CSC263H1: Data Structure and Analysis

Winter 2024

Professor. Marsha Chechik

Course Notes

Author: Xinyue Li

Last Edition: May 9, 2024

1. Complexity

1.1 Review

- Complexity - amount of resources required for running an algorithm, measured as a function of input size
- Resource - running-time or memory space
- Time Complexity - number of steps executed by an algorithm
- Space Complexity - number of unit of space required by an algorithm
- Basic Operation - any operation whose run-time **does not depend** on the input size
  - *e.g.* arithmetic operations, assignments, array accesses, comparisons, return statements
  - we do not try to precisely quantify the exact number of basic operations

Measuring by Counting Steps

1. Represent as a function $T(n)$ of **input size** n
2. Every chunk of instructions is represented by a constant
   - chunk - sequence of instructions that always get executed together

Runtime can be measured by counting the number of times all lines are executed, or the number of times some important lines are executed.

We prove bounds on $T(n)$ using asymptotic notation

- Upper Bound - $T(n) \in \mathcal{O}(f(n))$
- Lower Bound - $T(n) \in \Omega(f(n))$
- Tight Bound - $T(n) \in \Theta(f(n)) : T(n) \in \mathcal{O}(f(n))$ and $T(n) \in \Omega(f(n))$

If $T(n)$ is a polynomial of degree k, then $T(n) \in \mathcal{O}(n^k)$

If $T(n) = g(n) + f(n)$, and $f(n)$ asymptotically dominates $g(n)$, then $T(n) \in \mathcal{O}(f(n))$

$$1 \quad \log n \quad \sqrt{n} \quad n \quad n \log n \quad n^2 \quad n^3 \quad 2^n \approx n! \quad n^n$$

fast $\longrightarrow$ slow

Different Cases of Running Time

Let $t(n)$ represent number of steps executed by an algorithm A on input x

- Worst-Case - the maximum running time of A for all inputs of size n
- Best-Case - the minimum …
- Average-Case - the expected … $T(n) = \mathrm{E}[t_n]$1.2

1.2 Worst-Case Running Time Analysis

1. Identify the input size
   - *e.g.* number of bits, number of list elements, number of vertices and/or edges
2. Identify the case in which the performance of the algorithm is worst
3. Given an approximation of number of basic operations that execute in that case
4. Given an upper-bound/lower-bound/tight-bound for $T(n)$

Clarification

- $\mathcal{O}$ and Ω specify bounds over a mathematical functions
- Worst-case and best-case correspond to algorithms
- $\mathcal{O}$ and Ω can both be used to upper-bound and lower-bound the worst-case and best-case

Upper-Bound

Take arbitrary input x with size n, use at most keyword to count steps

Lower-Bound

Take specific input x with size n, use exactly keyword to count steps

1.3 Average-Case Running Time Analysis

Direct Computation

1. Define S_n, i.e. the space of all inputs of size n
2. Assume a probability distribution over S_n - specifying likelihood of each input
3. Define the random variable t_n over S_n
   - $t_n(x)$ - number of steps executed by A on an input x in S_n
4. Compute the expected value of $t_n(x)$

$$\mathrm{E}[t_n] = \sum_i i \times Pr[t_n = i]$$

   - $Pr[t_n = i]$ - probability of t_n obtaining the value i

Indicator Random Variables

Define indicator random variables $X_1, X_2, \ldots, X_m$ s.t.:

- $X = X_1 + X_2 + \ldots + X_m$;
- Each X_i has only two possible values: 0 or 1

Then

$$\mathrm{E}[X] = Pr[X_1 = 1] + \ldots + Pr[X_m = 1]$$

2. Priority Queues and Heaps

2.1 Priority Queue

- Objects - a set of elements, where each element has a priority
- Operations
  - *Insert(PQ, x, p)* - Add x to the priority queue *PQ* with the priority p
  - *FindMax(PQ)* - Return the item in *PQ* with the highest priority
  - *ExtractMax(PQ)* - Remove and return the item from *PQ* with the highest priority
  - *IncreaseKey(PQ, x, k)* - Increases the priority value p of the element x to the new value k

left child	right child	parent
$2i$	$2i+1$	$\lfloor \frac{i}{2} \rfloor$

  - k assumed to be at least as large as p

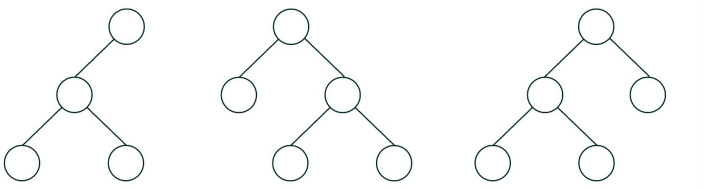
Time Complexity with List

	Unsorted List	Sorted List
Insert(PQ, x, p)	$\Theta(1)$	$\Theta(n)$
FindMax(PQ)	$\Theta(n)$	$\Theta(1)$
ExtractMax(PQ)	$\Theta(n)$	$\Theta(1)$
IncreaseKey(PQ, x, k)	$\Theta(1)$ (assume we know where x is placed)	$\Theta(n)$

2.2 Heaps

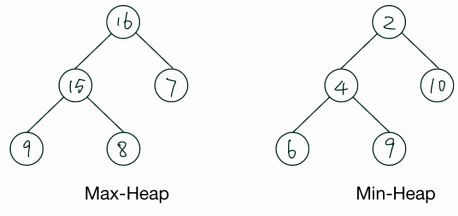
Complete Binary Tree

- A binary tree is complete iff it satisfies the following two properties
  1. All of its levels are full, except possibly the **bottom** one
  2. All of the nodes in the bottom level are as far to the left as possible
- There is only one complete tree shape for each number of nodes



violate 1 violate 2 correct

Max-Heap and Min-Heap

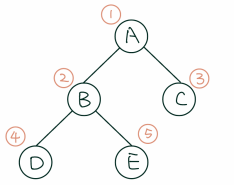


- Max/Min-Heap Property - A tree satisfies the max/min-heap property iff for each node in the tree, the value of that node is greater/less than or equal to the value of all of its descendants
- Max/Min-Heap - A complete binary tree that satisfies the max/min-heap property
  - Implication - Every sub-tree of a max-heap/min-heap is also a max-heap/min-heap

Array Representation

- Level Order Traversal - from left to right, level by level
- For a node corresponding to index i (assuming the items are stored starting at index 1)

Use for Priority Queue - Each item x has a key $x.p$ which represents its priority and A is a list-based max heap based on priorities of its items



2.3 Implementation

FindMax

- Implementation

  HeapMaximum(A): Return the root of A

  ```
  return A[1]
  ```

- Worst-Case Running Time - $\Theta(1)$

IncreaseKey

- Implementation

  HeapIncreaseKey(A, i, k) (assume that i is the index of x in the array)

  1. Set the priority of x to k

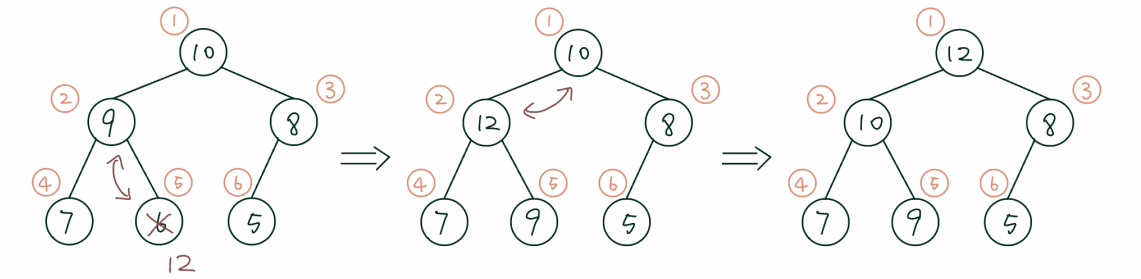
     ```
     A[i].p = k
     ```

  2. Bubble-up x to proper position, by swapping with parent until k is not greater than priority of parent of x

     ```
     i = A.size
     while i > 1:
         curr_p = A[i].p
         parent_p = A[i // 2].p
         if curr_p <= parent_p:      # heap property satisfied, break
             break
         else:
             A[i], A[i // 2] = A[i // 2], A[i]
             i = i // 2
     ```

e.g. *HeapIncreaseKey(A, 5, 12)*

- Worst-Case Running Time - $\Theta(\log n)$



- i.e. move a leaf all the way to the root
- $\Theta(h) = \Theta(\log n)$ where h is the height of the heap and n is the number of nodes of heap

MaxHeapInsert

- Implementation

  MaxHeapInsert(A, x, k):

  1. Insert x at the (only) spot that keeps the tree a complete binary tree

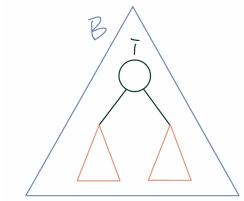
     ```
     A.size += 1
     A[A.size].item = x
     A[A.size].p = k
     ```

  2. Bubble-up x to a proper position to maintain the max-heap property

     ```
     ...  # same as IncreaseKey
     ```

- Worst-Case Running Time - $\Theta(\log n)$

MaxHeapify



- Pre-conditions - i is a node in a complete binary tree B. The binary trees rooted at *Left(i)* and *Right(i)* are max-heaps
- Post-condition - The binary trees rooted at i is a max-heap
- Implementation

  MaxHeapify(B, i): Bubble-down i to a proper position, by swapping with children

  1. Compare the root and its children
     - If the root is the largest, then the tree is already a Max-Heap
     - Otherwise, swap the root with largest child
  2. Fix the subtree rooted at swapped child to maintain the max-heap property by repeating Step 1 for the sub-tree which its root has been swapped

     ```
     while i * 2 <= B.size:
         curr_p = B[i].p
         left_p = B[2 * i].p
         right_p = B[2 * i + 1].p

         if curr_p >= left_p and curr_p >=right_p:
             break
     ```

```
        else if left_p >= right_p:
            B[i], B[2 * i] = B[2 * i], B[i]
            i = 2 * i
        else:
            B[i], B[2 * i + 1] = B[2 * i + 1], B[i]
            i = 2 * i + 1
```

- Worst-Case Running Time - $\Theta(\log n)$
  - i.e. move the root all the way down to a leaf

HeapExtractMax

- Implementation

  HeapExtractMax(H):

  1. Return the root of the tree
     ```
     root = H[1]
     ```
  2. Replace the root with a node f in the heap so that the tree remains a complete binary tree (we choose the last node)
     ```
     H[1] = H[H.size]
     H.size -= 1
     ```
  3. Bubble-down f to a proper position to maintain the max-heap property
     ```
     i = 1
     ...  # same as MaxHeapify
     ```
- Worst-Case Running Time - $\Theta(\log n)$

