Advanced Algorithms Seminar—Draft

Nikhil Pandey
np7803@rit.edu

December 10, 2019

Contents

1 Red Black Trees

1.1 Binary Search Tree

Binary Search Trees are data structures to store information for efficient access (minimum, maximum, other queries) and manipulations (insert, delete, modify) and can be used as a dictionary or a priority queue.

1.1.1 BST Property

$$\begin{aligned} y &\in LeftSubTree(x) \\ &\Rightarrow y.key < x.key \end{aligned}$$

$$\begin{aligned} y &\in RightSubTree(x) \\ &\Rightarrow y.key > x.key \end{aligned}$$

1.1.2 Searching

$$search(x, key) = \begin{cases} nil & \text{If } x \text{ is } nil \\ x & \text{If } x.key = key \\ search(x.left, key) & \text{If } x.key < key \\ search(x.right, key) & \text{Otherwise} \end{cases}$$

1.1.3 Insertion

$$insert(x, key, value) = \begin{cases} Node(key, value, nil, nil) & \text{If } x \text{ is } nil \\ Node(key, value, x.left, x.right) & \text{If } x.key = key \\ Node(x.key, x.value, insert(x.left, key, value), x.right) & \text{If } x.key < key \\ Node(x.key, x.value, x.left, insert(x.right, key, value)) & \text{Otherwise} \end{cases}$$

1.1.4 Deletion

$$delete(x, key) = \begin{cases} nil & \text{If } x \text{ is } nil \\ deleteRoot(x) & \text{If } x.key = key \\ Node(x.key, x.value, delete(x.left, key), x.right) & \text{If } x.key < key \\ Node(x.key, x.value, x.left, delete(x.right, key)) & \text{Otherwise} \end{cases}$$

$$deleteRoot(x) = \begin{cases} x.right & \text{If } x.left \text{ is } nil \\ x.left & \text{If } x.right \text{ is } nil \\ Node(l.key, l.value, x.left, delete(x.right, l.key)) & \text{Otherwise} \\ \quad where\ l = leftistNode(x.right) & \end{cases}$$

$$leftistNode(x) = \begin{cases} x & \text{If } x.left \text{ is } nil \\ leftistNode(x.left) & \text{Otherwise} \end{cases}$$

1.1.5 Time Complexit

Binary Search Trees are pretty efficient when they are balanced. The performance starts becoming linear for an imbalanced tree. The worst-case is because of the tree height deteriorating to n. A self-balancing tree

Operation	Average Case	Worst Case
Search	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Insert	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Delete	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$

automatically keeps its height small on performing insertions and deletion and ensures $\mathcal{O}(\log n)$ operations. Some examples of binary search trees are:

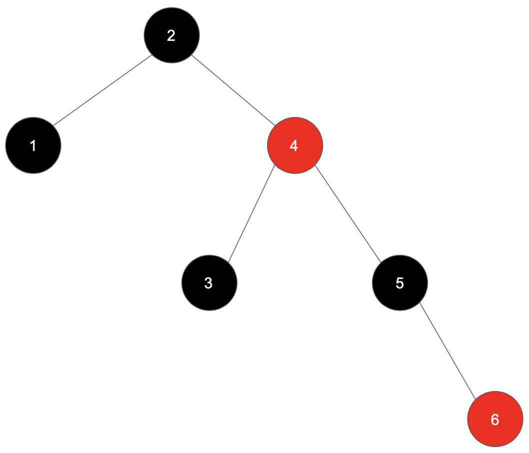
- Red-Black Trees
- AVL Trees
- B-Trees
- 2-3 Trees
- Splay Trees

1.2 Introduction

Red-black tree is a self-balancing binary search tree with each node colored either red or black. Red-black trees satisfy following invariant:

- No red node has a red parent.
- Every path from the root to an empty node contains the same number of black nodes.

Empty nodes are considered to be black. The root is also black. When invariant are maintained most operations take no more than $\mathcal{O}(\log n)$ because the longest possible path in a tree, one with alternative black and red nodes, is no more than twice as long as the shortest possible path, one with black nodes only.



Lemma: *The number of internal nodes of a sub tree rooted at* $x \geq 2^{bh(x)} - 1$

Proof: By mathematical induction

Let n be the number of internal nodes of a sub tree rooted at x, $h(x)$ be the height of sub tree and $bh(x)$ be the number of black nodes from x to any leaf in the sub tree, not counting x if it is black.

- Observe that when $h(x) = 0$ we have,

$$\begin{aligned} n &= 0 \\ &= 1 - 1 \\ &= 2^0 - 1 \\ &\geq 2^{bh(x)} - 1 \end{aligned}$$

- Assume for a node x with $h(x) = k$, $n \geq 2^{bh(x)} - 1$
- A node y, with $h(y) = k + 1$, will have at least two children with black height of $bh(y)$ or $bh(y) - 1$ depending on whether the child is red or black.

$$\begin{aligned} n &= \text{left subtree} + \text{right subtree} + 1 \\ &\geq (2^{bh(y)-1} - 1) + (2^{bh(y)-1} - 1) + 1 \\ &= 2 \cdot 2^{bh(y)-1} - 1 \\ &= 2^{bh(y)} - 1 \end{aligned}$$

Theorem: *The height of a red-black tree with n internal nodes:* $h \leq 2\log(n+1)$

Proof:

Let h be the height of the tree with n internal nodes and bh be its black height.

