Link to a pdf of the textbook:
https://dl.ebooksworld.ir/books/Introduction.to.Algorithms.4th.Leiserson.Stein.Rivest.Cormen.MIT.Press.9780262046305.EBooksWorld.ir.pdf

Another good resource:
https://static.packt-cdn.com/downloads/4874OS_Appendix_Big_O_Cheat_Sheet.pdf

Correctness, Readability, Efficiency

Data structures: To manage complexity
Algorithms: Finite set of instructions to complete a task, abstract

Insertion sort:
- Prefix array is the already sorted array in the beginning of the array that you expand upon
- Keep moving all elements larger than the pivot one position to the right
- Place the pivot in the position after the first element that is smaller than it

```
insertion_sort(vec)
{
  for(int i = 0; i < vec.size(); i++) //i is the position of the pivot
  {
    int pivot = vec[i];
    int j = i - 1;
    while(j >= 0 && vec[j] > pivot)
    {
      vec[j + 1] = vec[j];
      j--; //keep going back until you find a spot to insert the pivot
    }
    vec[j + 1] = pivot; //insert the pivot at the position one after the element we know is smaller than it
  }
}
```

Loop invariant: something that is always true before and after each iteration of the loop
- For insertion sort, the loop invariant is that the prefix array is always sorted
  - vec[from 0 to i-1] is sorted

Measuring Efficiency

We care only about the efficiency when the input size is large

Can't use clock method to time program because different architectures

For discussing algorithms, we only care about the rate of growth and the efficiency of the algorithm itself, not the time it takes for each instruction to complete or the smaller individual operations like integer vs float. Because this computation is negligible compared to the rest of the algorithm and it can vary from computer to computer.

Asymptotic notation: f(n), where n is the input size

Formal Definition for Big Theta:
θ(g(n)) = {f(n): **exists** c1, c2, n0 such that **for all** n > n0, c1g(n) <= f(n) <= c2g(n)}, f(n) ∈ θ(g(n))
- θ(g(n)) is a set of functions f(n) that satisfy the above
- Tight bound, bounded from above and below
- c1,c2 can be fractions, n0 must be an integer. All is positive since n is positive
- Negation of this is that for all c1,c2,n0, there exists an n that makes c1g(n) <= f(n) <= c2g(n) false

5n, 0.001n are both θ(n)

$$\sum_{i=0}^{k} a_i n^i \ is \ \theta\left(n^k\right)$$

- Pay attention to the highest power term

log_10(n) is θ(log_2(n)), bases don't matter since you just divide by a constant $\log_b a = \frac{\log_d a}{\log_d b}$

- **Constant** (1)
  - Simple statements
- **Logarithmic** $(\log N)$
  - Divide in half
- **Linear** (N)
  - Simple loop
- **Linearithmic** $(N \log N)$
  - Divide and conquer
- **Quadratic** (N^2)
  - Double loop
- **Cubic** (N^3)
  - Triple loop
- **Exponential** (2^N)/**Factorial** $(N!)$/etc.
  - Exhaustive search

- log_b(n^c) still belongs to the logarithmic class since you can change it to clog_b(n) where c is a constant
- log(log(n)) is a different class than log(n). It grows slower.
- We call quadratic and cubic polynomial running time, but they belong to different classes. n^2 is not θ(n^3).

In most cases, we can't have a big-theta for an algorithm because it is hard to get a lower bound and upper bound to match. This is why we use big-oh to give a general upper bound (worst case scenario) for efficiency

O(g(n)) = {for all n > n0, f(n) <= c*g(n)}
- n ∈ O(n^2), but n^2 is not O(n)

Big omega Ω for lower bound, lets us know the minimum time (best case) scenario

NP-complete: non-deterministic polynomial, set of problems. This class of problems cannot be solved in polynomial time (hasn't been proved). P = NP? Don't know if they are the same class

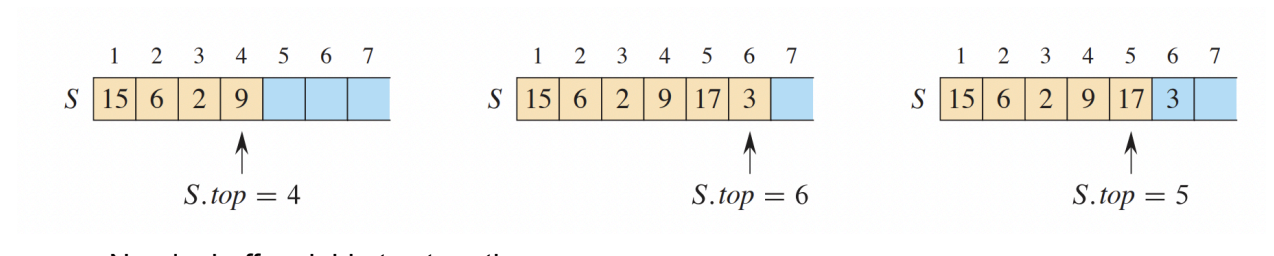
Recursive factorial example:

```
int factorial(int n) {
  if(n > 1)
    return n * factorial(n - 1);
  else
    return 1;
}
```

- This is O(n) because the function must execute the multiplication for each integer from n to 1

Data Structures

Stack: LIFO, push to top, pop from top.



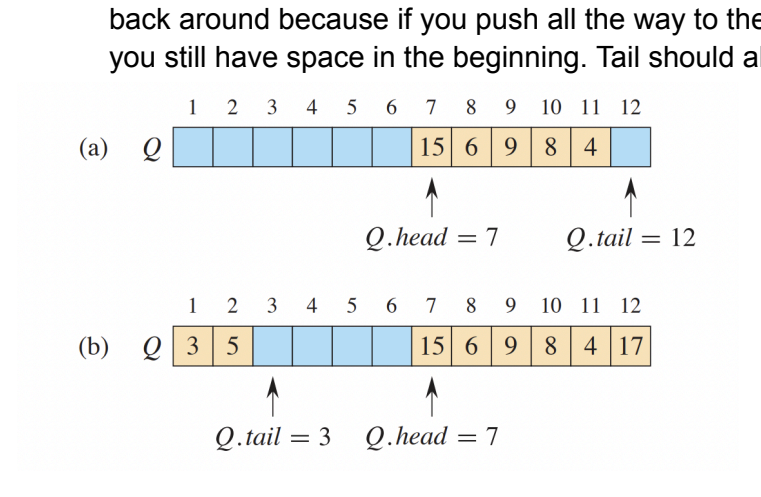
- Need a buff variable to store the array
- Need a capacity variable to record the total possible size of the array
- Need a top variable to keep track of the index of where you're popping/pushing to, in actual implementation it's at the position just above the current filled top and when you pop you decrement it before you return the element.

```
bool push(T buf[], T val)
{
  if (top < capacity)
  {
    buf[top] = val;
    top++;
    return true;
  }
  else
  {
    return false;
  }
}
T pop(T buf[])
{
  if(top > 0)
  {
    top--;
    return buf[top]
  }
  else
  {
    error, stack is empty
  }
}
```

- Don't need to remove element when you pop in this implementation since its an array implementation, and you'll overwrite it anyways

Queue: FIFO, push to end, pop from front

- Need head and tail variables. Push to the tail and pop from the head. Increment tail when pushing, increment head when popping. When the head or tail is at the end, wrap back around because if you push all the way to the end and pop all the way to the end, you still have space in the beginning. Tail should always point to an empty slot



- Need a capacity variable
- When Q.head = Q.tail + 1, or when Q.head = 0 and Q.tail = capacity-1, then the queue is full
  - Or when (head - tail + capacity) % capacity == 1
- When Q.head = Q.tail, then the queue is empty.
- The queue can only hold up to capacity-1 elements. When the tail is at the end and you push, the front of the array needs to be empty. Tail always needs to point to an empty slot. This is so we can distinguish between an empty queue (tail and head are the same) and a full queue.

```
T queue[capacity]; //initialize an array of type T and size capacity
int head = 0; int tail = 0;

bool push(T queue[], T val)
{
  if((head-tail + capacity) % capacity != 1) //+capacity added to fix negative mod bug
  {
    queue[tail] = val;
    tail = (tail + 1) % capacity; //make the tail wrap around
    return true;
  }
  else
  {
    return false;
  }
}
T pop(T queue[])
{
  if(head == tail)
  {
    queue is empty, error
  }
  else
  {
    int tempHead = head;
    head = (head + 1) % capacity;
    return queue[tempHead];
  }
}
```

Side Note: Inserting into an array is O(n) since worst case is needing to shift all elements one right
- When you do need to shift elements: the preferred way is taking the end element and shifting it forward and shifting the previous element forward, etc. Instead of shifting from any specific point in the array that you insert or remove from. They're both O(n) though.