Conclusion

- Intuition - Tree is partially sorted. Enough to make query operations fast while not requiring full sorting after each update
- Complete tree - Ensures height is small
- Heap Order - Supports faster heap operations

2.4 Heap Sort

Given a max-heap H, we can crated a sorted array out of H by keep extracting max element for n times, then the extracted keys are sorted in non-ascending order

- Pre-conditions - A is an arbitrary array of size n (starting index is 1)
- Post-condition - A is sorted in non-decreasing order

- Implementation

  HeapSort(A)

  1. Convert A to a max-heap
  2. Let count point to the end of A

     count = A.size
  3. Extract the max element
     - Call *HeapExtractMax(A[1: count + 1])*
     - Put the max element where count points to
     - Decrease count by one

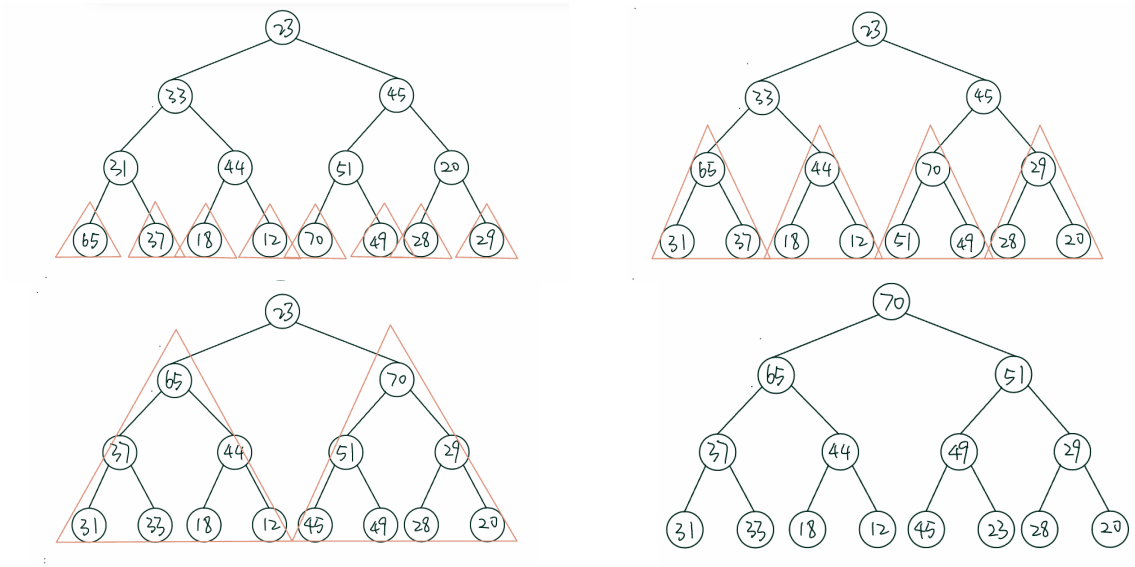
     ```
     while count > 0:
         extract_max = HeapExtractMax(A[1: count+1])
         A[count] = extract_max
         count -= 1
     ```
  4. Repeat Step 3 until count is 0

- Worst-Case Running Time - $\Theta(n \log n)$

2.5 BuildMaxHeap

General Idea - Build a max heap bottom-up

1. Interpret the list as the level order of a complete binary tree
2. Starting from bottom of the tree, call *MaxHeapify* for all non-leaf nodes



- Implementation

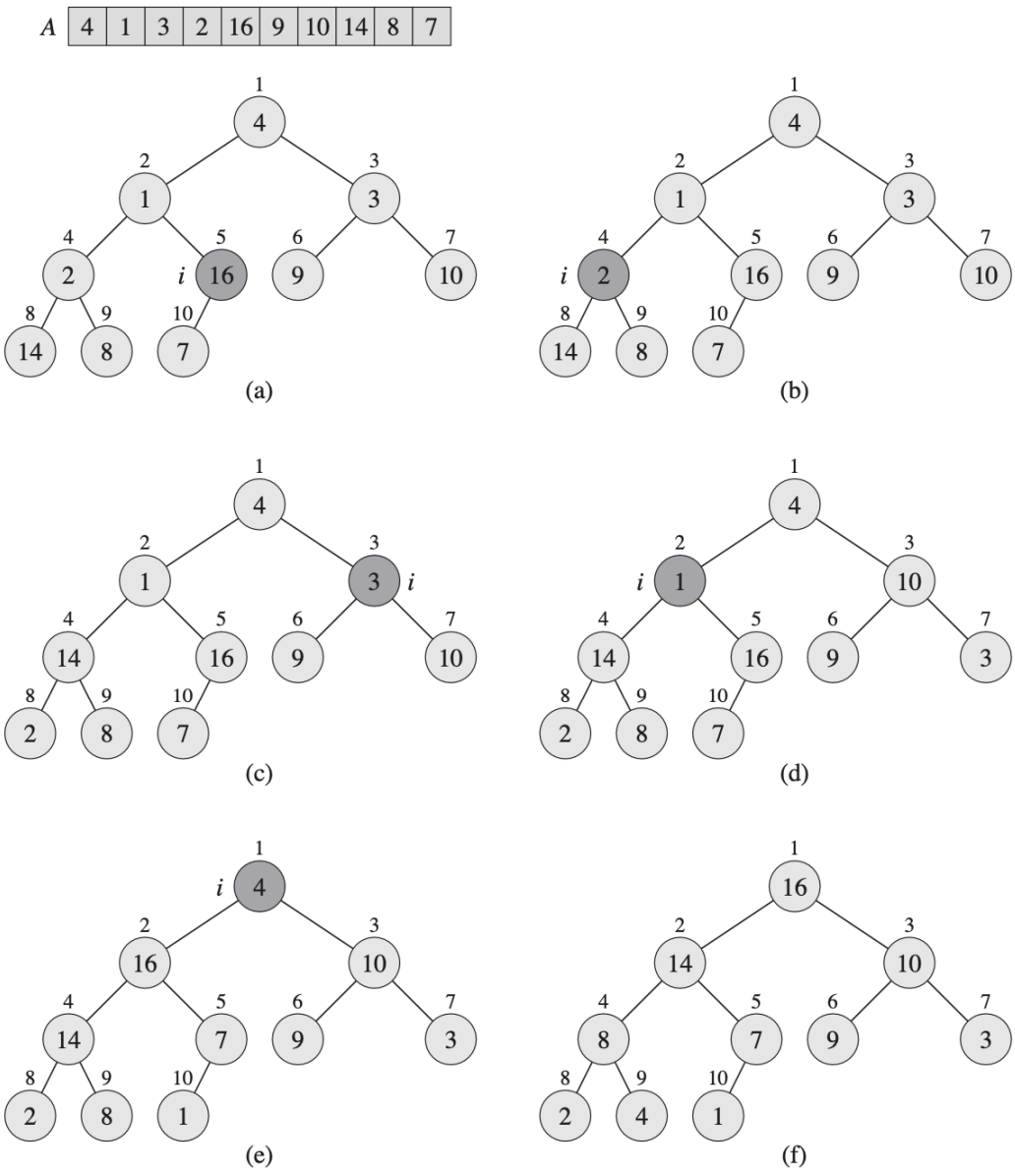
  BuildMaxHeap(A):

  A.heap_size = A.length

  for $i = \lfloor A.length/2 \rfloor$ down to 1:

  MaxHeapify(A, i)

- Worst-Case Running Time - $\Theta(\log n)$



3. Dictionaries and Binary Search Trees

3.1 Dictionary

- Objects - A collection of key-value pairs
- Operations
  - *Search(D, k)* - Return x in D s.t. $x.key = k$, or NIL if no such x in D
  - *Insert(D, x)* - Insert x in D; if some y in D has $y.key = x.key$, replace y by x
  - *Delete(D, x)* - Remove x from D

Note: k is a key, x is a node

Time Complexity with List

	Unsorted List	Sorted List
Search(D, k)	$\Theta(n)$	$\Theta(\log n)$
Insert(D, x)	$\Theta(1)$	$\Theta(n)$
Delete(D, x)	$\Theta(n)$	$\Theta(n)$

3.2 Binary Search Trees

Binary Search Tree Property - For every node x, $x.key$ is greater than every key in left sub-tree of x, and $x.key$ is less than every key in right sub-tree of x

- A binary tree that satisfies the binary search tree property
- Each node has at most two children
- Nodes doesn't have to be full, unlike binary heaps

NOTE: BST property is **recursive**

- Minimum value of a BST - left most node with no left child
- Maximum value of a BST - right most node with no right child
- Information - $x.key$, $x.left$, $x.right$, $x.p$ (the parent node)

Traversal

- Inorder - Depth First: left, root, right
- Preorder - Depth First: left, root, right
- Postorder - Depth First: left, root, right
- Level-by-level - Breath-First

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

Worst-Case Running Time: $\Theta(n)$

3.3 Implementation

Finding Minimum

- Implementation

  BSTMin(x): Return the node with the minimum key in the tree rooted at x

  1. Start from the x
  2. Keep going left, until the left child is NIL

     ```
     while x.left != NIL:
         x = x.left
     ```
  3. Return the final node

     ```
     return x
     ```

- Worst-Case Running Time - $\Theta(n)$

Search

- Implementation

  BSTSearch(root, k)

  1. Start from the root
  2. If going to **NIL**, not found

     ```
     if root == NIL:
         return NIL
     ```
  3. If equal, return the current node

     ```
     else if root.key == k:
         return root
     ```
  4. If k is smaller than the key of the current node, go left

     ```
     else if root.key > k:
         return BSTSearch(root.left, k)
     ```
  5. if k is larger than the key of the current node, go right

     ```
     else:
         return BSTSearch(root.right, k)
     ```

- Worst-Case Running Time - $\Theta(h)$, h can be as big as n when only one side

Insert

- Implementation

  BSTInsert(root, x)

  1. Start from the root
  2. When next position is **NIL**, insert there

     ```
     if root == NIL:
         root = x
     ```
  3. Go down, left and right like what we do in BSTSearch

     ```
     else if root.key > x.key:
         root.left = BSTInsert(root.left, k)
     ```

```
        else if root.key < x.key
            root.right = BSTInsert(root.right, k)
```

4. If find equal key, replace the node

```
        else:
            replace root with x
```

- Worst-Case Running Time - $\Theta(h) = \Theta(n)$

NOTE: If inserting the same set of data in a diff. sequence, the shape of the BST will be diff.