From the first invariant:

$$bh \geq \frac{h}{2}$$

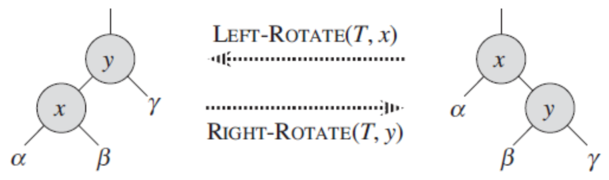
Using the lemma:

$$\begin{aligned} & n \geq 2^{\frac{h}{2}} - 1 \\ \implies & n + 1 \geq 2^{\frac{h}{2}} \\ \implies & \lg(n+1) \geq \frac{h}{2} \\ \implies & h \leq 2 \cdot \lg(n+1) \end{aligned}$$

Corollary: The dynamic-set operations search, minimum, maximum, successor, predecessor run in $\mathcal{O}(\log(n))$.

1.3 Rotations

During insertion and deletion of nodes, above properties or invariant might be violated. Rotations are a way to change the black height of the tree.



1.3.1 Left Rotation

Left rotation assumes that the right child is not *nil*.

$$\begin{aligned} leftRotate(x) = RBNode(&x.right.key, x.right.value, x.right.color, \\ &RBNode(x.key, x.value, x.color, x.left, x.right.left), \\ &x.right.right) \end{aligned}$$

1.3.2 Right Rotation

Right rotation assumes that the left child is not *nil*.

$$\begin{aligned} rightRotate(x) = RBNode(&x.left.key, x.left.value, x.left.color, \\ &x.left.left, \\ &RBNode(x.key, x.value, x.color, x.left.right, x.right)) \end{aligned}$$

1.4 Search

The search procedure is exactly same as Binary Search Tree.

$$search(x, key) = \begin{cases} nil & \text{If } x \text{ is } nil \\ x & \text{If } x.key = key \\ search(x.left, key) & \text{If } x.key < key \\ search(x.right, key) & \text{Otherwise} \end{cases}$$

1.5 Insertion

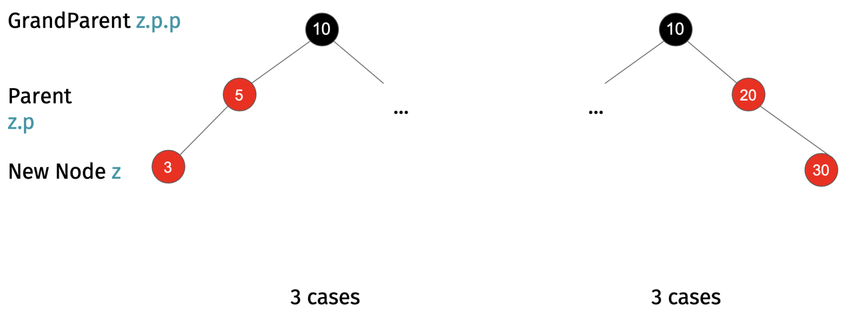
The color of the new node is marked as red. The insertion procedure is similar to binary search tree. After the insertion is done, the violations are fixed. There can be two types of violations.

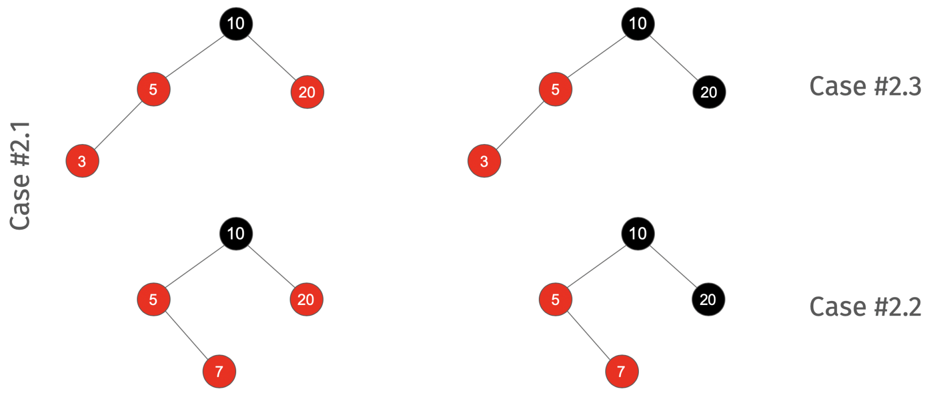
1. Root node might be red.
2. The parent might be red.

Case 1:
If the inserted node is the root, color it black.

