Basic Data Structures

Array

A contiguous area of memory, consisting of equal-sized elements, indexed by contiguous integers.

Operations:

Operation	Explanation	Time Complexity
Access(index)	Return the value of the element at a given index	O(1)
Add(index, value)	Add value at a given index to the array	O(1)
Remove(index)	Remove the value at a given index from the array	O(1)
Search(value)	Find a value in the array. Linear search	O(n)

Linked Lists

A list of elements, where each element contains a data field and a reference (link) to the next element in the list.
Elements are not stored contiguously in the memory.
Operations:

Operation	Explanation
PushFront(Key)	Add an element with value of key to the front of the list
Key TopFront()	Return the key of the first element (front item) of the list
PopFront()	Remove the first element (front item) of the list
PushBack(Key)	Add an element with the value of key, to the end (to the back) of the list
Key TopBack()	Returns the key of the last element (back item) of the list.
PopBack()	Removes the last element (back item) of the list.
Boolean Find(Key)	Returns true if value key is in the list, or false otherwise?
Erase(Key)	Removes key from list if it is included in the list, if not, does nothing.
Boolean Empty()	Returns True if list is empty, and False otherwise.
AddBefore(Node, Key)	Adds a value key before a given node
AddAfter(Node, Key)	Adds a value key after a given node

Singly Linked List

A linked list where each element contains a ***key*** and a ***next*** pointer

Pseudo Code Implementation (for some of operations):

```
PushFront(key)
node ←new node
node.key ← key
node.next ← head
head ← node
if tail = nil:
   tail ← head
```

```
PopFront()
if head = nil:
  ERROR: empty list
head ← head.next
if head = nil:
  tail ← nil
```

```
PushBack(key)
node ←new node
node.key ← key
node.next =nil
if tail = nil:
   head ← tail ← node
else:
   tail.next ← node
   tail ← node
```

```
PopBack()
if head = nil:  ERROR: empty list
if head = tail:
   head ← tail ←nil
else:
   p ← head
   while p.next.next ≠ nil:
      p ← p.next
   p.next ← nil; tail ← p
```

```
AddAfter(node, key)
node2 ←new node
node2.key ← key
node2.next = node.next
node.next = node2
if tail = node:
   tail ← node2
```

Doubly linked list

Is a list where each element contains a ***key***, a ***next*** pointer and a ***prev*** (previous) pointer.
Pseudo Code Implementation (for some of operations):

```
PushBack(key)
node ←new node
node.key ← key; node.next =nil
if tail = nil:
   head ← tail ← node
   node.prev ←nil
else:
   tail.next ← node
   node.prev ← tail
   tail ← node
```

```
PopBack()
if head = nil:  ERROR: empty list
if head = tail:
   head ← tail ←nil
else:
   tail ← tail.prev
   tail.next ←nil
```

```
AddBefore(node, key)
node2 ←new node
node2.key ← key
node2.next ← node
node2.prev ← node.prev
node.next ← node2
if node2.next ≠nil:
   node2.prev.next ← node2
if head = node:
   head ← node2
```

```
AddAfter(node, key)
node2 ←new node
node2.key ← key
node2.next ← node.next
node2.prev ← node
node.next ← node2
if node2.next ≠nil:
   node2.next.prev ← node2
if tail = node:
   tail ← node2
```

Operation	Explanation	Time Complexity	
		Singly linked list	Double linked list
PushFront(Key)	Add an element with value of key to the front of the list	O(1)	O(1)
Key TopFront()	Return the key of the first element (front item) of the list	O(1)	O(1)
PopFront()	Remove the first element (front item) of the list	O(1)	O(1)
PushBack(Key)	Add an element with the value of key, to the end (to the back) of the list	O(n) With a pointer to the tail of the list O(1)	O(n) With a pointer to the tail of the list O(1)
Key TopBack()	Returns the key of the last element (back item) of the list.	O(n) With a pointer to the tail of the list O(1)	O(n) With a pointer to the tail of the list O(1)
PopBack()	Removes the last element (back item) of the list.	O(n) With a pointer to the tail of the list O (n)	O(1)
Boolean Find(Key)	Returns true if value key is in the list, or false otherwise?	O(n)	O(n)
Erase(Key)	Removes key from list if it is included in the list, if not, does nothing.	O(n)	O(n)
Boolean Empty()	Returns True if list is empty, and False otherwise.	O(1)	O(1)
AddBefore(Node, Key)	Adds a value key before a given node	O(n)	O(1)
AddAfter(Node, Key)	Adds a value key after a given node	O(1)	O(1)

Stack

An abstract data type that holds an ordered, linear sequence of items.

- A stack is a LIFO – Last In, First Out - data structure.
- Stacks can be implemented usung an array or a linked list.

Stack supports the following operations:

Operation	Explanation	Time Complexity
Push (S, Key)	Pushes (adds) element Key to stack S. no return value	O(1)
Key Top(S)	Return most recently added key	O(1)
Key Pop (S)	Removes and returns the most recently added key	O(1)
Boolean Empty(S)	Returns true if stack is empty and false if not empty	O(1)

Array implementation

S.numElements is the current number of elements in stack S.

```
Boolean Empty(S)
if S.numElements  = 0
   return True
else
   return False
```

```
Key Pop(S)
if Empty(S)
   error "underflow"
else S.numElements ← S.numElements – 1
   return S[S.numElements + 1]
```

```
Push (S,  Key)
S.numElements ← S.numElements + 1
S[S.numElements] ← Key
```

```
Key Top()
return S[S.numElements]
```

Linked list implementation

Each node in the linked list contains a ***data*** element and a ***next*** pointer.

```
Boolean
Empty(S)
return head = nil
```

```
Key Top()
if head ≠ nil
   return head.key
else
   return error "isEmpty"
```

```
Key Pop(S)
if Empty(S)
   error "underflow"
else
   node ← head
   head ← node.next
   node.next ← nil
   return node.key
```

```
Push (S,  Key)
node ← new node
node.key ← key
node.next ←  head
head ←   node
```

Queue

An abstract data type that holds an ordered, linear sequence of items.

- A queue is a FIFO – First In, First Out - data structure.
- Queues can be implemented using an array or a linked list (with tail pointer).
- Each queue operation is O(1): Enqueue, Dequeue, Empty.

Operation	Explanation	Time Complexity
Enqueue (Q, Key)	Pushes (adds) element Key to stack S. no return value	O(1)
Dequeue(Q)	Return most recently added key	O(1)
Boolean Empty(Q)	Returns true if stack is empty and false if not empty	O(1)

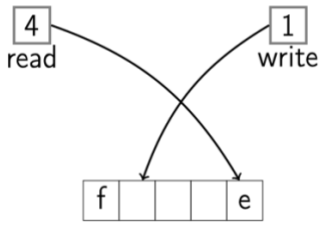
Array implementation

The array implementation includes an array Q, a read index, a write index, capacity & n.

- The read index "points" to the head of the queue, from which the next data should be read (initially 0).
- The write index "points" to the end of the queue, where the next item added to the queue will be inserted (initially 0).
- n is the current number of elements in the queue (initially 0).
- capacity is the size of the array.

For example: The array Q, read, write & n after 2 Enqueue operations: Enqueue (e), Enqueue (f).

n = 2



Enqueue(Q, x)

```
if n = capacity
        error "overflow"
Q[write] ← x
write ←( write + 1 )mod capacity
n ← n + 1
```

Empty(Q):

```
return n = 0
```

Dequeue(Q):

```
if Empty(Q)
        error "underflow"
x ← Q[read]
read ← (read + 1) mod capacity
n ← n - 1
return x
```

Linked list

Enqueue: use List.PushBack

Dequeue: use List.TopFront and List.PopFront

Empty: use List.Empty

Trees

Definition

A Tree is:

- empty, or
- a node with:
  - a key, and
  - a list of child trees.

Remark: (optional) parent

Tree Terminology

- Root: top node in the tree (has no parent)
- Child: a child has a line down directly from a parent
- Ancestor: parent, or parent of parent, etc.
- Descendant: child, or child of child, etc.
- Sibling: sharing the same parent
- Leaf: node with no children
- Interior node – a node which is not a leaf.
- Level: 1 + num edges between root and node
- Height: maximum depth of subtree node and farthest leaf
- Forest: collection of trees
- Path: sequence of nodes along the edges of a tree.

For binary tree, node contains:

- Key
- Left
- Right
- (optional) parent

```
Height(tree)
if tree = nil:
  return 0
return 1 + Max(Height(tree.left),
               Height(tree.right))
```

```
Size(tree)
if tree = nil
  return 0
return 1 + Size(tree.left) +
           Size(tree.right)
```

Types of tree

Depth-first: we completely traverse one sub-tree before exploring a sibling sub-tree:

- InOrderTraversal
- PreOrderTraversal
- PostOrderTraversal

Breadth-first: we traverse all nodes at one level before progressing to the next level:

- LevelTravesal

Depth-first

```
InOrderTraversal(tree)
if tree = nil:
  return
InOrderTraversal(tree.left)
Print(tree.key)
InOrderTraversal(tree.right)
```

```
PreOrderTraversal(tree)
if tree = nil:
  return
Print(tree.key)
PreOrderTraversal(tree.left)
PreOrderTraversal(tree.right)
```

```
PostOrderTraversal(tree)
if tree = nil:
  return
PostOrderTraversal(tree.left)
PostOrderTraversal(tree.right)
Print(tree.key)
```

Breadth-first

```
LevelTraversal(tree)
if tree = nil:  return
Queue q
q.Enqueue(tree)
while not q.Empty():
  node ← q.Dequeue()
  Print(node)
  if node.left ≠ nil:
    q.Enqueue(node.left)
  if node.right ≠ nil:
    q.Enqueue(node.right)
```

Summary

- Trees are used for lots of different things
- Trees have a key and children.
- Tree walks: DFS (ore-order, in-order, post-order) and BFS
- When working with a tree, recursive algorithms are common.
- In computer science, trees grow down.

Priority Queue

Priority queue is an abstract data type, which is a generalization of a queue where each element is assigned a priority and elements come out in order by priority.

Algorithms that use priority queues: Dijkstra, Prim, Huffman and heap sort.

