Data Structures Notes

Abstract Data Types (ADT)

- A mathematical model of a data structure that specifies the type of the data stored and the desired operations.
- ADT separates the specifications of a data structures from its implementations.
- ADT is analogous to an advanced data structure which is or is made using a primitive data structure.
- For example:
  - ADT: Stack
  - Implementation: Array, Linked List

Characteristics of Algorithms

- Time Complexity - Running time of an algorithm
- Space Complexity - Memory consumption of an algorithm

Measuring Time Complexity

- We characterize the time complexity of an algorithm as a function of its input size.
- Time complexity can be measured using:
  - Experimental Analysis
  - Big-O Analysis

Experimental Analysis

- We execute the algorithm on various inputs and record time spent in each execution. The gathered data is then plotted to get a sense of how the algorithms time complexity

changes with increasing input sizes.

- Challenges of experimental analysis:
  - Comparison of two algorithms is difficult unless the experiments are performed in the same software and hardware environments.
  - Our input test set may leave out important inputs.
  - Out input test set may not be large enough.
  - Experimental analysis requires the algorithm to be implemented before it can analyzed.

Big-O Analysis

Why Big-O analysis

- Big-O provides us with a general way of analyzing the running time of algorithms that:
  - Takes into account all possible inputs
  - Is independent of hardware and software configuration
  - Does not require implementing the algorithms
  - Analysis can be done using pseudo-code

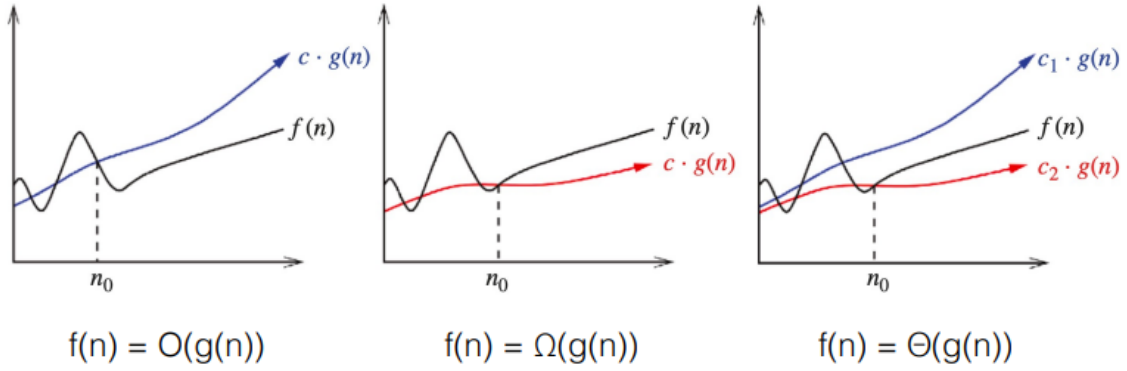
RAM Model of Computation

- Machine-independent algorithm design depends upon a hypothetical computer called the Random Access Machine:
  - Each primitive operation (e.g., +, -, =, if) takes 1 step
  - Loops and function calls are complex operations consisting of multiple steps
  - Each memory access takes exactly 1 step. Furthermore, we have as much memory as we need
- We measure the run time of an algorithm by counting the number of primitive operations

Different Big Notations (Types of Running Times)

Notation	Description

Notation	Description
Big O	Upper Bound - **Worst case** time complexity of an algorithm. (Running time on the most difficult input). Time complexity of an algorithm f(n) is g(n) if f(n) ≤ c*g(n) **for all n > n0**
Big Omega	Lower Bound - **Best case** time complexity of an algorithm. (Running time on the easiest input) Time complexity of an algorithm f(n) is g(n) if f(n) ≥ c*g(n) for **all n > n0**
Big Theta	Average - Complexity within upper and lower bound.

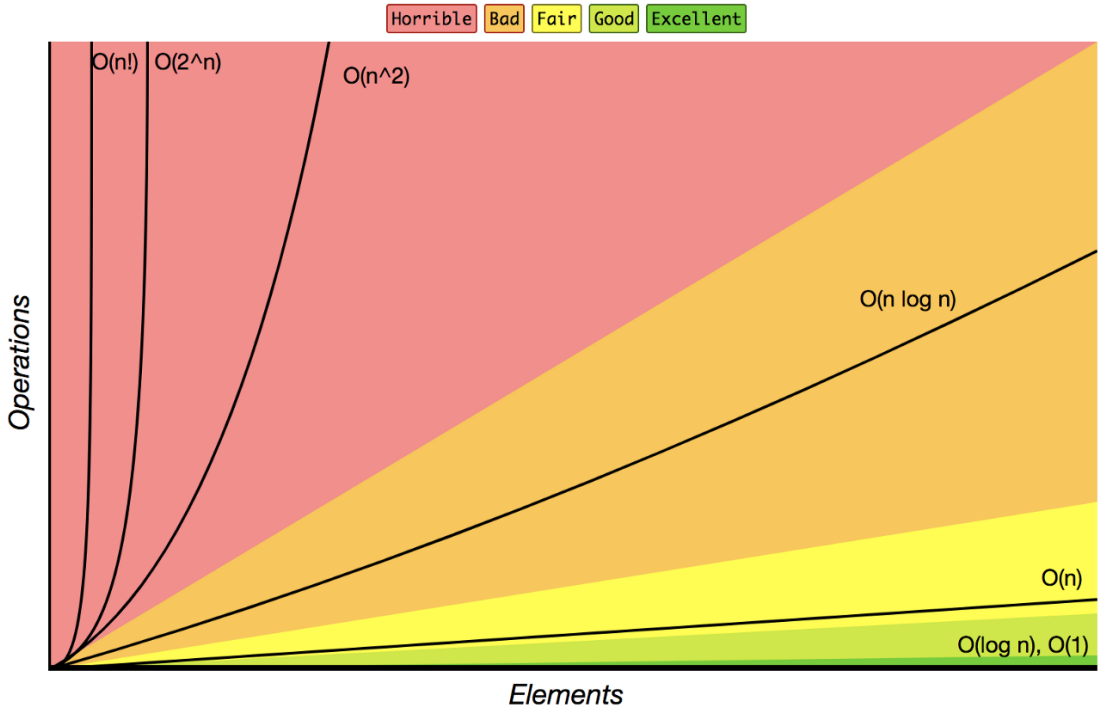


f(n) = O(g(n)) f(n) = Ω(g(n)) f(n) = Θ(g(n))

