02105 Algorithms and Data Structures 1

Jacopo Ceccuti (s215158)

Github: 02105 Algorithms and Data Structures 1 Algorithm Collection

Contents

Algorithms

Peaks

A[i] is a peak if A[i-1]≤A[i] and [i+1]≤A[i].

Algorithm 1

For each element, check if it is a peak and return the first peak element
Pseudocode:

```
Peak1(A, n)
        if A[0] ≥ A[1] return 0
        for i = 1 to n−2
                if A[i−1] ≤ A[i] ≥ A[i+1] return i
        if A[n−2] ≤ A[n−1] return n−1
```

Running time: O(n)

Algorithm 2

The maximum element in the list must be a peak
Pseudocode:

```
FindMax(A, n)
        max = 0
        for i = 0 to n−1
                if A[i] > A[max] max = i
        return max
```

Running time: O(n)

Algorithm 3

If an element has a neighbor that is higher than itself, then there must be a peak in the direction of that neighbor
Pseudocode:

```
Peak3(A, i, j)
        m = ⌊(i+j)/2⌋

        if A[m−1] ≤ A[m] ≥ A[m+1] return n
        elseif A[m−1] > A[m]
                return Peak(A, i, m−1)
        elseif A[m] < A[m+1]
                return Peak3(A, m+1, j)
```

Running time: O(log n)

Searching

Given a sorted list A and a number x, find whether x occurs in A.

Linear search

Go linearly through a list and see if x occurs
Pseudocode:

```
LinearSearch(A, x, n)
        for i = 0 to n
                if A[i] = x return true
        return false
```

Running time: O(n)

Binary search

If the middle element of a slice of A is lower than x, then x must be in the right half, otherwise, if the middle element is higher than x, then x must be in the left half
Pseudocode:

```
BinarySearch(A, i, j, x)
        if j < i return false

        m = ⌊(i+j)/2⌋

        if A[m] = x return true
        else if A[m] < x return BinarySearch(A, m+1, j, x)
        else return BinarySearch(A, i, m−1, x)
```

Running time: O(log n)

Sorting

Givet en list A, returner listen B med samme værdier i sorteret orden

Insertion Sort

Going from left to right, for each new item, insert it into the list so it's still sorted
Pseudocode:

```
InsertionSort(A, n)
        for i = 1 to n−1
                j=i
                while j > 0 and A[j−1] > A[j]
                        swap A[j] and A[j−1]
                        j = j − 1
```

Running time: $O(n^2)$

Merge Sort

Split A into two halves, sort these two recursively, combine the two subsets (divide and conquer)
Pseudocode:

```
MergeSort(A, i, j)
        if i < j
                m = ⌊(i+j)/2⌋
                MergeSort(A, i, m)
                MergeSort(A, m+1, j)
                Merge(A, i, m, j)
```

Running time: $O(n \log n)$

Graph searching

An algorithm designed to visit all vertices in a graph
Concepts:

- Discovery time: The first time an edge is visited
- Finish time: Last time an edge was visited

Depth-First Search

Unmark all vertices, visit all the neighbors of an edge that is not marked and mark these, now visit their neighbors recursively
Pseudocode:

```
DFS(s)
        time = 0
        DFS-visit(s)

DFS-visit(v)
        v.d = time++
        mark v
        for each unmarked neighbor u
                u.parent = v
                DFS-visit(u)
        v.f = time++
```

Running time: O(n + m)

Breadth-First Search

Unmark all vertices and make a queue Q. Select start edge and enqueue start edge s. While Q is not empty dequeue an element, find its neighbors, mark them and enqueue them. BFS always finds Shortest Paths from s
Pseudocode:

```
BFS(s)
        mark s
        s.d = 0
        Q.Enqueue(s)
        while not Q.IsEmpty()
                v = Q.Dequeue()
                for each unmarked neighbor u
                        mark u
                        u.d = v.d + 1
                        u.parrent = v
                        Q.Enqueue(u)
```

Running time: O(n + m)

