Red-black Trees

- They are *balanced search trees*, which means their height is $O(\log n)$.
- Most of the search and update operations on these trees take $O(\log n)$ time.
- The structure is well balanced, i.e. each subtree itself is a balanced search tree.
Rotation

- Is the basic operation for maintaining balanced trees.
- Maintains inorder key ordering:
  - For all $a$ in $L$, $b$ in $R$, $c$ in $S$ we have $a \leq b \leq c$.
- Depth($L$) decreases by 1.
- Depth($R$) stays the same.
- Depth($S$) increases by 1.
- Takes $O(1)$

Left-Rotate($T$, $x$)
1: $y=\text{right}[x]$
2: $\text{right}[x]=\text{left}[y]$
3: If $\text{left}[y]!=$NIL then $p[\text{left}[y]]=x$
4: $p[y]=p[x]$
5: If $p[x]!=$NIL then $\text{root}[T]=y$
6: else if $x=\text{right}[p[x]]$ then
  7: $\text{left}[p[x]]=y$
8: else $\text{right}[p[x]]=y$
9: $\text{left}[y]=x$
10: $p[y]=y$

Right-Rotate($B$)

Left-Rotate($A$)
Red-Black Trees

- Every node is either red or black.
- Root and Leaves (NIL) are black.
- If a node is red, then both its children are black.
- All paths from a node \( x \) to a leaf have same number of black nodes (Black-Height(\( x \)))

Example
**Height**

- A red-black tree with \( n \) keys has height \( \leq 2 \log (n+1) \).
- Proof (Intuition): Merge the red nodes into their parents

```
7 11 8
5 3 18
10 22
```

**Proof:**

- Produces a tree with nodes having 2, 3, or 4 nodes

```
2
```

- Height \( h' \) of new tree is black height of original tree:
  - \( h' \geq h/2 \)
  - \( n+1 \) leaves implies \( n+1 \geq 2^{h'} \)
  - \( \log (n+1) \geq h' \geq h/2 \)
Red-Black Insertion

- Insert $x$ into tree
- Color $x$ red.
- Red-black property 1 still holds.
- Red-Black property 2 still holds (inserted node has NIL’s for children).
- Red-black property 4 still holds ($x$ replaces a black NIL and has NIL children).

If $p[x]$ is red, then property 3 is violated.
- To correct, we move violation up in tree until it can be fixed.
- No new violations will be introduced during this process.
- For each iteration, there are six possible cases.
Insertion Cases

- $x$’s is the left child of $x$’s grandparent.
- $x$’s parent’s sibling ($x$’s uncle) is red.
- Then
  - $\text{Color}[p[x]] = \text{Black}$
  - $\text{Color}[\text{right}[p[p[x]]]] = \text{Black}$
  - $\text{Color}[p[p[x]]] = \text{Red}$
  - $x = p[p[x]]$

Insertion case 1
Insertion case 1

- x’s is the left child of x’s grandparent.
- x’s uncle is Black.
- Then
  - x = p[p[x]]
  - Left-Rotate(T, x)
  - Color[p[x]] = Black
  - Color[p[p[x]]] = Red
  - Right-Rotate(T, x)
Insertion case 2

Before insertion:

```
  A
 / 
B   C
 /   / 
1   2   3
```

After insertion:

```
  A
 /   
B     C
 / 
1   2
```

```
Insertion Cases

- $x$’s parent is the left child of $x$’s grandparent.
- $x$’s uncle is Black.
- $x$ is the left child of $p[x]$
- Then
  - $\text{Color}[p[x]] = \text{Black}$
  - $\text{Color}[pp[x]] = \text{Red}$
  - $\text{Right-Rotate}(T,x)$

Insertion case 3
Red-Black Insert

- Cases 4, 5, 6 are symmetric to 1, 2, 3 (x’s parent is the right child of x’s grandparent).
- After case 2 or 3, no further correction is needed.

