Lec03-data-structures

Thursday, August 31, 2023 5:34 PM

We just finished up with proving correctness of MST algorithms, let's circle back around to implementation.

Data structures are design specs

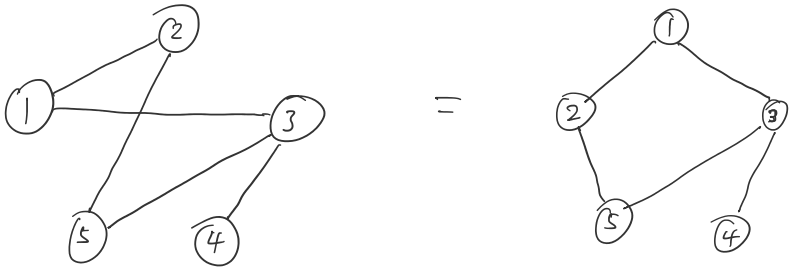
- how to store data in memory — storage
- what ops we can perform on data — ops
- the algs for those ops — algs
- time & space efficiency of algs — complexity

} organization of data can often lead to speed

abstract data type omits the implementation, but says what the black box should do

Ex. Graph ADT G

- G.add_vertex(u) – adds a vertex to G
- G.add_edge(u,v,d) – adds an edge $\{u,v\}$ with weight d.
- G.has_edge(u,v) – returns True iff $\{u,v\} \in E_G$.
- G.neighbors(u) – returns list of vertices adjacent to u in G
- G.weight(u,v)



Option 1: Adjacency matrix

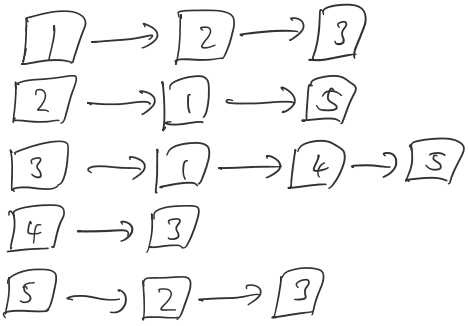
$$A = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

$A_{ij} = 1$ iff $\{i,j\} \in E_G$

or

$A_{ij} = w(\{i,j\})$

Option 2: Adj list



Option 3: Edge list

{1, 2}
{1, 3}
{2, 5}
{3, 4}

How to impl weights?
How long to find neighbors?
What about adding a new edge or node?

{3, 4}
{3, 5}

How do we check in Kruskal if adding an edge (u, v) would create a cycle?

- Would create cycle if u, v in same connected component.
- We start with a component for each node
- Components merge when we add an edge
- Need a way to check if u, v are in same component and to merge two components into one.

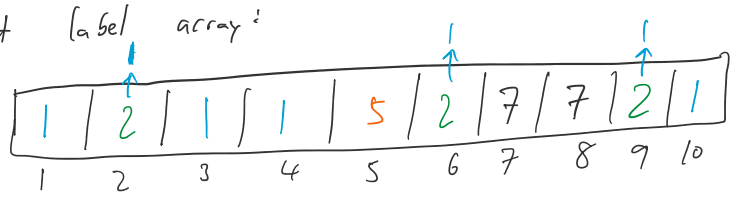
Union-Find Abstract Data Type (KT 4.6)

Supports following ops:

- Make Union Find (S) - create data structure containing |S| sets, each containing one item from S
- Find (i) - return label of set containing i.
- Union (a, b) - merge sets a and b into single set.

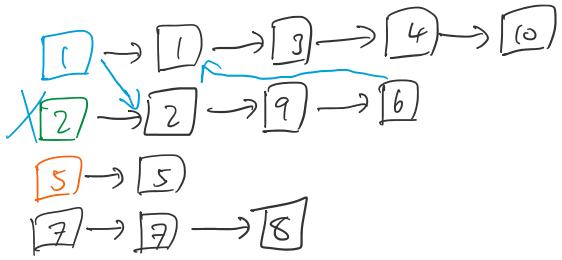
One way to build Union-Find: (array based)

Set label array:

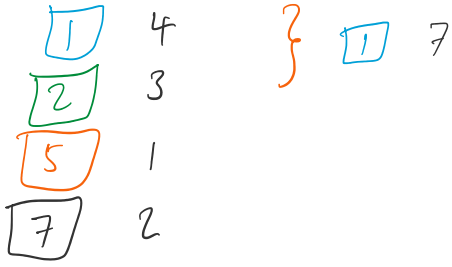


← says which set an item is in.

Items lists:



Sizes



MakeUnionFind(S): create data structures
$O(S)$ time (proportional to S time)

Find(i): Return UF.sets[i]
$O(1)$ time (constant time)

Union(x,y): Use size array to decide which set is smaller.
WLOG, assume $|x| \leq |y|$.
Walk down elements i in set x, setting sets[i] = y.
Set size[y] = size[y] + size[x]
(Make y point to start of x list and end of y list point to y
↳ prepend smaller list to larger list

Let's compute runtime of array-based union-find

Thm Any seq of k union operations on a collection of n items takes time proportional to $k \log k$

proof. After k unions, at most $2k$ items have been involved in a union
(each union can touch at most 2 new items;
if it was a union of 2 singleton items;
often only 1 or 0 new items touched.