Connected Components

A Connected Component is the maximum number of connected vertices
Pseudocode:

```
Connected(G)
        while G has unmarked node u
                BFS(u)
```

Running time: O(n + m)

Bipartite Graphs

A Bipartite Graph is a graph in which all vertices can be colored either red or blue and where the edges in the graph are connected to a red and a blue corner
Pseudocode:

```
Bipartite(G, s)
        mark s
        s.d = 0
        Q.Enqueue(s)
        while Q.IsEmpty()
                v = Q.Dequeue()
                for each unmarked neighbor u
                        mark u
                        u.d = v.d + 1
                        u.parrent = v
                        Q.Enqueue(u)

                for each neighbor u
                        if (u.d mod 2) = (v.d mod 2)
                                return false
        return true
```

Running time: O(n + m)

Topological sorting

Determine if a graph is topologically sorted

Algorithm 1

Create an inverse graph G^R. Find an edge v with in-degree 0. Remove v and all edges connected out of it. Place v at the leftmost point in the new graph. Recourse
Pseudocode:

```
TopSort1(G)
        reverse G
        while G has verticies left
                for each vertex in G with in-degree 0 v
                        remove v from G

                if G does not contian vertex with in-degree 0
                        return false
        return true
```

Running time: O(n^2)

Algorithm 2

Create an inverse graph G^R. Create a list with in-degree of all vertices and a linked list of vertices with in-degree 0. Remove the first element from the linked list and draw one from all vertices v points to. Check if any of the vertices have in-degree 0 and add them to the linked list. Recourse
Pseudocode:

```
TopSort2(G)
        reverse G

        for each vertex in G v
                if deg⁻(v) = 0
                        L.Insert(v)
                d[v] = deg⁻(v)

        for each vertex in L v
                L.Delete(v)
                for each vertex pointed to by v u
                        d[u]--
                        if d[u] = 0
                                L.Insert(u)
        return is G empty
```

Running time: O(n+m)

Directed Acyclic Graph

Check if G is a Directed Acyclic Graph (DAG), by checking if it can be topologically sorted
Pseudocode:

```
DAG(G)
        return TopSort(G)
```

Running time: O(TopSort)

Heap Building

Given a list of integers H[1..n], convert the list to a heap

Algorithm 1

For all nodes in ascending levels, use BubbleUp
Pseaudocode:

```
Build1(H)
        for each node in H increasing v
                BubbleUp(v)
```

Running Time: O(n log n)

Algorithm 2

For all nodes in descending levels, use BubbleDown
Pseaudocode:

```
Build2(H)
        for each node in H decreasing v
                BubbleDown(v)
```

Running Time: O(n)

Heap Sorting

How can we sort a given list H[1..n], using a heap?
Pseudocode:

```
HeapSort(H)
        BuildHeap(H)
        for each node in H v
                Hr.Insert(v)
        return Hr
```

Running Time: O(n log n)

Minimum Spanning Tree

A Minimum Spanning Tree is a weighted tree that occurs in a weighted graph where the sum of all edges is the smallest possible

Prim's Algorithm

At a given time, add to the list the lightest neighbor of one of the vertices already in the list
Pseudocode:

```
Prim(G, s)
        for all verticies v in G
                v.key = ∞
                v.parent = null
                P.Insert(v)

        P.DecreaseKey(s, 0)
        while P is not empty
                u = P.ExtractMin()
                for all neightbors v of u
                        if P contains v and G.Adjacent(v, u) < v.key
                                P.DecreaseKey(v, G.Adjacent(v, u))
                                v.key = G.Adjacent(v, u)
                                v.parent = u
```

Running Time: O(m log n)

Kruskal's Algorithm

Sort all edges from lightest to heaviest and add them to the list if it doesn't create a cycle
Pseudocode:

```
Kurskal(G)
        sort edges
        U.Init(n)
        for all edges (u, v) in G
                if not U.Connected(u, v)
                        Insert(u, v)

        return U
```

Running time: O(m log n)