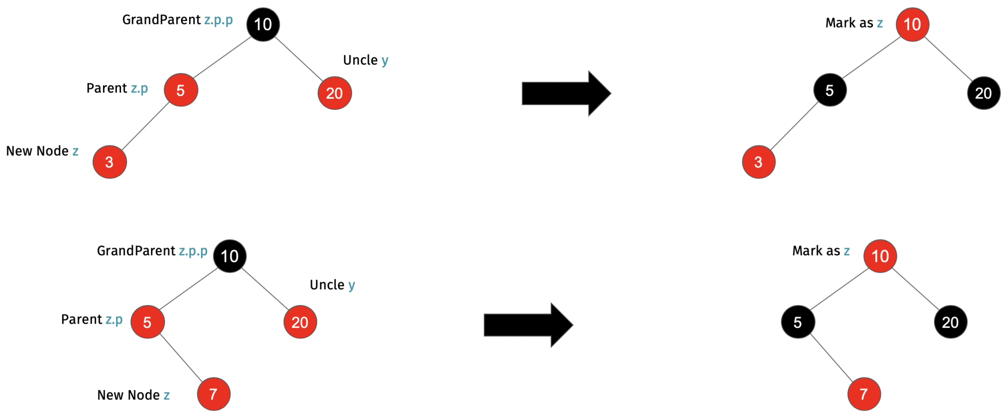
Case 2:
If the parent is red, we can divide this case further into 6 cases. 3 cases when the parent is the left child and

3 cases when the parent is the right child. These cases are symmetrical so only the first three cases when the parent is the left child are discussed.





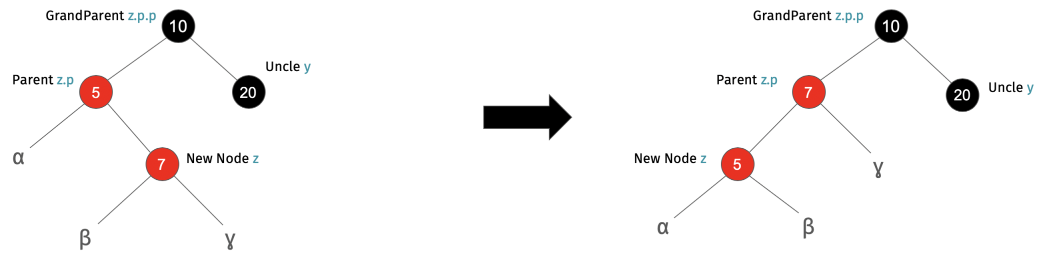
Case 2.1 Uncle is red



- Color parent and uncle black.

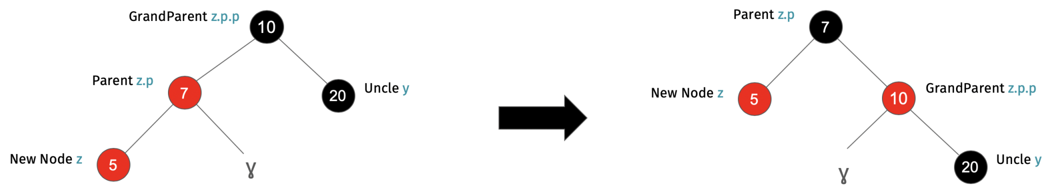
- Color grandparent red.
- Fix violations for grandparent if any.

Case 2.2 New node is the right child and uncle is black



- Left rotate to obtain Case 2.3.

Case 2.3 New node is the left child and uncle is black



- Color parent black.
- Color grand parent red.
- Right rotate grandparent.

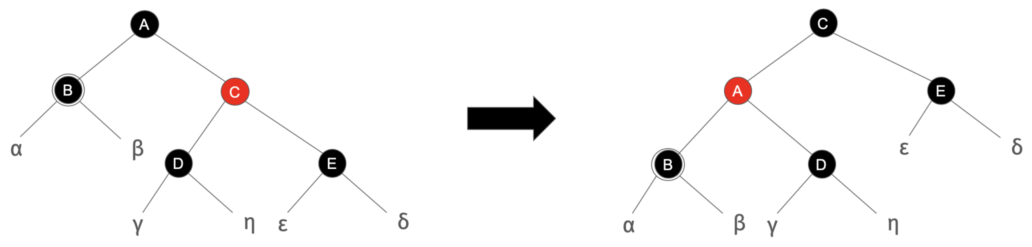
1.6 Deletion

The deletion is similar to that of binary search tree. We need to keep track of the color of the replacement node.

If a leaf node is being deleted, replace with the null node; one child is null, replace with the non-null child; both child are present, replace with the in-order successor of the node.

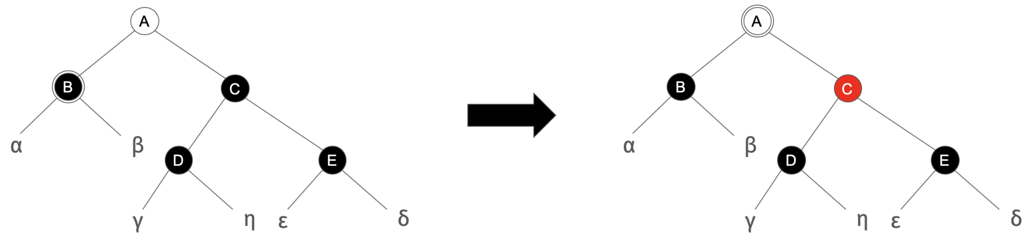
If the removed node was black, the resulting node is referred to as double-black; such double black node exists, the black height needs to be fixed. There are 8 cases we can look into: 4 cases are the symmetrical with the other 4 depending on whether the double-black is the left child or right child. We'll look into the 4 cases where the double-black node is a left-child.