```plaintext
RB-Insert(T, x)
1: Tree-Insert(T, x)
2: color[x] = Red
3: While x != root[T] and color[p[x]] = Red
4: If p[x] = left[p[x]]
5: y = right[p[x]]
6: If color[y] = Red then
7: color[x] = Black
8: color[y] = Black
9: color[p[x]] = Red
10: else if x = right[p[x]]
11: y = p[x]
12: Left-Rotate(T, x)
13: color[p[x]] = Black
14: color[p[p[x]]] = Red
14: Right-Rotate(T, x)
15: color[root[T]] = Black.
```

Example

- Use R-B Insert to insert element with key 4.
Example

- Use R-B Insert to insert element with key 4.

Example

Case 1

Case 2
### Example

![Graph Example]

**Case 3**

**Done**

### Introduction to Graph Theory

- A graph \( G = (V, E) \) is a pair of sets:
  - \( V \): vertex set.
  - \( E \): edge set.

- A graph may be weighted and its edges might be directed.

\[
V = \{ \text{Sea}, \text{Sfo}, \text{Lax}, \text{Msn}, \text{Stl}, \text{Dfw}, \text{Mia}, \text{Lga}, \text{Pvd} \}
\]

\[
E = \{ (\text{Sea}, \text{Sfo}), (\text{Sfo}, \text{Lax}), (\text{Sea}, \text{Msn}), \ldots, (\text{Lga}, \text{Pvd}) \}
\]
Preliminaries

- Things to know:
  - Path
  - Cycle
  - Sub-graph
  - Degree of a Node
  - Maximum and Minimum Degree
  - Maximum Number of Edges in an Undirected Graph
  - Connected Components of a Graph
  - Shortest Path in a Weighted Graph
  - Tree (rooted tree)
  - Spanning Tree of a Graph:
  - Acyclic Graph
  - Bipartite Graph

Notations:

- Given a graph $G=(V,E)$, where
  - $V$ is its vertex set, $|V|=n$,
  - $E$ is its edge set, with $|E|=m=O(n^2)$.

- If $G$ is connected then for every pair of vertices $u,v$ in $G$ there is path connecting them.

- In an undirected graph an edge $(u,v)=(v,u)$.

- In a directed graph $(u,v)$ is different from $(v,u)$.

- In a weighted graph there weights associated with edges or vertices.

- Running time of graph algorithms are usually expressed in terms of $n$ or $m$. 
Graph Representation in terms of Adjacency Matrix

- The adjacency matrix of a graph $G$, denoted by $A_G$, is an $n$ by $n$ defined as follows:

  $$A_G[i, j] = \begin{cases} 
  1 & \text{if } (i, j) \in E \\
  0 & \text{if } (i, j) \notin E 
  \end{cases}$$

- If $G$ is undirected then $A_G$ is symmetric.

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

Graph Representation in terms of Adjacency List

- In this method for each vertex $v$ in $V$, a list $Adj[v]$ will represent those vertices adjacent to $v$. The size of this list is the degree of $v$.

$Adj[1] = \{2, 3\}$

$Adj[2] = \{3\}$

$Adj[3] = \{\}$

$Adj[4] = \{3\}$
Note that:

- Number of 1’s in $A_G$ is $m$.
- Degree of a vertex is the sum of entries in corresponding row of $A_G$.
- Sum of all degree is $2m$.
- In a directed graph sum of the out degrees is equal to $m$.

Minimum Spanning Tree (MST) in a Weighted Graph

- Let $G=(V,E)$ be a graph on $n$ vertices and $m$ edges, and a weight function $w$ on edges in $E$.
- A sub-graph $T$ of $G$ through all vertices which avoids any cycle is a spanning tree.
- The weight of $T$ is define as sum of the weights of all edges in $T$:

$$w(T) = \sum_{(u,v) \in T} w(u,v)$$
Greedy MST

- The greedy algorithm tries to solve the MST problem by making locally optimal choices:
  1. Sort the edges by weight.
  2. For each edge on sorted list, include that in the tree if it does form a cycle with the edges already taken; Otherwise discard it.
- The algorithm can be halted as soon as \( n-1 \) edges have been kept.