Shortest Path

A Shortest Path is the shortest path between two points in a weighted graph

Dijkstra's Algorithm

Given a directed weighted graph, with positive weights and a start vertex s, find the shortest path. All vertices have a distance estimate, v.d, which is the length of the shortest path to that vertex
Pseudocode:

```
Dijkstra(G, s)
        for all vertices v in G
                v.d = ∞
                v.parent = null
                P.Insert(v)

        P.DecreaseKey(v)
        while P not empty
                u = ExtractMin(P)
                for all v that u points to
                        Relax(G, P, u, v)

Relax(G, P, u, v)
        if (v.d > u.d + G.Adjacent(v, u))
                v.d = u.d + G.Adjacent(v, u)
                P.DecreaseKey(v, v.d)
                v.parent = u
```

Running Time: O(m log n)

Shortest Path on DAG

Given a directed acyclic graph, find the shortest path from a start vertex to all other vertices
Pseudocode:

```
Dijkstra(G, s)
        for all vertices v in G
                v.d = ∞
                v.parent = null

        for all vertices u in G topologically sorted
                for all v that u points to
                        Relax(G, P, u, v)

Relax(G, P, u, v)
        if (v.d > u.d + G.Adjacent(v, u))
                v.d = u.d + G.Adjacent(v, u)
                v.parent = u
```

Running Time: O(m + n)

Tree Algorithms

Size

Check the size of the left and right nodes and add them together with 1, it gives the size of a node
Pseudocode:

```
Size(v)
        if v = null
                return 0
        return size(v.left) + Size(v.right) + 1
```

Running time: O(size(v))

Tree Traversals

Visit all nodes in a specific order
Sequence:

- Inorder: Visit left subtree, visit node, visit right subtree
- Preoder: Visit node, visit left subtree, visit right subtree
- Inorder: Visit left subtree, visit right subtree, visit node

Pseudocode:

```
Inorder(v)
        if v = null
                return
        Inorder(v.left)
        v.Visit()
        Inorder(v.right)

Preorder(v)
        if v = null
                return
        v.Visit()
        Preorder(v.left)
        Preorder(v.right)

Postorder(v)
        if v = null
                return
        Postorder(v.left)
        Postorder(v.right)
        v.Visit()
```

Running time: O(n)

Datastructures

Stack

A dynamic sequence of elements S
Operations:

- `PUSH(x)`: Add x to S
- `POP(x)`: Remove and return the most recently added item to S
- `ISEMPTY()`: Return true if S is empty

Pseudocode:

```
Push(S, x, top)
        S[top+1] = x
        top = top + 1

Pop(S, top)
        elm = S[top]
        top = top - 1
        return elm

IsEmpty(top)
        if top = -1 return true
        else return false
```

Running time:

- `PUSH(x)`: O(1)
- `POP(x)`: O(1)
- `ISEMPTY()`: O(1)

Space: O(N)

Queue

A dynamic sequence of elements Q
Operations:

- `ENQUEUE(x)`: Add x to Q
- `DEQUEUE(x)`: Remove and return the first element added to Q
- `ISEMPTY()`: Return true if Q is empty

Pseudocode:

```
Enqueue(Q, x, tail)
        Q[tail] = x
        tail = tail + 1

Pop(S, head)
        elm = Q[head]
        head = head - 1
        return elm

IsEmpty(head, tail)
        if head = tail return true
        else return false
```

Running time:

- `ENQUEUE(x)`: O(1)
- `DEQUEUE(x)`: O(1)
- `ISEMPTY()`: O(1)

Space: O(N)

Linked List

Several nodes, with data called keys, which are connected to each other by pointers. Can be single or double linked

Operations:

- `SEARCH(head, x)`: Return node with key x
- `INSERT(x)`: Insert node x at the front of the list. Return new head
- `DELETE()`: Delete node x in the list. Return new head