Operations:

Operation	Description	Time Complexity Implementation using an unsorted array	Time Complexity Implementation using a sorted array	Time Complexity Implementation using a Binary heap
Insert(d, p)	Adds a new element with priority p and data d	O(1)	O(n)	O(lg n)
ExtractMax(Q)	Extracts an element with maximum priority from priority queue Q	O(n)	O(1)	O(lg n)
Remove(it)	Removes an element pointed by an iterator it	Differs by implementation: Array – O(n) Singly linked list – O(n) Doubly linked list – O(1)	O(n)	O(lg n)
GetMax()	Returns an element with maximum priority (without changing the set of elements)	O(n)	O(1)	O(1)
ChangePriority(it, p)	Changes the priority of an element pointed by it to p	O(1)	O(1)	O(lg n)

Pseudo

Insert(Q, d, p)	
unsorted array/list	sorted array
• Add e to the end of array/list – O(1)	• find a position for e, (using binary search) - O(log n) • Shift all elements to the right of it by 1 - O(n) • insert e - O(1)
O(1)	**O(n)**

ExtractMax()	
unsorted array/list	sorted array
• Find max O(n) • Delete element O(1) • If array – shift all elements to the left – O(n)	• extract the last element – O(1)
O(n)	**O(1)**

Remove(it)	
unsorted array/list	sorted array
• Delete element O(1) • If array – shift all elements to the left – O(n) • If singly linked list – find previous – O(n) • If double linked list – O(1)	• Remove element – O(1) • Shift all elements to the left of it by 1 - O(n)
Differs by implementation: • **Array – O(n)** • **Singly linked list – O(n)** • **Doubly linked list – O(1)**	**O(n)**

GetMax()	
unsorted array/list	sorted array
• Find and return max O(n)	• Return the last element – O(1)
O(n)	**O(1)**

ChangePriority(it, p)	
unsorted array/list	sorted array
• Change elements' priority - O(1)	• Change elements' priority – O(1)
O(1)	**O(1)**

Binary Max heap

Definition

Binary max-heap is a binary tree (each node has zero, one, or two children) where the value of each node is at least the values of its children.

In other words

For each edge of the tree, the value of the parent is at least the value of the child.

Definition

A binary tree is complete if all its levels are filled except possibly the last one which is filled from left to right.

Lemma

A complete binary tree with n nodes has height at most $O(\log n)$.

Proof

- Complete the last level to get a full binary tree on $n' \geq n$ nodes and the same number of levels ℓ.
- Note that $n' \leq 2n$.
- Then $n' = 2^\ell - 1$ and hence $\ell = \log_2(n'+1) \leq \log_2(2n+1) = O(\log n)$. □

A complete binary tree with n nodes has two main advantages:

1. Keep the tree shallow – a height at most O(log n).
2. Can be stored as an array, such that:
   - parent(i) = ⌊i/2⌋
   - leftchild(i) = 2i
   - rightchild(i) = 2i + 1.

Helper operations and declarations used in the array implementation of Binary

- ***siftUp()*** - swap the problematic node with its parent until the heap property is satisfied
- ***siftDown()*** - swap the problematic node with its parent until the heap property is satisfied
- ***maxSize*** is the maximum number of elements in the heap
- ***size*** is the size of the heap
- ***H[1 … maxSize]*** is an array of length ***maxSize*** where the heap occupies the first ***size*** elements
- ***Parent(i)*** **–** the index of the parent of the node in index i of the array
- ***LeftChild(i)*** **–** the index of the left child of the node in index i of the array
- ***RIghtChild(i)*** **–** the index of the right child of the node in index i of the array
-

Array implementation of binary max heap pseudo code:

Remark:
To keep the tree complete:

- When inserting an element, insert it as a leaf in the leftmost vacant position in the last level and let it sift up.

- When extracting the maximum value, replace the root by the last leaf and let it sift down.

```
Parent(i)
return ⌊i/2⌋
```

```
LeftChild(i)
return 2i
```

```
RightChild(i)
return 2i + 1
```

For arrays with starting index of 1

```
SiftUp(i)
while i > 1 and H[Parent(i)] < H[i]:
  swap H[Parent(i)] and H[i]
  i ← Parent(i)
```

```
SiftDown(i)
maxIndex ← i
ℓ ← LeftChild(i)
if ℓ ≤ size and H[ℓ] > H[maxIndex]:
  maxIndex ← ℓ
r ← RightChild(i)
if r ≤ size and H[r] > H[maxIndex]:
  maxIndex ← r
if i ≠ maxIndex:
  swap H[i] and H[maxIndex]
  SiftDown(maxIndex)
```

```
Insert(p)
if size = maxSize:
  return ERROR
size ← size + 1
H[size] ← p
SiftUp(size)
```

```
ExtractMax()
result ← H[1]
H[1] ← H[size]
size ← size − 1
SiftDown(1)
return result
```

```
Remove(i)
H[i] ← ∞
SiftUp(i)
ExtractMax()
```

```
ChangePriority(i, p)
oldp ← H[i]
H[i] ← p
if p > oldp:
  SiftUp(i)
else:
  SiftDown(i)
```

Summary

When implementing a priority queue using a binary max heap, implemented with an array, the resulting implementation is:

- fast: all operations work in time O(log n) (GetMax even works in O(1))
- space efficient: we store an array of priorities; parent-child connections are not stored, but are computed on the fly
- easy to implement: all operations are implemented in just a few lines of code

Heap Sort

We can create a new sorting algorithm using a priority queue implemented as a binary max heap

```
HeapSort(A[1 . . . n])

create an empty priority queue
for i from 1 to n:
  Insert(A[i])
for i from n downto 1:
  A[i] ← ExtractMax()
```

- The resulting algorithm is comparison-based and has running time O(n log n)
- Natural generalization of selection sort: instead of simply scanning the rest of the array to find the maximum value, use a smart data structure.
- Not in-place: uses additional space to store the priority queue.

HeapSort(A[1 . . . n])	Time complexity	Space Complexity
create an empty priority queue for i from 1 to n: Insert(A[i]) for i from n downto 1: A[i] ← ExtractMax()	O(1) n times - O(lg n) n times - O(lg n)	This is not an in-place algorithms, so an additional O(n) array is allocated
	O(n lg n)	O(n)

In-place heap sort

- For this, we will first turn an array into a heap (in-place) by permuting its elements. (Build Heap)
  - In a complete tree with n nodes, represented by an array, the last $\lfloor n/2 \rfloor$ indexes represent tree leaves (nodes with no children). By default, the subtrees that originate from these nodes satisfy the heap property. Thus, the first node, that may not satisfy the heap property is the first none leaf node, which can be found in index $\lfloor n/2 \rfloor$ of the array.
  - We then start repairing the heap property in all subtrees of depth 1, from bottom to top, starting with index $\lfloor n/2 \rfloor$ until index 1.
  - When we reach the root, the heap property is satisfied in the whole tree.
  - Running time: O(n log n) no additional memory needed (not taking run time stack recursion allocation into account).
- After we built the heap, we will repeatedly move the current largest element to the end of the current array (while we reduce the size of the array in each iteration). (Heap sort)

```
BuildHeap(A[1...n])
size ← n
for i from ⌊n/2⌋ downto 1:
  SiftDown(i)
```

```
HeapSort(A[1...n])
BuildHeap(A)                    {size = n}
repeat (n − 1) times:
  swap A[1] and A[size]
  size ← size − 1
  SiftDown(1)
```

Building Running Time

- The running time of BuildHeap is O(n)
- The running time of in-place HeapSort is O(n lg n)

Partial sorting

Input: An array $A[1 \ldots n]$, an integer $1 \leq k \leq n$.

Output: The last k elements of a sorted version of A.

```
PartialSorting(A[1...n], k)
BuildHeap(A)
for i from 1 to k:
  ExtractMax()
```

- Partial sorting Running time: O(n + k log n)
- When $k \leq \frac{n}{\lg n} \rightarrow k \log n \leq n \rightarrow$ partial sorting running time O(n)

Additional definitions:

Min-heap – a heap (each node has a value and a list of its children) where the value of each node is at most the value of its children.

Binary Min heap - A binary tree (each node has zero, one, or two children) where the value of each node is at least the values of its children.

d-ary heap – a generalized heap, in which nodes on all levels except for possibly the last level, have exactly d children. The height of such a tree is $\lg_d n$. The running time of SiftUp is $O(\lg_d n)$. The running time of SiftDown is $O(g_d n)$.

Summary:

- Priority queue supports two main operations: Insert and ExtractMax.
- In an array/list implementation one operation is very fast (O(1)) but the other one is very slow (O(n)).
- Binary heap gives an implementation where both operations take O(lgn) time.
- Binary heap also has a space efficient implementation.

Disjoint Sets

Definition

A disjoint-set data structure supports the following operations:

- MakeSet(x) creates a singleton set {x}
- Find(x) returns ID of the set containing x:
  - if x and y lie in the same set, then Find(x) = Find(y).
  - otherwise, Find(x)≠ Find(y)
- Union(x, y) merges two sets containing x and y

Implementation:

- Represent each set as a rooted tree
- Use the root of the set as its ID
- Union by rank heuristic: hang a shorter tree under the root of a taller one (since we would like to keep the trees shallow as possible)
- To quickly find a height of a tree, we will keep the height of each subtree in an array rank[1…n]: rank[i] is the height of the subtree whose root is i
- Path compression heuristic: when finding the root of a tree for a particular node, reattach each node from the traversed path to the root
  - When using path compression, rank[i] is no longer equal to the height of the subtree rooted at i is at most rank[i]
  - Still, the height of the subtree rooted at
  - and it is still true that a root node of rank k has at least w^k nodes in its subtree: a root node is not affected by path compression
- Amortized running time: O(log* n) (constant for practical values of n)

Pseudo code

```
MakeSet(i)
parent[i] ← i
```

Union Rank & Path compression heuristic Find(i):

```
Find(i)
if i ≠ parent[i]:
  parent[i] ← Find(parent[i])
return parent[i]
```

```
Union(i, j)
i_id ← Find(i)
j_id ← Find(j)
if i_id = j_id:
  return
if rank[i_id] > rank[j_id]:
  parent[j_id] ← i_id
else:
  parent[i_id] ← j_id
  if rank[i_id] = rank[j_id]:
    rank[j_id] ← rank[j_id] + 1
```

Operation	Linked list implementation Time Complexity	Union Rank heuristic Time complexity	Union Rank & Path compression heuristic Amortized Time complexity
MakeSet	O(1)	O(1)	O(1)
Find(i)	O(1)	O(lg n)	Almost O(1)
Union(x, y)	O(n)	O(lg n)	Almost O(1)

Hashing

A **hash table** is an implementation of a set or a map using hashing.
A **Hash functions:** for any set of objects S and any integer m > 0,
a function ***h*** : S → {0, 1, . . . ,m −1} is called a hash function.
m is called the cardinality of hash function **h**.
Collisions: when h(o1) = h(o2) and o1 ≠ o2

Definition

Map from S to V is a data structure with methods HasKey(O), Get(O), Set(O, v), where O ∈ S, v ∈ V.

```
HasKey(O)
L ← A[h(O)]
for (O', v') in L:
  if O' == O:
    return true
return false
```

```
Get(O)
L ← A[h(O)]
for (O', v') in L:
  if O' == O:
    return v'
return n/a
```

```
Set(O, v)
L ← A[h(O)]
for p in L:
  if p.O == O:
    p.v ← v
    return
L.Append(O, v)
```

Lemma

Let c be the length of the longest chain in A. Then the running time of HasKey, Get, Set is Θ(c + 1).