Case 1 Parent is black, sibling is red with two black children.



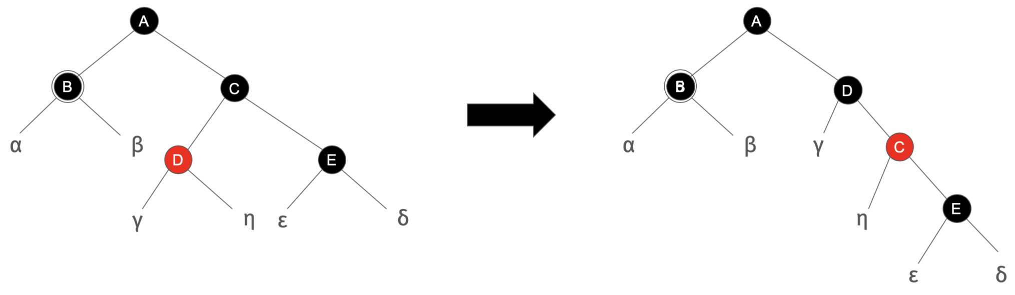
- Color sibling black.
- Color parent red.
- Left rotate parent.
- The double-black node is still double-black. Follow the resulting case.

Case 2 Sibling is black with two black children.



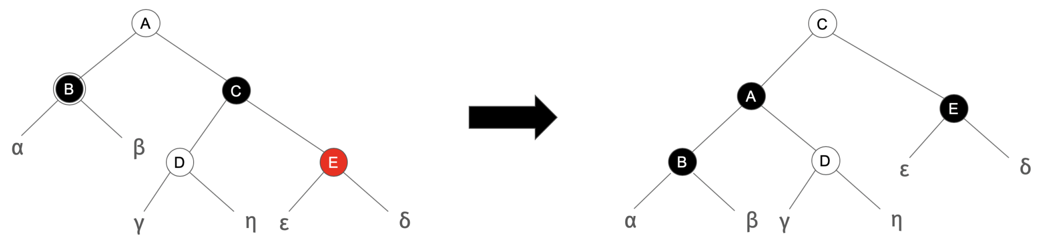
- Color sibling red.
- Increase the blackness of parent. If its red, it becomes black and terminates. If its black, it becomes double-black. Follow the resulting case.

Case 3 Parent is black, sibling is black with red left child and black right child.



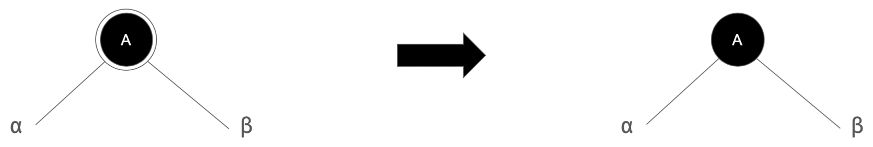
- Color sibling's left child black.
- Color sibling red.
- Right rotate sibling.
- The double-black node is still double-black. Follow the resulting case.

Case 4 Sibling is red with red right child.



- Color sibling with parent's color.
- Color parent black.
- Color sibling's right child black.
- Left rotate parent.
- This is a terminal case.

If the root is double black, it can be colored black without changing the black height.



1.7 Running Time

1.7.1 Insertion

Since the height of a red-black tree of n nodes is $\mathcal{O}(\log n)$, the total cost of the insert procedure without the procedure for fixing the red-black property violation is $\mathcal{O}(\log n)$. All cases are terminal except Case 2.1 which moves the pointer two levels up. The total number of times the fixing procedure can be executed is therefore $\mathcal{O}(\log n)$. Therefore, the overall time complexity of the insert procedure is $\mathcal{O}(\log n)$.

1.7.2 Deletion

Since the height of a red-black tree of n nodes is $\mathcal{O}(\log n)$, the total cost of the delete procedure without the procedure for fixing the red-black property violation is $\mathcal{O}(\log n)$. Cases 1, 3 and 4 lead to a termination after performing constant number of color changes and at most three rotations. Case 2 is the only case in which the procedure can be repeated because the pointer moves up at most $\mathcal{O}(\log n)$ times, performing no rotations. Hence, the fixing procedure takes at most $\mathcal{O}(\log n)$, performing at most three rotations. Therefore, the overall time complexity of the delete procedure is $\mathcal{O}(\log n)$.

Operation	**Average Case**	**Worst Case**
Search	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Insert	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Delete	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

1.8 Example

TODO ...

