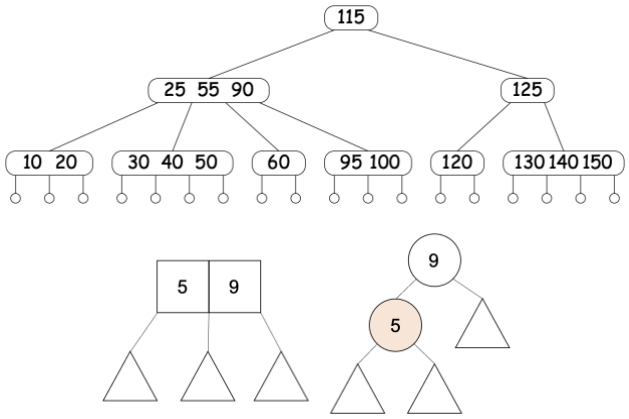
Prep: bring ID, jacket, glasses, water, lots of paper, pen, watch and this set of notes
You got this!

Continued from Midterm 2 Sheet

Balanced Search Structures

B-Trees

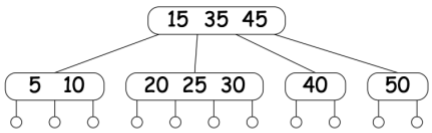


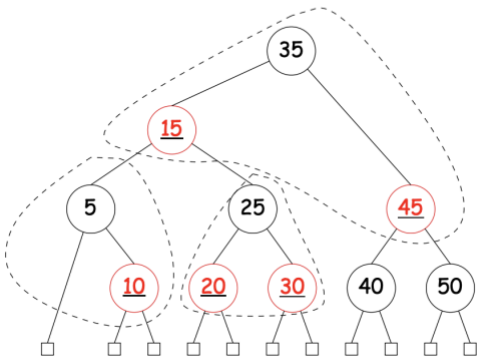
Invariants

- A node with k items must have $k+1$ children (null children for leaf nodes)
  - A node in 2-3 tree has at most size 2 and at most 3 children
  - A node in 2-4 tree has at most size 3 and at most 4 children
- All leaves are of same distance to root

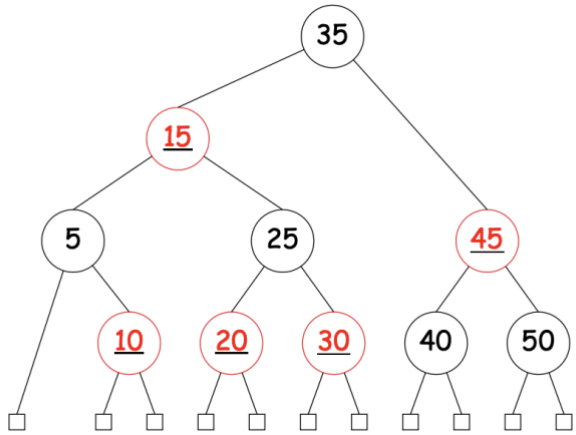
Properties

- The height is guaranteed $\Theta(\log N)$
- Searching each node is constant.
- 2-4: each parent can have 2 to 4 children
- Add from the bottom. If can contain, just put inside, else split the node, moving the middle element up.





Red-Black Tree



Invariants

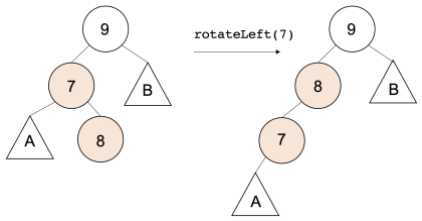
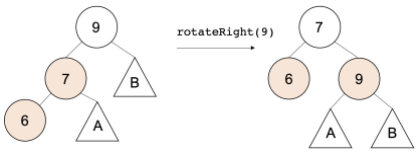
- Root is always a black node
- Every leaf has same number of black ancestors ("black height") including itself
- Every internal node has two children
- Every red node has two black children
- **No two red nodes in a row**
- [LLRB] **No red node is the right child of another node**