Proof

- If L = A[h(O)], len(L) = c, O ∉ L, need to scan all c items
- If c = 0, we still need O(1) time

Lemma

Let n be the number of different keys O currently in the map and m be the cardinality of the hash function. Then the memory consumption for chaining is Θ(n + m).

Proof

- Θ(n) to store n pairs (O, v)
- Θ(m) to store array A of size m

Definition

Set is a data structure with methods Add(O), Remove(O), Find(O).

Pseudo code set implementation:

```
Add(O)
L ← A[h(O)]
for O' in L:
  if O' == O:
    return
L.Append(O)
```

```
Remove(O)
if not Find(O):
  return
L ← A[h(O)]
L.Erase(O)
```

$h : S \to \{0, 1, \ldots, m-1\}$
$O, O' \in S$
$A \leftarrow$ array of m lists (chains) of objects O

```
Find(O)
L ← A[h(O)]
for O' in L:
  if O' == O:
    return true
return false
```

Definition
An implementation of a set or a map using hashing is called a hash table.

Conclusion

- Chaining is a technique to implement a hash table
- Memory consumption is O(n + m)
- Operations work in time O(c + 1)

Desirable Properties for a hash function

- Fast to compute
- Deterministic
- Distributes keys well into different cells
- Must have cardinality m (size of hash table)

Remark: if number of objects |S| is more than m – collisions will occur

Binary Search Trees

Binary Search Tree property - A node's key is larger than the key of any descendent of its left child, and smaller than the key of any descendant of its right child.

A binary search tree node has the following fields: Key, Parent (optional), Left child & Right child.

Operations (on binary search tree with root R)

Operation	Input	Output	Unbalanced binary search tree - Time Complexity	AVL - Time Complexity
Find(k, R)	Key k, Root R	The node in the tree of R with key k , if k not in tree R, return the place in the tree where k would fit	O(Depth) = O(n)	O(lg n)
Next	Node N	The node in the tree with the next largest key	O(Depth) = O(n)	O(lg n)
RangeSearch(x, y, R)	Numbers x,y, root R	A list of nodes with key between x and y	O(n)	O(n)
Insert(k, R)	Key k and root R	Adds node with key k to the tree	O(Depth) = O(n)	O(lg n)

Delete(N)	Node N	Removes node N from tree	O(Depth) = O(n)	O(lg n)
Merge(R_1, R_2)	root R_1, root R_2 with all keys in R_1's tree smaller than all keys in R_2's tree	The root of a new tree with all the elements of both trees	O(Depth) = O(n)	O(lg n)
Split(x, R)	Root R of a tree, key x	Two trees, one with elements ≤ x and one with elements > x.	O(Depth) = O(n)	O(lg n)

Pseudo code:

```
Find(k,R)
if R.Key = k:
  return R
else if R.Key > k :
  if R.Left ≠ null:
    return Find(k, R.Left)
  else
    return R
else if R.Key < k :
  if  R.Right ≠ null:
    return Find(k,R.Right)
  else
    return R
```

```
Next(N )
if N.Right ≠ null:
  return LeftDescendant(N.Right)
else:
  return RightAncestor(N)
```

```
RightAncestor(N )
if N.Key < N.Parent.Key
  return N.Parent
else:
  return RightAncestor(N.Parent)
```

```
LeftDescendant(N )
if N.Left = null
  return N
else:
  return LeftDescendant(N.Left)
```

```
RangeSearch(x , y , R )
L←∅
N ← Find(x,R) while N.Key ≤ y
  if N.Key ≥ x:
    L ← L.Append(N)
  N ← Next(N) return L
Return L
```

```
Insert(k , R )
P ← Find(k,R)
Add new node with key k as child of P
```

```
Merge(R1, R2)
T ← Find(∞, R1)
Delete(T )
MergeWithRoot(R1, R2, T )
return T
```

```
Split(R , x )
if R = null:
  return (null,null)
if x ≤ R.Key:
  (R1, R2) ← Split(R.Left, x)
  R3 ← MergeWithRoot(R2, R.Right, R)
return (R1,R3)
if x > R.Key: ...
```

```
Delete(N )
if N.Right = null:
  Remove N, promote N.Left
else:
  X ← Next(N)
  \\ X .Left = null
  Replace N by X, promote X.Right
```

```
MergeWithRoot(R1, R2, T )
T.Left ← R1
T .Right ← R2
R1.Parent ← T
R2.Parent ← T return T
```

The time complexity of many of the binary search tree operations are O(depth of tree). The maximum depth of a tree can be O(n). But if we insure that the tree is a balanced tree, the maximum depth of the tree is O(lg n), thus improving the time complexity of the binary search tree operations from O(n) to O(lg n). A balanced tree is when the left and right subtrees of every node are half the size of the whole tree (in which the node is the root).

The problem is that the insert and delete operations can destroy the trees' balance.

AVL trees are balanced binary search trees. This is accomplished by rebalancing the tree after each insertion or deletion.

Definition

The height of a node is the maximum depth of its subtree.

Recursive Definition – Height(N)

```
if N is leaf
  return 1
else
  return 1 + max(N.Left.Height, N.Right.Height)
```

AVL Properety

AVL trees maintain the following property:
For all nodes N,
| N.Left.Height – N.Right.Height| ≤ 1
Thus the Height of an AVL tree is O(lg n)

```
Rebalance(N)
P ← N.Parent
if N.Left.Height > N.Right.Height+1:
  RebalanceRight(N)
if N.Right.Height > N.Left.Height+1:
  RebalanceLeft(N)
AdjustHeight(N)
if P ≠ null:
  Rebalance(P)
```

```
AVLInsert(k, R)
Insert(k, R)
N ← Find(k, R)
Rebalance(N)
```

```
AVLDelete(N)
Delete(N)
M ← Left child of node replacing N
Rebalance(M)
```

```
AdjustHeight(N)
N.Height ← 1+ max(
                N.Left.Height,
                N.Right.Height)
```

```
AVLTreeMergeWithRoot(R1, R2, T)
if |R1.Height − R2.Height| ≤ 1:
  MergeWithRoot(R1, R2, T)
  T.Ht ← max(R1.Height, R2.Height) + 1
  return T
else if R1.Height > R2.Height:
  R' ← AVLTreeMWR(R1.Right, R2, T)
  R1.Right ← R'
  R'.Parent ← R1
  Rebalance(R1)
  return root
else if R1.Height < R2.Height:
  ...
```

```
RotateRight(X)
P ← X.Parent
Y ← X.Left
B ← Y.Right
Y.Parent ← P
P.AppropriateChild ← Y
X.Parent ← Y, Y.Right ← X
B.Parent ← X, X.Left ← B
```

```
RebalanceRight(N)
M ← N.Left
if M.Right.Height > M.Left.Height:
  RotateLeft(M)
RotateRight(N)
AdjustHeight on affected nodes
```

אופציה אלטרנטיבית למימוש של עץ AVL.
לתשומת לבכן – ניתן לעבוד עם כל אחת משתי האופציות המוצגות כאן.

מחיקה והוספת איבר לעץ AVL

בכל צמת q בעץ יש חמישה שדות: left[q], right[q], p[q], key[q], h[q]

שדה h מאחסן את גובה העץ (או התת-עץ) ש-q הוא השורש שלו.

(i) שגרות עזר:

שגרות פשוטות המבצעות פעולות שחוזרות על עצמן בשגרות הבאות.

height(q)

```
//q מחזירה את גובה הצומת

if q=NIL
then return -1
else return h[q]
```

compute-height(q)

```
//h מחשבת את גובהו של הצומת q ומציבה ערך בשדה

if q≠NIL
then h[q]←1+MAX(height(left[q]),height(right[q]))
```

balance-factor(q)

```
//q מחזירה את גורם האיזון של הצומת

return height(left[q])- height(right[q])
```

higher-son(q)

```
// מחזיר את התת-עץ הגבוה יותר מבין
//        .q שני התת-עצים של הצומת
//NIL אם התת-עצים באותו גובה מוחזר

if height(left[q])>height(right[q])
then return left[q]
else if height(left[q])<height(right[q])
     then return right[q]
     else return NIL
```

unbalanced(q)

```
//מחזירה ערך TRUE אם הצומת q אינו מאוזן
//                    FALSE אחרת מחזירה

return |balance-factor(q)|>1
```

(ii) ביצוע רוטציות

הרוטציות ממומשות כפי שמתואר בקורס.

left-rotate(T,x)

```
y←right[x]
right[x]←left[y]
if left[y]≠NIL
then p[left[y]]←x
p[y]←p[x]
if p[x]=NIL
then root[T]←y
else if x=left[p[x]]
     then left[p[x]]←y
     else right[p[x]]←y
left[y]←x
p[x]←y
compute-height(x)
compute-height(y)
```

right-rotate(T,x)

```
y←left[x]
left[x]←right[y]
if right[y]≠NIL
then p[right[y]]←x
p[y]←p[x]
if p[x]=NIL
then root[T]←y
else if x=right[p[x]]
     then right[p[x]]←y
     else left[p[x]]←y
right[y]←x
p[x]←y
compute-height(x)
compute-height(y)
```

(iii) איזון מחדש של צומת שאינו מאוזן

```
rebalance(T,q)
//q מאזנת מחדש את הצומת
r←higher-son(q)
s←higher-son(r)
if s=NIL
then if r=left[q]
     then s←left[r]
     else s←right[r]
if r=left[q]
then if s=right[r]
     then left-rotate(T,r)
     right-rotate(T,q)
else if s=left[r]
     then right-rotate(T,r)
     left-rotate(T,q)
```

הסבר השגרה rebalance

השגרה rebalance מקבלת כפרמטר צומת q שגורם האיזון שלו הוא 2 או (-2) ואשר כל הצאצאים שלו הם בעלי גורם איזון 1, 0 או (-1).

