Skip Lists - CS 126

What is a skip list?

- Skip list $S = \{S_0, S_1, \ldots, S_h\}$
- S_0 contains all elements
- S_i is a random subset of S_{i-1} for $i = 1, \ldots, h-1$
- S_h only contains $-\infty$

Inserting

- repeatedly toss a fair coin until we get tails.
  - i is the number of times the coin came up heads.
- If $i \geq h$, we add to the skip list lists $S_{h+1}, \ldots, S_{i+1}$
  - each only containing $-\infty$.
- We search for k and find positions $p_0, p_1, \ldots, p_i$ of the items with largest element $< k$ in lists $S_0, S_1, \ldots, S_i$.
- For $j = 0, \ldots, i$ we insert k into list S_j after position p_j.

Space usage

Fact 1: prob of i consecutive heads is $1/2^i$

Fact 2: If each of n entries is present in a set with prob p, expected size of the set is np.

Using these facts the expected size of each list $S_i = \frac{n}{2^i}$

Searching

Start at $-\infty$ in S_h

Repeat:

- Scan forward until we reach rightmost position p s.t. element$(p) \leq k$
- If you can not scan forward anymore, then drop down.

If you cannot scan forward in S_0, Stop.

Deletion

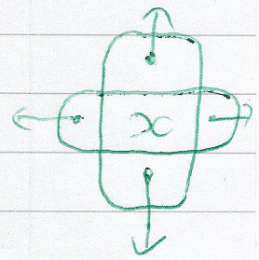
Search for x in skip list and find positions $p_0, p_1, \ldots, p_i$.

- we remove these from the lists.
- we then remove all but one empty list.

Implementation

using quad nodes

And define special nodes $-\infty$.



So total size is

$$\sum_{i=0}^{h} \frac{n}{2^i} = n \sum_{i=0}^{h} \frac{1}{2^i} < 2n$$

Hence space complexity is $O(n)$

Skip Lists 2 - CS126

Height

With high probability a skip list with n items has height $O(\log n)$.

Fact 3:

If each of n events has prob. p, the prob that atleast one event occurs is at most np.

Search and update times

Search time is proportional to:

- number of drop-down steps
- number of scan-forward steps.

drop-down steps are bounded by the height.

- $h = O(\log n)$ with high probability.

Scan-forward steps:

Fact 4: expected number of coin tosses required in order to get tails is 2.

By fact 4, in each list the expected number of scan forward steps is 2.

Thus the expected number of scan-forward steps is $O(\log n)$

Search takes $O(\log n)$ time

- By fact 1, each entry is in list S_i with prob $1/2^i$
- By fact 3, prob that list S_i has at least one item is at most $n/2^i$
- For $i = 3\log n$, the prob that $S_{3\log n}$ has atleast 1 entry is at most $1/n^2$
- Skip list with n entries has height at most $3\log n$ with prob at least $1 - 1/n^2$

Expected Vs. worst case

Expected:

- Show what is the typical running time of the algorithm
- Worst case time can be much larger.
- Large deviations from the expected time are untypical

Worst:

- Worst case scenario
- Applies to all instances of the problem
- May not be representative of what typically happens.

Binary Search Trees - CS126

Ordered Maps

Keys are assumed to come from a total order.
Items are stored in order by their keys.
Allows nearest neighbour queries:
- Item with largest key less than or equal to k
- Item with smallest key $\geq k$

Binary Search

Can perform nearest neighbour queries on an ordered map.
- Each step, the number of candidates is halved.
- terminates after $O(\log n)$ steps

Search Tables

ordered map implemented by an array-based sorted sequence.
performance:
- Search - $O(\log n)$
- insert - $O(n)$
- delete - $O(n)$.

Binary Search Tree

Binary tree storing keys at its internal nodes.
External nodes store nothing.
- for each internal p with key k_p
  - keys in left subtree are $< k_p$
  - keys in right subtree are $> k_p$

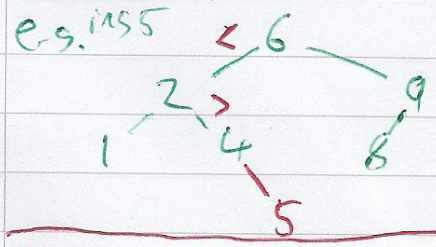
Inorder traversals of BST visits keys in increasing order.