For any item v, set[v] changes at most $\log_2(2k)$ times ← total number of items possible in largest set
because each time set[v] changes |set[v]| at least doubles.
(since we update labels of smaller set)

At most $2k$ items updated at most $\log_2(2k)$ times each
$\Rightarrow 2k \log_2(2k)$ work.

Running time of Kruskal using arrays

Sorting edges $\approx m \log m$ for m edges
$m \leq n^2$, so $\log m < \log n^2 = 2 \log n$
$\approx m \log n$

$m \geq n$, so $\log m \leq \log n^2 = 2 \log n$

$\approx m \log n$

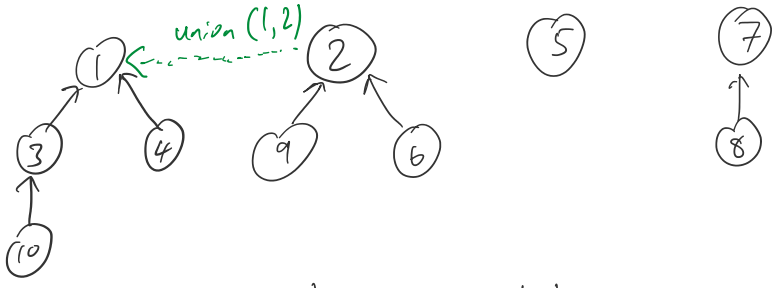
At most $2m$ "find" operations $\approx 2m$ time
(to check if u, v are in same component)

At most $n-1$ union operations $\approx n \log n$ time

$\Rightarrow$ total runtime $\approx m \log n + 2m + n \log n$

largest term since $m \geq n$ if graph is connected and not already a tree

Tree-based Union-Find



Make Union Find (S): create $|S|$ single-node trees. $\Theta(S)$ time

Find (i): Follow pointer from i to root of tree

Union (x,y): If $|x| < |y|$, make x point to y. $\Theta(1)$ time

Thm: Find (i) takes time $\Theta(\log n)$

proof. set [i] is renamed at most $\log_2(n)$ times
because each renaming doubles the size.
The depth of a tree is the max number of renamings for an item.

Runtime of Kruskal using trees:

Sorting edges $\Theta(m \log n)$

$\leq 2m$ "find" ops: $\log n$ time each $\rightarrow 2m \log n$

$\leq n-1$ union ops: $\Theta(1)$ time

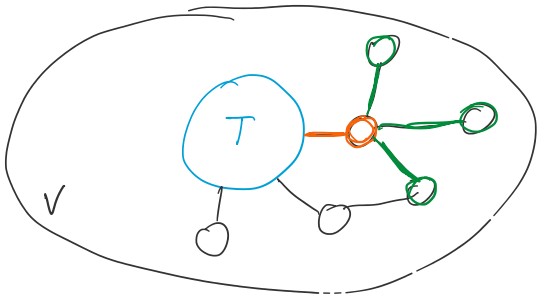
Total $\approx n + 2m \log n + n \leq 3m \log n + n$

$\leq n-1$ union ops : $O(n)$

Total running time $\approx m \log n + 2m \log n + n \leq 3m \log n + n$
$= \Theta(m \log n)$

Recall: Prim's starts from arbitrary node, and grows the tree adding the lightest edge on the frontier

- Need neighbors(u)
- Need ClosestVertex in frontier
  Updates at each step



could keep list of all neighbors and their distances, which would take $O(n)$ time to search through and update.

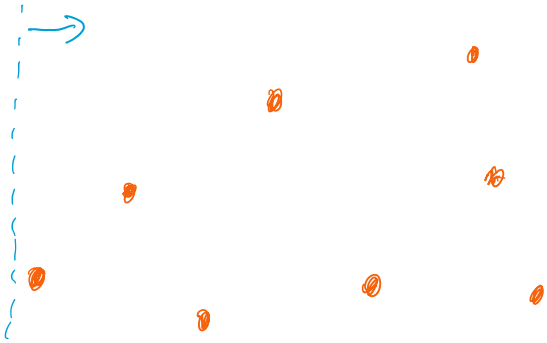
Priority Queue ADT & heaps

A heap H holds items i that have keys key(i), and make it easy to find the smallest key.

redundant
- H.insert(i)
- H.deletemin() – return smallest key & delete it } exactly the closestVertex operation
- H.makeheap(S)
- H.findmin()
- H.delete(i)