השגרה משתמשת ברוטציות כדי לגרום לכך שהתת-עץ המושרש ב-q יתאזן.

השגרה מבחינה בין שישה מקרים שונים, ומטפלת בכל אחד מהם.

בשלושה מקרים התת-עץ השמאלי של q גבוה יותר מהתת-עץ הימני שלו, ובשלושה מקרים התת-עץ הימני גבוה יותר מהתת-עץ השמאלי שלו. מכיוון ששלושת המקרים האחרונים סימטריים לשלושת הראשונים, נראה רק את הטיפול בשלושת המקרים בהם התת-עץ השמאלי גבוה יותר מהימני. ודאו שברור לכם שהעץ אחרי התיקון אכן מאוזן.

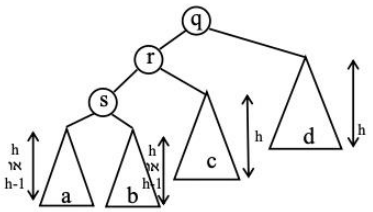
מקרה ראשון:

r הוא הבן השמאלי של q. התת-עץ השמאלי של r גבוה יותר מהתת-עץ הימני שלו.

הפתרון הוא לבצע רוטציה ימנית סביב q.

התת-עץ המושרש ב-q

לפני ביצוע הרוטציה:

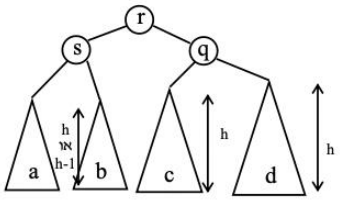


הערה:

a או b (או שניהם) בגובה h.

התת-עץ

אחרי ביצוע הרוטציה:



3

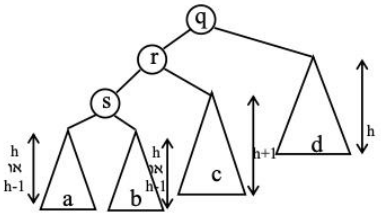
מקרה שני:

r הוא הבן השמאלי של q. התת-עץ השמאלי של r והתת-עץ הימני של r הינם באותו גובה.

הפתרון הוא לבצע רוטציה ימינה סביב q.

<u>התת-עץ המושרש ב-q</u>

<u>לפני ביצוע הרוטציה:</u>

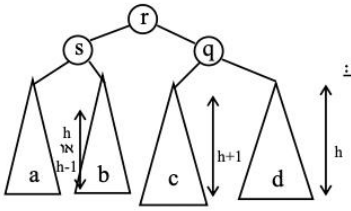


הערה:

a או b (או שניהם) בגובה h.

<u>התת-עץ</u>

<u>אחרי ביצוע הרוטציה:</u>



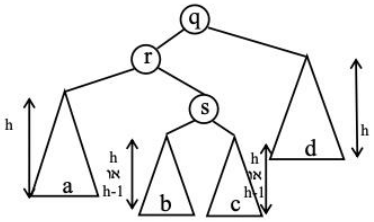
מקרה שלישי:

r הוא הבן השמאלי של q. התת-עץ השמאלי של r נמוך יותר מהתת-עץ הימני שלו.

הפתרון הוא לבצע רוטציה שמאלה סביב r, ואחר-כך רוטציה ימינה סביב q.

<u>התת-עץ המושרש ב-q</u>

<u>לפני ביצוע הרוטציה:</u>

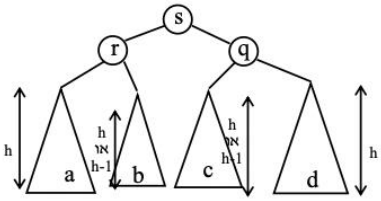


הערה:

c או b (או שניהם) בגובה h.

<u>התת-עץ</u>

<u>אחרי ביצוע הרוטציה:</u>



4

(iv) הכנסת איבר לעץ AVL:

```
TREE-INSERT(T,z)
y←NIL
x←root[T]
critical←NIL
while x≠NIL do
  if balance-factor(x)≠0 //  כלומר: גורם האיזון הוא 1 או
(-1)
  then critical←x
  y←x
  if key[z]<key[x]
  then x←left[x]
  else x←right[x]
p[z]←y
if y=NIL
then root[T]←z
else if key[z]<key[y]
     then left[y]←z
     else right[y]←z
while y≠critical do
   compute-height(y)
   y←p[y]
if y≠NIL and unbalanced(y)
then rebalance(y)
```

השורות בשחור הן השורות של אלגוריתם ההכנסה לעץ חיפוש בינרי, כפי שהוא מופיע בספר של קורמן.
בכחול מופיעות התוספות המתאימות לעץ AVL.

המשתנה critical יחזיק את הצמת העמוק ביותר שגורם האיזון שלו שונה מ 0 (הצומת הקריטי).
לאחר ההכנסה לעץ החיפוש, נטפס מאביו של העלה החדש ועד הצמת הקריטי ונעדכן את שדה הגובה לצמתים בדרך.
הצמת הקריטי הוא היחיד שיש לאזן (אם איזונו הופר).
שימו לב שהשגרה rebalance יכולה להיתקל במקרה הראשון או במקרה השלישי שתוארו לעיל, ולא במקרה השני. במקרים הראשון והשלישי גובה העץ לאחר האיזון חוזר להיות כמו גובה העץ לפני ההכנסה, ולכן אין צורך להמשיך ולבדוק את אבותיו הקדמונים של הצומת הקריטי.

(v) מחיקת איבר מעץ AVL:

```
TREE-DELETE(T,z)
if left[z]=NIL or right[z]=NIL
then y←z
else y←TREE-SUCCESSOR(z)
if left[y]≠NIL
then x←left[y]
else x←right[y]
if x≠NIL
then p[x]←p[y]
if p[y]=NIL
then root[T]←x
else if y=left[p[y]]
     then left[p[y]]←x
     else right[p[y]]←x
if y≠z
then key[z]⇔key[y]
r←p[y]
flag←TRUE
while r≠NIL and flag do
  tmp←height(r)
  compute-height(r)
  if unbalanced(r)
  then rebalance(r)
       r←p[r]
  if tmp=height(r)
  then flag←FALSE
  r←p[r]
return y
```

השורות בשחור הן השורות של אלגוריתם המחיקה מעץ חיפוש בינרי, כפי שהוא מופיע בספר של קורמן.
בכחול מופיעות התוספות המתאימות לעץ AVL.

הרעיון של האלגוריתם הוא לטפס מאביו של הצומת שנמחק עד לצומת העמוק ביותר שגובהו לא השתנה (או עד לשורש), לאזן את כל הצמתים שבדרך, ולעדכן את גובהם.
כאשר נגיע לצמת שגובהו לא השתנה ניתן לעצור, כי ברור שהגובה של אבותיו הקדמונים לא השתנה.
כדי לדעת מתי לעצור האלגוריתם הופך את הדגל `flag` ל `FALSE` כאשר הוא נתקל בצומת שגובהו לפני העדכון שווה לגובהו אחרי העדכון.

Data Structures Summary

Data Structure	Supported Operations	Explanation	Time Complexity
Array	Access(index)	Return the value of the element at a given index	O(1)
	Add(index, value)	Add value at a given index to the array	O(1)
	Remove(index)	Remove the value at a given index from the array	O(1)
	Search(value)	Find a value in the array. Linear search	O(n)
Singly Linked List	PushFront(Key)	Add an element with value of key to the front of the list	O(1)
	Key TopFront()	Return the key of the first element (front item) of the list	O(1)
	PopFront()	Remove the first element (front item) of the list	O(1)
	PushBack(Key)	Add an element with the value of key, to the end (to the back) of the list	O(n) With a pointer to the tail of the list O(1)
	Key TopBack()	Returns the key of the last element (back item) of the list.	O(n) With a pointer to the tail of the list O(1)
	PopBack()	Removes the last element (back item) of the list.	O(n) With a pointer to the tail of the list O (n)
	Boolean Find(Key)	Returns true if value key is in the list, or false otherwise?	O(n)
	Erase(Key)	Removes key from list if it is included in the list, if not, does nothing.	O(n)
	Boolean Empty()	Returns True if list is empty, and False otherwise.	O(1)
	AddBefore(Node, Key)	Adds a value key before a given node	O(n)
	AddAfter(Node, Key)	Adds a value key after a given node	O(1)
Doubly Linked List	PushFront(Key)	Add an element with value of key to the front of the list	O(1)
	Key TopFront()	Return the key of the first element (front item) of the list	O(1)
	PopFront()	Remove the first element (front item) of the list	O(1)
	PushBack(Key)	Add an element with the value of key, to the end (to the back) of the list	O(n) With a pointer to the tail of the list O(1)
	Key TopBack()	Returns the key of the last element (back item) of the list.	O(n)

