

$B - A - W$: MergeSort : $O(n \log n)$ HeapSort : $O(n \log n)$ QuickSort : $O(n \log n) - O(n \log n) - O(n^2)$ InsertionSort : $O(n) - O(n^2) - O(n^2)$ BubbleSort : $O(n) - O(n^2) - O(n^2)$

RemoveMin()-replace by last node. Bubble down-swap with smallest child.

RemoveMax() -replace by last node. Bubble down-swap with largest child

Insert() - add at the end of array (rightmost last node)

Unsuccessful search- $O(1 + \alpha)$. Successful - $\Theta(1 + \alpha)$.

Division method: $h(k) = k \bmod d, d = 2^r, r$ prime, not close to power of 2 or 10.

Multiplication method: $h(k) = Ak \bmod 2^w \ll w - r, 2^{w-1} < A < 2^w$.

Open addressing: Try hash function. If slot taken, take new function and retry.

Linear probing: $h(k, i) = (h'(k) + i) \bmod m$. **Quad:** $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$.

Double Hashing: $h(k, i) = (h_1(k) + i h_2(k)) \bmod m$.

Universal Hashing: #functions $h(k) = h(1) \leq H/m, k \in m$ keys.

Max-heap: Max @ top **Min-heap:** Min @ top.

Heap as array: Left[i]=A[2i], Right[i]=A[2i+1], Parent[i]=A[i/2] **Operations:** $O(\log n)$.

MaxHeapify: MaxHeapify(A, i, n)

1. $l \leftarrow \text{leftNode}(i)$
2. $r \leftarrow \text{rightNode}(i)$
3. if $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$
4. then $\text{largest} \leftarrow l$
5. else $\text{largest} \leftarrow i$
6. if $r \leq n$ and $A[r] > A[\text{largest}]$
7. then $\text{largest} \leftarrow r$
8. if $\text{largest} \neq i$
9. then exchange $A[i] \leftrightarrow A[\text{largest}]$
10. MaxHeapify(A, largest)

BuildMaxHeap: MaxHeapify(A, i, length(A)) i from length(A)/2 to 1.

Heapsort(A):

1. Build-Max-Heap(A)
2. for $i \leftarrow \text{length}[A]$ down to 2
3. exchange $A[1] \leftrightarrow A[i]$
4. MaxHeapify(A, 1, i-1)

AVL Tree: BST, $h_{\text{left}} - h_{\text{right}} \leq 1$

Insert at downmost leftmost child (as BST), $O(\log n)$.

Restore AVL property at x:

if x.rightchild is right unbalanced or balanced:

left rotate.

else right rotate then left rotate.

if x.leftchild is left unbalanced or balanced:

right rotate.

else left rotate then right rotate.

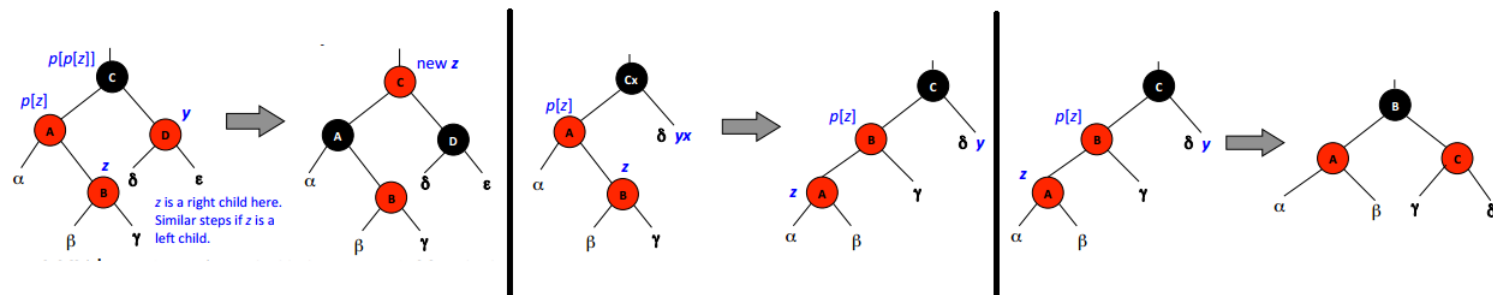
continue with x's ancestors.

In-Order on AVL /BST - \rightarrow increasing keys (sorted)

AVL sort -worst: $O(n \log n)$ if balanced. **BST sort**-best: $O(n \log n)$, worst: $O(n^2)$.

R-B Tree: BST. Root is black. NIL is black. No consecutive red nodes. All black heights are same from node to descendants. If no child-put NIL. **Black-height(x):** # black nodes from x to NIL (count NIL, don't count x).

R-B insert(x): BST insert x as red then restore R-B property.



Sets: In forest representation, root==representative.

Union: By size: smallest into biggest. By height: shortest into tallest.

Path compression: All nodes parent= representative.

Greedy: Local optimal choice, delegate subproblem recursively. For interval, start from beginning and minimize waste of space.

Huffman Trees: Highest frequency letters on top.

Graphs: $|E| = |V| - 1 \rightarrow$ it is a tree. Store weights or booleans in **adjacency matrix**.