Search

```
TreeSearch(k, v)
  if T.isExternal(v)
    return null
  if k < key(v)
    return TreeSearch(k, left(v))
  if k = key(v)
    return v
  return TreeSearch(k, right(v))
```

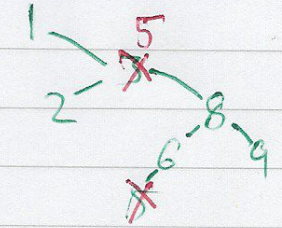
Insertion

e.g. insert 5



Deletion

Search for k, replace k with least element in the right subtree



Performance

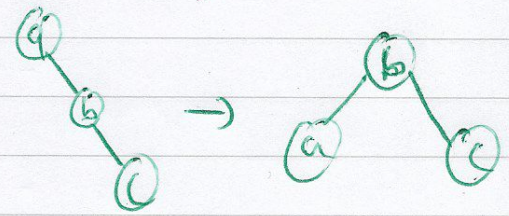
Space - $O(n)$
Search, insert & delete - $O(\log n)$

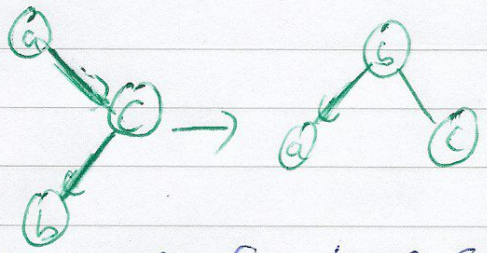
AVL Trees - CS126

Definition

- AVL trees are balanced
- An AVL tree s.t. every internal node v's children's height only differs by 1.

Balancing





Vice versa for trees spanning left.

AVL tree performance

Space - O(n)
Rebalancing - O(1)
Search - O(log n)
insert - O(log n)
removal - O(log n)

Height of AVL tree

height = log n
Read slide 3 for proof

Insertion

Same as normal BST but followed by balancing method

Removal

Uses BST method for removal followed by rebalancing the tree.

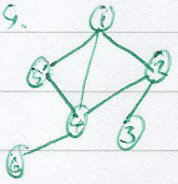
To do this we traverse up the tree from the node we removed. And if the subtree at that node is not balanced we proceed to balance it.

Graphs - CS126

What is a graph?

A pair (V, E)
- V = vertices
- E = Edges

e.g.



Graph types I

Directed
All edges are directed

undirected
All edges are undirected

Terminology

Adjacent vertices
2 vertices that share an edge.

Edges incident on a vertex
edges connected to a vertex.

End vertices of an edge
vertices that are connected by 1 edge.

Degree of a vertex
number of edges coming out of a vertex

parallel edges
edges with same vertices

self loop
both ends of an edge are the same vertex

path
sequence of alternating vertices and edges

Cycle
Circular sequence of alternating vertices and edges

Edge types

Directed

ordered pair

Undirected

unordered pair

Applications

Electronic circuits
Transportation Networks
Computer Networks
Databases

Tree

A graph without a cycle

Properties

1. Sum of the degrees of the vertices is an even number
2. In an undirected graph with 0 self-loops and no parallel edges
   $m \leq n(n-1)/2$
   where $n = \#$ of vertices
   $m = \#$ of edges

Graph ADT

numVertices()	vertices()
numEdges()	Edges()
getEdge(u, v)	endVertices(e)
opposite(v, e)	outDegree(v)
inDegree(v)	outgoingEdges(v)
incomingEdges(v)	insertVertex(o)
insertEdge(u, v, o)	removeVertex(v)
removeEdge(e)	

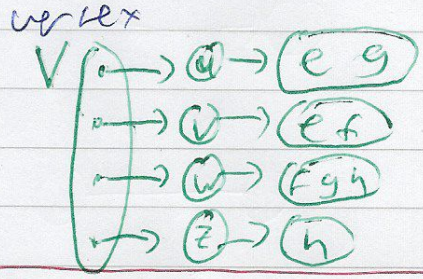
Graphs 2 - CS126

Edge List Structure

- Vertex object
  - element
  - reference to position in vertex sequence
- Edge object
  - element
  - origin vertex object
  - destination vertex object
  - ref. to position in edge sequence.
- Vertex sequence
- Edge sequence