2 Minimum Cut Problem

2.1 Min Cut

Let $G = (V, E)$ be an undirected graph. A *cut* in this graph is $|E(S, \overline{S})|$ for some $S \subsetneq V$ and $S \neq \varnothing$, the number of edges which cross a partition of the vertices. Formally:

$$\text{min-cut(G)} = \min_{\varnothing \neq S \subsetneq V} |E(S, \overline{S})|$$

Let n be the number of vertices and m be the number of edges. For finding the min-cut, a brute-force solution is to enumerate over all $\mathcal{O}(2^n)$ subsets. There is also a flow-based algorithm using the well-known Max-Flow Min-Cut Theorem. However, these algorithms are still inefficient.

2.2 Max-Flow Min-Cut Theorem

In every flow network with source s and target t, the value of the maximum (s, t)-flow is equal to the capacity of the minimum (s, t)-cut.

Finding a solution to one problem will lead to a solution to the other problem and vice-versa. When no more flow can be routed, we have found a minimum s-t cut.

We can solve the minimum cut problem by using the algorithm for finding the maximum flow as a black-box. Arbitrarily fix a node to be s. For each remaining vertex as t, find the maximum flow from s to t and the corresponding minimum s-t cut. Choose the best of these minimum s-t cuts to be the minimum cut. The run time would be polynomial in n and m and depend on the algorithm for maximum-flow.

2.3 Randomized Algorithms

2.3.1 Monte Carlo Algorithm

A randomized algorithm is called a Monte Carlo algorithm if it may fail or return incorrect answers, but has runtime independent of the randomness. e.g. Karger's Min-Cut Algorithm. The probability of such an incorrect result is bounded. One way of reducing the probability of erroneous results is to simply run the algorithm repeatedly with independent randomized choices each time. The probability of the algorithm to produce wrong result every time by running k times is:

$$\text{P(wrong every time)} = (1 - p)^k \leq e^{-pk}$$

To make the failure probability less than 1 in a million, just run the algorithm $20/p$ times.

2.3.2 Las Vegas Algorithm

A randomized algorithm is called a Las Vegas algorithm if it always returns the correct answer, but its runtime bounds hold only in expectation. e.g. QuickSort, Trapezoidal Map using Randomized Incremental Construction

A Las Vegas algorithm can be converted to Monte Carlo Algorithm by running it for a fixed amount of time and outputting an arbitrary answer if it fails to halt. The reverse is not true because of the strict requirement for Las Vegas algorithms.

2.4 Karger's Algorithm

The algorithm is very simple. Choose a random edge and contract it. Repeat n-1 times until only two vertices remain. Output this as a guess for the min-cut.

```
def BasicKarger(G)
    while there are more than 2 supernodes :
        Pick an edge(u, v) ∈ $E(G)$ uniformly at random
        Merge u and v
    Output edges between the remaining two supernodes
```

The initialization step of Karger's algorithm is as follows:

```
def Initialize(G)
    $\Gamma \leftarrow \varnothing$                                   ▷ Set of SuperNodes
    F $\leftarrow \varnothing$                                   ▷ Set of SuperEdges
    for v ∈ V :
        $\overline{v}$ ← new SuperNode
        V($\overline{v}$) ← $\{v\}$
        $\Gamma \leftarrow \Gamma \cup \{\overline{v}\}$
    for (u, v) ∈ E :
        $E_{uv} \leftarrow \{(u,v)\}$
        F ← F $\cup\{(u,v)\}\}$
```

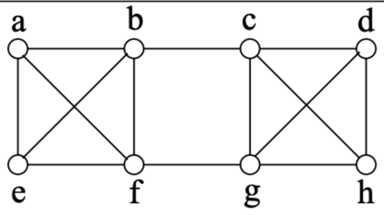
Edge Contraction: Let $G = (V, E)$ be an undirected graph with $|V| = n$ vertices, potentially with parallel edges. For an edge $e = (u, v) \in E$, the contraction of G along e is an undirected graph on $n - 1$ vertices, where we merge u,v to be a single supernode and delete any self-loops that may be created in this process. The edge contraction process is shown below:

```
def Merge(a, b, $\Gamma$)                                   ▷ a, b ∈ $\Gamma$
    x ← new SuperNode
    V(x) ← V(a) ∪ V(b)                ▷ Merge the set of vertices of a and b
    for d ∈ $\Gamma\backslash$\{a, b\} :                            ▷ $\mathcal{O}(n)$ iterations
        $E_{xd} \leftarrow E_{ad} \cup E_{bd}$                         ▷ $\mathcal{O}(1)$ using linked lists
    $\Gamma \leftarrow \Gamma\{a,b\} \cup \{x\}$
```

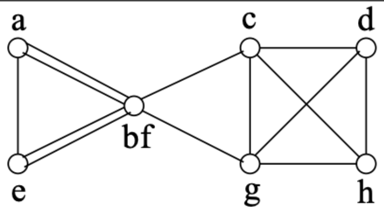
The algorithm in full detail is presented below:

```
def Karger(G)
    Initialize(G)
    while $|\Gamma| > 2$ :
        (u, v) ← uniform random edge from F
        Merge($\overline{u}$, $\overline{v}$, $\Gamma$)                            ▷ u ∈ $\overline{u}$, v ∈ $\overline{v}$
        F ← $F\backslash E_{\overline{uv}}$
    return one of the supernodes in $\Gamma$ and $|E_{xy}|$          ▷ $\Gamma = \{x, y\}$
```

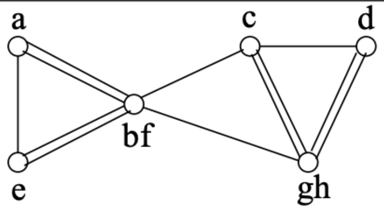
The following example illustrates one possible execution of Karger's algorithm.