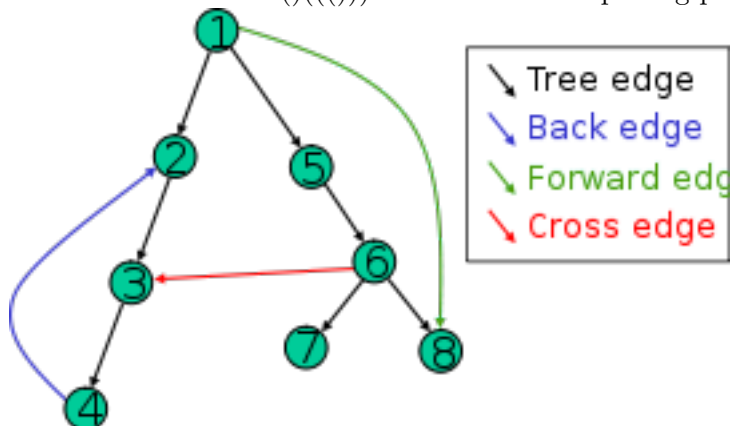
Color Code: **White:** undiscovered . **Grey:** neighbors unvisited . **Black:** done.

$d[u]$ = smallest # of edges from s to any u.

$\pi[u] = v$, v is the predecessor of u.

BFS: $O(V+E)$. **DFS:** $\Theta(V+E)$.

Parenthesis Thm: $()((()))$. Cannot finish exploring parent before child.



DAG: Directed Acyclic Graph (tree is a DAG). **Partial or-**

der: $a > b, b > c \rightarrow a > c$ but maybe $a=b$. **Total order:** Always $a > b$ or $a < b$.

Topological Sort: On DAG, no back edges. $\Theta(V + E)$. **Strongly Connected Component** if we can reach u from v, for any u and v in a subset of G. G^T has edges flipped (forward=backward).

Compute SCC: DFS(G), then G^T , then DFS(G^T) (in order of decreasing $f[u]$, as computed by DFS(G)).

$d[u]$: start time. $f[u]$: finish time.

MST: Connect all edges. Cut respects A - no edge in A crosses it. **Safe edge:** smallest weight edge between A and V-A.

Kruskal:

1. Starts with each vertex in its own component (1 partition / vertex).
2. Repeatedly merges two components into one by choosing a light edge that connects them (i.e., a light edge crossing the cut between them).
3. Scans the set of edges in monotonically increasing order by weight.
4. Uses a disjoint-set data structure to determine whether an edge connects vertices in different components.

Union by rank + path compression : total runtime = $O(E \log V)$

Prim's Algorithm:

1. Builds one tree, so A is always a tree.
2. Starts from an arbitrary root r .
3. At each step, adds a light edge crossing cut $(V_A, V - V_A)$ to A. Where V_A = vertices that A is incident on.

Find a light edge (for Prim):

1. Uses a priority queue Q to find a light edge quickly.
2. Each object in Q is a vertex in $V - V_A$.
3. Key of v is minimum weight of any edge (u, v) , where $u \in V_A$.
4. Then the vertex returned by Extract-Min is v such that there exists u $\in V_A$ and (u, v) is light edge crossing $(V_A, V - V_A)$.
5. Key of v is $w(u, v)$ if v is not adjacent to any vertex in V_A .

Prim - add lightest edges to neighbors until all reachable. Kruskal - connect partitions with safe edges until partition= whole graph. **Shortest paths:** $d[v]$ - **shortest path estimate.** (initially = ∞). $\pi[v]$ - predecessor of v (initially = NIL or if none).

Relax an edge (if exists shorter path):

if $(d[v] > d[u] + w(\text{edge between } u \text{ and } v))$ then $d[v] = d[u] + w(\text{edge between } u \text{ and } v)$, $\pi[v] = u$.

Dijkstra:

```
DIJKSTRA( V, E, w, s)
INIT-SINGLE-SOURCE( V, s)
S = {}
Q = V
while Q non empty do
  u = EXTRACT-MIN( Q)
  S = S U {u}
  for each vertex v in Adj[ u] do
    RELAX( u, v, w)
```

If we use binary heap, runtime = $O(E \log V)$

Gale-Shapley:

```
matching = empty
while there is a in A not yet matched do
    B = pref[a].removeFirst()
    if B not yet matched then
        matching = matching U {(a,B)}
    else
        G = B current match
        if B prefers a over G then
            matching = matching - {(G,B)} U {(a,B)}
return matching
```

Residual graph:

```
for each edge e=(u,v) in E
    if f(e) < c(e)
    then {
        put a forward edge (u,v) in Residual graph
        with residual capacity cf(e)=c(e)-f(e)
    }
    if f(e)>0
    then {
        put a backward edge (v,u) in Residual graph
        with residual capacity cf(e)=f(e)
    }
}
```

Augmenting path: From source to sink via residual graph. Follow lines and add capacity as you go
Final solution: add $G +$ residual graph paths . (take difference of edges flow if backwards + forward)

Ford-Fulkerson:

```
f =0
Residual=G
while (there is a source to sink path in Residual){
    f.augment(Path)
    update Residual based on new f
}
```

Flow through a cut: $|f| = \sum f(e) \text{ from A to B} - \sum f(e) \text{ from B to A}.$

Minimum cut:

1. Run Ford-Fulkerson for max flow.
2. Run BFS /DFS from s on Residual.
3. All reachable vertices define the cut.

$$\sum_{k=0}^{n-1} ar^k = a \frac{1-r^n}{1-r}.$$

$$BST \text{ amount of nodes} = 2^{h+1} - 1$$

$$RBT \ h \leq 2 \log_2(n+1)$$