Big-O Complexities

Complexity	Name	Example
`O(1)`	Constant	Accessing a specific element in an array
`O(log n)`	Logarithmic	Finding an element in sorted array (divide and conquer)
`O(sqrt(n))`	Root	
`O(n)`	Linear	Loop through array elements
`O(n log n)`	Loglinear	
`O(n^2)`	Quadratic	Looking at every index in an array twice (nested for loop)
`O(n^3)`	Cubic	
`O(c^n)` (c is a constant)	Exponential	Double recursion in Fibonacci
`O(n!)`	Factorial	

Big-O Complexity Chart



Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Measuring Time Complexity

Primitive Operations Example

2	1	def find_max(data):
-	2	"""Return the maximum element from a nonempty Python list."""
2	3	biggest = data[0] # The initial value to beat
n * 2	4	for val in data: # For each value:
n * 2	5	if val > biggest # if it is greater than the best so far,
0 to n	6	biggest = val # we have found a new best (so far)
1	7	return biggest # When loop ends, biggest is the max

- find_max executes 5n + 5 primitive operations in the worst case, 4n + 5 in the best case

- Let $T(n)$ be worst-case time of **find_max.** Then
  $a(4n + 5) \leq T(n) \leq b(5n + 5)$

- Define:
  a = Time taken by the fastest primitive operation
  b = Time taken by the slowest primitive operation

- Hence, the running time $T(n)$ is bounded by two linear functions.

- Do we really care about the 5 in T(n) 5n+5? (constant factor)
  - Well, for large input sizes constant terms will be dominated by higher order terms so they wouldn't matter much
  - Ex: $T_1(n) = 5n+5$ vs $T_2(n) = 8n$

- Do we really care about multiplicative factor 5 in 5n+5?
  - Sure, it matters in practice
  - However, when comparing algorithms that differ in their highest order terms, it matters much less
  - Ex: $T_1(n) = 5n+5$ vs $T_2(n) = n^2+7$

Why drop values?

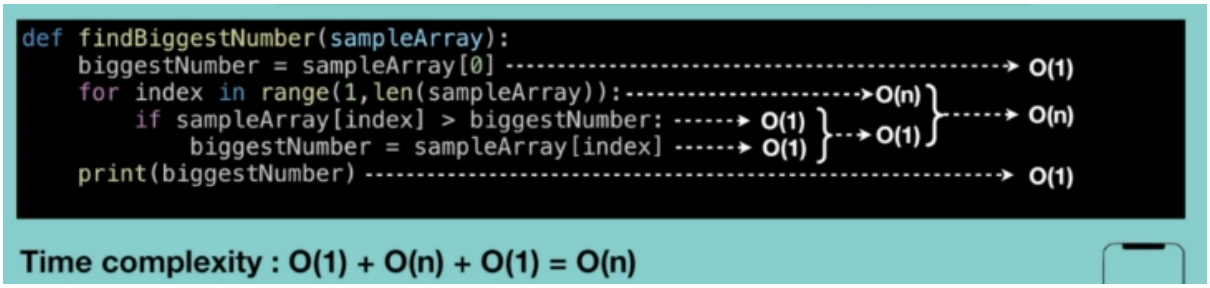
- The equation we get after the analysis of an algorithm is too precise, so we drop a few terms to get a more statistically appropriate term, which is easier to work with.
- What we care about is how an algorithm grows. (Linearly, exponentially, etc.)

Which values to drop?

- When calculating time complexity we drop:
  - Constant Factors
  - Multiplicative factors
  - **Lower Order Terms**