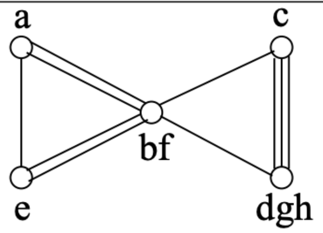
14 edges to choose from
Pick $b - f$ (probability 1/14)



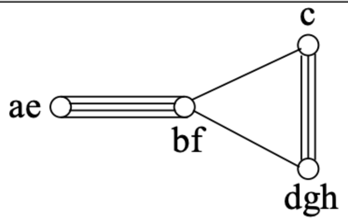
13 edges to choose from
Pick $g - h$ (probability 1/13)



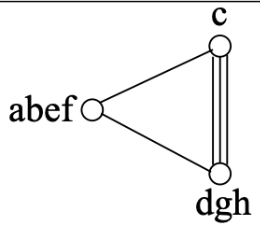
12 edges to choose from
Pick $d - gh$ (probability 1/6)



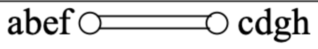
10 edges to choose from
Pick $a - e$ (probability 1/10)



9 edges to choose from
Pick $ab - ef$ (probability 4/9)



5 edges to choose from
Pick $c - dgh$ (probability 3/5)



Done: just two nodes remain

The example happens to give the correct minimum cut, but only because we carefully picked the edges for merging. There are many other choices of edges for merging, so it's possible we could have ended with a cut with more than 2 edges.

The run time of the algorithm is $\mathcal{O}(n^2)$ since each merge takes $\mathcal{O}(n)$ time and there are $n - 2$ merges until there are 2 supernodes left. We can get a better run time using the union-find data structure used for Kruskal's algorithm. In each iteration, we randomly select an edge and merge the two supernodes on the edge in amortized time $\mathcal{O}(\alpha(n))$ where $\alpha(n)$ is the inverse Ackermann function. Since there are m edges, there are at most m merging operations. For the random selection of edges, we generate a random permutation of edges in $\mathcal{O}(m)$ in the beginning and process the edges in that order. In total, the run time is $\mathcal{O}(m\alpha(n))$ with the union-find data structure.

2.5 Analysis of Karger's Algorithm

Lemma: *If degree(v) denotes the number of edge touching node v, then*

$$\sum_{v\in V} degree(v) = 2 \cdot |E|$$

Proof: To see this, imagine the following experiment: for each node, list all the edges touching it. The number of edges in this list is exactly the left-hand sum. But each edge appears exactly twice in it, once for each endpoint.

Lemma: *The size of minimum cut is at most* $2|E|/|V|$

Proof: For any vertex $v \in V$, we can form a cut by taking $E(v, V\backslash\{v\})$. It has *degree(v)* many edges.

$$\text{min-cut(G)} \leq \min_{v\in V}\text{degree}(v) \leq \frac{2\cdot|E|}{|V|}$$

Lemma: *Let $G = (V, E)$ be a graph, $S \subset V$ be a minimal cut in G. Let $e = (u, v) \in E$ be a uniform chosen edge. Then*

$$P_e(e \in E(S, \overline{S})) \leq 1 - \frac{2}{|V|}$$

Proof: We know that:

$$|E(S, \overline{S})| \leq \frac{2\cdot|E|}{|V|}$$

So the probability that $e \in E(S, \overline{S})$ is at most:

$$\frac{(\frac{2\cdot|E|}{|V|})}{|E|} = \frac{2}{|V|}$$

Theorem: *The probability that Karger's algorithm returns a min-cut is at least* $\frac{1}{\binom{n}{2}}$.

Proof: Find a min-cut $S \subset$ V. The algorithm outputs S if it never chooses an edge in S. So,

$$\begin{aligned}
\text{P(algorithm outputs S)} &= P(e_n, \cdots, e_3 \notin E(S, \overline{S})) \\
&= \prod_{i=n}^{3} P(e_i \notin E(S, \bar(S))|e_n, \cdots, e_{i+1} \notin E(S, \overline{S})) \\
&\geq \left(1 - \frac{2}{n}\right) \cdot \left(1 - \frac{2}{n-1}\right) \cdots \left(1 - \frac{2}{3}\right) \\
&= \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{2}{4} \cdot \frac{1}{3} \\
&= \frac{2}{n\cdot(n-1)}
\end{aligned}$$

Corollary: *The number of minimum cuts in G is at most $\binom{n}{2}$.*

Proof: Let $S_1, S_2, \cdots$ be the complete enumeration of all possible cuts of G. Let $p_1, p_2, \cdots$ be the probabilities such that p_i is the probability that Karger's algorithm returns cut S_i.

$$1 = \sum_i p_i \geq \sum_{i=1}^{l} p_i \geq l \cdot \frac{2}{n \cdot (n-1)}$$

The number of minimum cuts l is at most $\frac{n \cdot (n-1)}{2}$.