Linked Lists:

Each node in a singly linked list has a val and a next pointer. The tail's next pointer is nullptr. Head and Tail are both pointers to nodes.

To check if a linked list contains a cycle, create two head pointers p and q. Keep pushing q and p, but push q faster than p. if ever p == q then you know there is a cycle. O(n)

```
int detectLoop(Node* list)
{
    Node *slow_p = list, *fast_p = list;

    while (slow_p && fast_p && fast_p->next) {
        slow_p = slow_p->next;
        fast_p = fast_p->next->next;
        if (slow_p == fast_p) {
            return 1;
        }
    }
    return 0;
}
```

```
template<typename T>
void insert(T buf[], T val, int i) { //array implementation, insert at index i
  struct Node {
    T val;
    Node* next;
    Node(T v, Node* n) : val(v), next(n) {} //ctor
  };

  buf[i] = Node(val, buf[i]);
}

Node* find(Node* head, T val)
{//traverse from the head until you find val
  for( ; head != null; head = head->next) //we can modify head directly here since it is a copy of head (received as pass-by-value thru params, local variable). NOT pass-by-pointer even though the argument is a Node pointer because head itself is a Node pointer.
  {
    if(head->val == val)
    {
      return head;
    }
  }
  return null;
}
find(head, 50) //example of how to call find(), we don't pass in &head because head itself is already a pointer. Pass by value


void insert(Node** head, T val) //Node** because we want to change head. Head itself is a pointer and we pass-by-pointer the pointer
{


  Node* new_node = new Node(val, *head); //create a new node whose next pointer points to the current head node
  *head = new_node; //make the node you created the new head node
}
insert(&head, 50)
```

PSEUDO LL

Hash Tables:

Hash Tables are a data structure that maps keys to values. Use hash tables when you need fast O(1) look up times, and when you have large amounts of data (or the potential number of different inputs is a lot) in key-value pair format

Have an array called buffer. For each input (key), compute a hash (index/new key) to insert an element in the array.
- The hash is a numerical representation of the key. We can refer to it as the key/hash/index interchangeably

A hash function converts inputs into numerical hashes/keys. h(k): k -> [0, b-1]
- k becomes converted into the range 0 to b-1 where b is the size of the buffer array

For different inputs the hash function can produce the same hashes (collision). We use chaining to handle the collisions.
- Each element in the buffer array is a bucket (linked list or dynamic array)
  - It is important to keep the capacity of each bucket constant to maintain O(1). Don't expand the capacity of a bucket.
- When you have a collision, store the key-value pair as an element in the linked list
- In the same bucket, each key-value pair points to the next key-value pair (all the keys will be the same)
- When you want to access a specific value, go to the bucket using the key associated with that bucket and iterate through each element in the linked list to find the value you are looking for
  - This is always O(1) since each bucket is the same fixed size. It does not depend on input size.

Think of buckets as being specific keys, and within each key there is a linked list of values that belong to that key

RESIZE WHEN ANY BUCKET IS FULL

When a bucket becomes **full** due to too many collisions, you need to **resize** the entire hash table (buffer array). Create a new hash table with a size 2b + 1 where b is the size of the current buffer array.
- Now you need to **rehash** all the key-value pairs of the old hash table to the new one. We do this so they get scrambled and there won't be a full bucket again. Each key-value pair will receive a new hash/key.
  - We can't directly copy one hash table to a larger hash table because the original hashes would not necessarily hash to the same buckets in the new table, since the hash function is different because we change x%b to x%2b+1. Moreover, you would just have a full bucket in the new hash table since you copied the old hash table over.
- **Deallocate** the previous hash table and its associated buckets
- When you expand the size of the hash table, the **hash function will stay O(1)** since you are just changing the value of b in x % b. And **access will stay O(1)** since all buckets are the same size.

Max number of inputs before you need to resize:
- b * s (if buckets are kept at constant size) b is buffer size, s is the size of each bucket

Min number of keys to insert where you must have a collision: b + 1 (pigeonhole)

Hash tables have a time complexity of O(1) because you access the value directly at the bucket/key and each bucket is a constant size that you iterate through.

```
struct pair
{
  k key;
  v value;
}


struct Bucket
{
  pair* pairs; //dynamic array of key-value pairs
  int capacity, length;
}


struct HashTable
{
  Bucket* buckets; //dynamic array of buckets
  int length;
}


int hash(k key)
{
  int x = f(key); //do some stuff to convert the key into an integer hash
  return x % b; //where b is the size of the buffer
}
```

PSEUDO HASH TABLE

Binary Search Trees

All nodes to the left are less than nodes to the right

Levels of a tree start from 1 at the root node
Height of a tree is equal to the level of the deepest leaf node

Max number of nodes at level i: 2^(i-1), because each level you go down can have at max 2 times the number of nodes of the previous level.

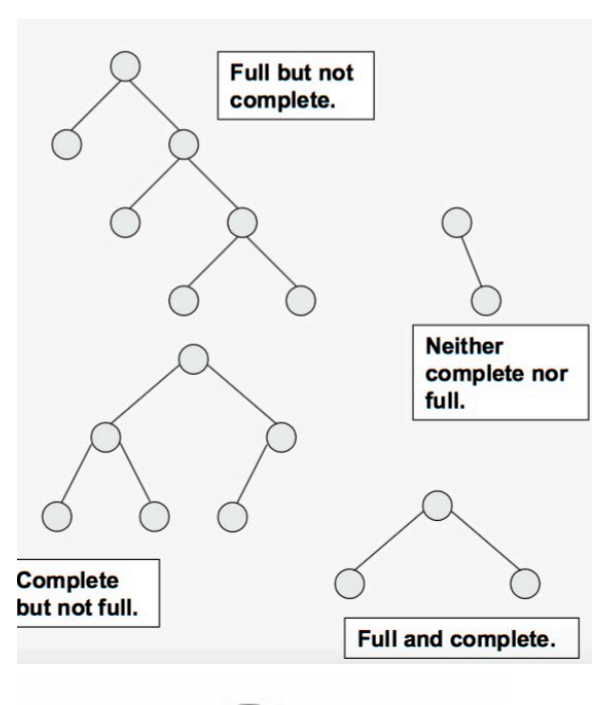
Max number of nodes in the entire tree of height i: (2^i) - 1 (just summation of all the number of nodes at level i for all levels)

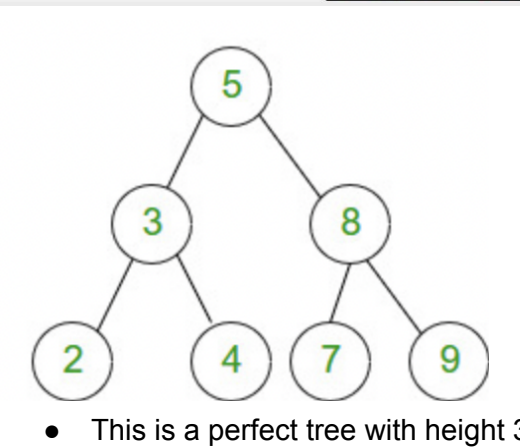
If n_0 is the number of nodes with zero children and n_2 is the number of nodes with 2 children
- Then for any binary tree with at least one node, n_0 - n_2 = 1
  - Proof by induction on the number of nodes on the tree. Base case is 1 node.

A **full** binary tree is a tree where all nodes have 2 or 0 children. No nodes can have one child.
- Total number of nodes is in the interval: [2h-1, (2^h) - 1]

A **complete** binary tree is where you fill a tree from top to bottom and all levels should be filled to their maximum, except perhaps the bottom where you would fill from left to right.
- Total number of nodes is the interval: [2^(h-1), (2^h) - 1]

A **perfect** binary tree is a complete and full tree AND all leaves are at the same level
- (2^h) -1





- This is a perfect tree with height 3

Complexity is determined by the balance of the tree. The complexity is O(h) where h is the height of the tree.
- log_2(n) <= Height <= n
  - We want the height of the tree to be log_2(n) and we can achieve this by creating a balanced tree.

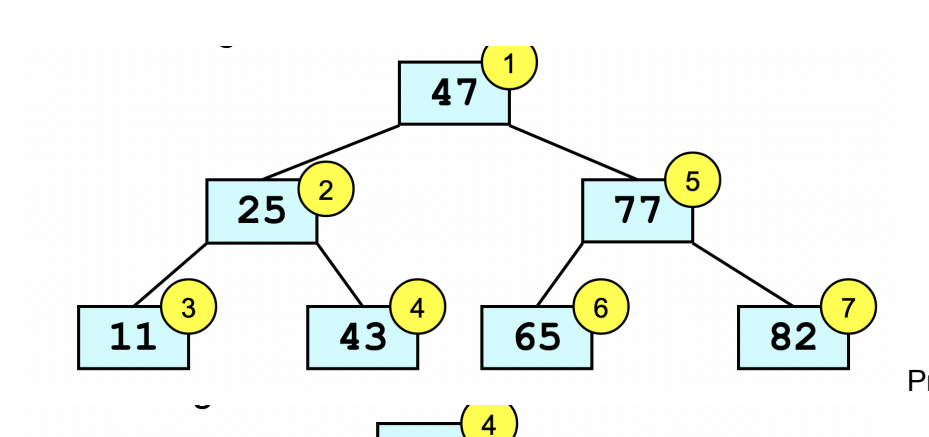
Traversal to all nodes in a tree:
- Pre-order traversal: CLR
- In-order traversal: LCR
- Post-order traversal: LRC

C is the current node, sometimes also represented as lowercase r.