3.3 Successor

Idea

Successor of x - the node with the smallest key larger than x

1. x has a **right** child
   - Its successor must be the minimum in the right subtree of x
2. x does not have a **right** child
   - x is the maximum in some subtree A
   - So, the successor y of x is visited right after finishing A in inorder traversal
   - So A must be the left subtree of y

Implementation

Successor(x):

1. Case one

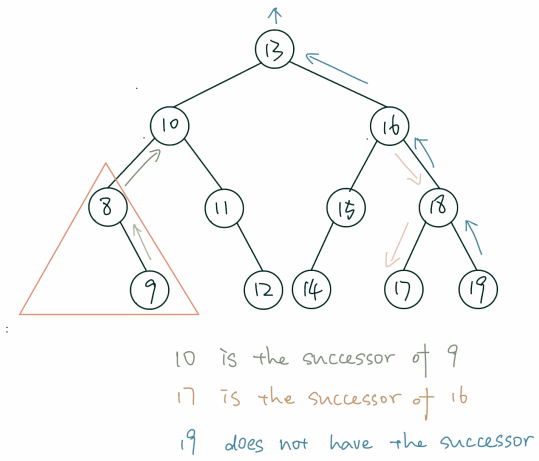
```
if x.right != NIL:
    return BSTMIN(x.right)
```

2. Case two
   - Go up to $x.p$

```
y = x.p
```

   - If x is a right child of $x.p$, keep going up
   - If x is a left child of $x.p$, stop, $x.p$ is what we find

```
while y != NIL and x == y.right:
    x = y
    y = y.p
return y
```



- Worst-Case Running Time - $\Theta(n)$
  - Case 1 - *BSTMin* takes $\Theta(h)$ or $\Theta(n)$
  - Case 2 - Going from a leaf at height h to root. Also $\Theta(n)$

NOTE: in case 2, if we reach the root of the tree, then x does not have a successor

3.4 Deletion

Idea

1. x has no child
   - Just delete it
2. x has one child
   - Delete x
   - Promote x's only child to x's spot, together with the child's subtree
3. x has two child
   - Delete x
   - Replace x by its successor y
   - Remove y from its original position
     - y must be the minimum of right subtree of x, so y has **at most one** child

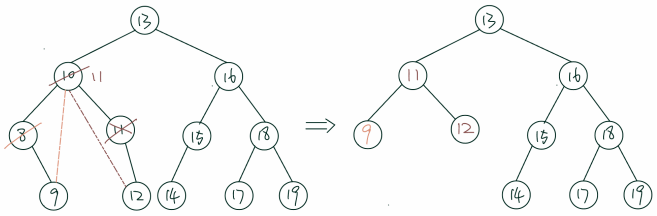
Implementation

- For case two

```
BSTTransplant(root, x, y): promote y to x's position
    If x.p == NIL:  # if x is the root
        root = y     # y replaces x as root
    else if x == x.p.left:
        x.p.left = y
    else:
        x.p.right = y
    if y != NIL:
        y.p = x.p    # update the parent of y
```

- Complete

```
BSTDelete(root, x):
    if root == NIL:
        pass    # do nothing
    else if root.key > x.key:
        BSTDelete(root.left, x)
    else if root.key < x.key:
        BSTDelete(root.right, x)
    else if x.left == NIL:    # Case 1 and 2
        BSTTransplant(root, x,  x.right)
    else if x.right == NIL:
        BSTTransplant(root, x, x.left)
    else:
        y = BSTExtractMin(root.right)      # successor of x.key
```



- Worst-Case Running Time - $\Theta(n)$

4. AVL Trees and Augmented Data Structure

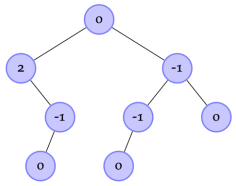
4.1 AVL Trees

- In a complete binary tree, the heights of the left and right sub-trees of any node differ by at most 1
- Balance Factor - The height of the right sub-tree minus the height of the left sub-tree

$$BF(n) = n.right.height - n.left.height$$

- AVL Invariant - A node n satisfies the AVL invariant if $-1 \leq BF(n) \leq 1$
- AVL-Balanced - A binary tree that all of its nodes satisfy the AVL invariant
- AVL Tree - A BST which is AVL-balanced

NOTE: Height is measured by the number of levels (number of nodes in the longest path from the root to a leaf)



Each node is labelled by its balance factor.

Property

- If $BF(x) = +1$, x is right heavy
- If $BF(x) = -1$, x is left heavy
- If $BF(x) = 0$, x is balanced

NOTE: The height of an AVL tree with n nodes is at most $1.44 \log_2(n+2)$, thus $h \in \Theta(\log n)$

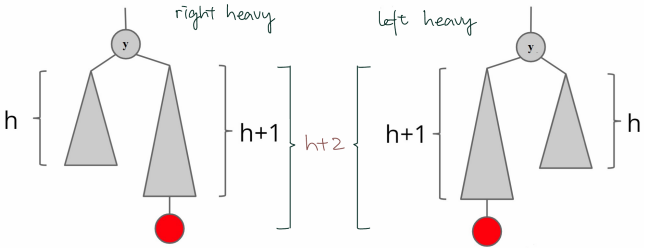
We also store $x.height$ in each node x

- AVLSearch: Same as BSTSearch
- AVLInsert and AVLDelete:
  1. Maintain AVL invariant for all affected nodes
  2. Maintain the BST property
  3. Update height of the affected nodes accordingly

4.2 Rotation

Observation

1. Inserting/deleting a node can only change the balance factors of its ancestors
2. The balance factor of the affected nodes changes by at most 1. The balance factor of the affected nodes can only be -2 or 2



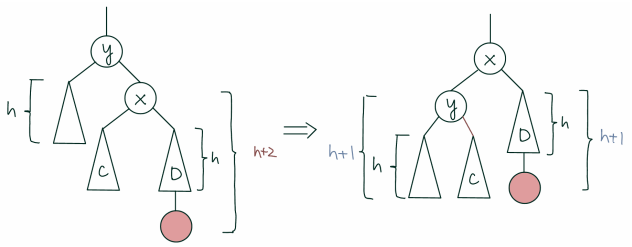
Assumption

y is the lowest ancestor that became unbalanced

i.e. all decedents of y satisfy the AVL invariant

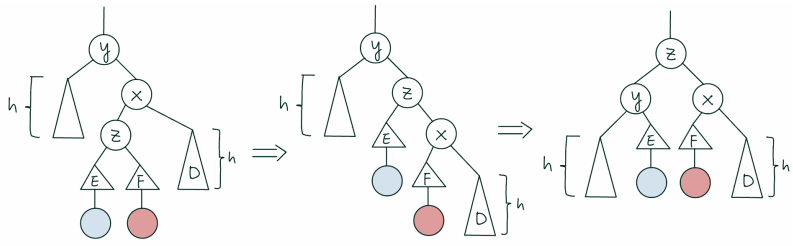
Idea

1. Inserting a new node into the right-subtree of y while y is right-heavy before insertion
    1. Insert the new node to the right subtree of x (x is the right child of y)



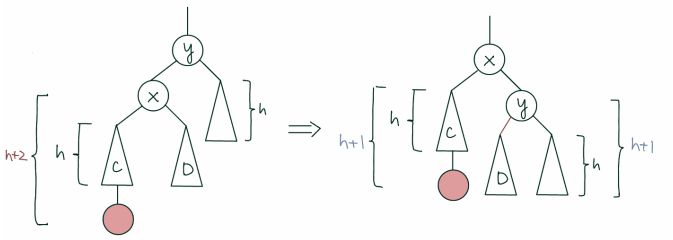
left rotation

    2. Insert the new node to the left subtree of x



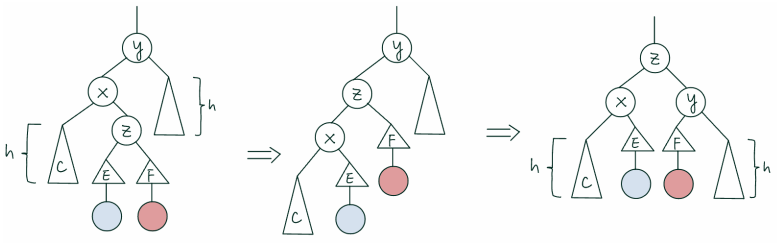
right-left rotation

2. Inserting a new node into the left-subtree of y while y is left-heavy before insertion
    1. Insert the new node to the left subtree of x (x is the left child of y)



right rotation

    2. Insert the new node to the right subtree of x



left-right rotation

4.3 Implementation

- Implementation

  AVLInsert(root, x) / AVLDelete(root, x):

  1. Insert/Delete like a BST -> $\Theta(h) = \Theta(\log n)$

     if root == NIL: *# example for insertion*
     root = x
     else if root.key > x.key:
     AVLInsert(root.left, x)
     else:
     AVLInsert(root.right, x)

  2. If still balanced, return

     BF = root.right.height - root.left.height

  3. Else (need re-balancing) ***rotation*** -> $\Theta(1)$

     if BF < -1 **or** BF > 1:
     # fix the imbalance for the root node
     fix_imbalance(root)

  4. Updated the height of affected nodes -> $\Theta(\log n)$

     root.height = max(root.right.height, root.left.height) + 1

- Worst-Case Running Time - $\Theta(\log n)$

```python
def fix_imbalance(D):
    # Check balance factor and perform rotations
    if D.balance_factor == -2:
        if D.left.left.height == D.right.height + 1:
            right_rotate(D)
        else:  # D.left.right.height == D.right.height + 1
            left_rotate(D.left)
            right_rotate(D)
    elif D.balance_factor == 2:
        # left as an exercise; symmetric to above case
        ...


def right_rotate(D):
    # Using some temporary variables to match up with the diagram
    y = D.root
    x = D.left.root
    A = D.left.left
    B = D.left.right
    C = D.right

    D.root = x
    D.left = A
    # Assume access to constructor AVLTree(root, left, right)
    D.right = AVLTree(y, B, C)
```

4.4 Augmentation

Augmented Data Structure - A modification of an existing data structure by storing additional information and/or performing addition operation

1. Choose data structure to augment
2. Determine additional information
3. Check additional information can be maintained during each original operation
4. Implement new operations

AVL Tree Augmentation

In augmenting AVL trees, if the additional information of a node only depends on the information stored in its children and itself, this information can be maintained efficiently during AVLInsert and AVLDelete without affecting their worst-case running time

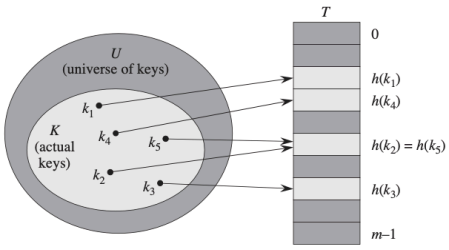
5. Hash Tables

5.1 Hash Table

- Universe $\mathcal{U}$ - The set of all possible keys
- Hash Function - A function from the set of all possible keys to integers between 0 and m - 1

