Course Notes for

CS 1501
Algorithm Implementation

By
John C. Ramirez
Department of Computer Science
University of Pittsburgh

- These notes are intended for use by students in CS1501 at the University of Pittsburgh and no one else
- These notes are provided free of charge and may not be sold in any shape or form
- These notes are NOT a substitute for material covered during course lectures. If you miss a lecture, you should definitely obtain both these notes and notes written by a student who attended the lecture.
- Material from these notes is obtained from various sources, including, but not limited to, the following:
  - Algorithms in C++ by Robert Sedgewick
  - Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne
  - Introduction to Algorithms, by Cormen, Leiserson and Rivest
  - Various Java and C++ textbooks
  - Various online resources (see notes for specifics)

Heap Implementation of PQ

- We need a PQ in Lazy and Eager Prim and in Kruskal
- Let's look a bit closer at PQs
  - We need 3 primary operations
    - Insert an object into the PQ
    - Find the object with best priority
      - Often called FindMin or FindMax
    - Remove the object with best priority
      - Often called DeleteMin or DeleteMax
  - How to implement?
    - 2 obvious implementations:
      - Unsorted Array
      - Sorted Array

Unsorted Array PQ:

- Insert: ***Add new item at end of array: Theta(1) run-time***
- FindMin: ***Search array for smallest: Theta(N) run-time***
- DeleteMin: ***Search array for smallest and delete it: Theta(N) run-time***

- Since we generally remove everything we insert into a PQ, let's calculate total work for N Inserts + N DeleteMins
- Assuming efficient array resizing, N Inserts will take Theta(N) time total
- Think about 1^{st} DeleteMin in PQ with N items → N-1 comparisons
  - Now 2^{nd} DeleteMin in PQ with N-1 items → N-2 comparisons
  - Total will be (N-1) + (N-2) + … + 1 = (N-1)(N)/2 → Theta (N^2)
- Total is thus Theta(N) + Theta(N^2) = **Theta(N^2)**

Sorted Array PQ:

- Insert: ***Add new item in reverse sorted order: Theta(N) run-time***
- FindMin: ***Smallest item at end of array: Theta(1) run-time***
- DeleteMin: ***Delete item from end of array: Theta(1) run-time***

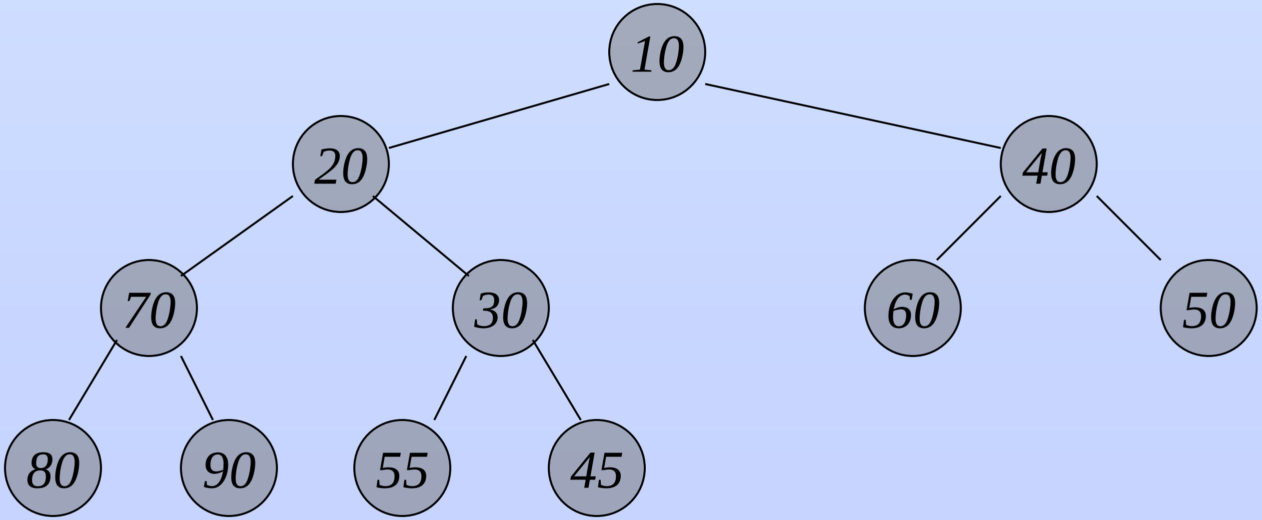
- Using analysis similar to that in previous slide, for a N Inserts and N DeleteMin, total time is **Theta(N^2)**
  - In this case the "work" is done during Insert rather than DeleteMin