The success probability of $1/\Theta(n^2)$ seems very small. However, the probability can be boosted to an arbitrarily large probability by performing many independent trials of Karger's algorithm. Let $C > 0$ be an arbitrarily large constant.

```
1: def AmplifiedKarger(G)                                          ▷ C is a constant
2:     Run $C \cdot \binom{n}{2} \cdot \ln(n)$ independent Karger procedures.
3:     return the best of the cuts computed so far
```

The run time of AmplifiedKarger is clearly $\mathcal{O}(n^4 \log(n))$ since we run $\mathcal{O}(n^2 \log(n))$ trials of an $\mathcal{O}(n^2)$ time algorithm.

Lemma:

$$\text{P(AmplifiedKarger is correct)} \geq 1 - \frac{1}{n^C}$$

where C is the constant used in the amplification algorithm.

Proof:

$$\begin{aligned}
\text{P(AmplifiedKarger is incorrect} &= \text{P(Karger is incorrect for all the } C \cdot \binom{C}{2} \cdot \ln(n) \text{ independent runs)} \\
&= (\text{P(Karger is incorrect)})^{C \cdot \binom{n}{2} \cdot \ln(n)} \\
&\leq \left(1 - \frac{2}{n \cdot (n-1)}\right)^{C \cdot \binom{n}{2} \cdot \ln(n)} \\
&\leq exp\left(-\frac{2}{n \cdot (n-1)} \cdot C \cdot \binom{n}{2} \cdot \ln(n)\right) \\
&= exp\left(-C \ln(n)\right) \\
&= \frac{1}{n^C}
\end{aligned}$$

2.6 Karger-Stein Algorithm

The runtime of the amplified version of Karger's algorithm is $\mathcal{O}(n^4 \log(n))$, without using the union-find data structure. The earlier merge operations in the algorithm are less risky i.e. a particular min–cut is more likely to be alive after the merge operation since there are more alive edges. The probability that a minimum cut is alive until there are t supernodes is $\frac{t(t-1)}{n(n-1)}$. For $t = \frac{n}{\sqrt{2}}$, this probability is approximately $\frac{1}{2}$. It is highly likely that a min-cut is alive even after the first $n - t$ merging operations. To reduce the amount of work, it makes sense to repeat the random merging operations from the intermediate multigraph with t supernodes, rather than from the beginning. The algorithm is given below:

```
def KargerStein(G)
    n ← $|G|$
    if n < $2\sqrt{2}$ :
        return min-cut by brute-force in $\mathcal{O}(1)$
    Run Karger's algorithm on G until $\frac{n}{\sqrt{2}}$ supernodes remain
    $G_1$ and $G_2$ be two copies of the resulting multigraph
    $S_1 \leftarrow KargerStein(G_1)$
    $S_2 \leftarrow KargerStein(G_2)$
    return the best of cuts $S_1$ and $S_2$
```

Lemma: *Karger-Stein algorithm runs in $\mathcal{O}(n^2 \log n)$ time.*

Proof: On a multigraph with n supernodes, Karger-Stein algorithm does $\mathcal{O}(n^2)$ amount of work to reduce the number of supernodes by a factor of $\sqrt{2}$ and also $\mathcal{O}(n^2)$ work to determine the best of two cuts returned by recursive calls. It makes 2 recursive calls on multigraphs with $\frac{n}{\sqrt{2}}$ supernodes each. The run time *T(n)* satisfies the following recurrence:

$$T(n) = 2 \cdot T\left(\frac{n}{\sqrt{2}}\right) + \mathcal{O}(n^2)$$

By Master's Theorem, $T(n) = \Theta(n^2 \log(n))$

Using the amplifying strategy, we can repeat Karger-Stein algorithm $\mathcal{O}(\log(n))$ times to obtain the overall failure rate at most some small constant. The overall run time of the amplified version of Karger-Stein algorithm is $\mathcal{O}(n^2 \log^2 n)$ which is better than the amplified version of Karger's algorithm.

3 The Knuth-Morris-Pratt (KMP) Algorithm

3.1 String Matching and Naive Algorithm

Given a text $T[1\cdots n]$ and a pattern $P[1\cdots m]$, both of which are strings over the same alphabet, find all the occurrences of P in T. We can say that P occurs in T with shift s if $P[1\cdots m] = T[s+1,\cdots,s+m]$. A simple algorithm considers all possible shifts.

```
def NaivePatternMatching(T, P)
    for s ← 0 to n-m :
        if P[1,...m] = T[s+1,...,s+m] :
            Print(s)
```

The total number of comparisons done by this algorithm is $(n-m+1)m = \Theta(nm)$. Any algorithm for the string matching problem must examine every symbol in T and P and so requires $\Omega(n+m)$ time.

3.2 The Algorithm

The traditional approach involved "backing up" the input text as we go through it. KMP algorithm uses information gleaned from partial matches of the pattern and text to skip over shifts that are guaranteed not to result in a match without "backing up" by using $\mathcal{O}(m)$ memory.