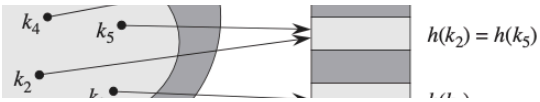
$$h : \mathcal{U} \to \{0,1,...,m-1\}$$

- Hash Table - A data structure containing an array of length m and a hash function h as before
  - $h(k)$ maps a key k to one of the m positions in hash table T
  - i.e. $h(k)$ is the index at which the key k us stored
  - Each array location called a slot or a bucket



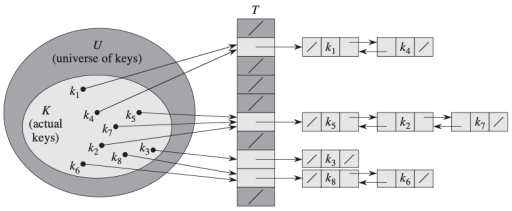
collision

- If $m \geq |\mathcal{U}|$, then there exists a has has function h which maps each key to a unique slot
  - such a function is called a perfect hash function
  - typically the number of possible keys is much bigger than the number of array slots
- If $m < |\mathcal{U}|$, then at least one collision occurs



5.2 Chaining

- Each bucket in the array points to a linked list of key-value pairs



Insertion

1. Compute $h(k)$. Set $i = h(k)$
2. Search the linked list stored at $T[i]$ to check whether an element with key k already exists
3. If so, replace the existing value with v. If not, insert a new node to the head of the list

Search

1. Compute $h(k)$. Set $i = h(k)$
2. Access index i in the table
3. Search the linked list stored at $T[i]$

Deletion

1. Compute $h(k)$. Set $i = h(k)$
2. Search the linked list stored at $T[i]$
3. If an element with key k is found, delete it from the list

Worst-Case Running Time

All elements in the table are hashed into the same bucket -> $\Theta(n)$

n is the total number of elements stored in the hash table

We assume that the hash value $h(k)$ is computed in constant time -> $\Theta(1)$

Average-Case running Time

Let $t_{m,n}(k)$ denote the number of steps executed by HashSearch to find k in a hash table containing m buckets and storing n elements

Simple Uniform Hashing Assumption (SUHA) - Any key equally likely to hash to any bucket

$$\mathbb{E}[t_{m,n}(k)] = 1 + \text{expected running time of searching for } k \text{ in } L_i \ \ (i = h(k) \in \Theta(1))$$

- In an Unsuccessful Search
  - Probability that key k is based to bucket i

$$Pr[h(k) = i] = \frac{1}{m}$$

  - The expected length of the linked list stored at a bucket i

$$\mathbb{E}[len_i] = \frac{n}{m}$$

  - Load Factor of a hash table

$$\alpha = \frac{n}{m}$$

$$\mathbb{E}[t_{m,n}(k)] = 1 + \frac{n}{m} = 1 + \alpha \in \Theta(1 + \alpha)$$

NOTE: If $\frac{n}{m} \in \Theta(1)$, then the average-case running time is also in $\Theta(1)$

- In an Successful Search
  - The probability that k is k_i is $\frac{1}{n}$

$$\mathbb{E}[t_{m,n}(k)] = \frac{1}{n}\sum_{i=1}^{n} S_i = \ldots = 1 + \frac{n}{2m} - \frac{1}{m} = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} \in \Theta(1+\alpha)$$

  where S_i denotes the expected number of steps to find k_i

All the function have $\Theta(\alpha + 1)$ average-case running time

If the number of hash-table slots is at least proportional to the number of elements in the table, then all dictionary operations can be implemented with constant average running time

5.3 Open Addressing

- Store all items directly in T (no chaining). If a collision occurred, look for another free spot in some systematic manner called probing

NOTE : The number of keys stored in the hash table cannot exceed the length of the array

Search

Follow the same probing approach used for insertion. Search returns None when encounters the first bucket that stores None

NOTE : Searching for an item requires examining more than just one spot

Probing

- Linear - Examine a linear sequence of slots

$$(h(k) + i) \mod m, \text{ for an integer } i \geq 0$$

- Quadratic - Examine a non-linear sequence of slots

$$(h(k) + c_1 \times i + c_2 \times i^2) \mod m, \text{ for an integer } i \geq 0$$

- Double Hashing - Use a second hash function to generate step values that depend on key

$$(h_1(k) + i \times h_2(k)) \mod m, \text{ for an integer } i \geq 0$$

Average-Case running Time

- Unsuccessful search - $\frac{1}{1-\alpha}$
- Successful search - $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$

- In practice open addressing works best when $\alpha < 0.5$

5.4 Hash Function

Decision Method

$$h(k) = k \mod m$$

- Pitfall : sensitive to the value of m; constrains the table size
- $k \mod 2^p$ depends on last p bits of k
- Good choice of m - A prime not too close to an exact power of 2x

NOTE : Keys are not spread out

Multiplication Method

1. Multiple k by a real constant $0 < A < 1$
2. Let x be the fractional part of $k \times A$ (note that $0 < x < 1$)
3. $h(k) = \lfloor m \times x \rfloor$

Not sensitive to the value of m

6. Amortized Analysis

6.1 Dynamic Arrays

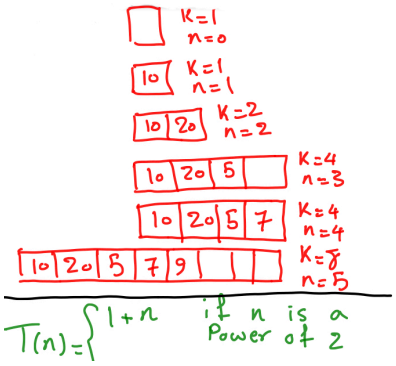
n - number of elements stored in the array

k - size of the array

Assumption: start with an array with $k = 1$ and $n = 0$

Append(A_1, x): Store element x in the first free position of array A_1. If A_1 is full, create new array A_2 twice the size of A_1, copy over all elements in A_1 to A_2, then append x

if n == k:
    Create a new array A_2 with size $2 \times k$
    Copy all the elements from A_1 to A_2
    k = k × 2
    n = n + 1
    A_2[n] = x
else:
    n = n + 1
    A_1[n] = x



6.2 Amortized Analysis

General Idea

- Worst-Case Sequence Complexity
  - total cost of performing a sequence of m operations in the worst case
  - Denote by T_m^{sq}, where m is the number of operations in the sequence

- Amortized Sequence Complexity:

$$\frac{\text{worst-case sequence complexity}}{m} = \frac{T_m^{sq}}{m}$$

Amortized sequence complexity for Dynamic Arrays - $\Theta(1)$

NOTE : considers an multiple operations; involves NO probability

6.3 Aggregate Method

1. Find the **total** cost of performing a sequence of operations in the worst case.
2. Divide the total cost by the number of operations in the sequence

c_i - the actual cost of i-th operations a sequence of m operations. Then

$$T_m^{sq} = \sum_{i=1}^{m} c_i$$

Suppose we perform a sequence of m Append operations

$T(i)$ - worst-case running time of the i-th operation in the sequence

Let $T'(i) = T(i) - 1$, then

$$\begin{aligned}
\sum_{i=1}^{m} T(i) &= \sum_{i=1}^{m} 1 + T'(i) \\
&= m + \sum_{i=1}^{m} T'(i) \\
&= m + \sum_{j=0}^{\lfloor \log_2 m \rfloor} 2^j \quad \text{since } 2^0 \le m \le 2^{\log_2 m} \\
&= m + 2^{\lfloor \log_2 m \rfloor + 1} - 1 \in \Theta(m)
\end{aligned}$$

Therefore,

$$\begin{aligned}
T_m^{sq} &= \sum_{i=1}^{m} T(i) \in \Theta(m) \\
&\Longrightarrow \frac{T_m^{sq}}{m} \in \Theta(1)
\end{aligned}$$

6.4 Accounting Method

Instead of considering the actual cost of each operation, assign an amortized cost a_i to each operation so that:

- Calculation of $\sum_{i=1}^{m} a_i$ is straightforward
- We can proof $\sum_{i=1}^{m} a_i \geq \sum_{i=1}^{m} c_i = T_m^{sq}$

How **amortized costs** should be assigned?

- Think of it as maintaining a bank account which is used to pay for the number of steps each operation executes.
- Some operations are charged a **more** than their **actual** cost.
  The surplus is deposited as **credit** into the bank account for later use.
- Some operations are charged **less** than their **actual** cost.
  The deficit is paid for by the savings (**credit**) in the bank account.
- The amortized cost to each operation must be set *large enough* that the **balance** in the bank account always remains **non-negative**, but *small enough* that no operation is charged significantly more than its actual cost.
- **Important:** The extra amortized charges to an operation does **not** mean that the operation really takes that much time. It is just a method of accounting that makes the analysis easier.

Step 1: Find appropriate **amortized** costs for all operation.

Step 2: Show that for all $k \in \mathbb{Z}^+$, and all possible sequences $\sum_{i=1}^{k} a_i \geqslant \sum_{i=1}^{k} c_i$.

- We might need to identify and prove a **credit invariant** to show that the above statement holds.
- After trying to prove the credit invariant and/or the inequality, you might realize that the amortized costs you considered are insufficient.
  You then must adjust the costs accordingly and re-do Step 2.

Step 3: Find an upper-bound for T_m^{sq} based on total amortized costs.

Example

Augmented Stack

- Push(S, x) - push object x onto stack S
  - actual cost - 1
- Pop(S) - pop the top of stack S and return the popped object. Calling Pop on an empty stack generates an error
  - actual cost - 1
- MutiPop(S, k) - remove the k top objects of stack S, popping the entire stack if the stack contains fewer than k objects
  - actual cost - $min(n, k)$

Step 1: Find appropriate amortized costs.

Push **amortized** cost: 2
Pop **amortized** cost: 0
MultiPop **amortized** cost: 0

Step 2: Show that for all $k \in \mathbb{Z}^+$, and all possible sequences $\sum_{i=1}^{k} a_i \geqslant \sum_{i=1}^{k} c_i$.
Proof:
Suppose we use a dollar bill to represent each unit of cost.
When we push an object x, we use 1 dollar to pay the actual cost of the push and x is left with a credit of 1 dollar.
When a *Pop* or *MultiPop* is executed on x, we pay its cost using the credit stored for x.
It is not possible to pop an element that has not been pushed!
So the total credit of stack objects is non-negative for every possible sequence.
Total credit is equal to the difference between total amortized cost and total actual cost.
That is, for any sequence of k *Push*, *Pop* and *MultiPop* operations we have

$(\sum_{i=1}^{k} a_i - \sum_{i=1}^{k} c_i) \geqslant 0,$

meaning that $\sum_{i=1}^{k} a_i \geqslant \sum_{i=1}^{k} c_i$ always holds.