			With a pointer to the tail of the list O(1)
	PopBack()	Removes the last element (back item) of the list.	O(1)
	Boolean Find(Key)	Returns true if value key is in the list, or false otherwise?	O(n)
	Erase(Key)	Removes key from list if it is included in the list, if not, does nothing.	O(n)
	Boolean Empty()	Returns True if list is empty, and False otherwise.	O(1)
	AddBefore(Node, Key)	Adds a value key before a given node	O(1)
	AddAfter(Node, Key)	Adds a value key after a given node	O(1)
Stack	Push (S, Key)	Pushes (adds) element Key to stack S. no return value	O(1)
	Key Top(S)	Return most recently added key	O(1)
	Key Pop (S)	Removes and returns the most recently added key	O(1)
	Boolean Empty(S)	Returns true if stack is empty and false if not empty	O(1)
Queue	Enqueue (Q, Key)	Pushes (adds) element Key to stack S. no return value	O(1)
	Dequeue(Q)	Return most recently added key	O(1)
	Boolean Empty(Q)	Returns true if stack is empty and false if not empty	O(1)
Priority Queue – implemented with an unsorted array	Insert(d, p)	Adds a new element with priority p and data d	O(1)
	ExtractMax(Q)	Extracts an element with maximum priority from priority queue Q	O(n)
	Remove(it)	Removes an element pointed by an iterator it	Differs by implementation: Array – O(n) Singly linked list – O(n) Doubly linked list – O(1)
	GetMax()	Returns an element with maximum priority (without changing the set of elements)	O(n)
	ChangePriority(it, p)	Changes the priority of an element pointed by it to p	O(1)
Priority Queue – implemented with a sorted array	Insert(d, p)	Adds a new element with priority p and data d	O(n)
	ExtractMax(Q)	Extracts an element with maximum priority from priority queue Q	O(1)
	Remove(it)	Removes an element pointed by an iterator it	O(n)
	GetMax()	Returns an element with maximum priority (without changing the set of elements)	O(1)

	ChangePriority(it, p)	Changes the priority of an element pointed by it to p	O(1)
Priority Queue – implemented with a Binary Heap	Insert(d, p)	Adds a new element with priority p and data d	O(lg n)
	ExtractMax(Q)	Extracts an element with maximum priority from priority queue Q	O(lg n)
	Remove(it)	Removes an element pointed by an iterator it	O(lg n)
	GetMax()	Returns an element with maximum priority (without changing the set of elements)	O(1)
	ChangePriority(it, p)	Changes the priority of an element pointed by it to p	O(lg n)
Disjoint Set – linked list implementation	MakeSet(x)	creates a singleton set {x}	O(1)
	Find(x)	returns ID of the set containing x: • if x and y lie in the same set, then Find(x) = Find(y). • otherwise, Find(x)≠ Find(y)	O(1)
	Union(x, y)	merges two sets containing x and y	O(n)
Disjoint Set – Union Rank Heuristic implementation	MakeSet(x)	creates a singleton set {x}	O(1)
	Find(x)	returns ID of the set containing x: • if x and y lie in the same set, then Find(x) = Find(y). otherwise, Find(x)≠ Find(y)	O(lg n)
	Union(x, y)	merges two sets containing x and y	O(lg n)
Disjoint Set – Union Rank & Path compression Heuristic implementation – amortized time complexity	MakeSet	creates a singleton set {x}	O(1)
	Find(i)	returns ID of the set containing x: • if x and y lie in the same set, then Find(x) = Find(y). otherwise, Find(x)≠ Find(y)	Almost O(1)
	Union(x, y)	merges two sets containing x and y	Almost O(1)
Hash Table – average time complexity	Add(value)	Add value to hash table	O(c + 1)
	Remove(value)	Remove value from hash table	O(c + 1)
	Find(value)	Find value in hash table	O(c + 1)
Binary Search Trees	Find(k, R)	Input: Key y, Root R Output: The node in the tree of R with key k , if k not in tree R, return the place in the tree where k would fit	O(Depth) = O(n)
	Next	Input: Node N Output: The node in the tree with the next largest key	O(Depth) = O(n)
	RangeSearch(x, y, R)	Input: Numbers x, y, root R Output: A list of nodes with key between x and y	O(n)

	Insert(k, R)	Input: key K, and root R Output: Adds node with key k to the tree	O(Depth) = O(n)
	Delete(N)	Input: Node N Output: Removes node N from tree	O(Depth) = O(n)
	Merge(R_1, R_2)	Input: root R_1, root R_2 with all keys in R_1's tree smaller than all keys in R_2's tree Output: The root of a new tree with all the elements of both trees	O(Depth) = O(n)
	Split(x, R)	Input: Root R of a tree, key x Output: Two trees, one with elements $\leq$ x and one with elements > x.	O(Depth) = O(n)
AVL Tree	Find(k, R)	Input: Key y, Root R Output: The node in the tree of R with key k , if k not in tree R, return the place in the tree where k would fit	O(lg n)
	Next	Input: Node N Output: The node in the tree with the next largest key	O(lg n)
	RangeSearch(x, y, R)	Input: Numbers x, y, root R Output: A list of nodes with key between x and y	O(n)
	Insert(k, R)	Input: key K, and root R Output: Adds node with key k to the tree	O(lg n)
	Delete(N)	Input: Node N Output: Removes node N from tree	O(lg n)
	Merge(R_1, R_2)	Input: root R_1, root R_2 with all keys in R_1's tree smaller than all keys in R_2's tree Output: The root of a new tree with all the elements of both trees	O(lg n)
	Split(x, R)	Input: Root R of a tree, key x Output: Two trees, one with elements $\leq$ x and one with elements > x.	O(lg n)

Graph Algorithms

Definition

An (***undirected) Graph*** is a finitie collection V of vertices, and a collection E of edges each of which connects a pair of vertices.

Ways of representing a graph:

1. Edge list – list of all edges
2. Adjacency matrix – Entry is 1 if there is an edge between two edges, and 0 if there is not.
3. Adjacency list - For each vertex, a list of adjacent vertices.

	Is Edge?	List of all edges	List
Adjacency Matrix	O(1)	$O(\|V\|^2)$	$O(\|V\|)$
Edge List	$O(\|E\|)$	$O(\|E\|)$	$O(\|E\|)$
Adjacency List	O(deg)	$O(\|E\|)$	O(deg)

Graph algorithm runtimes depend on |V | and |E|.

For instance, which is faster, $O(|V|^{3/2})$ or $O(|E|)$?

The answer depends on the density of the graph, namely how many edges you have in terms of the number of vertices.

Definition

Dense graph – when $|E| \approx |V|^2$ - A large fraction of pairs of vertices are connected by edges.

Definition

Sparse graph – when $|E| \approx |V|$ - Each vertex has only a few edges.

Definition
A directed Graph is a collection V of vertices, and a collection E of edges, where each edge has a start vertex and an end vertex.

Definition
Two vertices v,w in a directed graph are ***connected*** if you can reach v from w and can reach w from v.

Definition
A ***path*** in a graph G is a sequence of vertices $v_0, v_1, \ldots, v_n$ so that for all i, 0<=i<=n-1, (v_i, v_{i+1}) is an edge of G. Each edge appears at most once in this sequence.

Definition
Node u is ***reachable*** from node S if there is a path from S to u Input: Graph G and a vertex s Output: The collection of vertices v of G so that there is a path from s to v.

Definition
Strongly connected components (SCC) are graph components that include only nodes that have a path between all pairs of vertices in the component.

Definition
A ***Connected graph*** is a graph which has a path between every two vertices.

A directed graph is called **strongly connected** if it has a directed path from each vertex to each other vertex.

The vertices of a graph G can be partitioned into connected components so that v is reachable from w if and only if they are in the same connected component.

A directed graph can be partitioned into strongly connected components where two vertices are connected if and only if they are in the same component.

Definition
A ***cycle*** in a graph G is a path in which the start vertex is the same as its end vertex.

Definition
Directed Acyclic Graph (or DAG) - A directed graph G that has no cycles. Any DAG can be linearly ordered.

Definition
A ***source*** is a vertex with no incoming edges. A ***sink*** is a vertex with no outgoing edges.

Definition
Let G^R be the graph obtained from G by reversing all of the edges

Find sink components of G by running DFS on G^R

Reverse Graph Components
G^R and G have same SCCs
Source components of G^R are sink components of G

The vertex with largest postorder in G^R is in a sink SCC of G.

Definition
Metagraph(G) - a Metagraph of graph G, is a graph created by defining a node to represent every SCC – strongly connected component - in graph G, and including the edges of graph G that connect these components.

The metagraph of graph G is always a directed acyclic graph

Find if a graph contains a sink (Pseudo code):

Follow path as far as possible
$v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_n$. Eventually either:

- Cannot extend (found sink).
- Repeat a vertex (have a cycle).

LinearOrder (less efficient):

```
LinearOrder(G )
while G non-empty:
  Follow a path until cannot extend
  Find sink v
  Put v at end of order
  Remove v from G
```

Runtime:

- $O(|V|)$ paths.
- Each takes $O(|V|)$ time.
- Runtime $O(|V|^2)$.

Additional definition

A *cycle* in a graph G is a sequence of vertices $v_1, v_2, \ldots, v_n$ so that $(v_1,v_2),(v_2,v_3),\ldots,(v_{n-1},v_n), (v_n,v_1)$ are all edges. If G contains a cycle, it cannot be linearly ordered.

Definition

Length of the path L(P) is the number of edges in the path.

Definition

The **distance** between two vertices u, v in G is the length of the shortest path between u and v.
If there is no path between u and v, then the length between them is ∞.
If v=u, the length between them is 0.

Algorithms to explore and traverse a graph:

Algorithms to explore and traverse a graph:

DFS – Depth First Search

```
DFS(G)
for all v ∈ V:     mark v unvisite
for v ∈ V:
  if not visited(v):
    Explore(v)
```

```
Explore(v)
visited(v) ← true
for (v, w) ∈ E:
  if not visited(w):
    Explore(w)
```

Total runtime:

- $O(1)$ work per vertex.
- $O(1)$ work per edge.
- Total $O(|V| + |E|)$.

Modification of DFS to count the number of connected component in a graph:

```
DFS(G)
for all v ∈ V mark v unvisited
cc ← 1
for v ∈ V:
  if not visited(v):
    Explore(v)
    cc ← cc + 1
```

```
TopologicalSort(G)
DFS(G)
sort vertices by reverse post-order
```

Find Graph G's Scc's;

```
SCCs(G)

Run DFS(G^R)
for v ∈ V in reverse postorder:
  if not visited(v):
    Explore(v)
    mark visited vertices
      as new SCC
```

Runtime O(|V| + |E|)

```
Explore(v)

visited(v) ← true
for (v, w) ∈ E:
  if not visited(w):
    Explore(w)
```

Shortest Paths in graphs with equal edges lengths

Find the shortest paths from a single source, in a case all edges length's are equal.
Assuming graphs are undirected (though algorithm works on directed graphs as well).

BFS – Breadth First Search

In a BFS, we traverse the graph layerwise.
Starting from a source node s, first we visit all immediate children (all the nodes that're one edge away from the source node).
Then we'd move on to all *those* nodes' children (all the nodes that're *two edges* away from the source node).
And so on until we reach the end.