Consider the following example: $T = babcbabcabcaabcabcabcacabc$ and $P = abcabcacab$. Initially we place the pattern at the extreme left, shift of 0, and prepare to scan the leftmost character of the input text.

P	a	b	c	a	b	c	a	c	a	b																
T	b	a	b	c	b	a	b	c	a	b	c	a	a	b	c	a	b	c	a	b	c	a	c	a	b	c
	↑																									

The arrow indicates the current text character. It points to b which doesn't match a, the pattern is shifted by one space to the right and move to the next input character:

P		a	b	c	a	b	c	a	c	a	b															
T	b	a	b	c	b	a	b	c	a	b	c	a	a	b	c	a	b	c	a	b	c	a	c	a	b	c
		↑																								

Now we have a match, so the pattern stays put while the next several characters are scanned. We again get a mismatch:

P		a	b	c	a	b	c	a	c	a	b															
T	b	a	b	c	b	a	b	c	a	b	c	a	a	b	c	a	b	c	a	b	c	a	c	a	b	c
					↑																					

At this point, we have matched the first three pattern characters but not the fourth. So we know that the last four characters of the input have been $a\ b\ c\ x$ where $x \neq a$. In this case, no matter what x is, as long as it's not a, we deduce that the pattern can be immediately be shifted four more places to the right.

We get another partial match and a failure on the eighth pattern character.

P						a	b	c	a	b	c	a	c	a	b											
T	b	a	b	c	b	a	b	c	a	b	c	a	a	b	c	a	b	c	a	b	c	a	c	a	b	c
													↑													

We know last eight characters were $a\ b\ c\ a\ b\ c\ a\ x$ where $x \neq c$, the pattern should therefore be shifted three places to the right.

P									a	b	c	a	b	c	a	c	a	b								
T	b	a	b	c	b	a	b	c	a	b	c	a	a	b	c	a	b	c	a	b	c	a	c	a	b	c
													↑													

We have another mismatch, so we shift the pattern four places. That produces a match, and we continue scanning until reaching another mismatch on the eighth pattern character.

P													a	b	c	a	b	c	a	c	a	b				
T	b	a	b	c	b	a	b	c	a	b	c	a	a	b	c	a	b	c	a	b	c	a	c	a	b	c
																				↑						

Again the pattern is shifted three places to the right. This time a match is produced, and we discover the full pattern.

P																a	b	c	a	b	c	a	c	a	b	
T	b	a	b	c	b	a	b	c	a	b	c	a	a	b	c	a	b	c	a	b	c	a	c	a	b	c
																									↑	

3.3 Sliding Rule

Let $S = s_1 s_2 \cdots s_k$ be a string. Each string of the form $s_1 s_2 \cdots s_i, 1 \leq i \leq k$ is called a *prefix* of s. Similarly, each string of the form $s_i \cdots s_k, 1 \leq i \leq k$ is called a *suffix* of s.

Suppose that $P[1 \cdots q]$ is matched with the text $T[i-q+1, \cdots, i]$ and a mismatch occurs: $P[q+1] \neq T[i+1]$. Then, slide the pattern right so that the longest possible proper prefix of $P[1, \cdots, q]$ that is also a suffix of $P[1, \cdots, q]$ is now aligned with the text, with the last symbol of this prefix aligned at $T[i]$. If $\pi(q)$ is the number such that $P[1, \cdots, \pi(q)]$ is the longest proper prefix that is also a suffix of $P[1, \cdots, q]$, then the pattern slides so that $P[1, \cdots, \pi(q)]$ is aligned with $T[i - \pi(q) + 1, \cdots, i]$.

P	a	b	c	a	b	c	a	c	a	b
q	1	2	3	4	5	6	7	8	9	10
π	0	1	1	0	1	1	0	5	0	1

```
def KMP(T, P)
    q ← 0
    i ← 0
    while i < n :
        if P[q+1] = T[i+1] :
            q ← q + 1
            i ← i + 1
            if q = m :
                Print(i-q)
                q ← $\pi(q)$
        else
            if q = 0 :
                i ← i+1
            else
                q ← $\pi(q)$
```

3.4 Running Time

Each time through the loop, either we increase i or we slide the pattern right. Both of these events can occur at most n times, and so the repeat loop is executed at most *2n* times. The cost of each iteration of the repeat loop is $\mathcal{O}(1)$. Therefore, the running time is $\mathcal{O}(n)$, assuming that the values $\pi(q)$ are already computed.

3.5 Computing the π table

Once this table is computed, the actual matching takes $\mathcal{O}(n)$, if we can compute π in $\mathcal{O}(m)$, the total runtime of $\mathcal{O}(m+n)$ is achieved, which is linear in the input size.

```
def COMPUTEPI(P, m)
    $\pi$ ← array(1...m)
    j ← 0
    $\pi$[1] ← 0
    for i ← 2 to m :
        while j > 0 and P[i] ≠ P[j+1] :
            j ← $\pi$[j]
        if P[i] = P[j+1] :
            j ← j + 1
        $\pi$[i] ← j
    return $\pi$
```