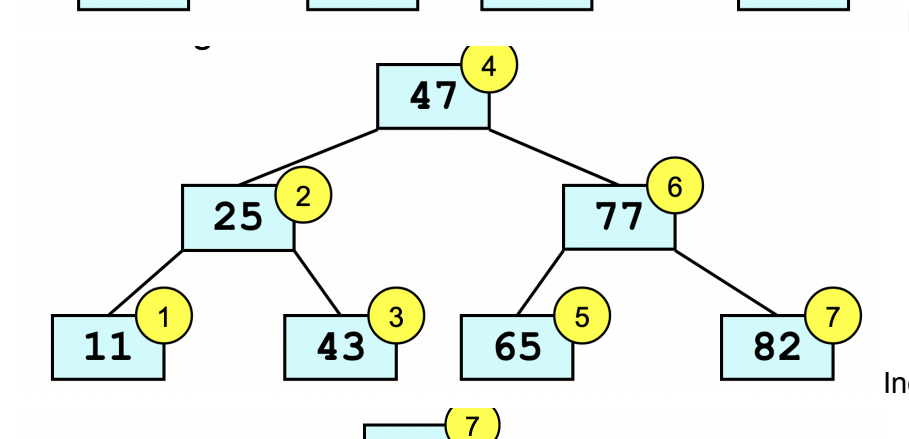
```
struct Node
{
  T key; //value of the node
  Node* left;
  Node* right; //recursive def, each node can have 0-2 children
}

Node* search(Node* tree, k key)
{
  if(tree == null)
  {
    return null;
  }
  if(tree->key == key)
  {
    return tree;
  }
  else if(key < tree->key)
  {
    return search(tree->left, key)
  }
  else
  {
    return search(tree->right, key)
  }
}

void inorder(Node* tree) //to print all nodes in a tree
{
  if(tree == null)
  {
    return;
  }
  inorder(tree->left);
  cout << tree->key; //current node, print it
  inorder(tree->right);
}
```

PSEUDO BST

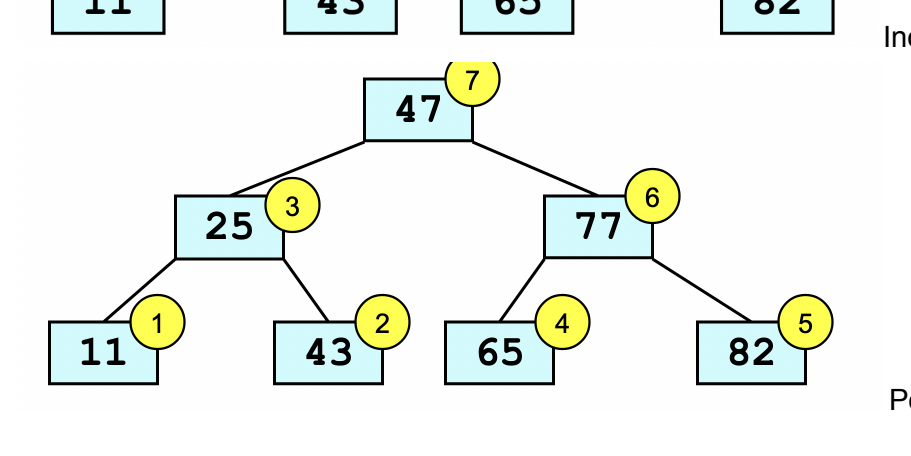
Visual representations of traversal methods:



Preorder



Inorder



Postorder

To remove an element from a normal binary search tree, you can follow these steps:

1. Start at the root node of the binary search tree.
2. If the value of the node you want to remove is less than the value of the current node, move to the left subtree.
3. If the value of the node you want to remove is greater than the value of the current node, move to the right subtree.
4. If the value of the node you want to remove is equal to the value of the current node, then you have found the node to remove.
5. If the node to remove is a leaf node (i.e., it has no children), simply remove the node from the tree.
6. If the node to remove has only one child, replace the node with its child.
7. If the node to remove has two children, find the minimum value node in its right subtree (or the maximum value node in its left subtree), replace the node to remove with this node, and then remove the minimum (or maximum) value node from the subtree.

To delete the entire binary tree use a recursive postorder

Red-black trees:
- We color each node as either red or black
- The root is black, all the leaves are black
- The children of a red node must be black
- Every simple path (moving downward) from any node to any leaf node contains the same number of black nodes.
- When a node is inserted into the red-black tree, it is initially colored red. However, this may cause a violation of the red-black tree properties, so the tree must be restructured by performing rotations and recolorings to restore the properties.

These properties ensure that the tree remains balanced, with a worst-case height of 2log(n+1), where n is the number of nodes in the tree. The properties also ensure that basic operations such as searching, insertion, and deletion can be performed efficiently in O(logn) time.

Let the black height bh(x) be the number of black nodes in the path from x to a leaf node, not including x itself
- Lemma: if a red-black tree has a black height bh(x) then it has least 2^bh(x) -1 nodes
- Lemma: A red-black tree with n internal nodes has height of at most 2log(n+1)
  - Number of nodes >= 2^bh(x) -1 >= 2^(h/2) -1
- Guarantees search time of O(logn)

Red-Black Trees Implementation:

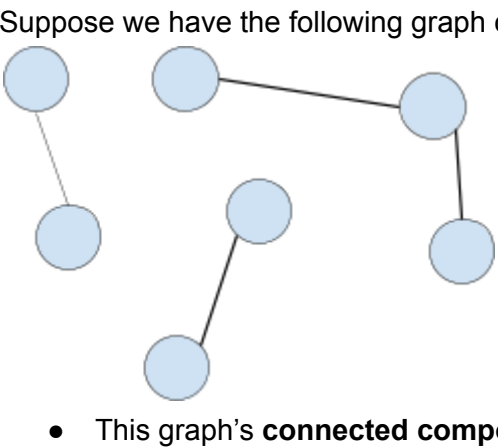
```
LEFT-ROTATE(T, x)
  y = x.right
  x.right = y.left          // turn y's left subtree into x's right subtree
  if y.left ≠ T.nil         // if y's left subtree is not empty ...
      y.left.p = x          // ... then x becomes the parent of the subtree's root
  y.p = x.p                 // x's parent becomes y's parent
  if x.p == T.nil           // if x was the root ...
      T.root = y            // ... then y becomes the root
  elseif x == x.p.left      // otherwise, if x was a left child ...
      x.p.left = y          // ... then y becomes a left child
  else x.p.right = y        // otherwise, x was a right child, and now y is
  y.left = x                // make x become y's left child
  x.p = y
```

```
RB-INSERT(T, z)
  x = T.root                // node being compared with z
  y = T.nil                 // y will be parent of z
  while x ≠ T.nil           // descend until reaching the sentinel
      y = x
      if z.key < x.key
          x = x.left
      else x = x.right
  z.p = y                   // found the location—insert z with parent y
  if y == T.nil
      T.root = z            // tree T was empty
  elseif z.key < y.key
      y.left = z
  else y.right = z
  z.left = T.nil            // both of z's children are the sentinel
  z.right = T.nil
  z.color = RED             // the new node starts out red
  RB-INSERT-FIXUP(T, z)     // correct any violations of red-black properties
```