Implementation using a queue:

```
BFS(G, S)

for all u ∈ V:
  dist[u] ← ∞
dist[S] ← 0
Q ← {S} {queue containing just S}
while Q is not empty:
  u ← Dequeue(Q)
  for all (u, v) ∈ E:
    if dist[v] = ∞:
      Enqueue(Q, v)
      dist[v] ← dist[u] + 1
```

The running time of breadth-first search is O(|E| + |V |).

Shortest-path tree is indeed a tree, i.e. it doesn't contain cycles (it is a connected component by construction).

Constructing shortest-path tree

```
BFS(G, S)
for all u ∈ V:
  dist[u] ← ∞, prev[u] ← nil
dist[S] ← 0
Q ← {S} {queue containing just S}
while Q is not empty:
  u ← Dequeue(Q)
  for all (u, v) ∈ E:
    if dist[v] = ∞:
      Enqueue(Q, v)
      dist[v] ← dist[u] + 1, prev[v] ← u
```

```
ReconstructPath(S, u, prev
result ← empty
while u ≠ S:
  result.append(u)
  u ← prev[u]
return Reverse(result)
```

Corollary:
If S → . . . → u → t is a shortest path from S to t, then d(S, t) = d(S, u) + w(u, t)

dist[v] will be an upper bound on the actual distance from S to v.
The edge relaxation procedure for an edge (u, v) just checks whether going from S to v through u improves the current value of dist[v].

```
Relax((u, v) ∈ E)

if dist[v] > dist[u] + w(u, v):
  dist[v] ← dist[u] + w(u, v)
  prev[v] ← u
```

Shortest Paths in graphs with different edges lengths
Find the shortest paths from a single source, in a case edges weights are non-equal and non-negative.
Referring to the general version of the problem and assuming that the graphs are directed.
Dijkstra's Algorithm –
Works for any graph with non-negative edge weights

```
Dijkstra(G, S)
for all u ∈ V:
  dist[u] ← ∞, prev[u] ← nil
dist[S] ← 0
H ← MakeQueue(V) {dist-values as keys}
while H is not empty:
  u ← ExtractMin(H)
  for all (u, v) ∈ E:
    if dist[v] > dist[u] + w(u, v):
      dist[v] ← dist[u] + w(u, v)
      prev[v] ← u
      ChangePriority(H, v, dist[v])
```

Total running time:

$$T(\texttt{MakeQueue}) + |V| \cdot T(\texttt{ExtractMin}) + |E| \cdot T(\texttt{ChangePriority})$$

Priority queue implementations:

- array:

$$O(|V| + |V|^2 + |E|) = O(|V|^2)$$

- binary heap:

$$O(|V| + |V| \log |V| + |E| \log |V|) = O((|V| + |E|) \log |V|)$$

If graph contains negative weight edges Dijkstra's algorithm will return a wrong solution. Therefore, there is Bellman-Ford Algorithm that will solve this problem:

```
BellmanFord(G, S)
{no negative weight cycles in G}
for all u ∈ V:
  dist[u] ← ∞
  prev[u] ← nil
dist[S] ← 0
repeat |V| − 1 times:
  for all (u, v) ∈ E:
    Relax(u, v)
```

The running time of Bellman–Ford algorithm is O(|V ||E|).

In a graph without negative weight cycles, Bellman–Ford algorithm correctly finds all distances from the starting node S.

Negative wheight Cycles
A graph G contains a negative weight cycle if and only if |V|-th (additional) iteration of BellmanFord(G, S) updates some dist-value.

Finding negative cycles in a graph Algorithm:
Run |V| iterations of Bellman–Ford algorithm, save node v relaxed on the last iteration.
v is reachable from a negative cycle
Start from x ← v, follow the link x ← prev[x] for |V | times _ will be definitely on the cycle
Save y ← x and go x ← prev[x] until x = y again

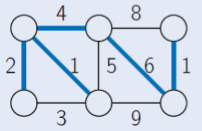
Minimum Spanning Tree – MST

Input: A connected, undirected graph $G = (V, E)$ with positive edge weights.

Output: A subset of edges $E' \subseteq E$ of minimum total weight such that the graph (V, E') is connected.

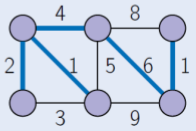
Kruskal's algorithm

repeatedly add the next lightest edge if this doesn't produce a cycle



Prim's algorithm

repeatedly attach a new vertex to the current tree by a lightest edge



Remark: the set E' always froms a tree.

Properties of Trees

A tree is an undirected graph that is connected and acyclic.

A tree on n vertices has n − 1 edges.

Any connected undirected graph G(V , E) with |E| = |V | − 1 is a tree.

An undirected graph is a tree iff there is a unique path between any pair of its vertices.

Kruskal's Algorithm

Algorithm: repeatedly add to X the next lightest edge e that doesn't produce a cycle

At any point of time, the set X is a forest, that is, a collection of trees

The next edge e connects two different trees_say, T1 and T2

The edge e is the lightest between T1 and V − T1, hence adding e is safe

Implementation Details

use disjoint sets data structure

initially, each vertex lies in a separate set each set is the set of vertices of a

connected component to check whether the current edge {u, v} produces a cycle, we check whether u and v belong to the same set

```
Kruskal(G)
for all u ∈ V:
  MakeSet(v)
X ← empty set
sort the edges E by weight
for all {u, v} ∈ E in non-decreasing
  weight order:
  if Find(u) ≠ Find(v):
    add {u, v} to X
    Union(u, v)
return X
```

Total running time: O(|E| log |V |)

Prim's Algorithm

X is always a subtree, grows by one edge at each iteration

we add a lightest edge between a vertex of the tree and a vertex not in the tree

very similar to Dijkstra's algorithm

```
Prim(G)
for all u ∈ V:
  cost[u] ← ∞,  parent[u] ← nil
pick any initial vertex u0
cost[u0] ← 0
PrioQ ← MakeQueue(V)     {priority is cost}
while PrioQ is not empty:
  v ← ExtractMin(PrioQ)
  for all {v, z} ∈ E:
    if z ∈ PrioQ and cost[z] > w(v, z):
      cost[z] ← w(v, z),  parent[z] ← v
      ChangePriority(PrioQ, z, cost[z])
```

the running time is:

|V| · T(ExtractMin)+|E| · T(ChangePriority)

for array-based implementation, the running time is O(|V|2)

for binary heap-based implementation, the running time is O((|V| + |E|) log |V|) = O(|E| log |V|)

Sample Questions

Question:

Describe how to implement a "minimum stack" – a stack, that in addition to the standard PUSH and POP operations, supports another operation "MIN" which returns the minimum value of the stack (only returns the value, does not erase it).

Running time of all three operations is constant.

גם עליה ממומשת PUSH -ו POP מפעולות שחוץ מחסנית – מינימום" מחסנית"לממש ביצד תארו פעולת MIN מוחקת ולא מחזירה במחסנית (רק המינימלי הערך את המחזירה).

קבועים יהיו הפעולות שלוש של הריצה זמני.

פתרון:

מבנה הנתונים יורכב משתי מחסניות. שתי המחסניות יכילו אותו מספר איברים. הראשונה תכיל את

אברי מחסנית המינימום, השנייה תאחסן בראש המחסנית את האיבר המינימלי.

תחילה נגדיר פעולת עזר המחזירה את האיבר שבראש המחסנית top –

```
top(s)
        x←pop(s)
        push(s,x)
        return x
push(s,x)
        push(s[1],x)
        if empty(s[2]) or x<top(s[2])
                then push(s[2],x)
        else push(s[2],top(s[2]))
pop(s)
        pop(s[2])
        return pop(s[1])
min(s)
        return top(s[2])
```

כל הפעולות הנ"ל מתבצעות בזמן ריצה קבוע ללא תלות בגודל הקלט ולכן

סיבוכיות זמן הריצה שלהן היא קבועה כשל הפעולות הבסיסיות על מחסנית.

Question

Given two very large decimal numbers x, y. Each number is stored in a stack in such a way that the most significant digit is in the bottom of the stack.

Write an algorithm that receives two stacks S1, S2 where x and y are stored and return stack S3 that stores the sum x+y (also in S3, the most significant digit is in the bottom of the stack).

You may use only the standard stack's operations.

At the end of the algorithms stacks S1 and S2 should be empty.

The algorithm should be written in pseudocode.

נתונים שני מספרים עשרוניים גדולים מאוד x, y.

כל אחד מהמספרים נמצא במחסנית , כך שהספרה המשמעותית ביותר נמצאת בתחתית המחסנית.

כתבו אלגוריתם המקבל שתי מחסניות S1 ,S2, שבהן מאוחסנים המספרים x ו– y ומחזיר מחסנית S3שבה נמצא הסכום, x + y (גם ב - S3 הספרה המשמעותית ביותר תהיה בתחתית המחסנית) .

מותר להשתמש אך ורק בפעולות הבסיסיות המוגדרות על מחסנית.

בתום ריצת האלגוריתם S1,S2 תהיינה ריקות. עליכם לכתוב את האלגוריתם בפסידוקוד.

פתרון:

```
c=0
while (not empty(s1) or not empty(s2)) do
        if empty(s1) //הספרה מהמחסנית הראשונה
        then x=0
        else x=pop(s1)
        if empty(s2) //הספרה מהמחסנית השניה
        then y=0
        else y=pop(s2)
        push(s3,(x+y+c) mod 10)//חיבור
        c=(x+y+c)/10 //חישוב הנשא
if c>0
        then push(s3,c) //הנשא האחרון
while not empty(s3) do // שלוש הלולאות הופכות את הסדר
```

```
        push(s2,pop(s3))
while not empty(s2) do
        push(s1,pop(s2))
while not empty(s1) do
        push(s3,pop(s1))
```

Question:

We'll add to a heap definition the following condition: in each node that has two children, the left son value is greater than or equal to the right son value.

Given a heap H that satisfied the above additional condition, describe a procedure (no need to write pseudocode) that performs the following operation (while saving the new heap condition): increasing element H[i] in d (d>0) and fixing the heap.

Running time should be O(lg(n)).

נוסיף להגדרת הערמה את התנאי: לכל צומת שהוא אב לשני בנים, ערך הבן השמאלי גדול או שווה לערך הבן הימני.