- How can we improve our overall run-time?
  - We could use a BST
    - This would give ave Theta(lgN) for each operation
      - Could be linear with out of balance tree
    - However, it is MORE than we need
      - It maintains a complete ordering of the data, but we only need a PARTIAL ORDERING of the data
  - Instead we will use a HEAP
    - Complete binary tree such that for each node T in the tree

      T.Val has a higher priority than T.lchild.val

      T.Val has a higher priority than T.rchild.val

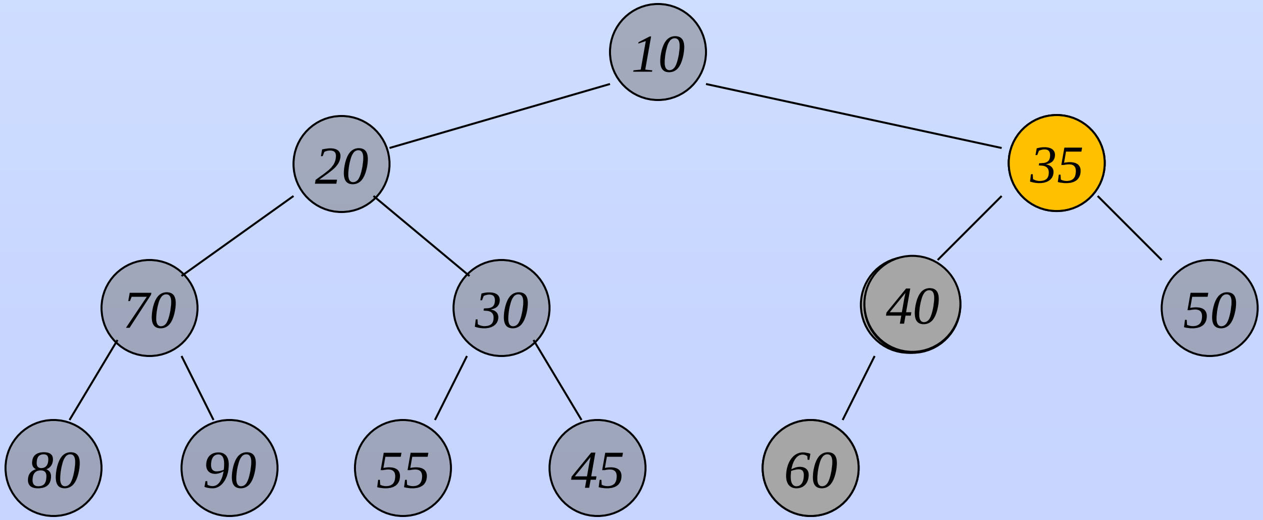
- The tree below is a Min Heap
  - Note that each root of a subtree has a priority that is higher than either of its children nodes
    - Note: In this case higher priority == smaller value
  - However, sibling order is arbitrary
    - Note 20 < 40 (left < right)
    - Note 70 > 30 (left > right)



- Note that we don't care how T.lchild.val relates to T.rchild.val
  - But BST does care, which is why it gives a complete ordering
    - We will see soon why we will ultimately prefer a heap to a BST for a PQ
- Ok, how do we do our operations:
  - FindMin is easy – ROOT of tree
  - Insert and DeleteMin are not as trivial
  - For both we are altering the tree, so we must ensure that
    - It remains a complete binary tree
    - The HEAP PROPERTY is reestablished

