Set ADT

Definitions, terminologies, notations, operations

- **Set**: A collection of (distinct) objects.
- **Sets in Computer Science**: A collection of (distinct) objects of same type along with a linear order; e.g., a set of integers, a set of characters, a set of strings, a family of sets.
  - Linear order denoted by $<$
  - Properties
    - For any two elements x, y of the set either $x < y$ or $x = y$ or $y < x$. [Every pair is comparable.]
    - For every x, y, z in the set, if $x < y$ and $y < z$ then $x < z$. [Transitivity.]
- **Operations**:

	Operations
Set	Minimum, Maximum, Sort, Split

Binary Operations	Element	Set
Element	Compare	Membership (is x a member of S?), Rank, Insert (x as an element of S), Delete (x from the set S), Find (the set in which x is present)
Set		Union, Intersection, Difference, Compare, Merge (union of disjoint, sorted sets)

Bit-vector representation/implementation of sets

- **Direct Address Table**: Assuming that all the sets (in the domain of discourse) are subsets of the universal set $[N] = \{1, 2, \ldots, N\}$. **Space used for a set of size n is $O(N)$.**

- **Complexity of Operations**:
  - Membership: Does $x \in S$? — 'Yes', if $S[x] = 1$ and 'No', otherwise. — $O(1)$
  - Insertion: Insert(x, S) — Assign $S[x] \leftarrow 1$ — $O(1)$
  - Deletion: Delete(x, S) — Assign $S[x] \leftarrow 0$ — $O(1)$
  - Find: Find$(x, \{S_1, S_2, \ldots, S_k\})$ — $O(k)$
    ```
    for i = 1 to k
        if S_i[x] = 1 then Return(S_i)
    ```
  - Union: Union(A, B, C) — $O(N)$
    ```
    Initialize C[ ] ← 0
    for x = 1 to N
        if (A[x] = 1 ∨ B[x] = 1) then C[x] ← 1
    ```
  - Intersection: Intersection(A, B, C) — $O(N)$
    ```
    Initialize C[ ] ← 0
    for x = 1 to N
        if (A[x] = 1 ∧ B[x] = 1) then C[x] ← 1
    ```
  - Difference: Difference(A, B, C) — $O(N)$
    ```
    Initialize C[ ] ← 0
    for x = 1 to N
        if (A[x] = 1 ∧ B[x] = 0) then C[x] ← 1
    ```

Array representation/implementation of sets

- **Array**: To represent unsorted set. Space used for a set of size n is $O(n)$.

- **Complexity of Operations**:
  - Membership: Does $x \in S$?
    for $i = 1$ to $|S|$
    if $S[i] = x$ then Found $O(n)$
  - Insertion: Insert(x, S)
    Assign $S[|S|+1] \leftarrow 1$, $|S| \leftarrow |S|+1$ $O(1)$
  - Deletion: Delete(x, S)
    for $i = 1$ to $|S|$
    if $S[i] = x$ then $S[i] = S[|S|]$, $|S| \leftarrow |S|-1$ $O(n)$
  - Find: Find$(x, \{S_1, S_2, \dots, S_k\})$
    for $i = 1$ to k
    if Member(x, S_i)=True then Return(S_i) $O(n^k)$
  - Union: Union(A, B, C)
    for $i = 1$ to $|A|$
    $C[i] \leftarrow A[i]$
    for $j=1$ to $|B|$
    if Member$(B[j], A)$ = False then $i \leftarrow i+1$,
    $C[i]=B[j]$ $O(n^2)$
  - Intersection: Union(A, B, C)
    $k \leftarrow 0$
    for $i = 1$ to $|A|$
    if Member$(A[i], B)$ = True then $k \leftarrow k+1$,
    $C[k]=A[j]$ $O(n^2)$

Linked list representation/implementation of sets

- Represent the Set by a linked list, where the items of the linked-list are the elements of the Set. **Space used to represent a set of size n is $O(n)$.**
- **Complexity of operations**:

Membership: Requires traversal of the linked list. $O(n)$

Insertion: No traversal required if the list is not sorted. $O(1)$

Requires traversal if the list is sorted. $O(n)$

Deletion: Requires traversal. $O(n)$

Find: Requires traversing each list. [A good use case for multi-threading.] $O(n^k)$

Union: Assume that the result of the union of A, B is stored in A.