Step 3: Find an upper-bound for T_m^{sq} based on amortized costs:

$$T_m^{Sq} = \sum_{i=1}^{m} c_i \leqslant \sum_{i=1}^{m} a_i \leqslant \sum_{i=1}^{m} 2 = 2m$$

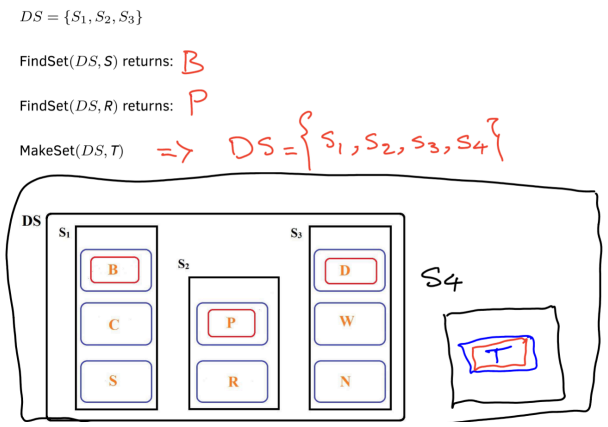
$$\frac{T_m^{Sq}}{m} \leqslant \frac{2m}{m} \Rightarrow \frac{T_m^{Sq}}{m} \in O(1)$$

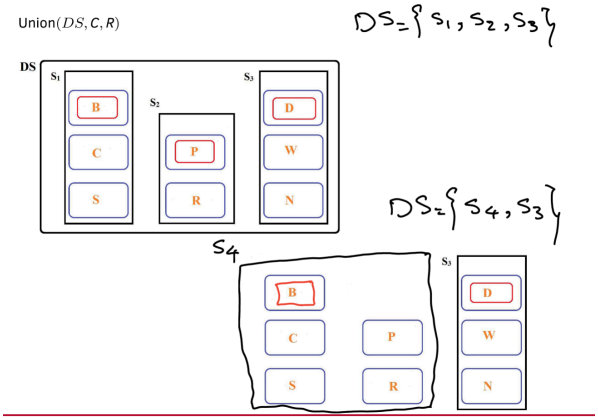
Amortized sequence complexity for **Stacks**: $O(1)$

7. Disjoint Sets

7.1 Disjoint Set

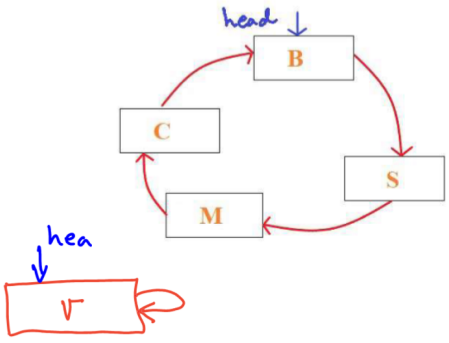
- Objects - A collection of nonempty disjoint sets $S = \{S_1, S_2, \ldots, S_k\}$
  - Each set is identified by a unique element called it representative
- Operations
  - *MakeSet(DS, v)* - Given element v that does not already belong to one of the sets, create a new set $\{v\}$. The new item is the representative element of the new set
  - *FindSet(DS, x)* - Return the representative element of the set containing x
    - NOTE : We know the set that contains x, we just need to find its representative
  - *Union(DS, x, y)* - Given two items x and y, merge the sets that contain these items. The representative of the new set might be one of the two representatives of the original sets, one of x or y, or nothing completely different
    - NOTE : If both x and y belong to the same set already, operation has no effect





7.2 Circularly-Linked Lists

- One circularly-linked list for each set
- Head of the linked list is the representative of the set



MakeSet

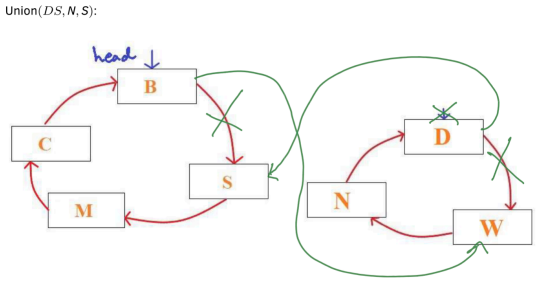
- Create a new linked list with a node x storing element v
- set $x.next = v$

FindSet

- Follow the links until reaching the head node r
- return r

Union

- locate the head of each list by calling
  l1 = FindSet(DS, x)
  l2 = FindSet(DS,y)
- Exchange $l_1.next$ and $l_2.next$



Time Analysis

- Worst-Case

MakeSet	FindSet	Union
$\Theta(1)$	$\Theta(L)$	$\Theta(L_1+L_2)$

NOTE : L is the length of the list containing x

- Amortized

Consider a bad sequence where you have $m/4$ MakeSet, then $m/4-1$ Union, then $m/2+1$ Find Set

After $\frac{m}{4}$ MakeSet and $\frac{m}{4}-1$ Union, there will be a list with size $\frac{m}{4}$

For each FindSet, run time is $\Theta(\frac{m}{4})$ in the worst case

For $(\frac{m}{2}+1)$ FindSet, total run time is

$$(\frac{m}{2}+1)\times\frac{m}{4}\in\Theta(m^2)$$

For $\frac{m}{4}$ MakeSet, total run time is $\Theta(m)$

For $(\frac{m}{4}-1)$ Union, total run time is

$$1+2+3+\ldots+(\frac{m}{4}-1)\in\Theta(m^2)$$

Then,

$$T_m^{sq}\in\Theta(m^2)$$
$$\frac{T_m^{sq}}{m}\in\Theta(m)$$

7.3 Linked Lists with Pointer to Head

- One linked list for each set
- All nodes in a list, except the head node, have a pointer to the head of the list
- Head of the linked list is the representative of the set
- The head node has a pointer to the tail of the list

MakeSet

- Create a new linked list with a node x storing element v
- Set the properties of x

```
x.rep = x
x.tail = x
```

FindSet

```
return x.rep
```

Union

- Append one list to the tail of the other
- Update the tail pointer
- Update the pointers to the head

Worst-Case Running Time
$\Theta(1)$
$\Theta(1)$
$\Theta(L_1 + L_2)$

Amortized Time Analysis

Consider a bad sequence where you have $(m/2+1)$ MakeSet, then $(m/2-1)$ Union, always appending longer list to the end of single-element list

For Updating head pointers

$$1 + 2 + 3 + \ldots + (\frac{m}{2} - 1) \in \Theta(m^2)$$

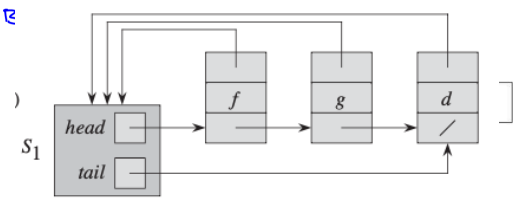
For $(\frac{m}{2} + 1)$ MakeSet, total run time is $\Theta(m)$

Then,

$$T_m^{sq} \in \Theta(m^2)$$
$$\frac{T_m^{sq}}{m} \in \Theta(m)$$

7.3 Linked Lists with Pointer to Head and Union-By-Weight

- One linked list for each set
- All nodes in a list, except the head node, have a pointer to the head of the list
- Head of the linked list is the representative of the set
- The head node has a pointer to the tail of the list
- The head node stores the size of each list



MakeSet

- Create a new linked list with a node x storing element v
- Set the properties of x

```
x.rep = x
x.tail = x
x.size = 1
```

FindSet

```
return x.rep
```

Union

- Append the shorter list to the longer list
- Update the tail pointer
- Update the size of the new list
- Update the pointers to the head

Worst-Case Running Time
$\Theta(1)$
$\Theta(1)$
$\Theta(L_1 + L_2)$

Amortized Time Analysis

- Consider a sequence of m operations.
- Let n be the number of MakeSet operations in the sequence. So there are **never more than** n elements in total.
- For some arbitrary element x, we want to prove an **upper bound** on the number of times that $x.rep$ can be updated.

- $x.rep$ gets updated only when the set that contains x is **unioned** with a set that is **not smaller**.
- So each time $x.rep$ is updated, the size of the resulting set must be at least **double** the size of the list containing x.
- That is, every time $x.rep$ is updated, set size **doubles**.
- There are only n elements in **total**, so we can double at most Lg n times
- So $x.rep$ cannot be updated more than Lg n times

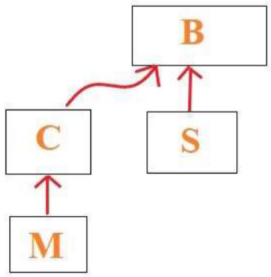
For a sequence of $(m/2 + 1) = n$ MakeSet, then $(m/2 - 1)$ Union

Total number of $x.rep$ updates in at most $n \log n$

For the other operations, total run time is $\Theta(m)$

$$T_m^{sq} \in \Theta(m + n \log n) \in \Theta(m \log m) \implies \frac{T_m^{sq}}{m} = \Theta(\log m)$$

7.4 Trees



- One inverted tree for each set
- Each element points to its parent only (or to itself if it's the root)
- The root of the tree is the representative of the set and points to itself

NOTE : Trees are not necessarily binary, number of children of a node can be arbitrary

MakeSet

- Create a tree with a node x storing element v
- Set the properties of x

```
x.p = x
```

FindSet

- Trace up the parent pointer until the root r is reached
- return r

Union

- locate the head of each list by calling

```
l1 = FindSet(DS, x)
l2 = FindSet(DS,y)
```

- Let one tree's root point to the other tree's root

```
l1.p = l2
```

Worst-Case Running Time
$\Theta(1)$
$\Theta(h)$~$\Theta(n)$, where h is the height of the tree containing x
$\Theta(h_1 + h_2)$

Amortized Time Analysis

Consider a bad sequence where you have $m/4$ MakeSet, then $(m/4 - 1)$ Union and $(m/2 + 1)$ FindSet

After $\frac{m}{4}$ MakeSet and $\frac{m}{4} - 1$ Union, there will be a long chain with $\frac{m}{4}$ elements

For $(\frac{m}{2} + 1)$ FindSet, total run time is

$$\frac{m}{4}(\frac{m}{2} + 1) \in \Theta(m^2)$$

Then,

$$T_m^{sq} \in \Theta(m^2)$$
$$\frac{T_m^{sq}}{m} \in \Theta(m)$$

7.4 Trees with Union-by-rank

- One inverted tree for each set
- Each element points to its parent only (or to itself if it's the root)
- The root of the tree is the representative of the set and points to itself
- The root stores the rank of the tree
- Rank - the height of the tree (for now)

MakeSet

- Create a tree with a node x storing element v
- Set the properties of x

```
x.p = x
x.rank = 0
```

FindSet

- Trace up the parent pointer until the root r is reached
- return r

Union

- locate the head of each list by calling

```
l1 = FindSet(DS, x)
l2 = FindSet(DS,y)
```