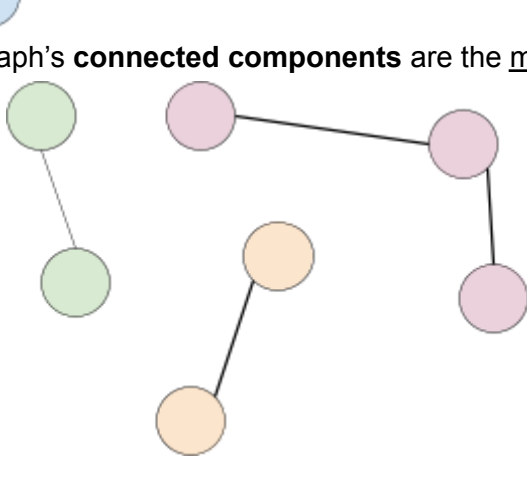
```
RB-INSERT-FIXUP(T, z)
1  while z.p.color == RED
2      if z.p == z.p.p.left              // is z's parent a left child?
3          y = z.p.p.right               // y is z's uncle
4          if y.color == RED             // are z's parent and uncle both red?
5              z.p.color = BLACK         ┐
6              y.color = BLACK           │ case 1
7              z.p.p.color = RED         │
8              z = z.p.p                 ┘
9          else
10             if z == z.p.right
11                 z = z.p               ┐ case 2
12                 LEFT-ROTATE(T, z)     ┘
13             z.p.color = BLACK         ┐
14             z.p.p.color = RED         │ case 3
15             RIGHT-ROTATE(T, z.p.p)    ┘
16     else // same as lines 3–15, but with "right" and "left" exchanged
17         y = z.p.p.left
18         if y.color == RED
19             z.p.color = BLACK
20             y.color = BLACK
21             z.p.p.color = RED
22             z = z.p.p
23         else
24             if z == z.p.left
25                 z = z.p
26                 RIGHT-ROTATE(T, z)
27             z.p.color = BLACK
28             z.p.p.color = RED
29             LEFT-ROTATE(T, z.p.p)
30 T.root.color = BLACK
```

Disjoint Sets

Suppose we have the following graph of vertices (nodes) and edges:



- This graph's **connected components** are the maximal subgraphs of connected nodes



Our goal is to design an algorithm that can compute the connected components of a graph

Assume we have a set of nodes S. Divide S into a set of k disjoint subsets.

$$S = \{ S_1, S_2, S_3, \ldots, S_k \}$$

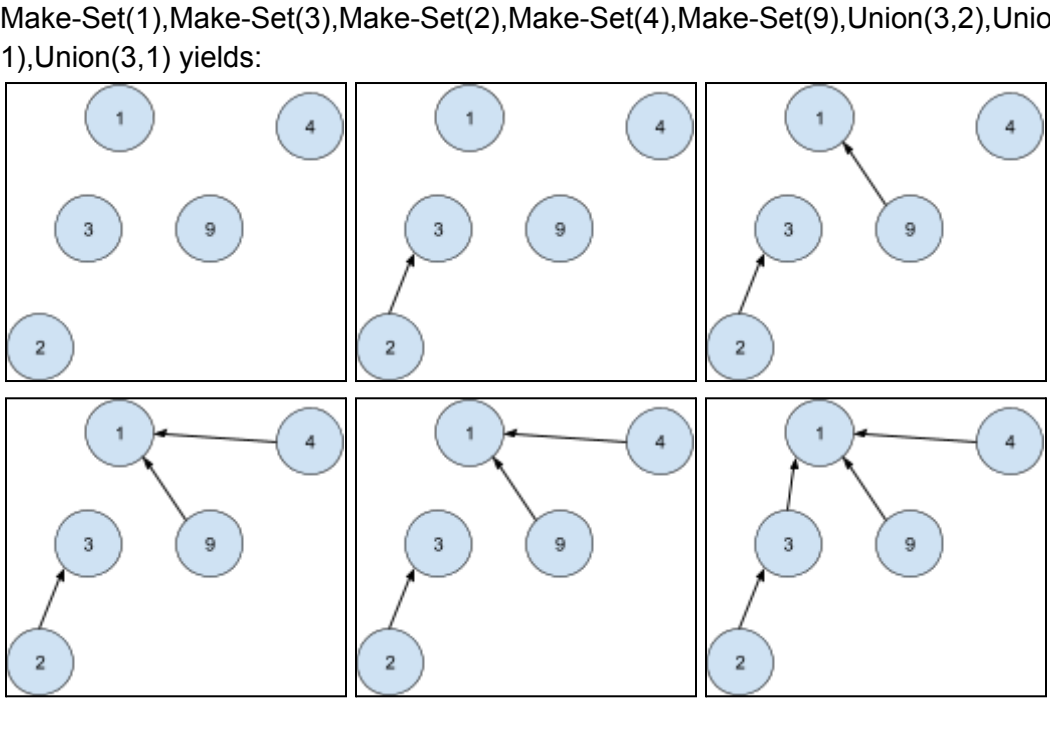
- Disjoint subsets means that the intersection of any pair of subsets is empty

Each disjoint subset has a representative element. This element will be used to refer to the subset as a whole.
- You can treat disjoint sets as trees with the representative element being the root

We will have a couple different operations:
- **Make-Set(x)**
  - We make a new subset in S that contains the element x. O(1)
  - Don't need to check if the subset already exists because we treat it as a precondition
- **Find-Set(x)**
  - Tells us which subset the element x belongs to. Climbs up x's parents until you reach the root. Returns the root/representative element. O(h), where h is the height of the tree
    - Assuming you union by rank, it will be O(logn).
- **Union(rep1,rep2)**
  - Take a union of the set which rep1 is representative and the set which rep2 is representative. O(1)
    - The Union operation merges the two subsets into a single subset with a new representative element.
      - Does this by setting the representative of one of the subsets to the other subset. Do this in a way so it minimizes the height. **Union by Rank:** always point the root of the shorter tree to the root of the taller tree. If the same height, point the arrow either way.

Calling m number of make-set(), find-set(), and union() operations, of which n are make-set(), takes O(mlogn) time.

Make-Set(1),Make-Set(3),Make-Set(2),Make-Set(4),Make-Set(9),Union(3,2),Union(1,9),Union(4,1),Union(3,1) yields:



Heaps

Max heap properties:
- Complete tree
- The root of any subtree is the **largest** element in the subtree

Min heap properties:
- Complete tree
- The root of any subtree is the **smallest** element in the subtree

For heaps the order of the children does not matter

For a binary search tree with two nodes:
- It is possible to make a max heap 2,1
- It is not possible to make a min heap (violates the complete rule for heaps)

Assuming we have a max-heap
- push(x): O(logn)
  - First insert the element at the bottom level at the rightmost position (that still makes it a complete tree)
  - Compare this newly inserted element with its parent, if the parent is less than than this element, swap them.
  - Percolate up as needed, don't need to check the children (also known as heapify up or sift up)
- pop(x): O(logn)
  - Store the root's value in a variable.
  - Take the rightmost value at the bottom level and make it the root's new value, delete the node you took from.
  - If the new root's value is less than a child's value, swap the root value with the child value. Make sure you compare to both children and swap with the child with the larger value.
  - Percolate down as needed.
  - Return the root's old value you stored in a variable.

For a min heap do the same but change all "less than" in the above steps with "greater than", and for pop swap with the child with the lowest value.

NOTES:
- Search is O(n). Have to look through every element

Sorting Algorithms

Heap Sort

To use a heap for sorting, just push elements into a heap and pop them out. Sorts in increasing order if you use a min-heap, sorts in decreasing order if you use a max-heap. O(nlogn)
- Push n times, pop n times. Both push and pop are O(logn)
- Space complexity is O(n), since you need to store all the keys of the tree in a heap
  - To get space complexity to O(1), use the array itself as the heap (in-place sorting).

Heapifying an array:

Index 0 of the array is the root, each subsequent element may be treated as a node filled in a complete tree format. Now we need to reorder the array to sort it

If a node is at index i, its children will be at index 2i+1 and 2i+2 (given that your root is at index 0).

Steps for in-place heap sort (using a max heap for descending order):
1. Parse through the leaf nodes (right to left in the array) until you reach an internal node (a node which if its index is i, there exists an element at at 2i+1). You can also do this by doing floor(n/2) - 1 where n is the size of the array. This gives the index of the last internal node
   - Thus: the elements from index floor(n/2) to n are all leaves

2. For the last internal node, if it has two children and one or more of the children are larger than the parent, then swap the parent with the largest child. If it has one child, swap if the child is bigger than the parent.

3. Find the next internal node and repeat until you reach the root

4. Once you are at the root, we know each subtree is a max heap so just percolate this last node down as needed.

Invariant: heap properties are guaranteed at every level (since we're starting from the bottom and going up)

IMPLEMENTATION:

```
heapsort(A, n)
{
    build_maxheap(A, n);
    for(i from n-1 down to 0) //sorts all elements in the array, one by one extraction from heap
    {
        swap(A[0], A[i]); //moves current root to the end
        max_heapify(A, i, 0); //call on the reduced heap, we pass in i as the size
    }
}

build_maxheap(A, n) //A is the array, n is the size, turns A into a max heap
{
    for(i from floor(n/2) - 1 down to 0) //for all internal nodes
    {
        max_heapify(A, n, i);
    }
}

max_heapify(A, n, i) //i is the index of the internal/parent node
{
    largest = i;
    left = 2*i + 1;
    right = 2*i + 2;

    if(left < n && A[left] > A[largest])
        largest = left;

    if(right < n && A[right] > A[largest])
        largest = right;

    if(largest != i)
    {
        swap(A[i], A[largest]);
        max_heapify(A, n, largest) //recursively heapify the sub-tree
    }
}
```

- Precondition of max_heapify is that both left and right children are heaps.