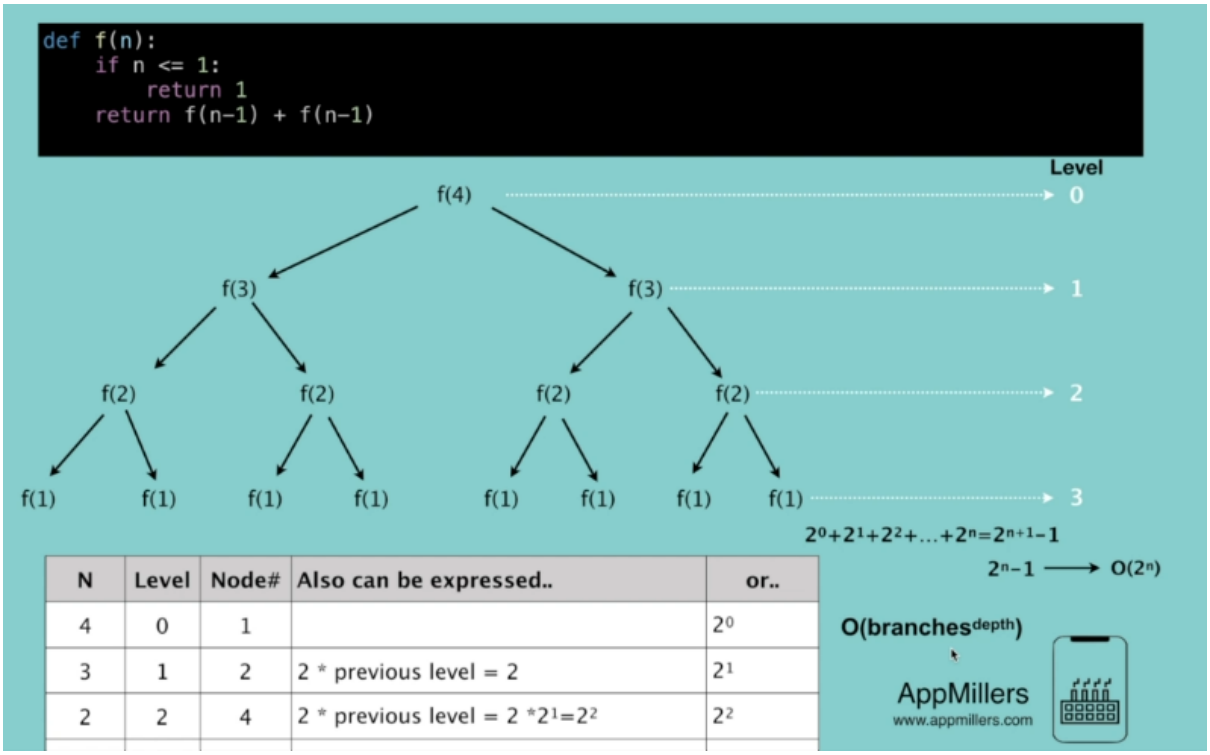
Time Complexity Cheat Sheet

No	Description	Complexity
Rule 1	Any assignment statements and if statements are executed once regardless of the size of the problem	`O(1)`
Rule 2	A `for` loop from `o` to `n`	`O(n)`
Rule 3	A nested loop of same sizes	`O(n^2)`
Rule 4	A loop whose controlling parameter is divided by two at each step	`O(log n)`
Rule 5	When dealing with multiple statements, add all of them upv	



Time Complexity of Recursive Function

NOTE: Not all recursive functions make multiple branches.

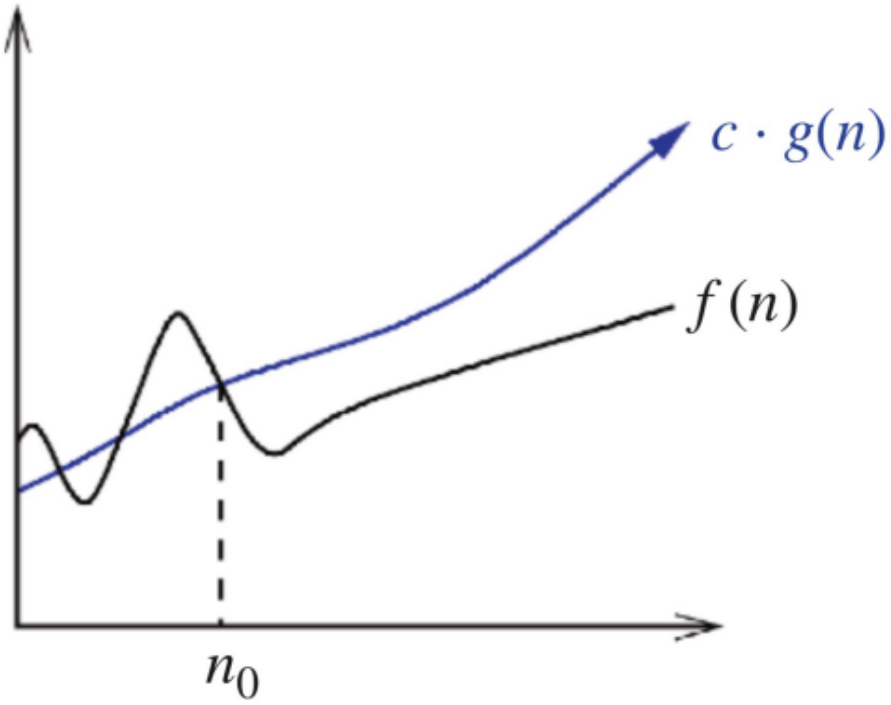


N	Level	Node#	Also can be expressed..	or..
4	0	1		2^0
3	1	2	2 * previous level = 2	2^1
2	2	4	2 * previous level = 2 *2^1=2^2	2^2

$$O(branches^{depth})$$

Understanding the Big-O (Dropping values)