Pseudocode:

```
Search(head, key)
        x = head
        while x not null
                if x.key = key return x
                x = x.next
        return null

Insert(head, x)
        x.prev = null
        x.next = head
        head.prev = x
        return x

Delete(head, x)
        if x.prev ≠ null
                x.prev.next = x.next
        else head = x.next
        if x.next ≠ null
                x.next.prev = x.prev
        return head
```

Running time:

- `SEARCH(head, x)`: O(n)
- `INSERT(x)`: O(1)
- `DELETE()`: O(1)

Space: O(N)

Graphs

Terminologies:

- Undirected graph: G=(V,E), where V are vertices, E are edges. n is the number of vertices, m is the number of edges
- Path: A sequence of vertices connected by edges
- Cycle: A path that starts and ends on the same edge
- Degree: deg(v) is the number of edges connected to v
- Connectivity: Two edges are connected if a path exists between them

Lemma:

- $\sum_{v \in V}$deg(v)=2m

operations:

- `ADJACENT(v, u)`: Find out if v and u are neighbors
- `NEIGHBORS(v)`: Return all neighbors of v
- `INSERT(v, u)`: Add (v, u) to G unless it is already there

Shortest Path: A Shortest Path is a path that passes by the fewest possible edges

Adjacency Matrix

An Adjacency Matrix is a matrix of n×n elements A, where each element in the list represents whether two vertices are neighbors, with a 1 or if they are not, with a 0
Pseudocode:

```
Adjacent(A, v, u)
        return A[v][u] = 1

Insert(A, v, u)
        A[v][u] = 1

Neighbors(A, v)
        return A[v]
```

Running time:

- `ADJACENT(v, u)`: O(1)
- `INSERT(v, u)`: O(1)
- `NEIGHBORS(v)`: O(n)

Space: O(n^2)

Adjacency List

An Adjacency List is a static list A, of size n, of dynamic lists, of size deg(v). Each dynamic list contains all the neighbors of the edge corresponding to the static list's index

Pseudocode:

```
Adjacent(A, v, u)
        return A[v].Search(u)

Insert(A, v, u)
        A[v].Insert(u)

Neighbors(A, v)
        return A[v]
```

Running time:

- `ADJACENT(v, u)`: O(deg(v))
- `INSERT(v, u)`: O(deg(v))
- `NEIGHBORS(v)`: O(deg(v))

Space: O(n + m)

Directed Graphs

Graphs but with edges that go in a certain direction
Terminologies:

- Path: A sequence of vertices connected by edges
- Shortest path: Path between two vertices in a graph such that the sum of the weights of its constituent edges is minimized
- Directed acyclic graph: A graph without cycles
- Topological sorting: An arrangement of vertices where all edges point in the same direction
- Strongly connected component: A subset of vertices, where given vertex v has a path to u and u has a path to v
- Transitive closure: A graph has transitive closure if, for a given point v, there is a direct path to all other vertices it is strongly connected to

Lemma:

- $\sum_{v \in V} \deg^{-}(v) = \sum_{v \in V} \deg^{+}(v) = m$

Operations:

- `POINTSTO(v, u)`: Find out if v points to u
- `NEIGHBORS(v)`: Return all vertices v points to
- `INSERT(v, u)`:Add the edge (v, u) to G unless it is already there

Adjacency Matrix

An Adjacency Matrix is a matrix of n×n elements A, where each element in the list represents whether the corner in the row points to the corner in the column, with a 1 or if it does not, with a 0
Pseudocode:

```
PointsTo(A, v, u)
        return A[v][u] = 1

Insert(A, v, u)
        A[v][u] = 1

Neighbors(A, v)
        return A[v]
```

Running time:

- `POINTSTO(v, u)`: O(1)
- `INSERT(v, u)`: O(1)
- `NEIGHBORS(v)`: O(n)

Space: $O(n^2)$

Adjacency List

An Adjacency List is a static list A, of size n, of dynamic lists, of size deg(v). Each dynamic list contains all the neighbors of the edge corresponding to the static list's index
Pseudocode:

```
PointsTo(A, v, u)
        return A[v].Search(u)

Insert(A, v, u)
        A[v].Insert(u)

Neighbors(A, v)
        return A[v]
```