Properties

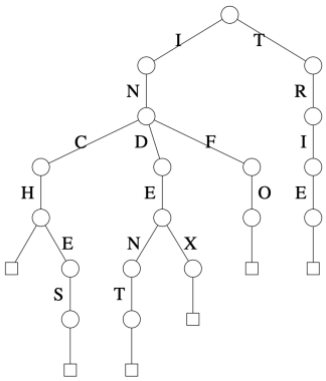
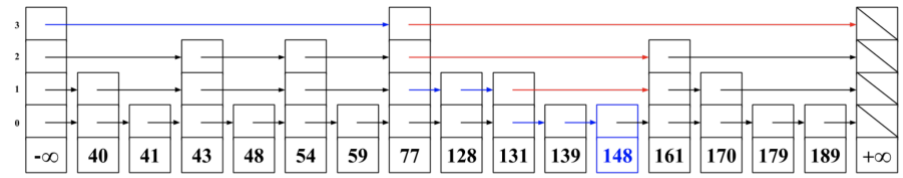
- Isomorphism between 2-3 trees and Left-leaning Red Black Trees (LLRB)
- Red nodes allow us to capture "overstuffed" nodes in a 2-3 tree.
- Four fix-up operations
  - `rotateLeft()`
  - `rotateRight()`
  - `colorFlip()`
  - change the root node to black
- `rotateLeft()` and `rotateRight()` can change height of tree by 1.
- `colorFlip()` is equivalent to exploding
- Black nodes: equivalent to any normal binary tree node
- Red nodes: equivalent to BTree node with more than one value.
- Make new root have the color of the old root and color the old root red.
- Always add values to a leaf node as a red node first
- If link is right leaning, rotate to make it left leaning

	• If node already has a red link to the left, temporarily add it to the right also as a red link • 2-4 tree is isomorphic to a RB Tree without the left leaning condition (a black node can have 2 red children) • Every node must have the same number of black nodes in between itself and the root (due to isomorphism) • Inserting: do normal binary tree insertion and color the inserted node red. restore LLRB properties • $\Theta(\log N)$ performance for searches, insertions and deletions • Searching always $O(\lg N)$

Fix-up Cases

Case A	Case B	Case C
		

Data Structures

Trie	Skip Lists (Probabilistic Data Structure)
	
• $\Theta(B)$ performance for searches, insertions and deletions, where B is length of key processed. • Hard to manage space efficiently.	• Start with the highest level, traverse the pointers till “overshot”. Then start from the previous block and traverse again. • Heights of nodes chosen randomly (almost $\frac{1}{2}$ as many nodes that are $> k$ high as there are k high) • Probable $\Theta(\log N)$ performance for searches, insertions and deletions

Graph Algorithms

Topological Sort $O(V+E)$	Union Find Disjoint Set (UFDS)
• Only DAGs have topological sorting. • Reverse DFS post-order traversal will always return a valid topological sort.	• With path compression and without path compression will return different trees. • For path compression, when checking if an edge should be added to the root, the path is

- Another approach: Take any node with 0 in-degree, add it into the topological array and remove out-going edges. Repeat. If at any step, there are multiple choices of nodes to remove, then there's multiple topological sorting.

compressed. Stop when all nodes are connected (no need to process till the end)

Graph Search

Dijkstra's Algorithm $O(V + E \log V)$

- Computes shortest path from single source to all other vertices
- Must ensure **nonnegative** edge weights
- Obtains a shortest path tree from source node

v	$\text{dist}(v)$					
	Init	A	D	B	C	E
A	0	0	0	0	0	0
B	∞	4	4	4	4	4
C	∞	∞	6	6	6	6
D	∞	2	2	2	2	2
E	∞	∞	9	8	7	7

Dijkstra Variants and Techniques

- Phantom nodes, phantom edges
- Modifying edge weights by $1/E$

A* Search

- Same as Dijkstra, with priority queue with $dist(v) + h(v, e)$ as the edge weight.
- Once a node is processed, update actual distance $\text{dist}(v) = \text{dist}(u) + w_{u,v}$ and push neighbors into priority queue
- Double check answer using table below:

v	$\text{dist}(v)$					
	Init	A	B	D	E	G
A	0	0	0	0	0	
B	∞	1	1	1	1	
C	∞	∞	4	4	4	
D	∞	2	2	2	2	
E	∞	∞	∞	5	5	
F	∞	∞	∞	∞	∞	
G	∞	∞	∞	∞	8	

- Heuristic is *admissible* if $h(v, e) \leq dist(v, e)$ (i.e. never overestimates actual distance)
- $h(v, e) = 0$ reduces to Dijkstra's algorithm
- Must fulfil the triangle inequality i.e. $h(A) \leq h(B) + d(A, B)$

Minimum Spanning Tree (MST)

- [Cut Property] Given any cut, the minimum-weight crossing edge is in MST
- If none of the edge weights are identical, there will only be one MST. (Kruskal is deterministic)

Prim's Algorithm ($O(E \log V)$)

- Maintain a set of vertices S, initially $\{v_1\}$.
- Among all v_i adjacent to S but not in S, add the one with smallest incident edge.
- Repeat.