בהינתן ערמה H המקיימת את התנאי הנוסף, תארו שגרה (אין צורך לכתוב פסידוקוד) שתבצע את הפעולה הבאה (עם שמירת התכונה החדשה): הגדלת האיבר $H[i]$ בערך d $(d>0)$ ותיקון הערמה. זמן הריצה יהיה $O(lgn)$.

פתרון:

מגדילים את ערך הצמת **i** , ומבצעים את שגרת התיקון הבאה:

כל עוד לא הגענו לשרש:

אם הוא בן ימני וגדול מאחיו מחליפים בינו לבין אחיו.

משווים אותו עם אביו: אם הוא גדול מאביו מחליפים ביניהם. אם הוא לא גדול מאביו: סיימנו.

מספר הפעמים שנבצע פעולה זו על ערימה המקיימת את התנאי הנוסף הוא לכל היותר תלוי בגובה הערימה - lg(n) – תואם לסיבוכיות זמן ריצה הנדרשת.

Question:

Given a heap H that satisfied an additional condition: for each node $x \in H$, all keys in x's left sub-tree are lower than (or equal to) all keys in x's right sub-tree.

What will be the result of H's post order traversal?

נתונה ערמה H המקיימת את התנאי הנוסף: עבור כל צומת $x \in H$, כל המפתחות בתת-עץ השמאלי של x קטנים מ- (או שווים ל-) כל המפתחות בתת-עץ הימני של x.

מה מתקבל מהסריקה בסדר סופי של H?

פתרון:

מקבלים את רשימת הצמתים ממוינת מהקטן לגדול.

Question:
Given a hash table T[1,…,m]. Each cell in the table points to an **n** size array. We want to implement a binary heap in each of the m arrays. We'll insert a sequence of n keys to the hash table in two different ways:

1. Insert all n keys to the described structure. After all keys are inserted, build m heaps. What is the worst case running time of this way?
   Is it possible to receive a better average running time?
2. Insert all n keys to the described structure. After every key's insertion, fix the appropriate heap. What is this way running time in the worst and average cases?

נתונה טבלת גיבוב $T[1..m]$; כל תא של הטבלה מצביע אל מערך בגודל n. ברצוננו ליישם בכל אחד מ-m המערכים ערימה בינרית. ברצוננו להכניס לטבלת הגיבוב סדרה של n מפתחות בשתי שיטות.

א. בשיטה הראשונה מכניסים את כל n המפתחות למבנה הנ"ל; אחרי שכולם הוכנסו, בונים את m הערימות. מהו זמן הריצה של השיטה במקרה הגרוע? האם יתכן שמתקבל זמן ריצה טוב יותר בממוצע?

ב. בשיטה השנייה מכניסים את n המפתחות למבנה הנ"ל; אחרי כל הכנסת מפתח, מתקנים את הערימה המתאימה. מהו זמן הריצה של השיטה במקרה הגרוע ובממוצע?

פתרון:

נניח ש $n=\Theta(m)$, כנדרש מטבלת גיבוב.

א. המקרה הגרוע O(n): כל המפתחות נכנסים לאותו תא. בממוצע בכל תא יהיו O(1) מפתחות אך זמן הריצה יהיה עדיין O(n).

ב. במקרה הגרוע O(nlgn): כל המפתחות נכנסים לאותו תא. בממוצע בכל תא יהיו O(1) מפתחות ולכן זמן הריצה יהיה O(n).

Question:
Suggest a data structure S the supports the following operations:

SEARCH (S, k): searches key k in S and returns an element with key value k.

INSERT (S, k): insert a new element with key value k to structure S.

DELETE (S, k): delete an element with key value k from structure S.

FREQUENCY (S, k): returns the number of elements with key value k in structure S

Required running time of each operation is O(1) in average.

הציעו מבנה נתונים S התומך בפעולות הבאות:

SEARCH (S,k): חיפוש אחר המפתח k במבנה S, והחזרת איבר כלשהו בעל מפתח k;

INSERT (S,k): הכנסת איבר חדש בעל המפתח k למבנה S;

DELETE (S,k): מחיקת איבר כלשהו בעל המפתח k מהמבנה S;

FREQUENCY (S,k): החזרת מספר האיברים בעלי המפתח k שבמבנה S;

זמן הריצה הנדרש של כל אחת מהפעולות הינו O(1) בממוצע.

פתרון:

מבנה הנתונים הוא טבלת גיבוב בעלת פונקציית גיבוב המגבבת בשיטת הפיזור האחיד כך שלכל צמת המייצג מפתח מסוים יש מצביע לרשימה מקושרת לכל האיברים בעלי אותו המפתח. בצמת יהיה, בנוסף, מונה לאורך הרשימה המקושרת.

הכנסה - ייבדק האם קיים איבר בעל מפתח k בטבלה. אם כן – יתווסף האיבר לרשימה המקושרת המתאימה ומונה מספר איברי הרשימה יגדל ב-1. אם לא – תתווסף רשימה מקושרת חדשה שגודלה 1.

חיפוש איבר בעל מפתח k – הפעלת פונקצית הגיבוב על ערך k. גישה לתא בטבלת הגיבוב ושאר הפעולות על רשימה מקושרת פועלות בסיבוכיות זמן הריצה קבועה - O(1)

במחיקה, אם יש איבר בעל מפתח k, יימחק האיבר הראשון ברשימה המתאימה, וגודלה יקטן ב-1. אם גודל הרשימה התאפס, יימחק המפתח מן הטבלה.

בדיקה אם יש איבר בעל מפתח k בטבלה – הפעלת פונקצית גיבוב על הערך k, גישה לתא המתאים בטבלת הגיבוב ובדיקה האם בתא זה יש רשימה מקושרת. ויתר הפעולות על רשימה מקושרת פועלות בסיבוכיות זמן ריצה קבועה – O(1).

בחיפוש יוחזר האיבר הראשון ברשימה המקושרת של המפתח k (אם יש רשימה כזאת) – חיפוש מתבצע באמצעות הפעלת פונקציית הגיבוב, גישה לתא המתאים בטבלת הגיבוב ובדיקה האם התא מכיל ערך או לא. פעולות אלו מתבצעות ב O(1).

כדי לדעת את מספר האיברים בעלי המפתח k, ייקרא המונה של הרשימה של המפתח k. אם אין רשימה כזאת מספר המופעים הוא 0.

פונקציה זו תבצע את אותה פעולה של פונקציית החיפוש ובדיקת מונה הרשימה אינה משפיעה על סיבוכיות זמן הריצה.

Question:

Write an algorithm that receives as parameters a binary search tree t and a value x. The algorithm returns a pointer to a node that its value is closest to the value of x.

כתבו אלגוריתם המקבל עץ חיפוש בינרי וערך x. האלגוריתם מחזיר מצביע לצמת שערכו קרוב ביותר לערך x.

פתרון:

הצמת המבוקש חייב להימצא על מסלול החיפוש של x בעץ: אם x נמצא בעץ אז הצמת המכיל אותו הוא הצמת המבוקש; אם x לא בעץ, אז הצמת המבוקש הוא הקודם בסדר תוכי או העוקב בסדר תוכי של x (אם x היה בעץ).

```
findClosest(t,x)
c←t
while t≠null do
  if key[t]=x
  then return t
  if |key[c]-x|>|key[t]-x|
  then c←t
  if key[t]<x
  then t←right[t]
  else t←left[t]
return c
```

Question:

Following is an alternative algorithm to erase node z from a binary search tree.

In the third case, when a node has two children, the algorithm will find the node's successor y, then change left[y] with left[z]. now it's possible to remove z as in the second case of erasing an element from a binary search tree.

1. Describe one advantage and one disadvantage of this algorithm in comparison to the original erasing algorithm (as described in course material)
2. Will the new algorithm work for erasing a node from an AVL tree?

נתאר אלגוריתם חלופי עבור מחיקת צומת z מעץ חיפוש בינרי.

במקרה השלישי, כאשר לצומת שני בנים, מאתרים את העוקב שלו y, ואז מחליפים בין $left[y]$ לבין $left[z]$; עבשיו אפשר להסיר את z כמו במקרה השני.

א. תארו יתרון אחד וחיסרון אחד לפחות של אלגוריתם זה יחסית לאלגוריתם המחיקה המקורי (המתואר בספר).

ב. האם ניתן להשתמש באלגוריתם החדש למחיקת צומת מעץ AVL?

פתרון:

א. היתרון הוא שיש כאן החלפה בין צמתים ולא בין תכולה של צמתים. אם יש איברים נוספים שמצביעים לצמת מסוים, הם לא יצביעו אליו יותר אחרי המחיקה.
החיסרון הוא שהעץ עלול להפוך למאוד לא מאוזן. המחיקה המקורית משנה את גבהי התת-עצים ב-1 לכל היותר.

ב. אלגוריתם המחיקה ב-AVL מסתמך על כך שגובהי התת-עצים קטן ב-1 לכל היותר. לכן האלגוריתם החלופי לא יכול לשמש למימוש AVL.

Question:

Prove or refute the following statement:

1. Adding a new edge to any directed graph G=(V,E) causes the number of strongly connected components to decrease mostly by one.

הוכיחו או הפריכו כל אחת משתי הטענות הבאות:

א. לכל גרף מכוון $G=(V,E)$ הוספת קשת חדשה לגרף גורמת למספר הרכיבים הקשירים היטב בגרף לקטון לכל היותר ב-1.

פתרון:

א. הטענה אינה נכונה: למשל, גרף מכוון ובו שלושה צמתים {1, 2, 3} וקשתות (1, 2), (2, 3). בגרף שלושה רכיבים קשירים היטב – כל צומת הוא רכיב בפני עצמו. אבל, אם נוסיף את הקשת (3, 1) יהיה בגרף רכיב קשיר היטב אחד, המכיל את כל הצמתים. כלומר, מספר הרכיבים קטן ב-2.

Question:

Given a directed, **acyclic** graph G=(V,E) with weighted edges. In addition, given two vertices s,t ϵ V. Describe an linear in input length algorithm, that finds the shortest path (track) from s to t.

נתון גרף G=(V, E) מכוון, **וחסר מעגלים** עם משקלים על הקשתות. עוד נתונים שני קדקודים s, t ∈ V. תאר אלגוריתם לינארי באורך הקלט, המוצא את המסילה הקצרה ביותר מ-s ל-t.