- Let the root with lower rank point to the root with higher rank
- if the two roots have the same rank, choose either root as the new root and increment the rank of the new root by one

Worst-Case Running Time
$\Theta(1)$
$\Theta(h)$~$\Theta(\log n)$
$\Theta(h_1 + h_2)$

Amortized Time Analysis

Theorem Let T he a tree generated by a series of MakeSet and Union operations using the union-by-rank heuristic. Let r be the rank of T, and n be the number of nodes in T. Then

$$2^r \leq n \implies r \leq \log_2 n$$

Consider a sequence of operations. Then each operation in the sequence takes at most $\log n$

For a sequence of $m/4 = n$ MakeSet, then $(m/4 - 1)$ Union and $(m/2 + 1)$ FindSet

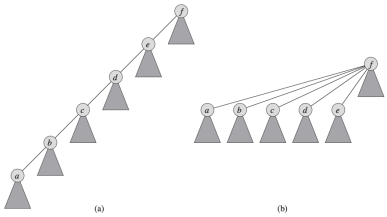
Then,

$$T_m^{sq} \in \Theta(m \log m)$$

$$\frac{T_m^{sq}}{m} \in \Theta(log m)$$

7.4 Trees with Path-Compression

- When calling FindSet for some x, keep track of nodes visited on path from x to the root
- Once the root is found, make each visited node to point directly tot he root



MakeSet

- Create a tree with a node x storing element v
- Set the properties of x

```
x.p = x
```

FindSet

- Trace up the parent pointer until the root r is reached
- once the root is found, make each visited node to point directly to the root
- return r

Union

- locate the head of each list by calling

```
l1 = FindSet(DS, x)
l2 = FindSet(DS,y)
```

- Let one tree's root point to the other tree's root

```
l1.p = l2
```

Worst-Case Running Time
$\Theta(1)$
$\Theta(h)$
$\Theta(h_1 + h_2)$

Amortized Time Analysis

For a sequence of n MakeSet (and hence at most $n-1$ Union) and f FindSet:

$$T_m^{sq} \in \Theta(n + f \times (1 + \log_{2+\frac{f}{n}} n))$$

7.5 Trees with Union-by-Rank and Path-Compression

Combine previous two part but path compression does not maintain height info

- A node's rank is an upper-bound on its height

For a sequence of m operations with n MakeSet (so at most $n-1$ union)

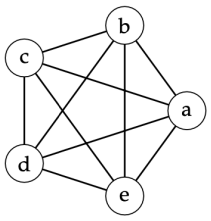
$$T_m^{sq} \in \mathcal{O}(m \times \alpha(n))$$

where $\alpha(n)$ is the inverse Ackerman function, which grows super slowly.
So we can treat it as constant

$$T_m^{sq} \in \mathcal{O}(m)$$

8. Graph and BFS

8.1 Graphs



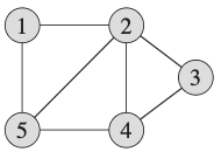
Category

- Graph - a tuple of two sets $G = (V, E)$, were V is a set of vertices, and E is a set of edges
  - E - a set of tuples (v_i, v_j) where $v_i, v_j \in V$
- Directed Graph - a graph in which edges have orientations
  - i.e. $(v_i, v_j) \neq (v_j, v_i)$
- Undirected Graph - a graph in which edges have no orientation
  - i.e. $(v_i, v_j) = (v_j, v_i)$
- Weighted Graph - a graph in which a weight is assigned to each edge
  - NOTE : Weights of edges might represent costs, lengths or capacities, etc.

Basic Definition

- Adjacent - two vertices are adjacent if there is an edge between them
  - they are neighbours
- Path - between vertices u and w is a sequence of distinct edges $(V_0, v_1), \ldots, (V_k, v_{k-1})$ where $u = v_0$ and $w = v_k$, and for all tuples (v_i, v_j) in the sequence, v_i and v_j are distinct
  - Length of path - the number of edges in the path
- Distance - the length of the shortest path between the two vertices
  - The distance between a vertex and itself is always **0**
- Connected Graph - there is a path between every pair of vertices in the graph
  - Minimum number of edges in an undirected **connected** graph with n vertices - $n - 1$
- Cycle - a path from an vertex to itself
  - NOTE : length of a cycle is at least 3
- Tree - connected graph and contains no cycle
- Forest - a collection of disjoint trees

8.2 Data Structures



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Adjacency Matrix

Let $V = \{v_1, \ldots, v_n\}(|V| = n)$

Let A be a $n \times n$ matrix, $A = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$

For an **undirected** graph, adjacency matrix is **symmetric**

For a **weighted** graph, if $(v_i, v_j) \in E$, store $w(v_i, v_j)$ in $A[i, j]$;

otherwise, store $-1/0/\infty$

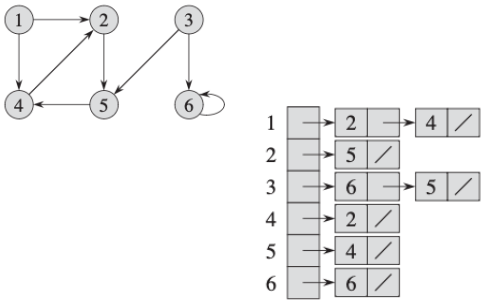
Adjacency List

Let $V = \{v_1, \ldots, v_n\} (|V| = n)$

Let L be a list of size n

Each position $L[i]$ corresponds with a vertex v_i and stores a list A_i of all vertices that have an edge from v_i

- i.e. A_i includes v_j iff $(v_i, v_j) \in E$



NOTE : For an undirected graph, edge (u, v) is stored twice

Comparison

- *Edge Query* - Given vertices u, v, return whether G contain edge (u, v) or (v, u)
- *Neighbourhood* - Given vertex v, return set of vertices u such that $(u, v) \in E$

	Adjacency Matrix	Adjacency Lists
Space Complexity	$\Theta(n^2)$	$\Theta(n + m)$
Add/Remove a Vertex	$\Theta(n)$	$\Theta(1)$ - add $\Theta(m)$ - remove
Edge Query	$\Theta(1)$	$\mathcal{O}(n)$ ~ $\mathcal{O}(min(m, n))$
Neighbourhood	$\Theta(n)$	$\mathcal{O}(n)$ ~ $\mathcal{O}(min(m, n))$

NOTE : $|V| = n, |E| = m$

8.3 BFS

Starting from s, first visit **all** (unvisited) neighbours of s, then visit **all** (unvisited) neighbours of neighbours of s, and so on, until all reachable vertices are visited

NOTE : BFS in a tree (starting from root) is a level-by-level traversal

Improvement

To avoid visiting a vertex twice -> labelling the visited vertices by colours

- While - Unvisited vertices; Have not been enqueued
- Grey - Encountered vertices; Have been enqueued
- Black - Explored vertices; Have been enqueued and all of their neighbours are enqueued

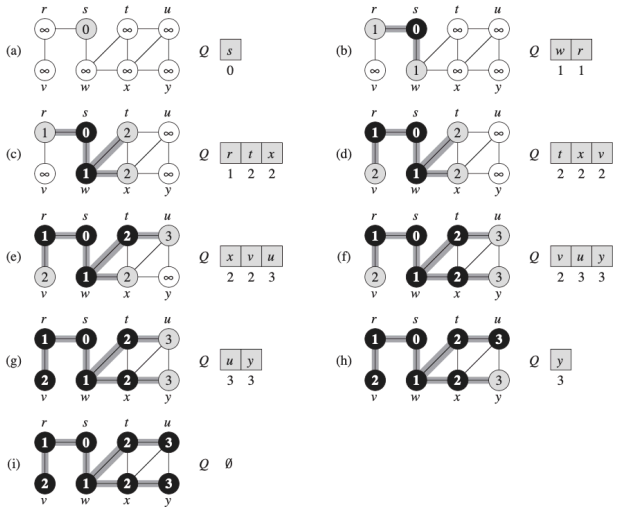
Idea

- Initially all vertices are **white**
- Change a vertex's colour to **grey** the first time visiting it
- Change a vertex's colour to **black** when all its neighbours have been encountered
- Avoid visiting grey or black vertices
- In the end, all vertices that are reachable from the source are **black**

Implementation

BFS(G, s):

```
for each vertex v ∈ V − {s}:  # initialize vertices
    v.colour = White
    # the distance from v to the
    # source vertex of BFS
    v.d = ∞
    # the vertex from which v is encountered
    v.p = NIL
Q = ∅
# start BFS by encountering the source vertex
s.colour = Grey
# distance from s to s is 0
s.d = 0
Enqueue(Q, s)
while Q not empty:
    u = Dequeue(Q)
    for each v ∈ G.adj[u] :
        # only visit unvisited vertices
        if v.colour = White:
            v.colour = Grey
            v.d = u.d + 1   # v is "1-level" father from s than u
            v.p = u         # v is introduced as u's neighbour
            Enqueue(Q, v)
    u.colour = Black    # u is explored as all its neighbours have been encountered
```



NOTE : If G is disconnected graph, then there exists vertices having infinite distance values

To get the shortest-path, follow $.p$ attribute from v to s backward

Worst-Case Running Time

Use adjacency list, the total amount of work

1. When visiting each vertex, Enqueue, Dequeue, assign values to v.colour, v.d, v.p… -> $\Theta(1)$
   - Total - $\Theta(n)$
2. At each vertex, check all its neighbours. Each edge is checked at most twice -> $\Theta(1)$
   - Total - $\Theta(m)$

Total running time - $\Theta(n+m)$

NOTE : $|V| = n, |E| = m$

9. DFS

9.1 DFS

Traverse the depth of any particular path before exploring its breadth. Starting from s, first visit a **neighbour** n_1 of s, then visit **an unvisited neighbour** n_2 of n_1, and so on. If visiting a neighbour n_k of n_{k-1} and all neighbours of n_k are already visited, try another neighbour of n_{k-1}

Improvement

To avoid visiting a vertex twice -> labelling the visited vertices by colours

- While - Unvisited vertices;
- Grey - Encountered vertices;
- Black - Explored vertices; Have been visited and all of their neighbours are explored

Idea

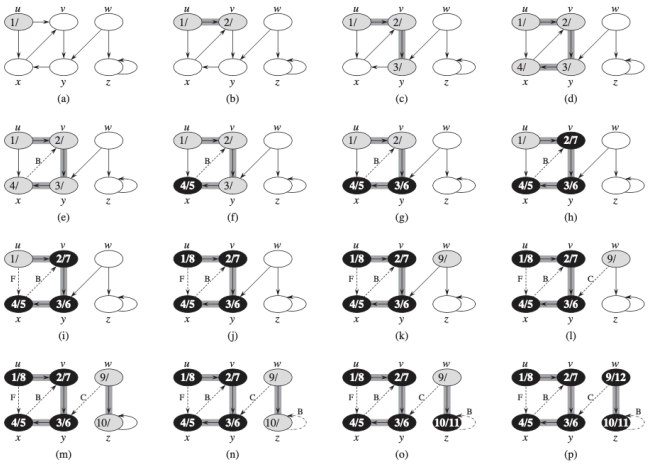
- Initially all vertices are **white**
- Change a vertex's colour to **grey** the first time visiting it
- Change a vertex's colour to **black** when all its neighbours have been encountered
- Avoid visiting grey or black vertices
- In the end, all vertices that are reachable from the source are **black**