- Step 1. takes \( O(m \log m) = O(m \log n) \).
- Today, we will see that Step 2 can be done in \( O(n \log n) \) time, later we will present a linear time implementation from this step.

Set Operation

- In the proof of running time for our MST algorithm we will use the following set operations:
  - **Make-Set**\((v)\): creates a set containing element \( v \), \( \{ v \} \).
  - **Find-Set**\((u)\): returns the set to which \( v \) belongs to.
  - **Union**\((u,v)\): creates a set which is the union of the two sets, one containing \( v \) and one containing \( u \).
- As an example, we can use a pointer to implement a set system: **Make-Set**\((v)\) will create a single node containing element \( v \).
  **Find-set**\((u)\) will return the name of the first element in the set that contains \( u \), and finally the **union**\((u,v)\) will concatenate the sets containing \( u \) and \( v \).
Example of a set Operations

- Use linked list to show a set
- Make-Set($w$):
- Find-Set($u$): (will return $w$)
- Union($u,v$):

Running Time of Set Operations

- The Make-Set and Find-Set will run in $O(1)$-time.
- How fast can we compute the union.
- Let us ask a different question. Let $N=\{1,\ldots,n\}$ be a set of $n$ integers, and let $P=\{(u,v)\mid u \text{ and } v \text{ in } N\}$ be a subset of pairs from $n \times n$.
- For $u=1$ to $n$ Make-Set($u$);
  For every pair $(u,v)$ in $P$
    If Find-Set($u$)! = Find-Set($v$)
      Union($u,v$)
- Question: How many times does the pointer for an element get redirected?
Union Operation

- Each merge of two sets might take linear number of pointer changes.
- We might have up $O(n^2)$ pointer changes.
- Let us keep a number associated with each set in its root, $\text{Rank}(u)$, which tells how many elements a set has.
- When merging two lists, always change the pointers in the list with smaller rank.

Now each time a pointer changes its corresponding set doubles in the size.

During the whole process the maximum set can become of size at most $n$.

For a specific pointer this happens at most $\log n$ times,

$$2^0, 2^1, 2^2, ..., 2^k = m$$

which means $k = \log n$

Over all $n$ elements, this will result in an $O(n \log n)$ number of pointer updates.
## Kruskal’s MST Algorithm

- It is directly based on Generic MST.
- At each iteration, it finds a light edge, which is also safe, and adds it to an ever growing set, $A$, which will eventually become the MST.
- During the course of algorithm, the structure generated by algorithm is a forest.

1. $A \leftarrow \emptyset$
2. for each $v \in V_G$ do
3. \hspace{1em} Make-Set($v$)
4. Sort Edges in $E_G$
5. for each $(u, v) \in E_G$
   \hspace{1em} (In order of increasing weights)
6. \hspace{1em} if Find-Set($u$) \neq Find-Set($v$)
7. \hspace{1em} \hspace{1em} $A \leftarrow A \cup \{(u, v)\}$
8. \hspace{1em} \hspace{1em} Union($u, v$)
9. Return $A$

## Running time of Kruskal’s Algorithm

- **Step 1**: $O(1)$
- **Steps 2,3**: $O(n)$
- **Step 4**: $O(m \log m)$
- **Steps 5-8**: $O(m \log n)$

1. $A \leftarrow \emptyset$
2. for each $v \in V_G$ do
3. \hspace{1em} Make-Set($v$)
4. Sort Edges in $E_G$
5. for each $(u, v) \in E_G$
   \hspace{1em} (In order of increasing weights)
6. \hspace{1em} if Find-Set($u$) \neq Find-Set($v$)
7. \hspace{1em} \hspace{1em} $A \leftarrow A \cup \{(u, v)\}$
8. \hspace{1em} \hspace{1em} Union($u, v$)
9. Return $A$