Running time:

- `POINTSTO(v, u)`: O(deg(v))
- `INSERT(v, u)`: O(deg(v))
- `NEIGHBORS(v)`: O(deg(v))

Space: O(n + m)

Priority Queue

A dynamic set S, where each element has a key, x.key, and satellite data, x.data
Operations:

- `MAX()`: Return the element with the largest key
- `EXSTRACTMAX()`: Return and remove the element with the largest key
- `INCREASEKEY(x, k)`: Sets x.key = k
- `INSERT(x)`: Inserts x into S

Linked List

Make a doubly linked list S
Pseudocode:

```
Max(S)
        for each element in S s
                if s.key > largest.key
                        largest = s
        return s

ExtractMax(S)
        max = Max(S)
        S.Delete(max)
        return s

IncreaseKey(x, k)
        x.key = k

Insert(S, x)
        S.Insert(x)
```

Running time:

- `MAX()`: O(n)
- `EXSTRACTMAX()`: O(n)
- `INCREASEKEY(x, k)`: O(1)
- `INSERT(x)`: O(1)

Space: O(n)

Sorted Linked List

Create a sorted doubly linked list S
Pseudocode:

```
Max(S)
        return S[0]

ExtractMax(S)
        max = S[0]
        S.Delete(max)
        return s

IncreaseKey(S, x, k)
        x.key = k
        move x linearly to correct new position

Insert(S, x)
        S.Insert(x)
        move x linearly to correct new position
```

Running time:

- `MAX()`: O(1)
- `EXSTRACTMAX()`: O(1)
- `INCREASEKEY(x, k)`: O(n)
- `INSERT(x)`: O(n)

Space: O(n)

Heap

We can use a heap to create a priority queue

```
Max()
        return H[1]

ExtractMax()
        r = H[1]
        H[1] = H[n]
        n = n - 1
        H.BubbleDown(1)
        return r

IncreaseKey(x, k)
        H[x] = k
        H.BubbleUp(x)

Insert(x)
        n = n + 1
        H[x] = k
        H.BubbleUp(x)
```

Running time:

- `MAX()`: O(1)
- `EXSTRACTMAX()`: O(log n)
- `INCREASEKEY(x, k)`: O(log n)
- `INSERT(x)`: O(log n)

Space: O(n)

Tree

A tree is a graph with a root node, connected to a series of children. The graph is always acyclic
Terminologies:

- Children: The nodes a parent node is connected to
- Parent: The node a child node is connected to
- Descendant: The nodes that are connected to a parent node through several layers
- Ancestor: The nodes that are the parent of the node, the parent of its node, etc.
- Leaf: The nodes that have no children
- Depth: The length of the path from a node to the root or from the root to the farthest leaf
- Height: The length of the path from a node to its leaf

Binary Tree: A tree with a maximum of two children, called left and right children
Complete Binary Tree: A binary tree where all levels of the tree are filled in
Almost Complete Binary Tree: A complete binary tree with 0 or more missing right children
Lemma: The height of an almost complete binary tree is O(log n)

Heap

An almost complete binary tree where all nodes store an element. Heap order must be satisfied, for all nodes v, both children's keys must be less than v.key

Operations:

- `PARENT(x)`: Return x's parents
- `LEFT(x)`: Return x's left child
- `RIGHT(x)`: Return x's right child
- `BUBBLEUP(x)`: Move node x up if it goes against heap order
- `BUBBLEDOWN(x)`: Move node x down if it goes against heap order

Pseudocode:

```
BubbleUp(x)
        if x.Parent().key < x.key
                swap x.Parent() and x
                BubbleUp(x.Parent())

BubbleDown(x)
        if x.Left().key > x.key
                if x.Right().key > x.key and x.Left().key < x.Right().key
                        swap x.Right() and x
                        BubbleDown(x.Right())
                else
                        swap x.Left() and x
                        BubbleDown(x.Left())
        else if x.Right().key > x.key
                swap x.Right() and x
                BubbleDown(x.Right())
```