$$f(n) \leq c \cdot g(n)$$

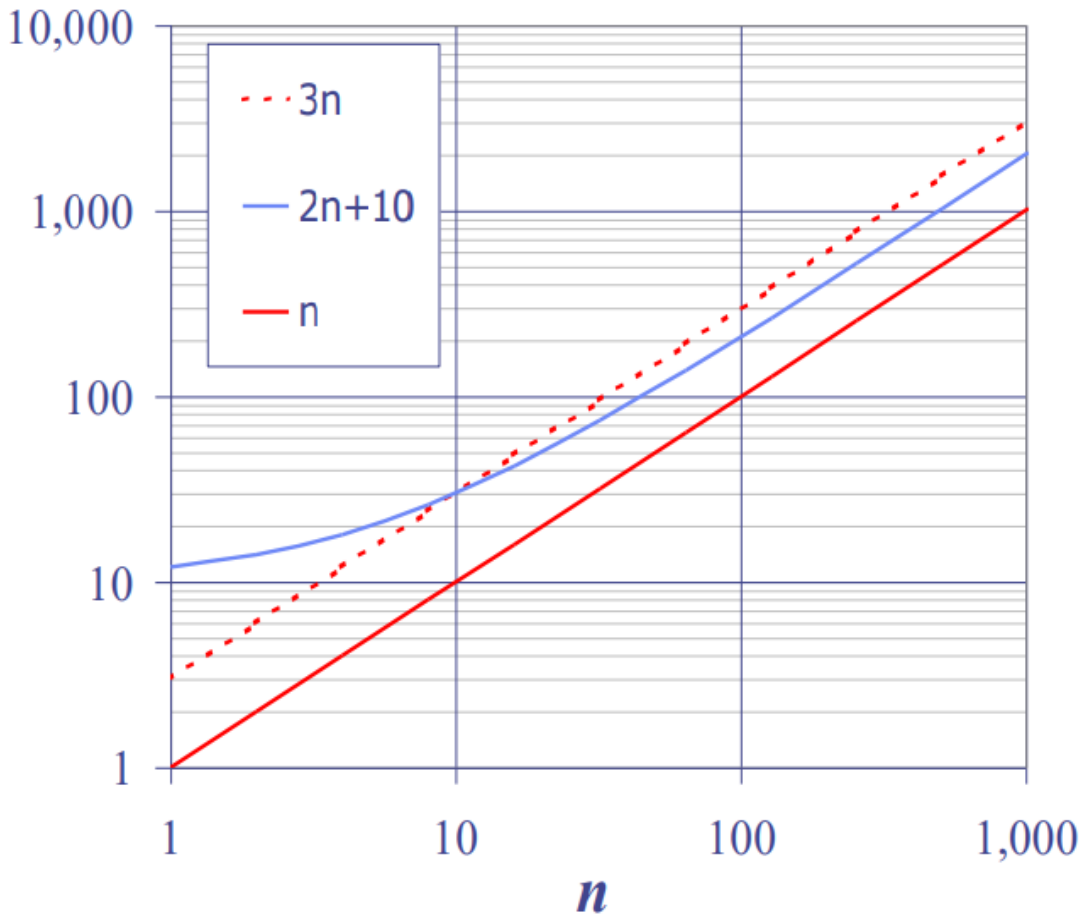


Is $2n + 10 \in O(n)$?

- The time complexity $f(n)$ of an algorithm is is $g(n)$, where $g(n)$ can be any of the Big-O Complexities if it satisfies the following equation for values of n beyond n_0:

$$f(n) \leq c \cdot g(n)$$

- $f(n)$ is the time complexity → $2n + 10$
- $g(n)$ is one of the big-o complexities → n
- c is a constant such that → $2n + 10 \leq c \cdot n$
- In this example, $c = 3$ and $n = 10$ satisfies the equation. (The values are found using trial and error)
- To satisfy the equation, we can use any value above 3 but in algorithmic analysis we try to use the closest possible bounds.
- In theory, the statement $2n + 10 \in O(n^2)$ is also true, but we only consider the closest bound.



Limitations of Big-O

- In practice the dropped values matter.
- When algorithms have different complexities constants, lower terms only matter when the problem size is small.

Also See

- Summation Formulas

Closed-form summations

$$\sum_{i=1}^{n} c = cn \tag{1}$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \tag{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \tag{3}$$

$$\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4} \tag{4}$$

$$\sum_{i=0}^{n} ar^i = a\left(\frac{1-r^{n+1}}{1-r}\right) \tag{5}$$

$$\sum_{i=0}^{\infty} ar^i = \frac{a}{1-r}, r < 1 \tag{6}$$

Summation Operations

Summation is a linear operator, so the usual rules for linear operators apply. Remember that sums are inclusive, so their endpoints are included in the computation.

$$\sum_{i=1}^{n}(x_i + y_i) = \sum_{i=i}^{n} x_i + \sum_{i=1}^{n} y_i \tag{7}$$

$$\sum_{i=1}^{n}(x_i + y_i)^k = \sum_{i=1}^{n}\sum_{j=0}^{k}\binom{k}{j} x_i^{k-j} y_i^j \tag{8}$$

$$\sum_{i=i}^{n} cx_i = c\sum_{i=1}^{n} x_i \tag{9}$$

It is sometimes helpful to rewrite sums, especially if there is a particular sub-sum of interest:

$$\sum_{i=0}^{n} i = \sum_{i=0}^{k} i + \sum_{i=k+1}^{n} i \tag{10}$$

$$\sum_{i=a}^{0} i, a < 0 = \sum_{i=0}^{-a}(-i) = -\sum_{i=0}^{-a} i \tag{11}$$

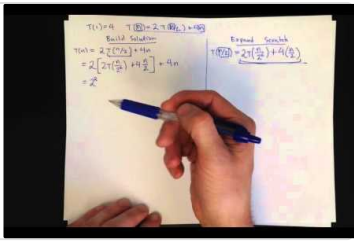
$$\sum_{i=0}^{n}(x_i + y_i)^k = (x_i + y_i)^0 + \sum_{i=1}^{n}(x_i + y_i)^k \tag{12}$$

- Recurrence Relations

Solved Recurrence - Iterative Substitution (Plug-and-chug) Method

This is an example of the Iterative Substitution Method for solving recurrences. Also known sometimes as backward substitution method or the iterative method...

https://www.youtube.com/watch?v=Ob8SM0fz6p0



- Logarithmic Identities

ULTIMATE PROPERTIES OF LOGARITHMS FORMULA SHEET

Property	Logarithm base b	Natural Log (base e)	Examples
1. Product Property	$\log_b(xy) = \log_b x + \log_b y$	$\ln(xy) = \ln x + \ln y$	$\log_3(10) = \log_3(5 \cdot 2) = \log_3 5 + \log_3 2$
2. Quotient Property	$\log_b\left(\frac{x}{y}\right) = \log_b x - \log_b y$	$\ln\left(\frac{x}{y}\right) = \ln x - \ln y$	$\log_3\left(\frac{5}{7}\right) = \log_4 5 - \log_4 7$
3. Powers Property	$\log_b x^p = p\log_b x$	$\ln x^p = p\ln x$	$\ln 27 = \ln 3^3 = 3\ln 3$
4. Root Property	$\log_b \sqrt[p]{x} = \frac{1}{p}\log_b x$	$\ln \sqrt[p]{x} = \frac{1}{p}\ln x$	$\log_2 \sqrt[3]{y} = \frac{1}{3}\log_2 y$
5. Inverse Property	$\log_b b^x = x$ or $b^{\log_b x} = x$	$\ln e^x = x$ or $e^{\ln x} = x$	$\log_3 3^4 = 4$
6. Identity Property	$\log_b b = 1$	$\ln e = 1$	$\log_{\sqrt{4}} \sqrt{4} = 1$
7. Zero Property	$\log_b 1 = 0$	$\ln 1 = 0$	$\log_4 1 = 0$
8. Change of base Property	$\log_b x = \frac{\log_a x}{\log_a b}$	$\ln x = \frac{\log_a x}{\log_a e}$	$\log_5 6 = \frac{\log 6}{\log 5}$
9. Equality Property	If $\log_b x = \log_b y$ then $x = y$	If $\ln x = \ln y$ then $x = y$	$\log_5 x = \log_5 6$ $x = 6$
10. Reciprocal Property	$\log_b \frac{1}{x} = -\log_b x$	$\ln \frac{1}{x} = -\ln x$	$\ln \frac{1}{5} = -\ln 5$