Heap sort has a time complexity of O(nlogn) since we have one call to build_maxheap which takes O(n) and max_heapify takes O(logn) and gets called n-1 times so we have O(n + nlogn) = O(nlogn)

Merge Sort (Divide and Conquer)

1. First divide up the array into n subarrays where n is the number of elements. Each subarray should have only one element

2. Merge adjacent subarrays into new subarrays and sort those

3. Continue doing step 2 until you get back to an array the same size as your original one

Invariant: each subarray being merged is already sorted

Time complexity: O(nlogn). Need to run logn iterations, each iteration is n dependent
Space complexity: O(n), can't sort in-place because you would ruin the already sorted arrays, need a temp array

Heap sort and merge sort have the same worst case running time and merge sort has a worse space complexity, but merge sort may be more effective depending on the nature of the data. Merge sort is a stable sort (meaning if two elements have the same value, their relative positions in the sorted array will be the same as their positions in the original array)

Quick Sort (also Divide and Conquer)

Most commonly used sorting algorithm
1. Pick a random pivot (any element)

2. Partition the array into two subarrays: elements less than the pivot and elements greater than or equal to the pivot. All elements less than the pivot are positioned to the left of it. Elements greater than the pivot are to be placed to the right of it. The pivot itself will be in its final sorted position
   - This can be accomplished by having two pointers: left and right. Left is the leftmost element in the array. Right is the rightmost element. Keep shifting the left pointer towards the right until you find an element larger than the pivot. Once you do, stop. Now keep moving the right pointer towards the left until you find an element smaller than the pivot. Once you do, stop. Swap the elements pointed to by left and right. Keep doing this until left and right meet.

3. Recursively apply steps 1 and 2 to the subarrays created in the previous step. This is done separately for the subarray to the left of the pivot and the subarray to the right of the pivot.

Invariant: After each partitioning step, all elements to the left of the pivot are smaller than the pivot, and all elements to the right of the pivot are greater than the pivot. (some elements can also be equal to the pivot if there are duplicates)

Time complexity: O(nlogn)
Space complexity is O(1) if you do it iteratively. O(logn) if you do it recursively.

If you accidentally pick an extreme value (min or max) for the pivot, time will be O(n^2)
- Picking the first or last index as your pivot in an already sorted array will trigger O(n^2)
- If every element in the array is the same, then it will also be O(n^2)

Summary

Complexities: average time, worst time, space complexity
Heap Sort: O(nlogn), O(nlogn), O(1)
Merge Sort: O(nlogn), O(nlogn), O(n)
Quick Sort: O(nlogn), O(n^2), O(1)

For space complexity, we only care about the memory allocated by the algorithm, not the original array.

Worst case is dependent on the arrangement of the array

Asymptotically heap sort would be the best but practically we use quick sort more. If you know nothing about the data, it's better to use quick sort. This is because it is in-place, has a simple implementation, and in practice has a lower coefficient of nlogn in its time complexity.

Can't have a comparison sorting algorithm better than O(nlogn) because we need to look at all elements at least once (n) and we need to compare the elements, which takes at least logarithmic time (logn) in a comparison-based sorting algorithm.

Graph Algorithms

Definition of a **tree**: there exists exactly one path between any two pair of vertices and there are no cycles

We have two ways of representing graphs: adjacency lists and adjacency matrices

Adjacency List: Space is O(n)
- Have an array of vertices where each vertex is a linked list
- Represent all neighbors of a vertex as elements in the linked list associated with that vertex
  - Like a hash table. Think of vertices as buckets and adjacent vertices as elements in those buckets

Adjacency Matrix: Space is O(n^2)
- Make a matrix where each row represents a vertex and each column represents a vertex
- Each cell in the matrix is either 1 or 0, depending on if there exists an edge between the vertex represented by the row of that cell and the vertex represented by the column of that cell. 1 for edge, 0 for no edge

Adjacency list is beneficial for sparse graphs (not a lot of edges). Saves space
Adjacency matrix is beneficial for telling if an edge exists in constant time

Two types of access:
1. Find if an edge exists between two vertices
   - O(n) for adjacency list
   - O(1) for adjacency matrix
2. Find all adjacent vertices of a vertex.
   - O(1) for adjacency list because we can just return the sublist
   - O(n) for adjacency matrix

Saving space with undirected graphs:
- For an adjacency matrix, you can just make half of the matrix since everything across the diagonal will be mirrored. Still O(n^2).
- For an adjacency list, you only need to store each vertex once. Instead of storing both i-j and j-i, you can just store i-j (have a convention like the largest vertex holds all smaller adjacent vertices inside its linked list). Saves half the space but still O(n)
  - For example, O-I-J-K would only be stored once

EXTRA GRAPH INFO:
Remember, a complete graph is where every pair of vertices is connected by an edge:
|V| = n
|E| = n(n-1)/2

A connected graph is where every vertex has a path to any other vertex. There can be a minimum of n-1 edges and a maximum of n(n-1)/2 edges (thus making it a complete graph)

Some rules:
- A graph with n vertices and n-1 edges is a connected graph
- A graph which is connected, has no cycles, and has n vertices has n-1 edges
- A graph which is connected, has n vertices, and n-1 edges, has no cycles

All these graphs are also Trees

Sparse if E < V^2
Dense if E ~= V^2

Breadth-First Search

BFS visits all the closest nodes of a graph first. It does this by using a queue

So we don't visit the same vertex twice:
- Color a vertex white if you haven't visited
- Color a vertex black if you have visited
- Initially all vertices are white

We want to traverse the closest vertices first. To do this we:
1. Create a queue and put the starting vertex into the queue
2. Every iteration, pop a vertex from the queue and visit that vertex, then push all of the neighbors of that vertex into the queue.
   - Remember to color vertices black when you visit them
3. Eventually all vertices that you can reach will be visited

Note: Can't use a stack for BFS because you would not traverse the closest nodes first

We can also keep a distance value for each node so we can find the shortest distance (number of edges) between the starting node and any other node:

```
vector<T> BFS(s) //where s is the starting vertex
{
    vector<T> vec;
    for(v : V) //V is the set of all vertices
    {
        v.color = white;
        v.distance = -1; //default distance val
    }

    s.color = black;
    s.distance = 0;

    queue<Vertex<T>> q;
    q.push(s)

    while(!q.empty())
    {
        Vertex<T> u = q.front();
        vec.push_back(u.value);
        q.pop();

        for((u, v) : E) //u,v is any edge that connects u to a neighbor, aka v is a neighbor
        {
            if(v.color == white) //if it is unvisited
            {
                v.color = black;
                v.distance = u.distance + 1; //cause we know u.distance is the shortest distance for u
                q.push(v);
            }
        }
    }

    return vec;
}
```

- *This implementation of BFS also works on disconnected graphs*

For BFS, running time: O(n + |E|) where n is number of vertices, |E| is the number of edges

Explanation: The for-loop that sets everything to white iterates |V| times, the while loop iterates |V| times, the inner for-loop iterates |V|*|Eaj| times (|V| times from the while loop and |Eaj| times from the loop itself). Therefore we get O(|V| + |V| + |V|*|Eaj|) == O(2|V| + |E|) == O(|V| + |E|) == O(n + |E|)
- Eaj is the set of adjacent edges to a vertex
- If we wish to express the running time of BFS in purely terms of n, it would be O(n) amortized and O(n + n^2) == O(n^2) worst case. This is less precise than n+|E| though
  - Refer to the bit about complete graphs and connected graphs in the section above to understand why

Depth-First Search

BFS goes as wide as possible, DFS goes as deep as possible. DFS can use a stack or can be recursively called (shown below)

```
DFS_visit(vertex<T> u, int& time, list<T> order)
{
    time = time + 1;
    u.discovery_time = time;
    u.color = gray;

    for(v : NeighborsOf(u)) //for each neighbor of u
    {
        if(v.color = white)
        {
            DFS_visit(v.value, time, order);
        }
    }

    time = time + 1;
    u.finish_time = time;
    u.color = black;
    order.push_front(u);
}

list<T> DFS() //return a list in topological order
{
    for(v : V) //for all vertices in the graph
    {
        v.color = white;
        v.finish_time = 0;
        v.discovery_time = 0;
    }

    list<T> order;
    int time = 0;

    for(v : V)
    {
        if(v.color == white)
        {
            DFS_visit(v, time, order);
        }
    }
    return order;
}
```

- *This implementation of DFS also works on disconnected graphs*

DFS is O(n+|E|) because we call DFS visit on each vertex only once (n) and we look at each edge only once (|E|). Similar to BFS

EXTRA NOTES:
- Can obtain a tree from BFS and DFS by following the path of visits
- Both BFS and DFS are not unique

Topological Sort

Assuming you have a spreadsheet with each cell having an expression in it that may depend on expressions in other cells, which order would you use to evaluate the cells? We use topological sorting to determine the order.