- v.p - the vertex from which v is encountered
- Keep Track of two timestamps for each vertex v
  - Discovery Time - the time when v is first encountered, stored in $v.d$
  - Finishing Time - the time when all the neighbours of v have been completely visited, stored in $v.f$

Implementation

DFS(G):

```
for each t ∈ G.V:        # total Θ(n)
    t.colour = White
    t.p = NIL
time = 0
for each s ∈ G.V :
    # make sure no vertex is left unvisited
    if s.colour == White :
        DFSVisit(G, s)
```



DFSVisit(G, s):

```
time = time + 1    # time is a global variable
s.d = time
s.colour = Grey
for each t ∈ G.adj[s] :
    if t.colour == White    # only visit unvisited vertices
        t.p = s       # t is introduced as s's neighbour
        DFSVisit(G, t)
s.colour = Black    # s is explored as all its neighbours have been encountered
time = time + 1
s.f = time   # keep finishing time after exploring all neighbours
```

Worst-Case Running Time

Use adjacency list, the total amount of work

1. When visiting each vertex, assign values to v.colour, v.d, v.p… -> $\Theta(1)$
   - Total - $\Theta(n)$
2. At each vertex, check all its neighbours. Each edge is checked at most twice -> $\Theta(1)$
   - Total - $\Theta(m)$

Total running time - $\Theta(n+m)$

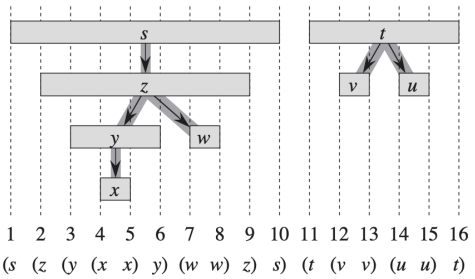
NOTE : $|V| = n, |E| = m$; worst case running time when using an adjacency matrix $\Theta(n^2)$

9.2 Properties

Parenthesis Theorem

In any DFS of a graph G, for any two vertices u and v

- interval $[v.d, v.f]$ contains $[u.d, u.f]$, or
- interval $[u.d, u.f]$ contains $[v.d, v.f]$, or
- $[v.d, v.f]$ and $[u.d, u.f]$ are disjoint



Nesting of Descendant's Interval

In the depth-first forest for a graph G, vertex v is a proper descendant of vertex u iff the interval $[u.d, u.f]$ contains $[v.d, v.f]$. That is

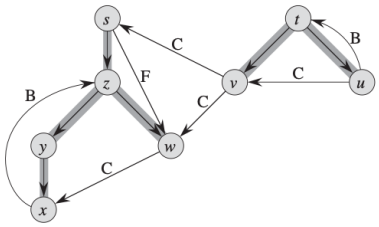
$$u.d < v.d < v.f < u.f$$

9.3 Applications

Detecting Cycles

If we know that there exits and edge between v and u, and also there exists another path between u and v, we can say that there exists a path from u to itself, and therefore the graph has a cycle

1. Consider a graph G
2. Suppose u is an ancestor of v in a DFS-forest of G
3. This means that there exists a path from u to v
4. now assume that there is an edge from v to u
5. Then we can say that a cycle is detected

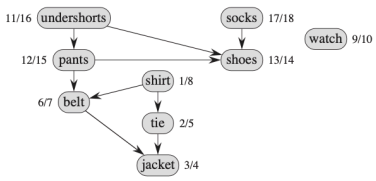


Tree Edge - an edge in the DFS-forest
Back Edge - a non-tree edge pointing from a vertex to its ancestor in the DFS forest
Lemma A graph contains a cycle iff DFS yields a back edge

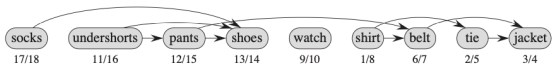
Identify a beck edge - look for edges to Grey vertices. If such an edge exists, it is a back edge

Topological Sort

Is a linear ordering of all its vertices such that if G contains an edge (u, v) then u appears before v in the ordering. If the graph contains a **cycle**, then no linear ordering is possible



NOTE : A topological sort of a graph is an ordering of its vertices along a horizontal line so that all directed edges go from left to right



Theorem

For any pair of distinct vertices $u, v \in V$, if G contains an edge from u to v, then $v.f < u.f$

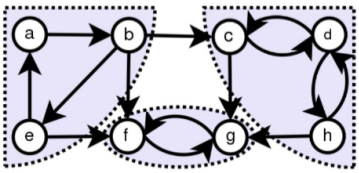
TopologicalSort(G)

1. Call DFS(G)
2. During DFS make sure that G does not contain any circles. At any points if a circle is detected return an empty list
3. As each vertex is finished, insert it onto the front of a linked list
4. Return the linked list of vertices

NOTE : Topological sorting is different from the usual kind of sorting

Finding Strongly Connected Components

Strongly Connected Component - of a directed graph G is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v, there's path from u to v and a path from v to u



G^T - transpose of G, where $G^T = (V, E^T)$

$E^T = \{(v, u) \in V \times V : (u, v) \in E\}$ - consists of the edges of G with directions reversed

NOTE: A directed graph G and its transpose G^T have the same strongly connected component

Let C and C' be distinct SCC in G.

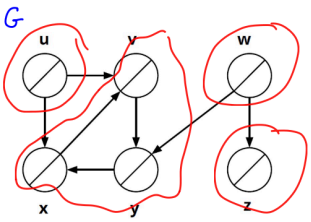
Suppose there is an edge $(u, v) \in E$ s.t. $u \in C$ and $v \in C'$

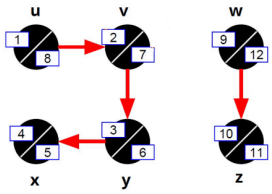
Then $f(C) > f(C')$ where $f(U) = max_{u \in U}\{u.f\}$ i.e. latest finishing time of any vertex in U

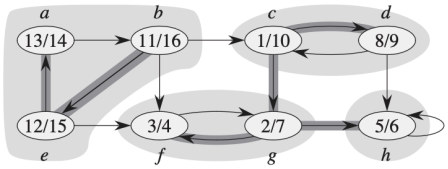
1. Search C before C'
   - since all $y \in C'$ are descendants of u, so $u.f > f(C')$, then $f(C) \geq u.f > f(C')$
2. Search C' before C
   - since C, C' are distinct SCC, thus we must search all vertices in C', then search C

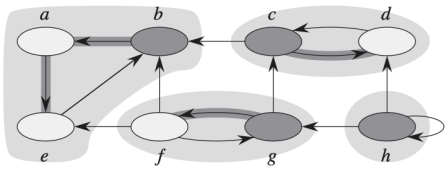
STRONGLY-CONNECTED-COMPONENTS(G)

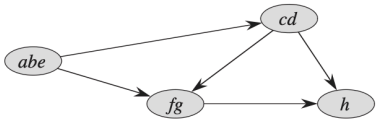
1 call DFS(G) to compute finishing times $u.f$ for each vertex u
2 compute G^T
3 call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component





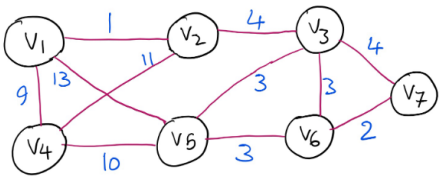






10. Minimum Spanning Trees

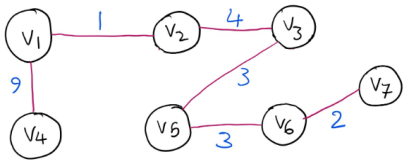
10.1 Minimum Spanning Trees



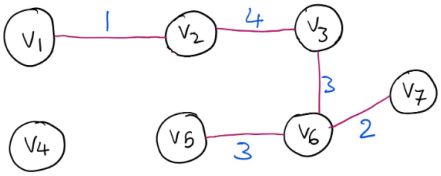
Let $G = (V, E)$ be a connected undirected weighted graph

Minimum Spanning Tree T of G -

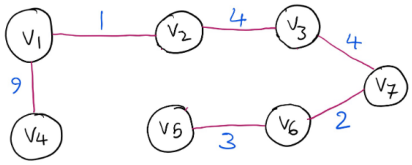
- a sub-graph of G
- includes all vertices of G -> spanning
- is acyclic and connected -> tree
- its total weight is minimized -> minimum



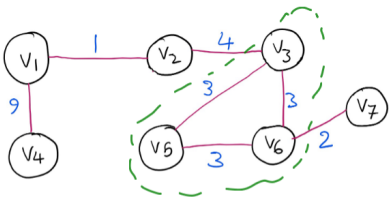
NOTE : A graph might have more than one MST



disconnected



not minimum weight



includes a cycle

Review

Let T be a tree with n vertices. Then

- T has exactly $n - 1$ edges
- Adding one edge to T will create a cycle
- Removing one edge from T will disconnect the graph

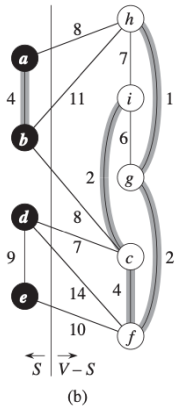
The MST of a connected undirected graph $G = (V, E)$ has $|V|$ vertices and $|V| - 1$ edges

10.2 Initial Implementation

Idea

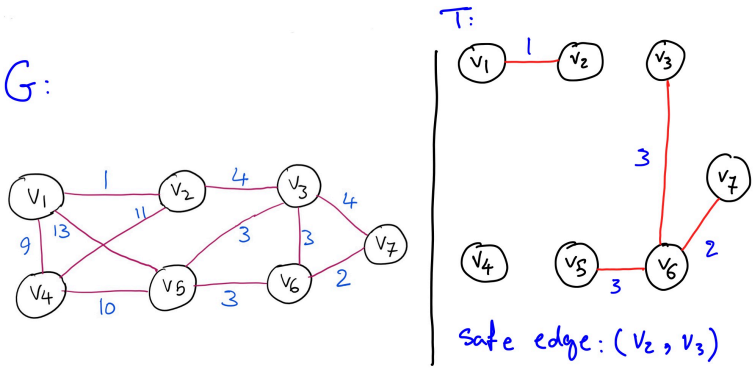
1. Let $T.V = G.V$. Start with $T.E = G.E$, keep deleting edges until an MST remains
   - In the worst-case, $\frac{n(n-1)}{2} - (n-1) \in \Theta(n^2)$ edges are deleted
   - An undirected graph G with n vertices has at most $\frac{n(n-1)}{2}$ edges
2. Let $T.V = G.V$. Start with **empty** $T.E$; keep adding edges until an MST is built
   - In the worst-case, $n - 1 \in \Theta(n)$ edges is added

