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

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

Unsuccessfull search- $O(1+\alpha)$. Successfull - $\Theta(1+\alpha)$.

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

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

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

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

Double Hashing: $h(k,i) = (h_1(k)+ih_2(k)) modm$.

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

Max-heap: Max @ top **Min-heap:** Mean @ 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 $\leftarrow$ leftNode(i)
2. r $\leftarrow$ rightNode(i)
3. if l heap-size[A] and A[l]¿A[i]
4. then largest $\leftarrow$ l
5. else largest $\leftarrow$ i
6. if r $\leq$ n and A[r] $>$ A[largest]
7. then largest $\leftarrow$ r
8. if largest $\neq$ i
9. then exchange A[i] $\leftrightarrow$ A[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 length[A] down to 2
3. exchange A[1] A[i]
4. MaxHeapify(A, 1, i-1)

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

Insert at downmost leftmost child (as BST), $O(logn)$.

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.

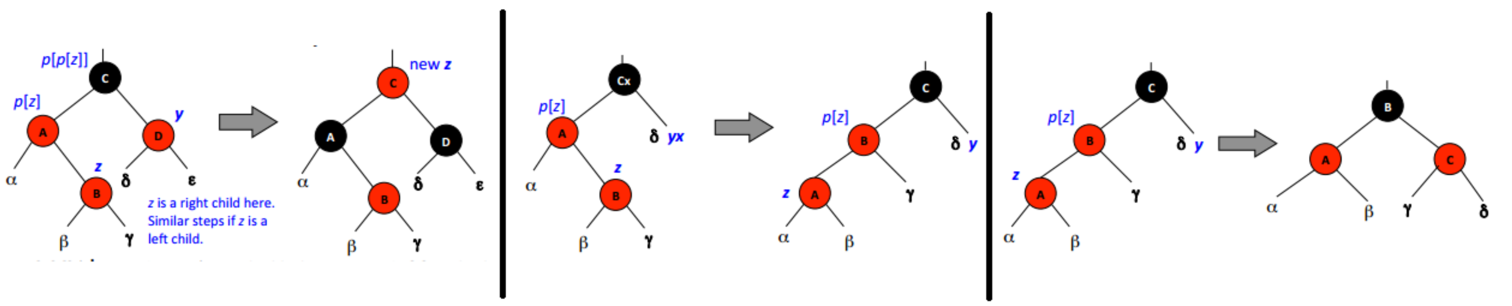
continue with x's ancestors.

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

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

R-B Tree: Root is black. NIL is black. Alternate parent-child colors.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, dont 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.

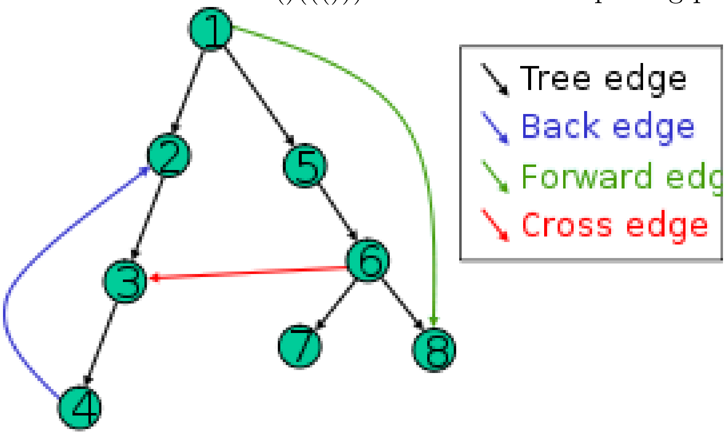
Greedy: Local optimal choice, delegate subproblem recursively. For interval, start from beginning and minimize waste of space.

Huffman Trees: Highest frequency letters on top.

Graphs: $|E| = |V| - 1->$ 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.
π[u]=v, v is the predecessor of u.
BFS: O(V+E). **DFS:** Θ(V+E).
Parenthesis Thm: ()((())). Cannot finish exploring parent before child.



DAG: Directed Acyclic Graph (tree is a DAG).**Partial order:** $a > b, b > c \rightarrow a > c$ but maybe a=b. **Total order:** Always $a > b$ or $a < b$.
Topological Sort: On DAG, no back edges. $\Theta(V + E)$. **Strongly Connectd 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 deacreasing 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 - VA.
3. Key of v is minimum weight of any edge (u, v), where u V_A.
4. Then the vertex returned by Extract-Min is v such that there exists u V and (u, v) is light edge crossing $(V_A, V - V_A)$.
5. Key of v is if v is not adjacent to any vertex in VA.

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(edge between u and v)) then d[v]=d[u]+w(edge between u and v), π[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      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: $|f| = \sum f(e) from A to B - \sum f(e) from B to A$.
Minimum cut:
1. Run Ford-Fulkerson for max flow.
2. Run BFS /DFS from s.
3. All reachable vertices define the cut.