Adjacency List Structure

- Incidence sequence for each vertex



Adjacency Matrix Structure

- Edge List Structure

	u	v	w	z
u		e	g	
v	e		f	
w	g	f		h
z			h	

Depth-First Search - CS126

Subgraph
- A subgraph S of a graph G is a graph S.t.
  - vertices of S are a subset of the vertices of G
  - edges of S are a subset of the edges of G

Spanning subgraph
a subgraph where vertices of S = vertices of G.

Connected graph
A graph is connected if there exists a path between every pair of vertices.

Spanning trees and forests
Spanning tree is a spanning subgraph that is a tree.

Same for spanning forest

Trees and Forests
Tree
Connected graph with 0 cycles.
Forest
Unconnected graph with 0 cycles

DFS - O(n+m)
Technique for traversing a graph.
A DFS traversal:
- visits all vertices and edges
- Determine connectivity
- Computes connected components
- Computes a spanning forest

DFS Algorithm
DFS(G, u)
  Mark u as visited
  for each of u's outgoing edges
    if V not visited
      record edge e as discovery edge for V.
      Recursively call DFS(G, V)

Analysis of DFS
- Setting/getting a vertex/edge label takes O(1) time.
- Each vertex is labelled twice
  - unexplored and visited
- Each edge is labelled twice
  - unexplored and Discovery/back
- DFS runs in O(n+m) time provided the graph is represented by an adjacency list.

properties of DFS
Property 1
DFS(G, v) visits all edges and vertices in the connected component of V
Property 2
Discovery edges labeled by DFS(G, v) form a spanning tree of the connected component of V

Depth-First Search 2 - CS126

Path finding

Alter DFS slightly for path finding

```
pathDFS(G, v, z)
  setLabel(v, VISITED)
  S.push(v)
  if v=z
    return S.elements()
  for all e∈G.incidentEdges(v)
    w=opposite(v,e)
    if getLabel(w)=UNEXPLORED
      setLabel(e, DISCOVERY)
      S.push(e)
      pathDFS(G, w, z)
      S.pop(e)
    else
      setLabel(e, BACK)
  S.pop(v)
```

DFS for an entire graph

```
DFS(G)
  for all u∈G.vertices()
    setLabel(u, UNEXPLORED)
  for all e∈G.edges()
    setLabel(e, UNEXPLORED)
  for all v∈G.vertices()
    if setLabel(v)=UNEXPLORED
      DFS(G, v)
```

Cycle finding

Alter DFS slightly for cycle finding

```
cycleDFS(G, v)
  S.push(v)
  for all e∈G.incidentEdges(v)
    if setLabel(e)=UNEXPLORED
      w=opposite(v,e)
      S.push(e)
      if getLabel(w)=UNEXPLORED
        setLabel(e, DISCOVERY)
        cycleDFS(G, w)
        S.pop(e)
      else
        T=new empty stack
        repeat
          o=S.pop()
          T.push(o)
        until o=w
        return T.elements()
  S.pop(v)
```

Breadth-First Search - CS126

BFS Algorithm

```
BFS(G, s)
  L0 = new empty sequence
  L0.addLast(s)
  setLabel(s, VISITED)
  i = 0
  while ¬Li.isEmpty()
    Li+1 = new empty sequence
    for all v ∈ Li.elements()
      for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
          w = opposite(v, e)
          if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            setLabel(w, VISITED)
            Li+1.addLast(w)
          else
            setLabel(e, CROSS)
    i = i + 1
```

Analysis

BFS runs in $O(n+m)$ time. - Given it is represented by an adjacency list structure.

DFS Vs. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	✓	✓
Shortest paths		✓

Cross Edge (V, W)

W is in the same or next level of V.