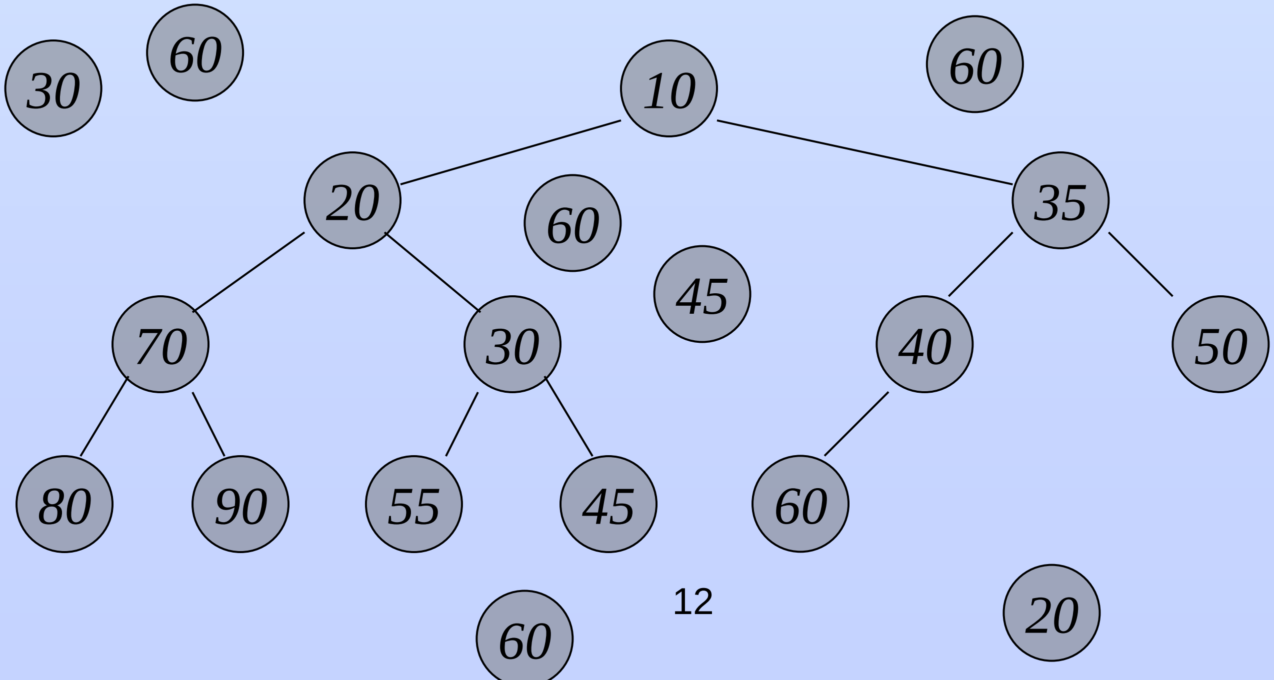
- Idea of Insert:
  - Add new node at next available leaf
    - Where is this?
      - First open leaf in last unfilled level of the tree
    - This will keep the tree complete but it may no longer be a valid heap
  - Push the node "up" the tree until it reaches its appropriate spot
    - If value has a higher priority than its parent swap them and move up a level
    - Repeat until value is correctly placed
    - Author calls this swim

- Trace insert(35)
  - Add value in new leaf node
  - upHeap:
    - Swap with parent (60)
    - Swap with parent (40)
    - > parent so stop



- Idea of DeleteMin:
  - We need to remove the root **value**
    - Since this is the mininum value
  - We need to remove the last leaf **node**
    - To keep our tree complete
  - Instead of deleting root node, we overwrite its value with that of the last leaf
    - Then we delete the last leaf node
  - But now root value may not be the min
  - Push the node "down" the tree until it reaches its appropriate spot
    - Author calls this sink

- Trace DeleteMin
  - Copy last leaf value (60) into root and delete leaf
  - Find min of (60, 20, 35) and swap into root
  - Find min of (60, 70, 30) and swap into subtree root
  - Find min of (60, 55, 45) and swap into subtree root
  - Heap has been restored