If cell B depends on cell A, evaluate A first. Always evaluate the dependent after the independent. Having a dependency is an asymmetric relation.

Must use a **directed graph** for this situation
- Each vertex represents a cell
- Each edge represents a dependency
  - A->B means B depends on A
    - The arrow goes to the dependent

We sort a linear order:
- For A->B, A must come before B in the order, but they don't necessarily have to be next to each other.
  - If X->Z and Y->Z, the order of X relative to Y doesn't matter but they both must come before Z.
- Formal Definition: Topological order of a directed graph is a linear order of all the vertices such that if there is an edge from U to V, then U appears before V in the order.

Topological Sorting:
1. Evaluate all independent vertices. Independent vertices have no incoming edges/arrows.
2. Remove each node you evaluate
3. Repeat steps 1 and 2 until you evaluate all the nodes

This works since you destroy the nodes that you evaluate, thereby making more nodes independent, etc

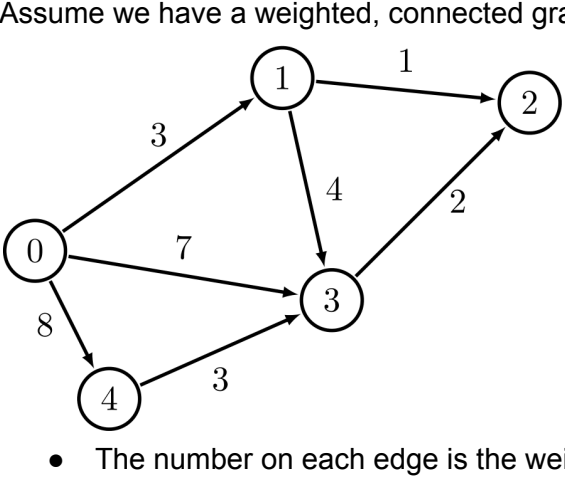
```
for(v in V)
{
    count[v] = 0; //count is the # of incoming edges for a specific vertex
}
for(u,v) in E
{
    count[v]++;
}
for(v in V)
{
    if(count[v] == 0)
    {
        container.push(v);
    }
}

while(container is not empty)
{
    u = container.pop();
    print(u)
    for(u,v) in E
    {
        count[v]--;
        if(count[v] == 0)
        {
            container.push(v);
        }
    }
}
```

Overall complexity is O(n+|E|), where n is the number of vertices and |E| is the number of edges

EXTRA NOTES:
- Topological graphs cannot have cycles (they are acyclic graphs)
- All acyclic graphs have a topological order
- A directed graph may have multiple different topological orders

Minimum Spanning Trees

Assume we have a weighted, connected graph:



- The number on each edge is the weight of that edge

Then the minimum spanning tree is the subset of edges (that connect all the vertices) with the minimum total weight.

- This subset will have exactly n-1 edges (where n is the number of vertices)
  - Because if it has any more than n-1 edges, it will have a cycle. So there are extra edges that you can remove to reduce the weight of the graph.
- In general, spanning trees and MSTs are not unique
  - Reworded: for a given graph, there can be multiple minimum spanning trees.
    - However, all MSTs must have the same total weight

Extra Rules:

- All connected graphs have at least one (minimum) spanning tree. All (M)STs are connected graphs.
- The subset of edges with the minimum total weight of any connected graph always makes a tree

The essence of computing a MST is:

1. Create an empty subgraph (candidate MST). Let's call it A
2. Given a vertex u, find an edge (u,v) that is safe for A
   - A safe edge is an edge that can be added to A while ensuring that A is a subset of a MST
3. Add (u,v) to A
4. Repeat steps #2 and #3 until A has n-1 edges

Invariant: A is always a subset of a MST

To find a safe edge in step#2, we need some more definitions:

- A ***cut*** (S, V - S) is a partition of V (set of vertices) into the disjoint subsets S and V-S.
- An edge (u,v) ***crosses*** the cut if one of its endpoints belongs to S and the other belongs to V-S.
- A cut ***respects*** a set A of edges if no edge in A crosses the cut
- An edge is a ***light edge*** crossing the cut if its weight is the minimum of any edge crossing the cut. There can be more than one light edge crossing a cut in the case of ties.

Thus, we can find a safe edge using the following theorem: if A is a subset of edges of a MST and a cut respects A, then any light edge crossing the cut is safe for A.

- Any cut that respects A can be utilized to find a safe edge for A

Brief proof by contradiction: Assume that the light edge (u,v) is not part of the minimum spanning tree. Then there must exist another pair of vertices x and y with a "lighter edge" crossing the cut. Implies the existence of another valid MST with a lower total weight. Contradicts definition of a MST

Corollary: Let A be a subset of a MST, and C is a connected component in G_A=(V,A). If (u,v) is a light edge connecting C to another connected component of G_A, then (u,v) is safe for A.

- G_A is a subgraph that contains all the vertices from the original graph but only the edges from A (candidate MST).
- G_A doesn't have to be connected initially. We want to grow A such that G_A becomes connected and an MST
- Reworded: If there is a light edge (u,v) connecting a connected component to another connected component in G_A, then (u,v) is a safe edge

IMPLEMENTATIONS:

Kruskal's Algorithm: Uses the corollary

1. Create a forest A (a set of trees), where each vertex in the graph is a separate tree
2. Create a sorted set containing all the edges in the graph in increasing weight
3. While A is not yet spanning
   a. Take the next smallest edge from the sorted set. If the edge connects two different trees then add it to the forest A, combining two trees into a single tree

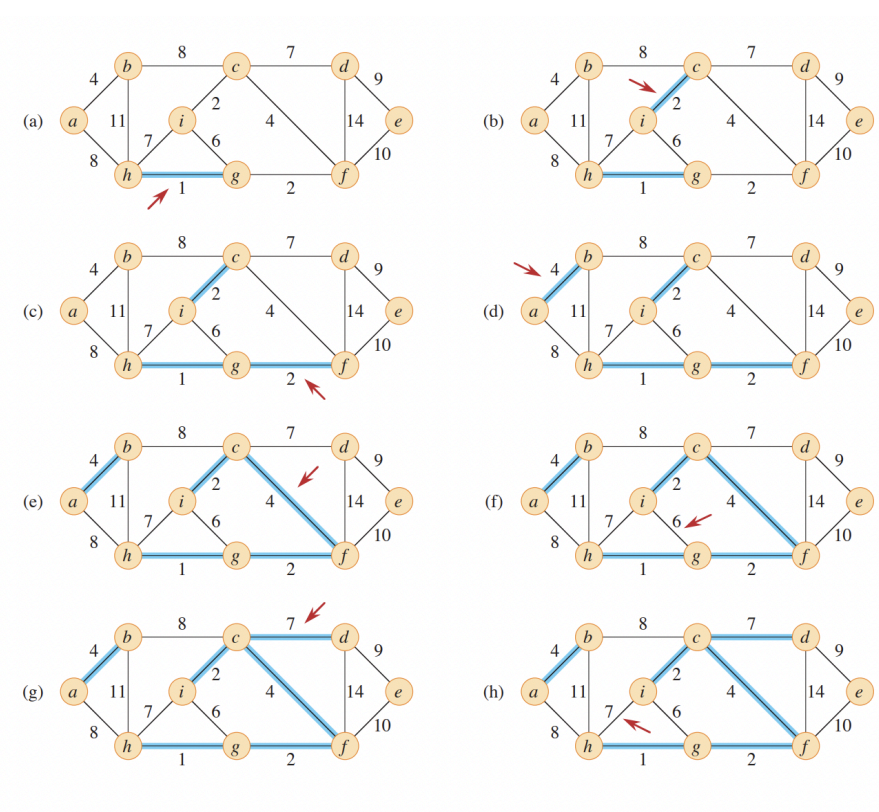


Figure 21.4 The execution of Kruskal's algorithm on the graph from Figure 21.1. Blue edges belong to the forest *A* being grown. The algorithm considers each edge in sorted order by weight. A red arrow points to the edge under consideration at each step of the algorithm. If the edge joins two distinct trees in the forest, it is added to the forest, thereby merging the two trees.

```
Kruskal(G)
{
  A = {} //A is a forest
  for v in V //n times
  {
     make-set(v);
  }
  for (u, v) in sort(E) //each edge in sorted order, sort takes ElogE, the for loop runs E times
  {
     if (find(u) != find(v)) //if u and v are not in the same connected component. We use the find function from disjoint sets. Each find takes logn
     {
        A = A union {(u, v)}  //merges the two connected components containing u and v
        union(u, v);

        if (|A| = |V| - 1) //A has n-1 edges, so we are done
        {
           break;
        }
     }
  }
  if (|A| < |V| - 1)
  {
     //A is not a spanning tree or not connected
  }

  // We want to map each connected component to a disjoint set. We union the two sets whenever we have a connection between the components.
}
```