} redundant

other applications includes process priority or plane sweep



Store points in priority queue, ordered by x-coordinate

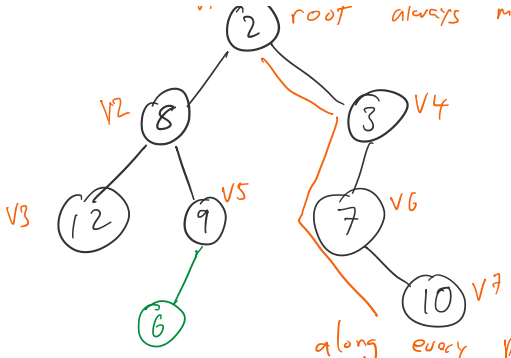
more efficient than keeping a sorted list if we only need to repeatedly find the smallest item. (both creation and updating)

Heap-ordered trees

(heap data structure ≠ heap memory)

root always minimum

- keys of child nodes ≥ key of parent

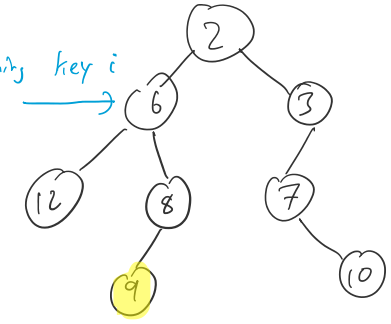


- keys of child nodes ≥ key of parent

Reminder: nodes are not just key values

along every path, keys monotonically non-decreasing

add nodes as leaf
and "sift up" by swapping with parent if smaller

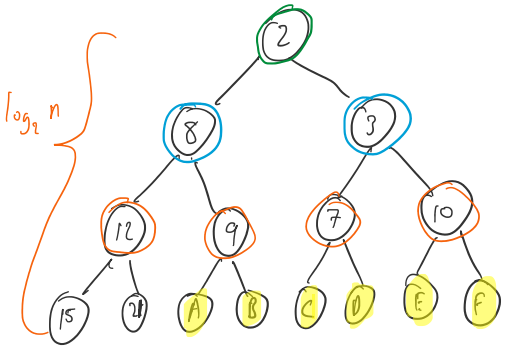


delete node containing key i
1. need pointer to node
2. replace key with key j of a leaf
3. delete leaf
4. If i > j, sift up
5. If i < j, sift down by swapping current node with smallest child until done

findMin: O(1)
insert/deleteMin: O(tree height) + time to find leaf
how to keep low
how to find

- Use last inserted node for deletes
- Choose next empty spot in complete tree.

Complete Tree ⟺ Array



left(i) = 2i if 2i ≤ n else None
right(i) = 2i+1 if 2i+1 ≤ n else None
parent(i) = ⌊i/2⌋ if i ≥ 2 else None

Can also create d-heap with higher fan-out

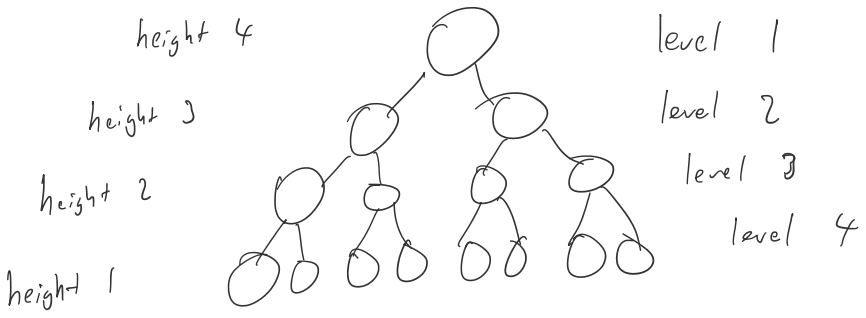
H.decreaseKey(u, j) – reduce key for item u by j.
Could just delete and reinsert in O(log n) time.
But more efficient to just reduce the key and siftup in a smaller O(log n)

H.makeHeap(S) – could insert n times, but that takes O(n log n)
Can do better O(n):
1 Put items arbitrarily into array (i.e. complete tree)

Can do better $O(n)$:

1. Put items arbitrarily into array (i.e. complete tree)
2. for all elements in reverse order, siftdown. — notice that the bottom is always heap-ordered compared to the active node

Would it work to start from the top and sift up?



at height h, there are $\leq \frac{n}{2^h}$ nodes
↳ only need to siftdown h levels

Runtime: $\sum_h h \cdot \frac{n}{2^h} = n \sum_h \frac{h}{2^h} \leq 2n = O(n)$

Claim: $\sum_{h=1}^{\infty} \frac{h}{2^h} = 2$

ratio test

proof: $\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \frac{5}{32} + \cdots$

$= \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots = 1$

$+ \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots = \frac{1}{2}$

$+ \frac{1}{8} + \frac{1}{16} + \cdots = \frac{1}{4}$

$+ \frac{1}{16} + \cdots = \frac{1}{8}$

$\vdots$

$= 2$

heaps can also be used for sorting by repeatedly deleting the root. $O(n \log n)$