- Run-time?
  - Complete Binary Tree is always balanced
    - Thus it height is ≈ lgN
  - upheap or downheap at most traverse height of the tree
    - Think about how these both work – see trace
  - Thus Insert and DeleteMin are always Theta(lgN) worst case
- For N Inserts + N DeleteMins total = Theta(NlgN)
  - Recall that our simple sorted and unsorted arrays each required Theta(N^2) for these operations
  - This is a definite improvement

- How to Implement a Heap?
  - We could use a linked binary tree, similar to that used for BST
    - Will work, but we have overhead associated with dynamic memory allocation and access
      - To go up and down we need child and parent references
    - How do we keep track of the last leaf?
  - But note that we are maintaining a complete binary tree for our heap
  - It turns out that we can easily represent a complete binary tree using an array

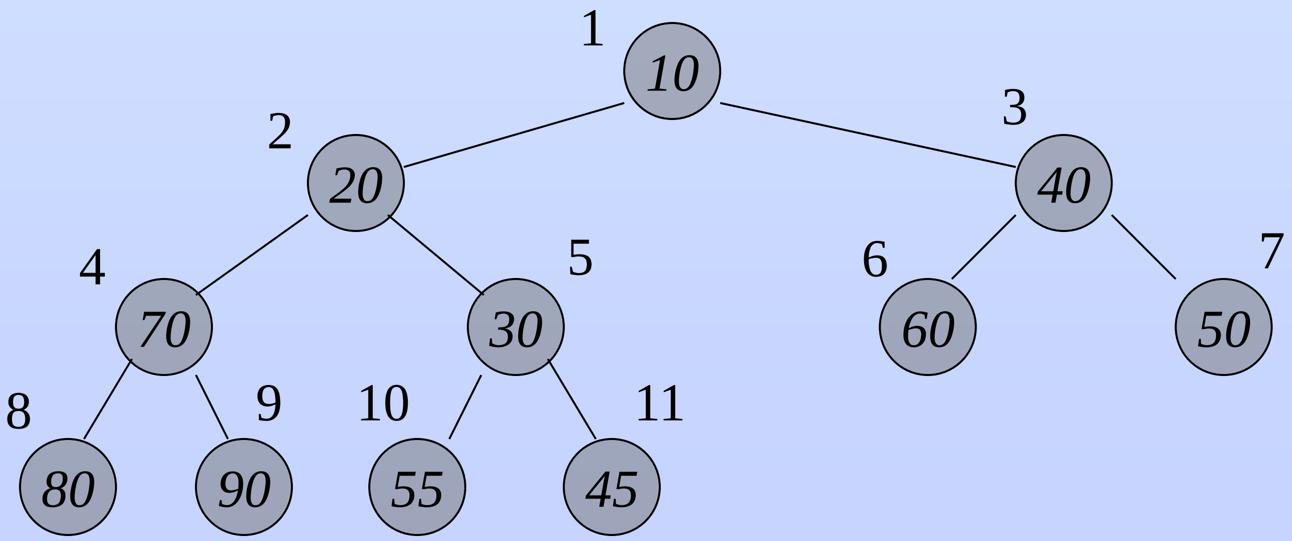
- Idea:
  - Number nodes row-wise starting at 1
  - Use these numbers as index values in the array
  - Now, for node at index i

    Parent(i) = i/2
    LChild(i) = 2i
    RChild(i) = 2i+1

- Now we have the benefit of a logical tree structure with the speed of a physical array implementation
- See next slide and MinPQ.java

- Adding / removing leaf is now simple
- Accessing child / parent nodes is also easy due to indexing of array

1	2	3	4	5	6	7	8	9	10	11	12
10	20	40	70	30	60	50	80	90	55	45	



- Recall the PQ operations:
    - Insert
    - FindMin
    - DeleteMin
- Recall operations needed for Eager Prim:
    - Insert
    - DeleteMin
    - Change
        - Change the priority of the vertex within the PQ when a better edge is found for the vertex
        - How to do this?
        - Do you see why it is a problem?