- Running time is O(Elogn)/O(ElogE)

Prim's Algorithm (using heap): Uses the theorem

1. Create a tree A with just the starting vertex
2. Find all edges connecting any tree vertex with a vertex not included in A, pick the minimum of these edges to include into A such that it does not form a cycle
3. Repeat step#2 until you have an MST

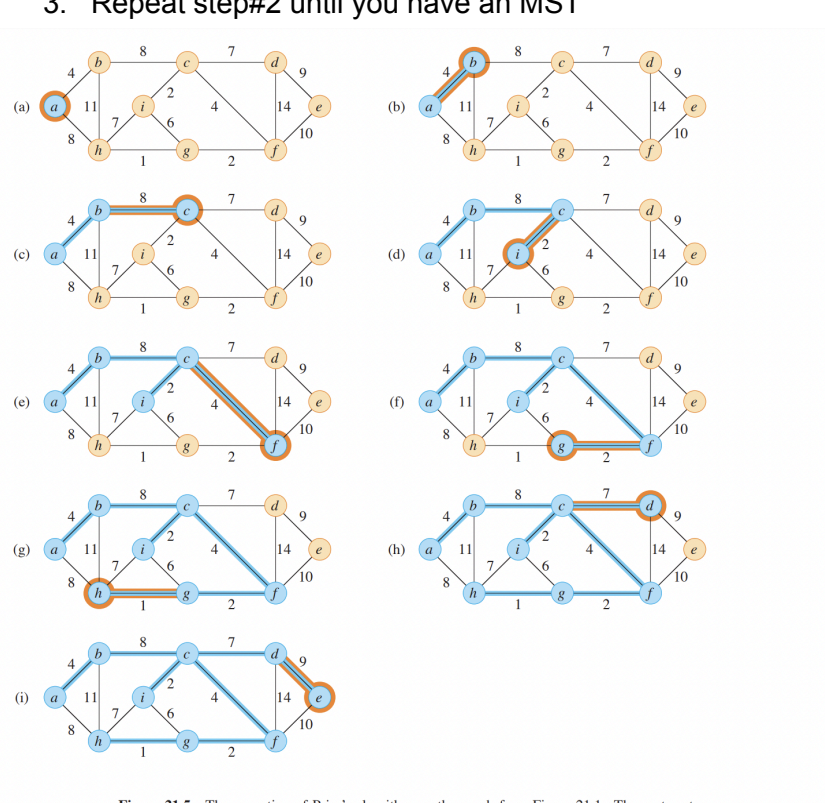


Figure 21.5 The execution of Prim's algorithm on the graph from Figure 21.1. The root vertex is *a*. Blue vertices and edges belong to the tree being grown, and tan vertices have yet to be added to the tree. At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing the cut is added to the tree. The edge and vertex added to the tree are highlighted in orange. In the second step (part (c)), for example, the algorithm has a choice of adding either edge (*b*, *c*) or edge (*a*, *h*) to the tree since both are light edges crossing the cut.

```
PrimUsingHeap(G, s) //s is the starting vertex
{
  for(v in (V-{s}))
  {
     v.key = infinity
  }
  s.key = 0 //key values are the min distance from source
  Q = build_min_heap(V); //O(n)
  while(Q isnt empty) //num iterations: V-1
  {
     u = extract-min(Q) //get the vertex with the minimum weight edges
     for((u,v) in E) //look at all the neighbors, runs E iterations CUMULATIVELY
     {
        if(v is in Q and v.key > Weight(u,v)) //takes logn
        {
           v.key = Weight(u,v); //whenever you change the value of a key, percolate as needed (increase-key and decrease-key). These functions are part of the heap and each take logn
           v.parent = u; //adds u,v to the tree
        }
     }
     A = A union (u,v)
     Q = Q - {v}
  }
}
```

- Running time is O(Elogn)/O(ElogE)
- We use a min-heap to keep track of all the vertices. The key of a vertex is the minimum weight of an edge from it to any neighbor

IMPORTANT DISTINCTIONS

- In Kruskal, A is always a forest until the end
- In Prim, A is always a tree
- Prim needs a starting vertex while Kruskal does not
- Both are greedy algorithms

Single-Source Shortest Path

We want to find the path with the least weight from a source vertex to any destination vertex given a connected graph.

Each vertex will have a parent pointer and a distance value
Each edge will have a weight value

We will follow Dijsktra's algorithm:

1. Create a set of visited nodes S. Push the start node into it and set its distance value to 0. Set all other nodes distance values to infinity
2. Create a min heap Q and push all nodes into it
3. Extract the node with the minimum distance in Q and add it to S. (initially this is just the start node)
4. For this node U, examine all neighbors of it. If U.distance + the weight of the edge connecting U and its neighbor < neighbor.distance, then you need to update neighbor.distance. Neighbor.distance becomes U.distance + connecting weight. You also need to set neighbor.parent = U. (relaxation)
5. Repeat steps #3 and #4 until all nodes have been visited
6. To trace the shortest path, start with the destination node and go back through the graph to the start by jumping through the parent pointers.

```
Dijkstra(G,startNode)
{
  S = {}
  Q = minheap
  for(all nodes)
  {
     node.d = INF
     Q.push(node)
  }
  startNode.d = 0

  while(!Q.empty()) //iterations: n
  {
     u = Q.extract_min()
     S.union(u)
     for(all edges from u to neighbor v) //iterations: cumulatively E (n*Eaj)
     {
        relax (u,v)
     }
  }

relax(u,v)
{
  if(u.d + weight(u,v) < v.d)
  {
     v.d = u.d + weight(u,v) //takes logn to find the new position of the vertex in the heap (decrease_key)
     v.parent = u;
  }
}
```

Total running time: O(Elogn)/O(ElogE)

- θ(n) <= E <= θ(n^2), we don't know specifically what E is

EXTRA NOTES:

- Can use BFS for shortest path if all edges are the same weight
- Dijsktra's algorithm doesn't work with negative weights
- The distance value of any node cannot increase at any point, it must decrease or stay the same

Advanced Data Structures

B-Trees

B-trees are self-balancing trees optimized for search. We minimize the worst case complexity.

Disk accesses are expensive so we minimize the number of times we need to access a node. As a result each node will have more children and contain more keys.

Given a complete tree where each node can have k children, time complexity for all operations will be $\log_k(n)$ instead of $\log_2(n)$ (as for a normal binary tree). Still O(logn) but smaller coefficient

- k = (max) degree of the tree

We want k to be as high as possible provided that the node can be stored in a disk block

In a binary tree, each node can hold one key and two child pointers (left and right). So in total we have 3 fields for each node.

In a B-tree of degree k, each node can hold k-1 keys and k child pointers. So in total we have 2k-1 fields for each node. ptr1,key1,ptr2,key2, … ptrk

Properties of B-Trees:

- Keys and children are sorted
- All leaves appear at the same level
- Each node has at most 2t-1 keys
- Each node has at least t-1 keys, except the root which has at least 1 key
- If there are n keys in a node, there are n + 1 children, therefore…
- Each node has at most 2t children
- Each node has at least t children, except the root which has at least 2 children

t = minimum degree of the B-Tree (aka the min number of children each node can have). This is different from k (maximum degree of the tree)

- Caveat: t must be greater than 1 for a b-tree.

B-tree ⊄ binary tree

Assuming we have a B-tree with t = 2

- Each node except the root may have 2,3, or 4 children. Therefore k = 4
- Called a 2-3-4 tree
- Time complexity is $\log_4(n)$

Minimum number of nodes at each depth:
Level 1 (root): 1
Level 2: 2
Level 3: 2t
Level 4: 2(t^2)
etc..