Running Time:

- `BUBBLEUP(x)`: O(log n)
- `BUBBLEDOWN(x)`: O(log n)

Linked List

Each node in the list contains key, parent, left and right
Pseudocode:

```
Parent(x)
        return x.parent

Left(x)
        return x.left

Right(x)
        return x.right
```

Running Time:

- `PARENT(x)`: O(1)
- `LEFT(x)`: O(1)
- `RIGHT(x)`: O(1)

Space: O(n)

Array List

A list H[0..n], where H[0] is not used and H[1..n] are the nodes
Pseudocode:

```
Parent(x)
        return ⌊x/2⌋

Left(x)
        return 2x

Right(x)
        return 2x+1
```

Running Time:

- `PARENT(x)`: O(1)
- `LEFT(x)`: O(1)
- `RIGHT(x)`: O(1)

Space: O(n)

Union

A union is a dynamic family of sets
Operations:

- `INIT(n)`: Construct set 0, 1, ..., n-1
- `UNION(i, j)`: Joins two sets containing i and j, if they are not already joined
- `FIND(i)`: Return a representative of the set containing i

Quick Find

Make a list id[0..n-1] where id[i] is the representative of i
Pseudocode:

```
Init(n)
        for k = 0 to n − 1
                id[k] = k

Find(i)
        return id[i]

Union(i, j)
        iID = Find(i)
        jID = Find(j)

        if iID≠jID
                for k = 0 to n − 1
                        if id[k] = iID
                                id[k] = jID
```

Running Time:

- `INIT(n)`: O(n)
- `UNION(n)`: O(n)
- `FIND(n)`: O(1)

Quick Union

Let each set be a tree. Each tree is represented by the parent node of each node, where a root is p[root] = root

Pseudocode:

```
Init(n)
        for k = 0 to n − 1
                p[k] = k

Find(i)
        while i≠p[i]
                i = p[i]
        return i

Union(i, j)
        ri = Find(i)
        rj = Find(j)
        if ri≠rj
                p[ri] = rj
```

Running Time:

- `INIT(n)`: O(n)
- `UNION(n)`: O(d)
- `FIND(n)`: O(d)

Weighted Quick Union

Let each set be a tree. Each tree is represented by the parent node of each node, where a root is p[root] = root. Also make a list sz[0..n-1], where sz[i] is the size of the subtree that has i as its root
Pseudocode:

```
Init(n)
        for k = 0 to n − 1
                p[k] = k
                sz[k] = 1

Find(i)
        while i≠p[i]
                i = p[i]
        return i

Union(i, j)
        ri = Find(i)
        rj = Find(j)
        if ri≠rj
                if sz[ri] < sz[rj]
                        p[ri] = rj
                        sz[rj] = sz[ri] + sz[rj]
                else
                        p[rj] = ri
                        sz[ri] = sz[ri] + sz[rj]
```

Running Time:

- `INIT(n)`: O(n)
- `UNION(n)`: O(log n)
- `FIND(n)`: O(log n)

Path Compression

Compress a Weighted Quick Union and Quick Union on Find, make all nodes on the path to the root, child to the root
Pseudocode:

```
Find(i)
        oi = i
        while i≠p[i]
                i = p[i]

        while oi≠p[oi]
                tmp = oi
                oi = p[oi]
                p[tmp] = i

        return i
```

Running Time: $O(n + m\ \alpha(m, n))$

Dynamic Connectivity

A dynamically connected graph is a graph in which we can easily find whether two vertices are connected.
Operations:

- `INIT(n)`: Creates a graph with n vertices and no edges
- `CONNECTED(u, v)`: Return whether u and v are connected
- `INSERT(u, v)`: Adds an edge between u and v if it does not already exist

Union

If we use a Union Find where every node in the dynamically connected graph is a node in the Union Find graph
Pseudocode:

```
Init(n)
        U.Init(n)

Connected(u, v)
        return U.Find(u) = U.Find(v)

Insert(u, v)
        U.Union(u, v)
```