Kruskal's Algorithm ($O(E \log V)$)

- Maintain a sorted edge list E.
- Add the smallest edge in E to S that doesn't form a cycle in S.
- Repeat.

Final Checks

- After every fix up operation, ensure the LLRB tree satisfies invariants
- Run graph algorithms during checks
- All graphs are simple in 61B - no edge from node to itself, no multiple edges between two nodes

Last Resorts

- Do HW9 again and check understanding
- check minimum spanning tree homework, add in tips from there.

Appendix VI: Implementations

Red Black Tree

```
public class RedBlackTree<T extends Comparable<T>> {
    private RBTreeNode<T> root;
    public RedBlackTree() { root = null; }
    RBTreeNode<T> rotateRight(RBTreeNode<T> node){
        if (node.left == null) return node;
        RBTreeNode<T> newRoot = node.left;
        newRoot.isBlack = node.isBlack;
        node.isBlack = false;
        node.left = newRoot.right;
        newRoot.right = node;
        return newRoot;
    }

    RBTreeNode<T> rotateLeft(RBTreeNode<T> node){
        if (node.right == null) return node;
        RBTreeNode<T> newRoot = node.right;
        newRoot.isBlack = node.isBlack;
        node.isBlack = false;
        node.right = newRoot.left;
        newRoot.left = node;
        return newRoot;
    }

    void flipColors(RBTreeNode<T> node){
        node.isBlack = !node.isBlack;
        node.left.isBlack = !node.left.isBlack;
        node.right.isBlack = !node.right.isBlack;
    }

    private boolean isRed(RBTreeNode<T> node){
        return node != null && !node.isBlack;
    }

    void insert(T item){
        root = insert(root.item);
        root.isBlack = true;
    }


    private RBTreeNode<T> insert(RBTreeNode<T> node, T
item){
        if (node == null) return new
RBTreeNode<>(false, item);
        int comp = item.compareTo(node.item);
        if (comp == 0){
            return node;
        } else if (comp < 0){
            node.left = insert(node.left, item);
        } else {
            node.right = insert(node.right, item);
        }

        // handles "right leaning"
        if (isRed(node.right) && !isRed(node.left)){
            node = rotateLeft(node);
        }

        // handles both children red
        if (isRed(node.left) && isRed(node.right)){
            flipColors(node);
        }
        return node;
    }

    static class RBTreeNode<T>{ /* OMIT */ }

}
```

Dijkstra's Algorithm

```
import java.util.Comparator;
import javafx.util.Pair;
import java.util.PriorityQueue;

public class Dijkstra {

  public PriorityQueue<Pair<Integer, Integer> > pq;
  public int[] dist;

  public Dijkstra(int n) {
      pq = new PriorityQueue<>(_cmp);
      dist = new int[n];
      for (int i = 0; i < n; i++) dist[i] = -1;
  }

  public void dijkstra(int s, ArrayList<Pair<Integer,
Integer> > adjList[]){
      dist[s] = 0;
      pq.add(new Pair(s, 0));
      while (!pq.isEmpty()) {
          Pair p = pq.poll();
          for (int i = 0; i < adjList[p].size(); i++){
              int v = adjList[p].get(i).getKey();
              int w = adjList[p].get(i).getValue();
              if (dist[v] == -1 || dist[i] + w <
dist[v]){
                  dist[v] = dist[i] + w;
                  pq.push(new Pair(dist[v], v));
              }
          }
      }
  }

  private static final Comparator<int[]> _cmp =
      new Comparator<int[]>() {
          @Override
          public int compare(Pair e0, Pair e1) {
              if (e1.getKey() != e2.getKey()) return
e1.getKey() - e2.getKey();
              return e1.getValue() - e2.getValue();
          }
      };
}
```

Random

```
import java.util.Random;
import java.util.Math;
import java.util.Collections;

public static void main(String[] args){
    double rand = Math.random(); // rdm double [0, 1)
    System.out.println("Random number: " + rand);

    ArrayList<String> lst = new ArrayList<>();
    lst.add("jz");
    lst.add("jj"); // add more here
    Collections.shuffle(lst);
}
```