PQ Needed for Eager Prim

- In order to allow change() in our PQ, we must be able to do a general Find() of the data
  - PQ is only partially ordered, and further it is ordered on priority VALUES, not vertex ids
  - Finding an arbitrary id will take Theta(N), ruining any benefit the heap provides
- Luckily, we can do it better with a little thought
  - We can think of each entry in 2 parts:
    - A vertex id (int)
    - A priority value

PQ Needed for Eager Prim

- To change, we need to locate the id, update the priority, then reestablish the Heap property
- We can do this using a reverse mapping array
  - Keep an array indexed on the vertex ids which, for vertex v gives the location of v in the PQ
  - When we move v in the PQ we change the value stored in our array
  - This way we can always "find" a vertex in Theta(1) time
  - Now, total time for update is simply time to reestablish heap priority using swim or sink
    - > This is just height of the tree or lg(v)
- See next slide and IndexMinPQ.java

Indexable PQ

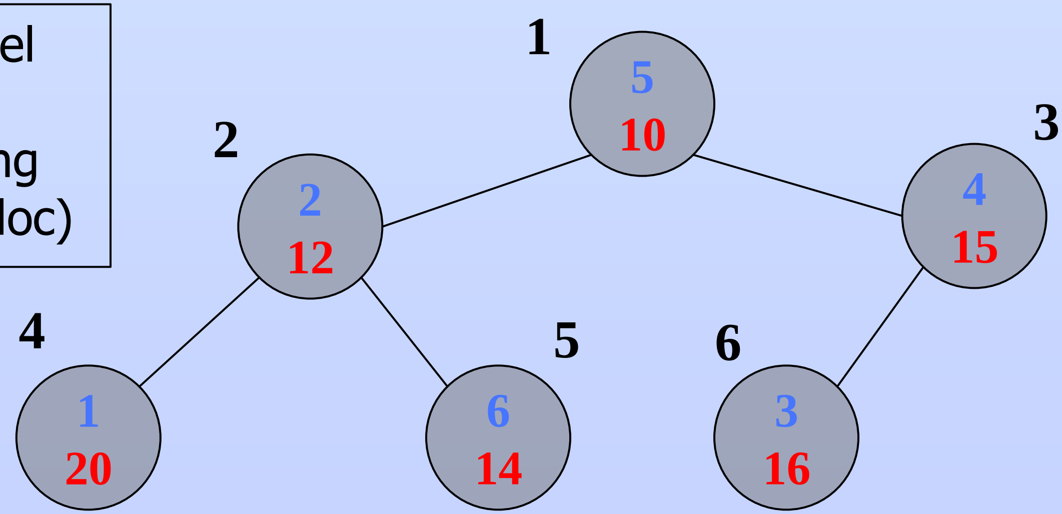
pq	1	2	3	4	5	6	7
	5	2	4	1	6	3	

keys	1	2	3	4	5	6	7
	10	12	15	20	14	16	

qp	1	2	3	4	5	6	7
	4	2	6	3	1	5	

- pq and keys are parallel arrays
- qp is a reverse mapping array (vertex id to pq loc)

blue → vertex id
red → current best edge for vertex
black → pq index



- Let's just quickly look at a snippet of the code

```
public void change(int k, Key key)
{
    if (!contains(k)) throw
        new RuntimeException("item is not in pq");
    keys[k] = key;
    swim(qp[k]);
    sink(qp[k]);
 }
```

- k is the vertex id that we want to change
- Note that it is calling both swim() and sink() – change could increase or decrease key value
- Note that the swim() and sink() calls are to qp(k)
  - This will access the pq location of vertex k

- We will consider the single-source shortest path problem:
  - Given a graph, G and a starting vertex, v, find the shortest path from v to each of the other vertices in G
  - Since direction is implied here, we will use a directed weighted graph
    - Idea is analogous to our undirected weighted graph
      - DirectedEdge class to represent an edge
      - EdgeWeightedDigraph class to represent the graph
    - See DirectedEdge.java
    - See EdgeWeightedDigraph.java