Therefore, the **total** number of nodes n in a B-tree with height h is bounded by:

$$n \geq 1 + 2\sum_{i=0}^{h-2} t^i \quad = \; = 1 + 2\frac{t^{h-1} - 1}{t - 1}$$

- Remember the height of a tree is equal to the number of nodes from the root to the deepest leaf (height = total number of levels)
- If you want to find the maximum number of nodes in a B-tree replace t with k (max degree)

Insertion into a B-tree

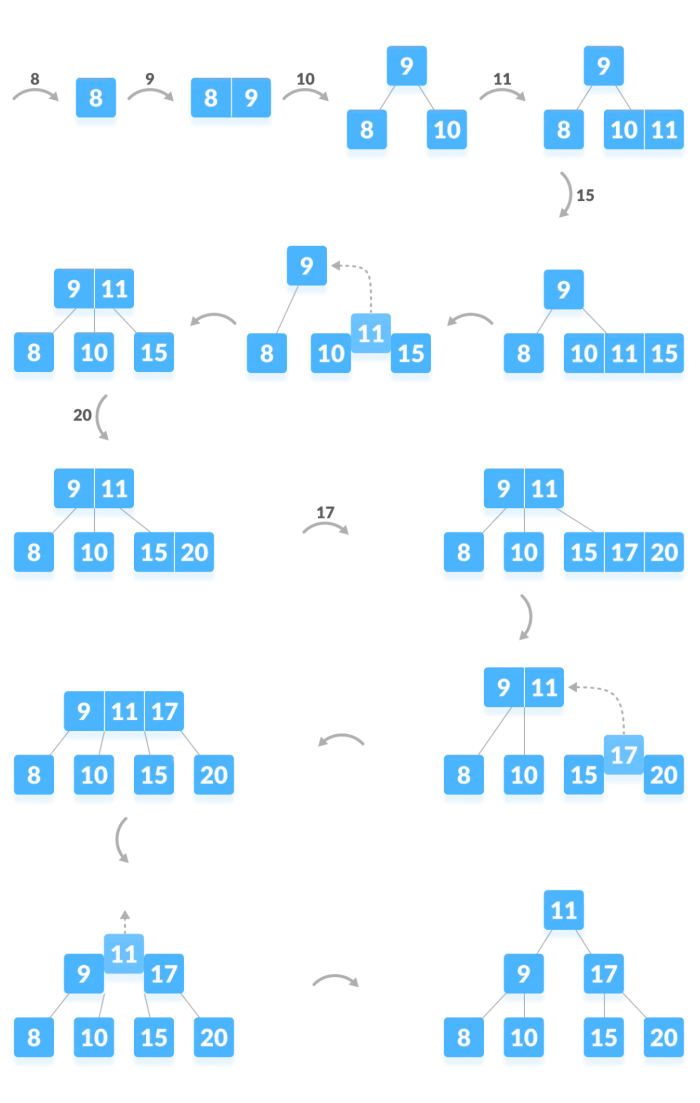
Insertion properties:

- A B-tree grows and shrinks from the root, unlike BSTs which grow and shrink from leaves
- Insertion into a B-tree can only happen at a leaf node

Steps:

1. Start at the root of the B-tree and traverse it to find the appropriate leaf node where the value should be inserted. To do this, compare the value to be inserted with the keys in each node and proceed down the tree by following the appropriate child pointer
2. Once you reach a leaf node, check if there is enough space to insert the new value (has less than 2t - 1 filled keys). If there is, proceed to step #4. Otherwise, continue to step #3.
3. If the leaf node is full, you need to perform a split operation. Splitting a full leaf node involves dividing the keys and values into two smaller leaf nodes and promoting the median value to the parent node. This split operation may propagate up the tree if necessary.
4. Insert the new value into the appropriate position within the leaf node, maintaining the order of the keys.

If your insertion resulted in a split and promotion of a value to the parent node, you need to update the parent node by inserting the promoted value into the correct position. If this parent node was already full, then you need to split it as well. This can propagate up to the root node. If the root node is split, a new root node will be created with the promoted value and the height of the tree will increase by one



Implementation of insert:

```
B-TREE-INSERT(T, k)
1  r = T.root
2  if r.n == 2t − 1
3      s = B-TREE-SPLIT-ROOT(T)
4      B-TREE-INSERT-NONFULL(s, k)
5  else B-TREE-INSERT-NONFULL(r, k)
```

```
B-TREE-SPLIT-ROOT(T)
1  s = ALLOCATE-NODE()
2  s.leaf = FALSE
3  s.n = 0
4  s.c_1 = T.root
5  T.root = s
6  B-TREE-SPLIT-CHILD(s, 1)
7  return s
```

```
B-TREE-SPLIT-CHILD(x, i)
1  y = x.c_i                          // full node to split
2  z = ALLOCATE-NODE()                // z will take half of y
3  z.leaf = y.leaf
4  z.n = t − 1
5  for j = 1 to t − 1                 // z gets y's greatest keys ...
6      z.key_j = y.key_{j+t}
7  if not y.leaf
8      for j = 1 to t                 // ... and its corresponding children
9          z.c_j = y.c_{j+t}
10 y.n = t − 1                        // y keeps t − 1 keys
11 for j = x.n + 1 downto i + 1       // shift x's children to the right ...
12     x.c_{j+1} = x.c_j
13 x.c_{i+1} = z                      // ... to make room for z as a child
14 for j = x.n downto i               // shift the corresponding keys in x
15     x.key_{j+1} = x.key_j
16 x.key_i = y.key_t                  // insert y's median key
17 x.n = x.n + 1                      // x has gained a child
18 DISK-WRITE(y)
19 DISK-WRITE(z)
20 DISK-WRITE(x)
```

Search, insertion, and deletion are all O(logn). Same as a balanced binary search tree but the bases of the logs are larger which makes the coefficient smaller and operations more efficient

Complexity Guide

MT1

AVERAGE RUNNING TIME

	Random Access	Random Search	Insertion	Deletion
Array (fixed)	O(1)	O(n)	Not supported	Not supported
Array (dynamic)	O(1)	O(n)	See notes below	See notes below
Stack	O(n)	O(n)	O(1) only push at top	O(1) only pop from top
Queue	O(n)	O(n)	O(1) only push at tail	O(1) only pop from head
Linked List	O(n)	O(n)	O(1) only prepend/append	O(1) only delete head or tail
Hash Table	O(1)	O(1)	O(1)	O(1)
Binary Search Tree	O(logn)	O(logn)	O(logn)	O(logn)
Red Black Tree	O(logn)	O(logn)	O(logn)	O(logn)

NOTE:

- For a dynamic array, if you insert/remove at the beginning OR middle, complexity is O(n). Insert/remove at the end is O(1) amortized (on avg)
- In practice, a dynamic array is faster than a linked list for all operations if it holds under ~20,000 elements (due to memory contingency)
- For linked list, if you want to insert in the middle or end then the complexity becomes O(n)
- For hash table, if the distribution of keys is not uniform: it still may be O(1) if the size of the buckets is capped but you will need to resize the hash table more frequently which is expensive. If you don't cap buckets it becomes O(n) for all operations
- For binary search tree, if the tree is unbalanced then all operations become O(n)
  - A red black tree will never be unbalanced

In summary:

Data Structure	Average cases			Worst cases		
	Insert	Delete	Search	Insert	Delete	Search
Array/stack/queue	O(1)	O(1)	O(n)	O(1)	O(1)	O(n)
Linked list	O(1)	O(1)	O(n)	O(1)	O(1)	O(n)
Doubly linked list	O(1)	O(1)	O(n)	O(1)	O(1)	O(n)
Hash table	O(1)	O(1)	O(1)	O(n)	O(n)	O(n)
Binary search tree	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)

FINAL

	Make-Set	Find-Set	Union
Disjoint Sets	O(1)	O(height)	O(1)

- If you union by rank, find-set will be O(logn), otherwise O(n)

	Random Search	Insertion	Deletion
Heaps	O(n)	O(logn) only push at leaf	O(logn) only pop at root
B-Trees	O(logn)	O(logn)	O(logn)

SORTING ALGORITHMS

	Average Time	Worst Time	Space Complexity
Insertion Sort	O(n^2)	O(n^2)	O(1)
Heap Sort	O(nlogn)	O(nlogn)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(n^2)	O(1)

- Best case for Insertion Sort is O(n) if it's already sorted in ascending order. If it's descending order then it is O(n^2)

GRAPH ALGORITHMS

Breadth-First Search	O(|V| + |E|)/O(n + |E|)
Depth-First Search	O(|V| + |E|)/O(n + |E|)
Topological Sort	O(|V| + |E|)/O(n + |E|)
MST (Kruskal)	O(|E|log|V|)/O(|E|logn)/O(|E|log|E|)
MST (Heap Prim)	O(|E|log|V|)/O(|E|logn)/O(|E|log|E|)
Single-Source Shortest Path	O(|E|log|V|)/O(|E|logn)/O(|E|log|E|)

GRAPH REPRESENTATION

Node/edge management	Storage size	Add vertex	Add edge	Remove vertex	Remove edge	Query
Adjacency list	O (|V| + |E|)	O(1)	O(1)	O (|V| + |E|)	O(|E|)	O(|V|)
Adjacency matrix	O(|V|²)	O(|V|²)	O(1)	O(|V|²)	O(1)	O(1)

Find if an edge exists between two vertices

- O(n) for adjacency list
- O(1) for adjacency matrix

Find all adjacent vertices of a vertex.

- O(1) for adjacency list
- O(n) for adjacency matrix