Topological Sorting

```
import java.util.ArrayList;
import javafx.util.Pair;

public class Topological {

    private int _n, _counter;
    private boolean[] _visited;
    private int[] _topo;

    public Topological(int n){
        _n = n;
        _visited = new boolean[n];
        _topo = new int[n];
```

Kruskal's Algorithm

```
import java.util.Arrays;
import java.util.Comparator;

public class Kruskal {
    public static int[][] mst(int V, int[][] E){
        // E[i] in the form of [v1, v2, edge_weight]
        E = Arrays.copyOf(E, E.length);
        int[][] result = new int[V - 1][];
        Arrays.sort(E, _cmp);
        UnionFind ufds = new UnionFind(V);

        int c = 0;
        for (int i = 0; i < E.length; i++){
```

```
    }

    public void dfs(int x, ArrayList<Pair<Integer,
Integer> > adjList[]){
        if (_visited[x]) return;
        _visited[x] = true;
        for (int i = 0; i < adjList[x].size(); i++) {
            if (_visited[adjList[x][i]]) continue;
            dfs(adjList[x][i]);
        }
        _topo[_counter] = x;
        _counter += 1;
    }

    public void topological(ArrayList<Pair<Integer,
Integer> > adjList[]){
        for (int i = 0; i < _n; i++){
            if (_visited[i]) continue;
            dfs(i);
        }
        //reverse array
        for (int i = 0; i < _n - 1 - i; i++){
            int temp = _topo[i];
            _topo[i] = _topo[_n - 1 - i];
            _topo[_n - 1 - i] = temp;
        }
    }
}
```

Union Find Disjoint Set (UFDS)

```
public class UFDS {

    public UFDS(int N) {
        p = new int[N];
        sz = new int[N];
        for (int i = 0; i < N; i++){
            p[i] = i;
            sz[i] = 1;
        }
    }

    // with path compression
    public int parent(int v) {
        if (p[v] == v) return v;
        return p[v] = parent(p[v]);
    }

    public boolean sameParent(int u, int v) {
        return parent(u) == parent(v);
    }

    public int merge(int u, int v) {
        int pu = parent(u);
        int pv = parent(v);
        if (pu == pv) {
            return pu;
        }
        int res;
        if (sz[pu] > sz[pv]) {
            p[pv] = pu;
            sz[pu] += sz[pv];
            return pu;
        } else {
            p[pu] = pv;
            sz[pv] += sz[pu];
            return pv;
        }
    }

    private int[] p, sz;
}
```

```
            if (ufds.samePartition(E[i][0], E[i][1]))
continue;
            ufds.union(E[i][0], E[i][1]);
            result[c] = E[i];
            c += 1;
        }
        return result;
    }

    private static final Comparator<int[]> _cmp = new
Comparator<int[]>() {
        @Override
        public int compare(int[] e0, int[] e1){
            return e0[2] - e1[2];
        }
    };
}
```

61B Graph Interface and Implementation

```
public interface Graph {
    Iterator<Integer> vertices();
    Iterator<Integer> successors(int v);
}

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;

public class SimpGraph implements Graph {
    public SimpGraph(){}
    @Override
    public Iterator<Integer> vertices(){
        return _edges.keySet().iterator();
    }
    @Override
    public Iterator<Integer> successors(int v){
        return _edge.get(v).iterator();
    }
    public void add(int v){
        _edges.put(v, new ArrayList<Integer>());
    }
    public void add(int v0, int v1){
        _edges.get(v0).add(v1);
    }
    private HashMap<Integer, ArrayList<Integer>> _edges
= new HashMap();
}
```

61B Traversal

```
import java.util.function.Consumer;
import java.util.Iterator;

public class Traverser {
    public Traverser(Graph G){ _G = G; }
    public void traverseReachable(int v0,
Consumer<Integer> func){
        _func = func;
        _marked.clear();
        traverse(v0);
    }
    public void traverse(int start){
       if (_marked.add(start)){
           _func.accept(start);
           for (Iterator<Integer> i =
_G.successors(start); i.hasNext();) traverse(i.next());
       }
    }
    private Graph _G;
    private HashSet<Integer> _marked = new HashSet<>();
    private Consumer<Integer> _func;
}
```