10.3 Growing Implementation



- **Intuition** - While growing T, if we make sure that T is always a subgraph os some MST of G, then eventually T will become an MST
- Safe Edge e - $T.E \cup \{e\}$ is a subset of edges of some MST of G
- Cut (S_1, S_2) - of an undirected graph G is a partition of V into two disjoint subsets S_1 and S_2
  - i.e. $S_1 \cap S_2 = \varnothing$ and $S_1 \cup S_2 = V$
- Crosses - an edge crosses the cut if one of its endpoints is in S_1 and the other is in S_2

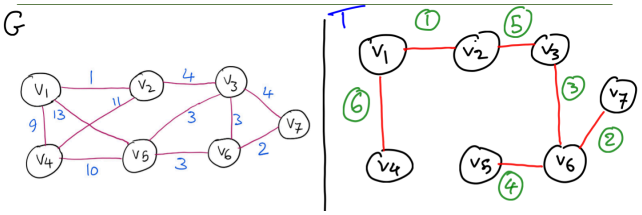
Theorem Let G be a connected undirected weighted graph, and T be a subgraph of G which is a subset of some MST of G. Let edge e be the minimum weighted edge among all edges that cross different connected components of T. Then e is safe for T.



Generic

GenericMST(G):

```
T.V = G.V        # Initializing vertices of the MST
T.E = ∅          # Starting with an empty set of edges
While T does not form a spanning tree:
    find e ∈ G.E with min weight that crosses two connected components of T
    # Add the edge e to the MST
    T.E = T.E ∪ {e}
return T
```



10.4 Kruskal's MST Algorithm

Idea

1. Go through the edges in order of non-decreasing weight
   - **Sort** the edges according to their weights
2. Select and add each edge that crosses two connected components of T
   - Go through the sorted list from lightest to heaviest
   - Add (u, v) to T if FindSet(u) $\neq$ FindSet(v)
   - After adding (u, v) to T, Union(u, v)

Implementation

Pre-condition: G is an undirected graph.

```
KruskalMST(G):
1. T.V = G.V
2. T.E = ∅
3. for each vertex v ∈ G.V:    # Create a separate connected component for each vertex
4.       MakeSet(v)
5. Let e_1, e_2, ..., e_m be the edges in G.E sorted by weight in non-decreasing order
6. for i = 1 to m:
7.       Let (u, v) = e_i
8.       if FindSet(u) ≠ FindSet(v):  # Check u and v are in different components of T
9.           T.E = T.E ∪ {(v, u)}      # Add the safe edge (v, u) to T
10.          Union(u, v)       # Now u and v are in the same connected component of T
11. return T
```

NOTE : If G is **not connected**, Kruskal's Algorithm will generate a Minimum Spanning Forest

Worst-Case Running Time

Assume that Disjoint-Sets implemented by tree with union-by-rank and path-compression

1. Sorting edges - $\mathcal{O}(m \log m)$
2. for loop on Lines #3 and #4 (MakeSet) - $\mathcal{O}(n)$
3. for loop on Lines #6 to #10 (FindSet and Union) - $\mathcal{O}(m\alpha(n)) \approx \mathcal{O}(m)$

Total running time - $\mathcal{O}(n + m \log m)$

Since $m \leq \dfrac{n(n-1)}{2}$, then $m \in \mathcal{O}(n^2) \implies \log m \in \mathcal{O}(\log n^2 = 2 \log n) \in \mathcal{O}(\log n)$

Total running time - $\mathcal{O}(n + m \log n)$

If the input graph G is connect ($m \geq n - 1 \implies n \in \mathcal{O}(m)$), the run time is $\mathcal{O}(m \log n)$

10.5 Prim's MST Algorithm

Idea

Pick an arbitrary vertex r of G as the starting vertex of T

Grow T by adding one edge at a time

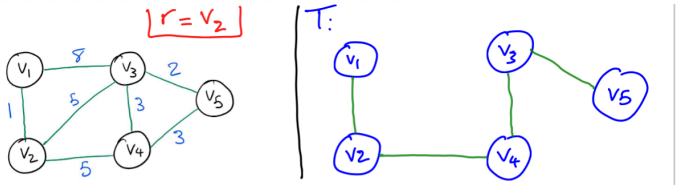
- At every step, T consists of one **connected component** that contains r, and **singleton** components that are not connected with r (each one is called an isolated vertex)
- At each step, pick a crossing edge with the minimum weight

Store all isolated vertices in a min-heap
Keys of the min-heap are weights of the crossing edges

Implementation

Precondition: G is a **connected** undirected graph. r is a vertex of G.

```
PrimMST(G, r):
1. T.V = G.V
2. T.E = ∅
3. for each vertex v ∈ G.V:
4.       v.key = ∞      # v.key keeps the shortest distance between v and T
5.       v.p = nil      # v.p keeps who in T is v connected to via lightest edge
6. r.key = 0            # Set r to be the root of min-heap
7. H = BiuldMinHeapGraph(G)
8. While H not empty:
9.       u = ExtractMin(H)
10.      if u.p ≠ nil:      # u.p is nil only when u = r
11.          T.E = T.E ∪ {(u.p, u)}      # Add a safe edge to the MST
12.      for each v ∈ G.adj[u]:    # all u's neighbours' distances to T need update
13.          if v ∈ H and w((u, v)) < v.key:
14.              v.p = u
15.              DecreaseKey(H, v.pos, w((u, v)))
16. return T
```



It	$v_1.key/v_1.p$	$v_2.key/v_2.p$	$v_3.key/v_3.p$	$v_4.key/v_4.p$	$v_5.key/v_5.p$	u	H root
0	∞/nil	0/nil	∞/nil	∞/nil	∞/nil	-	V_2
1	1/V_2		5/V_2	5/V_2	∞/nil	V_2	V_1
2			5/V_2	5/V_2	∞/nil	V_1	V_4
3			3/V_4		3/V_4	V_4	V_3
4					2/V_3	V_3	V_5
5						V_5	empty

Worst-Case Running Time

Assuming using binary min-heap and adjacency lists

1. for loop on Lines #3 to #5 - $\mathcal{O}(n)$
2. Building Min-Heap - $\mathcal{O}(n)$
3. Heap Operations - $\mathcal{O}(\log n)$
4. Total run time for all ExtractMin calls - $\mathcal{O}(n \log n)$
5. Total run time for all DecreaseKey calls - $\mathcal{O}(m \log n)$

Total running time - $\mathcal{O}((n+m)\log n)$

If G is connected, then the run time is $\mathcal{O}(m \log n)$

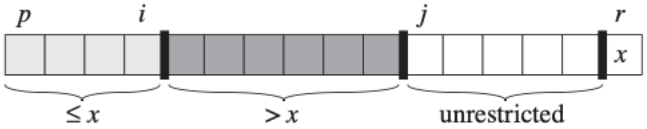
	Input Graph G	**Keep track of Connected Components**	**Find Minimum Weight Edge**
Kruskal's	If G is not connected generates a minimum spanning forest	Disjoint-Sets ADT	Sort all edges according to weight
Prim's	G has to be connected	Keep One Tree plus Isolated Vertices	Use Priority Queue ADT

11. Randomized Quick Sort

11.1 Quick Sort

Idea

1. If $len(A)$ is 1, return
2. Pick a pivot p
3. Partition the array into
   - A_1 - elements of A less than p
   - A_2 - elements of A equal to p

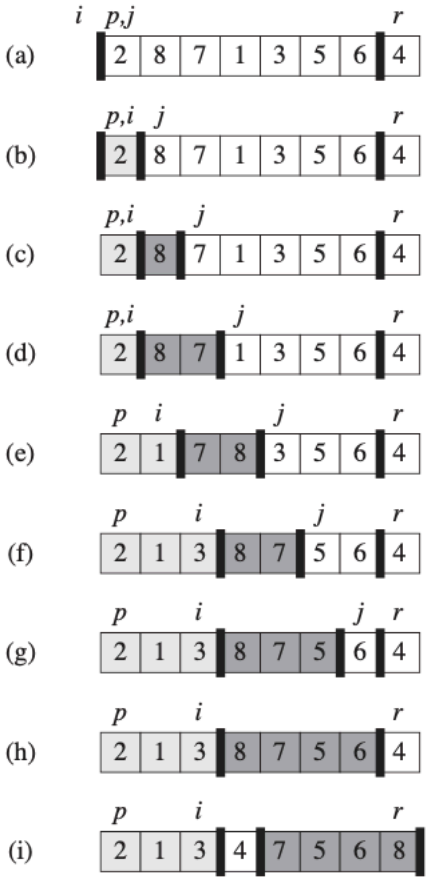


BuildMinHeap	ExtractMin	DecreaseKey
$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

   - A_3 - elements of A greater than p
4. Recursively repeat this procedure for A_1 and A_3

Implementation

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```



Worst-Case Running Time

Choose the number of Line #4 performed during Quick-Sort

- Each element in A can be chosen as pivot at most once
- In each recursive call, elements in the sub-array are **only compared to pivots**
- After being a pivot, that pivot will never be compared with anyone anymore

NOTE : Every pair (a, b) in A, are compared with each other **at most once**

Total #comparison $\leq$ total #pairs -> $\binom{n}{2} = \frac{n(n-1)}{2} \implies T(n) \in \mathcal{O}(n^2)$

Average-Case Running Time

Theorem z_i and z_j are compared by QuickSort iff out of the numbers $\{z_i, z_{i+1}, z_{i+2}, \ldots, z_j\}$, one of them is selected to be pivot first

$$\begin{aligned}
\mathrm{E}[t_n] &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} Pr(z_i \text{ is compared with } z_j) \\
&= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= 2(n+1) \sum_{k=1}^{n-1} \frac{1}{k+1} - 2(n-1) \\
&\in \Theta(n \log n)
\end{aligned}$$

11.2 Randomized QuickSort

- Shuffle the input array uniformly randomly

NOTE : After shuffling, the arrays look like drawn from a uniform distribution

- Las Vegas Algorithm - the solutions generated by the algorithm is guaranteed to be correct, but runtime depends on random choices
  - Randomized QuickSort (randomize the array and call QuickSort)
  - Choose pivot randomly
- Monte Carlo Algorithm - runtime of the algorithm is deterministic, but the output is based on random choices