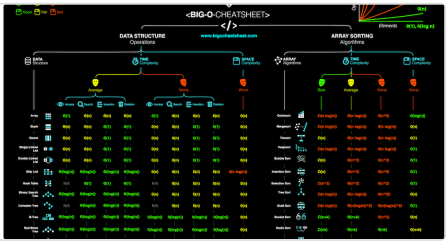
mathgotserved.com

Big-O Cheat Sheet

Know Thy Complexities!

Hi there! This webpage covers the space and time Big-O complexities of common algorithms used in Computer Science. When preparing for technical interviews in the past, I found myself spending hours crawling

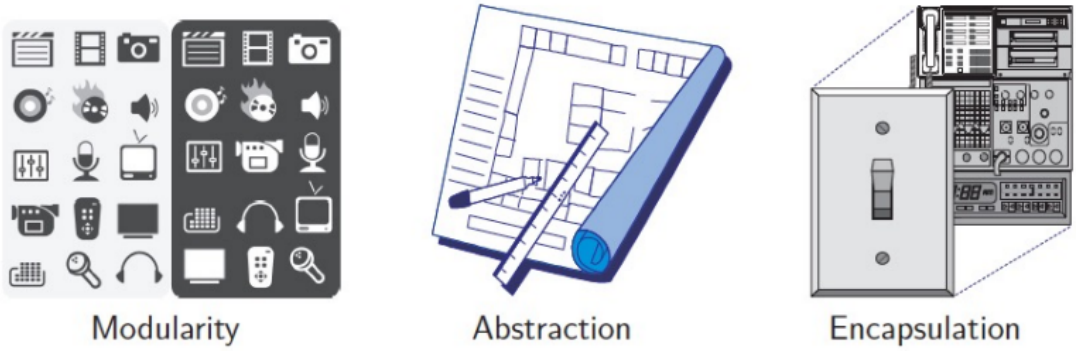
https://www.bigocheatsheet.com/



Python

Principles of Object-Oriented Programming

- Modularity - designing independent components
- Abstraction - hiding details under many classes and interface
- Encapsulation - interface for interacting with the lower level APIs



https://github.com/millionhz/python-roadmap

Contiguous vs. Linked Data Structures

- Data structures can be neatly classified as either contiguous or linked depending upon whether they are based on arrays or pointers:
  - Contiguously-allocated structures are composed of single slabs of memory, and include arrays, heaps, and hash tables.
  - Linked data structures are composed of multiple distinct chunks of memory bound together by pointers, and include lists, trees, and graph adjacency lists

Array List

- Contiguous area of memory consisting of equal size elements indexed by contiguous integers.
- Array list have O(1) indexing time.

Insert

- Just insert the element at a certain index
- `arr[idx] = 10`
- Time Complexity: $O(1)$

Insert (sorted)

- Find the appropriate index; insert at the element at the index; move all the other values forward.

- Time Complexity: $O(n)$

Delete (Do not retain order)

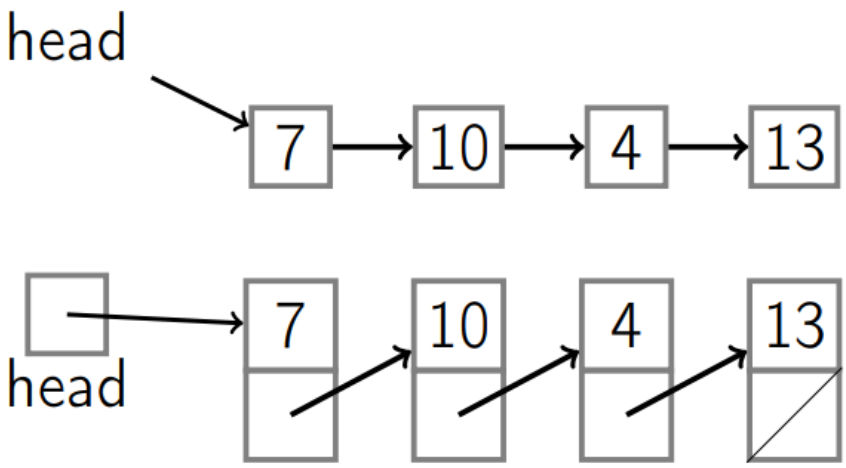
- Remove the last element and replace with the index to delete. (Decrease array size)
- `arr[idx_to_del] = arr[len-1]`
- Time Complexity: $O(1)$

Delete (retain order)

- Remove the specified element and move back all the other elements so no index remains empty.
- Recursive solution: Replace the current element with the next one → Delete the next element recursively.
- Time Complexity: $O(n)$

