapify-Up

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
- 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

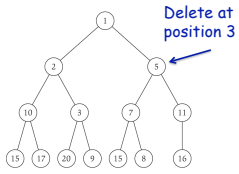
- Can insert a new element in a heap of *n* elements in O(log n) time

Proof. By induction

- If i=1, is already a heap
- If i>1,
  - Swaps are O(1)
  - Swaps continue up to root (max)

Deleting an Element

Delete at position *i*



Deleting an Element

Delete at position *i*

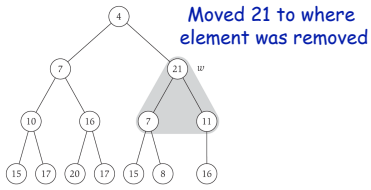
Not only removes an element

- 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



Two possibilities: element *w* is

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

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)
```