More on BFS

```
BFS(G)
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
      BFS(G, v)
```

properties

Property 1

BFS(G, s) visits all the vertices and edges of G_s

Property 2

discovery edges labelled by BFS(G, s) form a spanning tree T_s of G_s

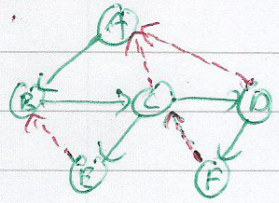
Property 3

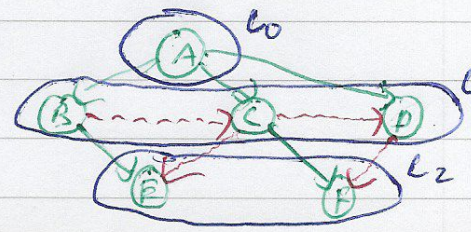
Consider BFS(G, s)

- For each vertex v in L_i, we have $dist(s, v) = i$
- The path in the BFS tree T_s from s to v is one of the shortest paths between the 2 vertices

Back edge (V, W)

W is an ancestor of V

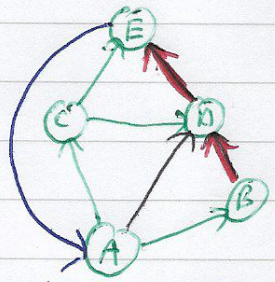




Directed Graphs - CS126

Disraph
A graph whose edges are all directed.

Directed DFS
In directed DFS, we have 4 types of edges
- discovery
- back
- forward
- cross



determines vertices reachable from a node.

Strong Connectivity
where every vertex can reach every other vertex

Strongly Connected Components
- Maximal ~~subdigraph~~ subgraphs s.t. each vertex can reach all other vertices in the subgraph
- Can be achieve in $O(n+m)$ using DFS

Transitive Closure
Given a disraph G, the transitive closure of G is the disraph G^* s.t.:
- G^* has same vertices of G
- If G has a directed path from v to w, G^* has a directed edge from v to w.

Disraph properties
If G is simple:
$M \leq n \cdot (n-1)$

If we keep in-edges and out-edges in seperate adjacency lists, we can perform listing of incoming edges and outgoing edges in time proportional to their size

Reachability
A node v is reachable from u, if there exists a path from u to v.

Strong Connectivity Algorithm
- Pick vertex v in G
- perform DFS from v in G
  - if w not visited print "no"
- Let G' be G with edges reversed.
- perform DFS from v in G
  - if w not visited print "no"
  - else, print "yes"
- running time $O(n+m)$

Computing Transitive Closure.
either
- perform DFS starting at each vertex - $O(n(n+m))$
- Or, use the Floyd-Warshall algorithm - $O(n^3)$

Directed Graphs 2 - CS126

Floyd-Warshall's Algorithm

Number vertices $v_1, \ldots, v_n$

Compute digraphs $G_0, \ldots, G_n$

- $G_0 = G$
- G_k has directed edge (v_i, v_j) if G has a directed path from v_i to v_j, with intermediate vertices in $\{v_1, \ldots, v_k\}$

- $G_n = G^+$
- In phase k, digraph G_k is computed from G_{k-1}
- We add (v_i, v_j) in G_k if edges (v_i, v_k) and (v_k, v_j) appear in G_{k-1}.

```
FloydWarshall(G)
  i = 1
  for all v ∈ G.vertices()
    denote v as v_i
    i += 1
  G_0 = G
  for k ← 1 to n
    G_k = G_{k-1}
    for i ← 1 to n (i ≠ k)
      for j ← 1 to n (j ≠ i or k)
        if G_{k-1}.areAdjacent(v_i, v_k) ∧
           G_{k-1}.areAdjacent(v_k, v_j)
          if ¬G_k.areAdjacent(v_i, v_j)
            G_k.insertDirectedEdge(v_i, v_j, k)
  return G_n
```

DAGs and Topological Ordering

(Directed Acyclic Graph)

- A DAG is a digraph that has no directed cycles.
- A topological ordering of a digraph is a numbering $v_1, \ldots, v_n$ of the vertices s.t. for every edge (v_i, v_j) $i < j$

A digraph admits a topological ordering if and only if it is a DAG.

Topological Sorting

```
TopologicalSort(H)
  i = n
  while H is not empty
    - Let v be a vertex with
      no outgoing edges.
    - Label v = i
    - i -= 1
    remove v from H.
```