Linked List

Singly-Linked



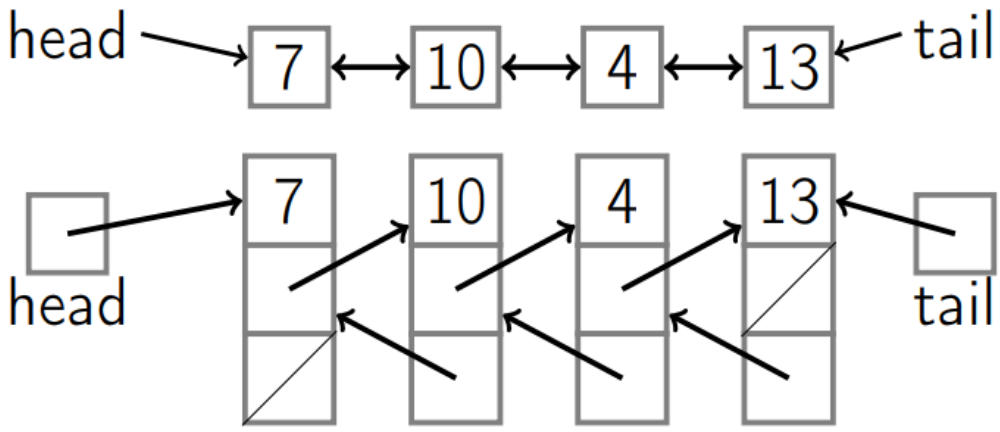
Node contains:

- key
- next pointer

`PushFront(Key)`	add to front
`Key TopFront()`	return front item
`PopFront()`	remove front item
`PushBack(Key)`	add to back
`Key TopBack()`	return back item
`PopBack()`	remove back item
`Boolean Find(Key)`	is key in list?
`Erase(Key)`	remove key from list
`Boolean Empty()`	empty list?
`AddBefore(Node, Key)`	adds key before node
`AddAfter(Node, Key)`	adds key after node

Singly-Linked List	no tail	with tail
`PushFront(Key)`	$O(1)$	
`TopFront()`	$O(1)$	
`PopFront()`	$O(1)$	
`PushBack(Key)`	$O(n)$	$O(1)$
`TopBack()`	$O(n)$	$O(1)$
`PopBack()`	$O(n)$	
`Find(Key)`	$O(n)$	
`Erase(Key)`	$O(n)$	
`Empty()`	$O(1)$	
`AddBefore(Node, Key)`	$O(n)$	
`AddAfter(Node, Key)`	$O(1)$	

Doubly-Linked



Doubly-Linked List	no tail	with tail
PushFront(Key)	$O(1)$	
TopFront()	$O(1)$	
PopFront()	$O(1)$	
PushBack(Key)	$O(n)$	$O(1)$
TopBack()	$O(n)$	$O(1)$
PopBack()	~~$O(n)$~~ $O(1)$	
Find(Key)	$O(n)$	
Erase(Key)	$O(n)$	
Empty()	$O(1)$	
AddBefore(Node, Key)	~~$O(n)$~~ $O(1)$	
AddAfter(Node, Key)	$O(1)$	

Stack [LIFO]

- We use arrays and linked lists to implement stack.
- Each stack operation is O(1): Push, Pop, Top, Empty.
- Stacks are occasionally known as LIFO queues.
- In a linked-list stack, the top of the stack is the head.
- in an arrayStack, the current highest filled index is the stack top

Definition

Stack: Abstract data type with the following operations:

- `Push(Key)`: adds key to collection
- `Key Top()`: returns most recently-added key
- `Key Pop()`: removes and returns most recently-added key
- `Boolean Empty()`: are there any elements?

Queues [FIFO]

- Each queue operation is O(1): Enqueue, Dequeue, Empty.

Linked-List implementation

- The linked list must have a tail pointer operations implemented, for lower time complexity.
- Elements are enqueued from the tail and are dequeued from the head.

Definition

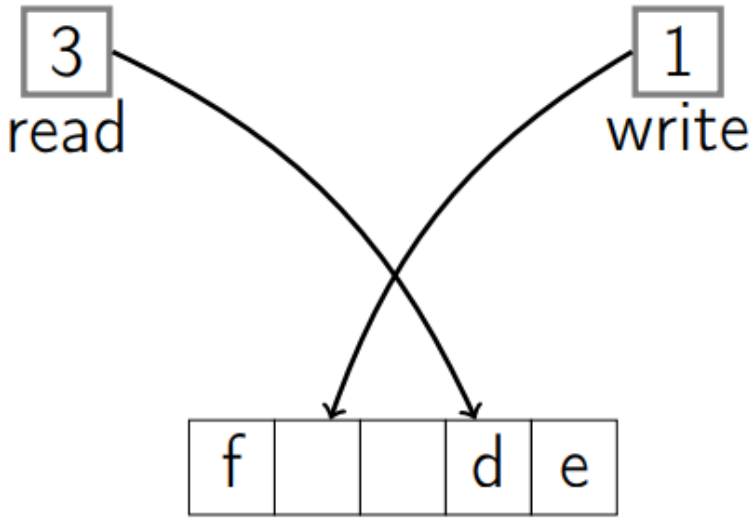
Queue: Abstract data type with the following operations:

- `Enqueue(Key)`: adds key to collection
- `Key Dequeue()`: removes and returns least recently-added key
- `Boolean Empty()`: are there any elements?

FIFO: First-In, First-Out

Array implementation

- We keep track of a read and write index which changes with each enqueue and dequeue operation.
- The read and write pointer can point to the same location only if the queue is empty.
- The arrays used in queues wrap around.



Trees

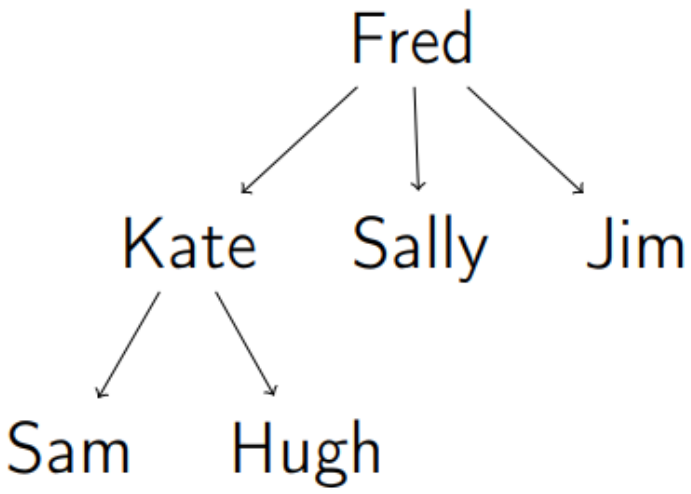
Definition

A Tree is:

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

- A tree is a recursive data structure (the members of the tree are tree themselves)
- Terminologies:
  - Root: Top node of the tree
  - Level: Fred is at level 1, Kate, Sally and Jim is at level 2
  - Parent: Fred is the parent of Kate, Sally and Jim
  - Child: Kate is the child of Fred

- Height: Distance to child node leaf; a root node has the height 1



Tree Height

- Total number of nodes on the longest path from root to leaf.
- A tree with a single root node is a tree with height 1.
- Complexity: $O(n)$

```
Height(tree)

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

Tree Size

- Total number of nodes.
- Complexity: $O(n)$

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

Traversal

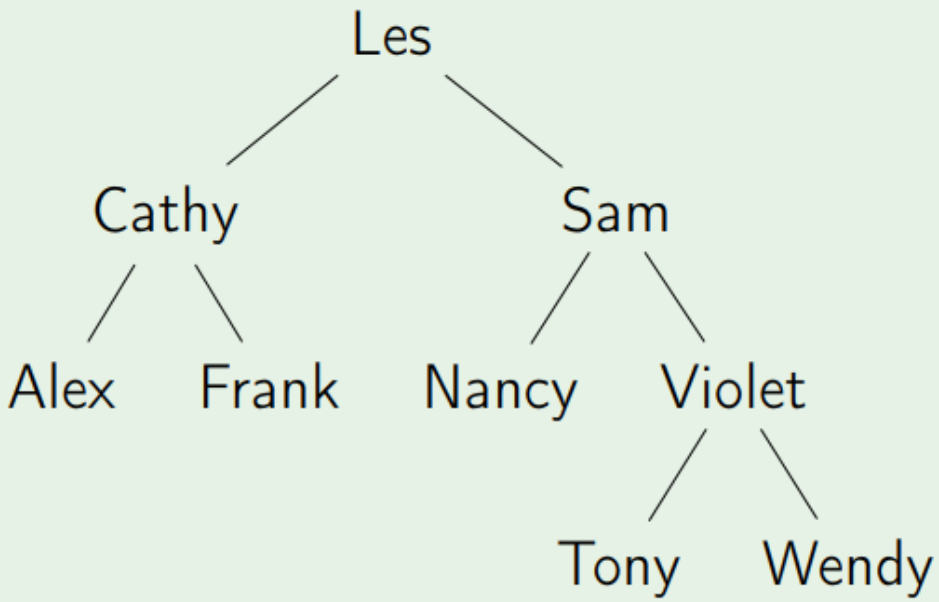
- Pre-Order gives the sequence in which elements were originally inserted in. [AVL]
- In-Order gives the elements in a sorted ascending order.

Depth-First Traversal

- We completely traverse one sub-tree before exploring a sibling sub-tree. (recursive) (Usually we explore left side and then move on to the right side)
- In-Order Traversal:

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

InOrderTraversal

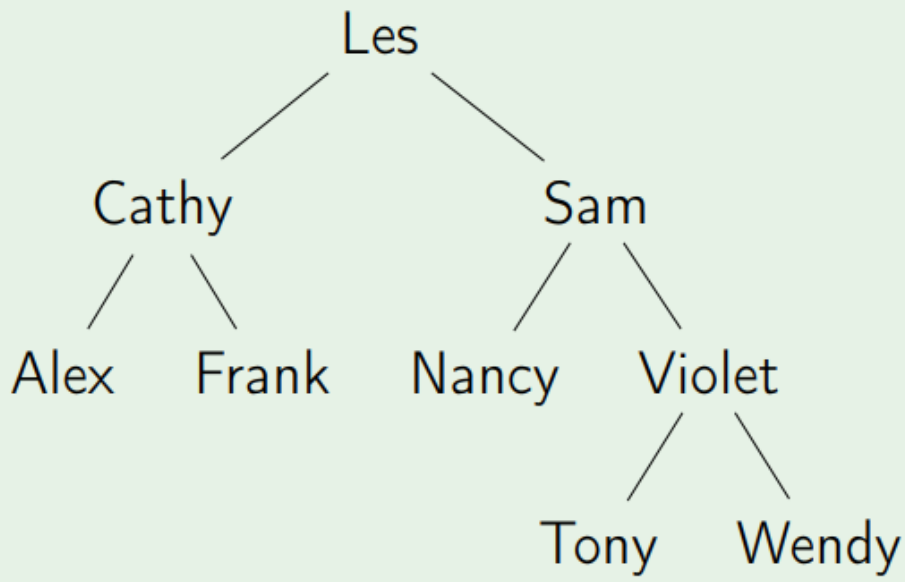


Output: Alex Cathy Frank Les Nancy Sam Tony Violet Wendy

- Pre-Order Traversal:

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

```
PreOrderTraversal
```



Output: Les Cathy Alex Frank Sam Nancy Violet Tony Wendy

- Post-Order Traversal:

```
PostOrderTraversal(tree)