Running Time:

- `INIT(n)`: O(n)
- `CONNECTED(u, v)`: O(log n)
- `INSERT(u, v)`: O(log n)

Weighted Graphs

A weighted graph is a graph where the edges have a weight
Terminologies:

- Cut: A cut is a division of vertices into two groups, by removing edges

Operations: The operations on a weighted graph are the same as on a normal graph
Lemmas:

- Cut property: For any cut in a weighted graph, the edge with the smallest weight will be part of the MST
- Cycle property: For every cycle in a graph, the heaviest edge will not be part of the MST

Adjacency Matrix

An Adjacency Matrix is a matrix of $n \times n$ elements A, where each element in the list represents whether two vertices are neighbors, with the weight of the edge, or if they are not, with a 0
Space: $O(n^2)$

Adjacency List

An Adjacency List is a static list A, with size n, of dynamic lists, of size deg(v). Each dynamic list contains all the neighbors of the edge corresponding to the index of the static list and a value corresponding to the weighting of the edge between the two nodes
Space: $O(n + m)$

Directed Weighted Graphs

A directed weighted graph is a weighted graph where the edges are also directed

Nearest Neighbor

Create a dynamic set S where each element has key, x.key, and data, x.data
Operations:

- `Predecessor(k)`: Return the element with largest key≤k
- `Successor(k)`: Return the element with middle key≤k
- `Insert(x)`: Adds x to S
- `Delete(x)`: Removes x from S

Linked List

Let S be a Doubly Linked List
Pseudocode:

```
Predecessor(k)
        large = null
        for all elements in S n
                if n.key ≤ k and n.key > large.key
                        large = n
        return large

Successor(k)
        small = null
        for all elements in S n
                if n.key ≥ k and n.key < large.key
                        small = n
        return small

Insert(x)
        S.Insert(x)

Delete(x)
        S.Delete(x)
```

Running Time:

- `Predecessor(k)`: O(n)
- `Successor(k)`: O(n)
- `Insert(x)`: O(1)
- `Delete(x)`: O(1)

Space: O(n)

Sorted Linked List

Let S be a Sorted Doubly Linked List
Pseudocode:

```
Predecessor(k)
        return BinarySearch(S, k)

Successor(k)
        return BinarySearch(S, k)

Insert(x)
        S.Insert(x)

Delete(x)
        S.Delete(x)
```

Running Time:

- `Predecessor(k)`: O(log n)
- `Successor(k)`: O(log n)
- `Insert(x)`: O(n)
- `Delete(x)`: O(n)

Space: O(n)

Binary Search Tree

A Binary Search Tree is a tree which satisfies that for a given node v, has a left and right node whose key is respectively less than and greater than v

Pseudocode:

```
Predecessor(v, k)
        if v = null
                return null
        if v.key = k
                return v
        if k < v.key
                return Predecessor(v.left , k)
        t = Predecessor(v.right , k)
        if t ≠ null
                return t
        else
                return v

Successor(v, k)
        if v = null
                return null
        if v.key = k
                return v
        if k > v.key
                return Successor(v.right , k)
        t = Successor(v.left , k)
        if t ≠ null
                return t
        else
                return v

Insert(x, v)
        if v = null return x
        if x.key ≤ v.key
                v.left = Insert(x, v.left)
        if x.key > v.key
                v.right = Insert(x, v.right)

Delete(x)
        if x.left = null and x.right = null
                x.parent.Child(x) = null
        else if x.left = null and x.right not null
                x.parent.Child(x) = x.right
                x.right.parent = x.parent
        else if x.right = null and x.left not null
                x.parent.Child(x) = x.left
                x.left.parent = x.parent
        else
                x.key = Successor(x.right).key
                Delete(Successor(x.right))
```

Running Time:

- `Predecessor(k)`: O(h)
- `Successor(k)`: O(h)
- `Insert(x)`: O(h)
- `Delete(x)`: O(h)

Space: O(n)