When A and B are disjoint then append list B to list A. $O(1)$

When A and B are not disjoint:

for every x in list A

Delete(x, B)

Append list B to list A $O(len(A) X len(B)) + O(1) = O(n^2)$

Note: Convenient to choose the shorter list as A.

Linked list representation/implementation of sets

- Represent the set by a linked list, where the items of the linked-list are the elements of the set. **Space used to represent a set of size n is $O(n)$.**
- Complexity of operations:

Intersection: for every x in list A

if Membership(x, B) is FALSE then Delete(x, A) $O(n^2)$

Difference: for every x in list A

if Membership(x, B) is TRUE then Delete(x, A) $O(n^2)$

- **Does the efficiency of membership, union, intersection, difference improve if the linked lists are sorted?**
- **Study the implementation of sets through the remaining data structures.**
- **Can trees, particularly binary trees be used to implement Sets?**

Comparison of various (element, Set) operations

Data Structure	Insert(x, S)	Delete(x, S)	Member(x, S) = Search(x, S)
Direct address table = Bit-vector	O(1)	O(1)	O(1)
Array (unsorted)	O(1)	O(1) [ignoring the time taken to locate the key]	O(n)
Array (sorted)	O(n) [requires data moves to the right; ignore the time taken to locate the position]	O(n) [requires data moves to the left; ignore the time taken to locate the position]	O(n) if using <u>linear search</u> O(log n) if using <u>binary search</u>

Search algorithms – Linear Search

- **Input**: Array A, key k
- **Output**: Answer to the query Membership(k, A)
- **Idea**: Traverse the array from the start until the element/key is found.

LinearSearch(k, A)

1. $c \leftarrow 1$
2. If $c > n$ then STOP // k is not found in A
3. If $A[c] = k$ then STOP // k is found in A
4. $c \leftarrow c+1$
5. Goto Step-2

Ex.: If A = <7, 14, 6, 12, 3>, then what is the result of LinearSearch(8, A), LinearSearch(6, A)?

Note: If k is found at position p then (i) $c=p$ (ii) LinearSearch STOPs
Note: If k is not found then (i) $c=n+1$ (ii) LinearSearch STOPs.

Run-time: $O(n)$

Search algorithms – Binary Search

- **Input**: Sorted array A, key k
- **Output**: Answer to the query Membership(k, A)
- **Idea**: If the search element is not the element in the middle position then (recursively) search in the left or the right sub-array according as the middle element is larger or smaller than the search element.

BinarySearch(k, A)

```
1.   L ← 1; R ← n
2.   while k not found
     2.1  if L > R then STOP                 // k is not found
     2.2  M ← L + floor((R-L)/2)
     2.3  if A[M] < x then L ← M+1
     2.4  if A[M] > x then R ← M-1
     2.5  if A[M] = x then STOP              // k is found
       endwhile
```

Ex.: If A = <2, 8, 13, 17, 23, 48, 50, 62>, then what is the result of BinarySearch(25, A)? What is the number of iterations? If A = <2, 3, 5, 7, 11, 13, 17> then what is the result of BinarySearch(11, A)? What is the number of iterations?

Run-time: $O(\log n)$

Comparison of various (element, Set) operations

Data Structure	Insert(x, S)	Delete(x, S)	Member(x, S) = Search(x, S)
Linked list (unsorted)	O(1)	O(1) [ignoring the time taken to locate the key]	O(n)
Linked list (sorted)	O(1) [ignoring the time taken to reach the required position] O(n) [including the traversal]	O(1) [ignoring the time taken to reach the required position] O(n) [including the traversal]	O(n)
Closed address table = Hashing by chaining/Open hashing/Closed addressing*	O(1) [assuming that the chain is an unsorted list]	O(1) [ignoring the time taken to locate the key on the chain]	O(n)
Open address table = Hashing by probing/Open addressing*	O(n) [ignoring the time taken to resize the table]	O(n) [ignoring the time taken to resize the table]	O(n)
* Ignoring the time taken to compute the hash value			
Tree	O(1)	O(h) [requires locating a replacement node]	O(n) [requires traversing]
Binary tree	O(1)	O(h) [requires locating a replacement node]	O(n) [requires traversing]

It is important to use data structures which are search efficient.

Ex.: *What is the time-complexity for the unary operations while representing Sets through various data structures?*