פתרון

תאור האלגוריתם:

1. נריץ TopSort על G. נסמן את הגרף הממוין G'=(V, E').

   הערה: ה-DFS המופעל ב-TopSort יופעל מקודקוד s.

2. נבנה גרף הפוך $G^T=(V, E^T)$ ביחס ל-G', כאשר:

$$E^T = \{ (u, v) \mid (v, u) \in E[G'] \}$$

3. נריץ על G^T את השגרה CalculateWeight, החל מקודקוד t, כמתואר להלן:

<u>CalculateWeight *(G^T, t)*</u>

```
for i ← 1 to t
        μ[v_i] ← ∞
        β[v_i] ← NIL
μ[t] ← 0
for i ← t downto 1
        for each vertex u∈Adj[v_i]
                if  ( μ[u] > μ[v_i] + w(v_i, u) )
                        μ[u] ← μ[v_i] + w(v_i, u)
                        β[u] ← v_i
```

4. משקל המסלול המבוקש הוא הערך שנמצא ב-μ[s].

   המסלול עצמו ימצא על-ידי מעבר מקודקוד s, לפי שדה הבנים - β.

<u>*סיבוכיות:*</u>

1. TopSort: O(m + n).
2. בניית G^T: O(m + n).
3. CalculateWeight: O(m + n).
4. מציאת המסלול: O(m).

סה"כ: O(m + n).

נכונות:

מכיוון שהגרף חסר מעגלים – ניתן להפעיל את אלגוריתם TopSort.

מתכונות ומנכונות TopSort אנו יודעים שלכשנסיים טיפול בקודקוד מסוים – לא נחזור אליו יותר.

מכיוון שעל בסיס עובדה זו כתובה השגרה CalculateWeight – ניתן ישירות לראות בנכונותה, שכן אופן עדכון ותחזוק שדה המשקל - μ - ברור מאליו – אם קיים משקל מצטבר קטן יותר מהנוכחי – מתבצע עדכון.

כך גם ברור אופן תחזוק שדה הבנים - β (שהוא בהפוך לשדה האב - π, המוכר לנו מאלגוריתמים רבים אחרים).

Question:

Write an efficient algorithm, that receives a un-directed graph G=(V,E) as an input, with non-negative weighted edges. Each edge is colored in either black or red. The input includes also one vertex s ϵ V. The algorithm should find for every vertex v ϵ V, the shortest path among all paths from s to v that begin in a red edge and end in a black edge.

Analyze algorithm's complexity and prove its correctness.

כתוב אלגוריתם יעיל ככל שתוכל, המקבל כקלט גרף לא-מכוון $G = (V, E)$ עם משקלות אי-שליליים על הקשתות, ובו כל קשת צבועה באחד משני צבעים: אדום ושחור. כמו כן, הקלט כולל צומת ***s*** בגרף. על האלגוריתם למצוא לכל צומת $v \in V$ את אורך המסלול הקצר ביותר מבין כל המסלולים מ-***s*** ל-***v*** המתחילים בקשת אדומה ומסתיימים בקשת שחורה.

נתח את סיבוכיות האלגוריתם והוכח את נכונותו.

פתרון:

נבנה מהגרף הנתון, גרף חדש ***G*** ללא צביעה על הקשתות, באופן הבא:

נגדיר גרף $G^* = (V^*, E^*)$ עבור $V^* = \{s\} \cup V' \cup V''$

כאשר V',V'' הם שכפולים של V, ונסמן לכל $v \in V$: $v' \in V', v'' \in V''$.

1. $E^* \leftarrow \phi$
2. לכל $(u,v) \in E$:
3. $E^* \leftarrow E^* \cup \{(u', v')\}$
4. אם **(*u, v*)** אדומה אז
5. אם ***u*=*s*** אז $E^* \leftarrow E^* \cup \{(s, v')\}$
6. אחרת (**(*u, v*)** שחורה)

7. $E^* \leftarrow E^* \cup \{(u', v'')\}$

משקלות הקשתות החדשות יהיה כמשקל הקשתות המקבילות להן בגרף המקורי. כלומר,

$w(u, v')=w(u', v')=w(u', v'')=w(u, v)$.

סיבוכיות בניית ***G*** ליניארית, $|E^*| \le 3|E|, |V^*| \le 2|V|$.

כעת נפעיל את האלגוריתם של דייקסטרה על הגרף החדש מהצומת ***s***. התוצאות המאוחסנות ב-**d[v'']** הן הרצויות.

סיבוכיות: $O(|E^*|+|V^*|\log|V^*|) = O(|E|+|V|\log|V|)$

הסבר: לכל מסלול ב-***G*** המתחיל מ-***s*** עם קשת אדומה ומסתיים ב-***v*** בקשת שחורה מתאים מסלול ב-***G**** המתחיל ב-***s*** ומסתיים ב-***v''*** (מסלול כזה יעבור מ-***s*** ל-***V'***, יישאר שם עד הצומת לפני האחרון והקשת האחרונה תהיה מצומת ב-***V'*** אל ***v''***), <u>ולהיפך</u>. כלומר קבוצת המסלולים ב-***G**** מ-***s*** ל-***V''*** מתאימה לקבוצת המסלולים החוקיים המוגדרים בשאלה, והאלגוריתם של דייקסטרה מחזיר את אורך המסלול המינימלי מביניהם.

Question:

Given an undirected, weighted and connected graph G=(V,E). Each edge is colored in blue or white or in any other color. Describe an algorithm that checks if there is a minimum spanning tree that does not include blue and white edges. All edges weighting 10 or less are blue and all edges weighting 20 or more are white. Other edges colors have no meaning.

נתון גרף לא מכוון, קשיר וממושקל G=(V, E). כל קשת צבועה בכחול או בלבן או בכל צבע אחר. תאר אלגוריתם הבודק האם קיים עץ פורש מינימלי ללא הקשתות הכחולות והלבנות. כל הקשתות ממשקל 10 או פחות הן כחולות, וכל הקשתות ממשקל 20 ויותר הן לבנות. אין משמעות לשאר הצבעים של שאר הקשתות.

פתרון:

תאור האלגוריתם:

1. נריץ על G את האלגוריתם של Kruskal, למציאת עץ פורש מינימלי T.

   נסמן: w(T) – המשקל של העץ הפורש מינימלי.

2. נסיר מ-G את הקשתות הלבנות שמשקלן קטן/שווה ל-10, ואת הקשתות הכחולות שמשקלן גדול/שווה ל-20.

   נסמן גרף זה ב-G'.

3. נריץ על G' את האלגוריתם של Kruskal.

אם לא נצליח לבנות עץ פורש – אזי לא קיים עץ כנדרש.

אחרת - נסמן את העץ 'T.

4. אם w(T') = w(T), אזי 'T הוא העץ כנדרש.

אחרת – לא קיים עץ כנדרש.

<u>סיבוכיות:</u>

1. Kruskal: O(m log n).
2. בניית 'G: O(m + n).
3. Kruskal: O(m log n).
4. בדיקה: O(1).

סה"כ: O(m log n).

<u>נכונות:</u>

הנכונות נובעת ישירות מנכונות Kruskal.

Question:

Given an un-directed, weighted, connected graph G and a minimum spanning tree T on G. After removing one edge e=(u,v) from G, find the minimum spanning tree in graph G'=G\e. time complexity O(m).

נתונים גרף לא מכוון, קשיר וממושקל G, ועץ פורש מינימלי T בתוכו. מורידים מ-G קשת e=(u, v).

מצא עץ פורש מינימלי בגרף G' = G \ e, בזמן O(m).

פתרון:

תאור האלגוריתם:

1. אם $e \notin T$ אזי T הוא גם העץ הפורש המינימלי של 'G.

אחרת:

2. T \ {e} הוא יער המורכב מ-2 עצים.

<u>נסמן</u>: T_v את העץ ששורשו הוא v.

T_u את העץ ששורשו הוא u.

נפעיל BFS מקודקוד v על T \ {e}.

נפעיל BFS^* מקודקוד u על T \ {e}.

BFS^* הנו אלגוריתם הזהה לחלוטין לאלגוריתם ה-BFS "הרגיל", פרט לכך שהוא צובע את הקדקודים באדום במקום בשחור.

3. נעבור על כל הקשתות ב-'G, ונחפש קשת e' = (i, j), שמשקלה מינימלי, וכן שקדקודיה, i ו-j צבועים בצבעים שונים (האחד אדום והשני שחור).

אם לא נמצאת קשת כזו – לא קיים עץ פורש ב-'G.

אם קיימת קשת כזו – אזי העץ הפורש המינימלי ב-G' הוא: $T' = T_v \cup T_u \cup \{e'\}$.

סיבוכיות:

1. בדיקת כל קשתות העץ: O(m).
2. הרצת BFS: O(m + n), ומכיוון שהגרף קשיר – O(m).

   הרצת BFS^*: O(m + n), ומכיוון שהגרף קשיר – O(m).
3. בדיקת כל הקשתות ב-G': O(m).

סה"כ: O(m).

נכונות:

ברור שאם $e \notin T$ אזי T הוא גם העץ הפורש המינימלי של G'.

הוצאת הקשת e=(u, v) מגדירה את החתך: (קדקודי T_u, קדקודי T_v) = (S, V\S).

נגדיר: A קבוצת הקשתות השייכות ל-T_u או ל-T_v. כלומר: $A = \{ e \in E(T) \mid (e \in T_v) \vee (e \in T_u) \}$.

ברור ש-A היא קבוצת קשתות השייכות לעץ פורש מינימלי כלשהו של G'.

על פי משפט (משפט 24.1 בעמוד 110 בספר "מבוא לאלגוריתמים") – A היא קבוצת קשתות השייכות לעץ פורש מינימלי כלשהו של G', ומכבדת את החתך (S, V\S) ⇐ הקשת הקלה e' = (i, j), החוצה חתך זה היא קשת בטוחה עבור A.

הוספת הקשת הבטוחה e' ל-A מחברת בין רכיבים זרים (שקדקודיהן צבועים בצבעים שונים – שחור או אדום) ⇐ A (לאחר הוספת e') מכילה n-1 קשתות, והגרף $G_A = (V, A)$ הנו חסר מעגלים.

⇐ הקשתות שב-A מהוות את העץ הפורש המינימלי של G': $T' = T_v \cup T_u \cup \{e'\}$.