Directed Graph

```
public class DirectedEdge {
    private final int v;
    private final int w;
    private final double weight;

 public int from() {
        return v;
    }

 public int to() {
        return w;
    }
```

- Compare to the either() and other() methods of the Edge class

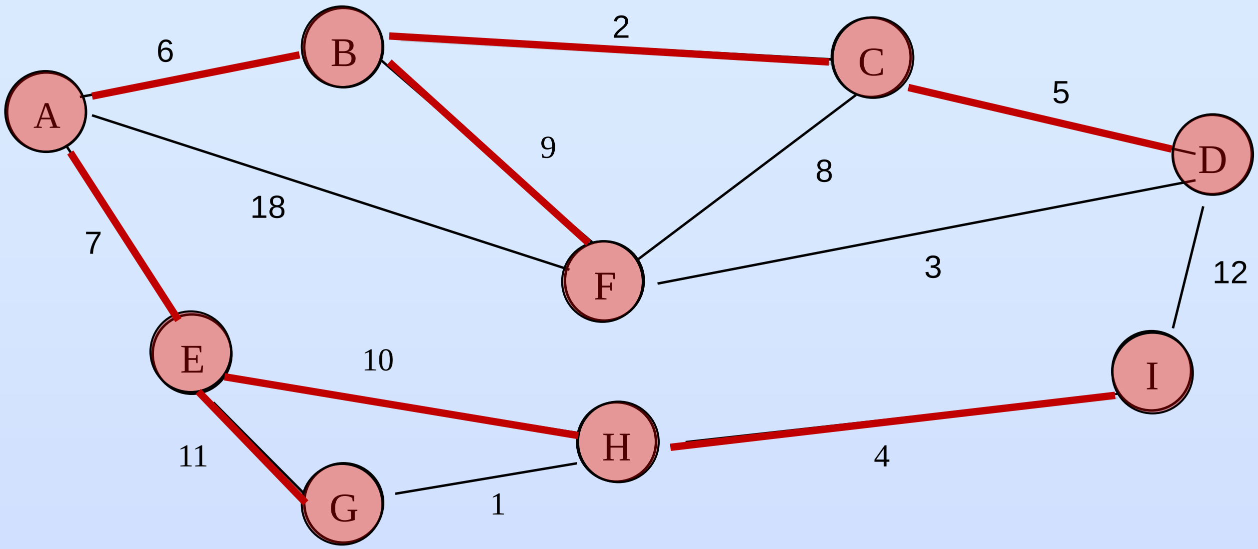
- Dijkstra's Algorithm
  - Very similar in idea to Eager Prim algorithm
  - We will build a shortest path (SP) tree vertex by vertex from a source vertex
    - Use a PQ to store candidate vertices
    - Key difference between Dijkstra and Eager Prim is the priority used to choose a vertex to add to the tree
      - With Eager Prim the priority was the **edge weight** that would connect the vertex to the tree
      - With Dijkstra it is the overall **path length** from the source to that vertex

- Other than that the code for Eager Prim and Djikstra is virtually identical
  - Even though Eager Prim uses an undirected graph and Djikstra uses a directed graph, we could easily adapt Djikstra to an undirected graph
    - For example, for given edge (A,B) we could always add both (A,B) and (B,A) to make it an “undirected” directed graph
- Thus, the run-times are also identical:
  - **Theta(ElgV)**

- See DijkstraSP.java
  - Compare to PrimMST.java
- Also see trace in next slide

Djikstra's Algorithm

Original graph showing edge weights



Tree shown in RED

Indexable Priority Queue

B	E	F	C	F	G	H	D	F	F	I	G	I
AB	AE	AF	BC	BF	EG	EH	CD	CF	DF	DI	HG	HI
6	7	18	8	15	18	17	13	16	16	25	18	21

- Now priorities are path lengths rather than edge weights
- Other than that the algorithm is the same as Eager Prim
- But the resultant trees are different