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

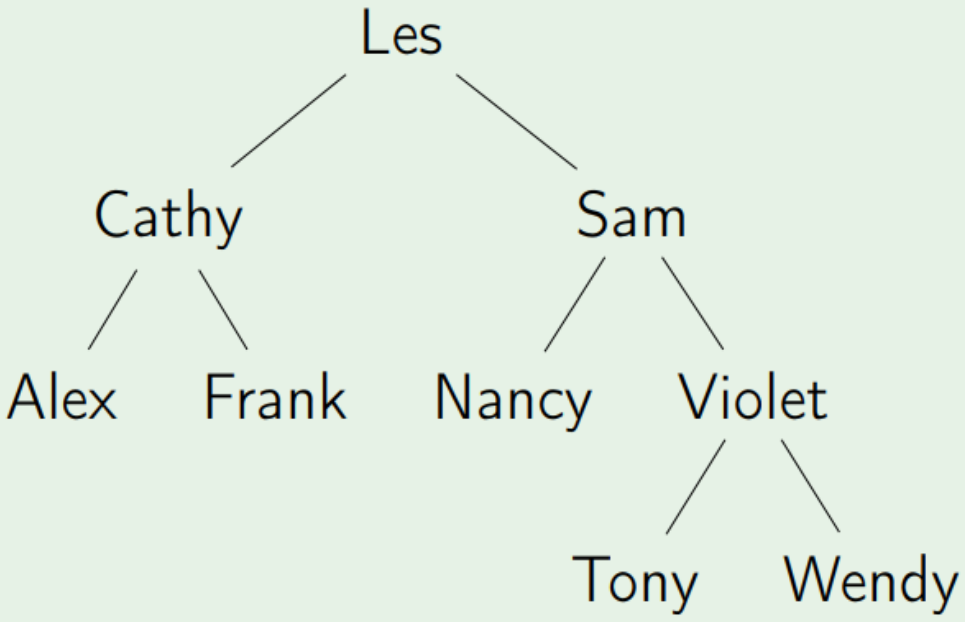
Breadth-First Traversal

- We traverse all nodes at one level before progressing to the next level.

LevelTraversal(*tree*)

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

LevelTraversal



Output: Les Cathy Sam Alex Frank Nancy Violet Tony Wendy

Local Search*

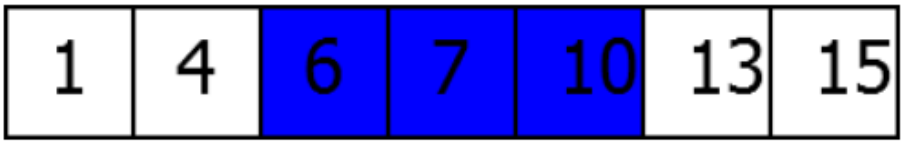
Definition

A Local Search Datastructure stores a number of elements each with a key coming from an ordered set. It supports operations:

- `RangeSearch`(x, y): Returns all elements with keys between x and y.
- `NearestNeighbors`(z): Returns the element with keys on either side of z.

Example

`RangeSearch`$(5, 12)$



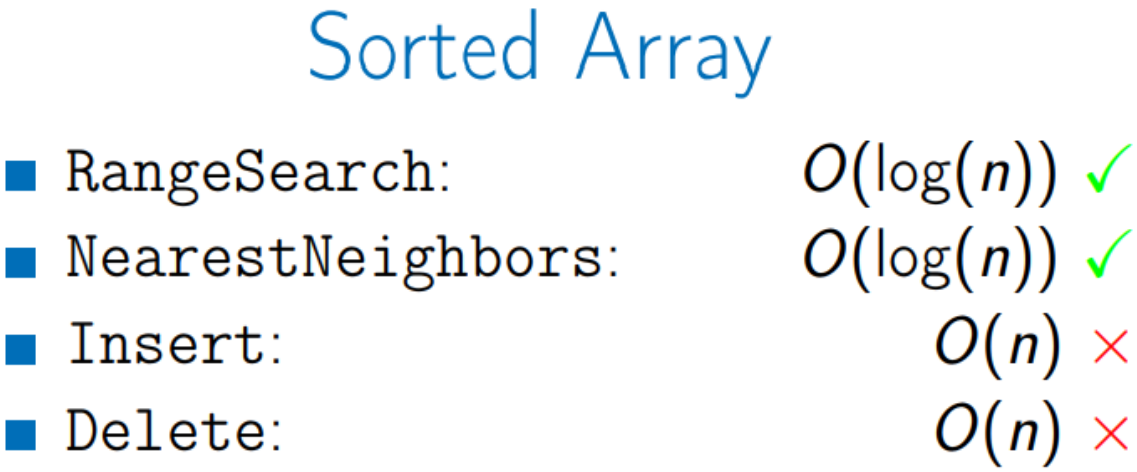
`NearestNeighbors`(3)

Unsorted Array

Array

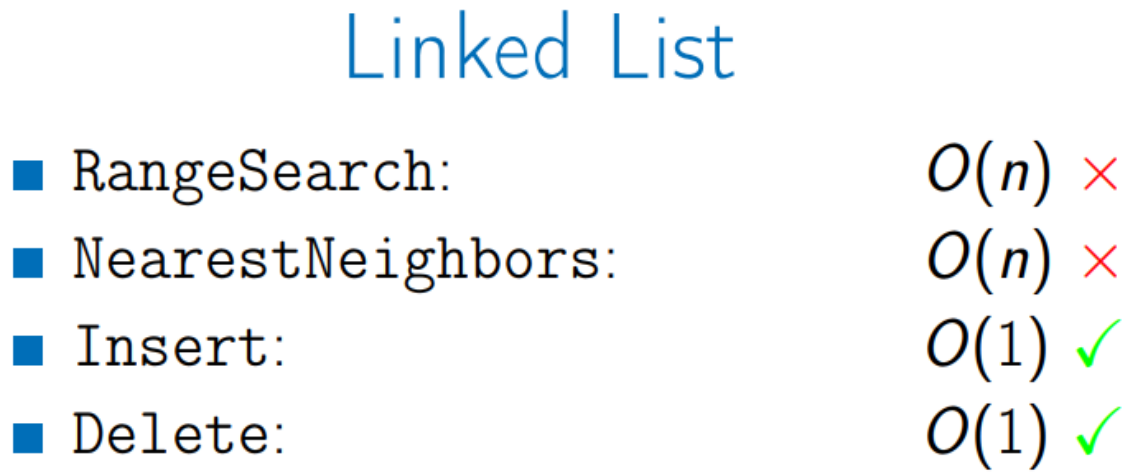
- `RangeSearch`: $O(n)$ ×
- `NearestNeighbors`: $O(n)$ ×
- `Insert`: $O(1)$ ✓
- `Delete`: $O(1)$ ✓

Sorted Array



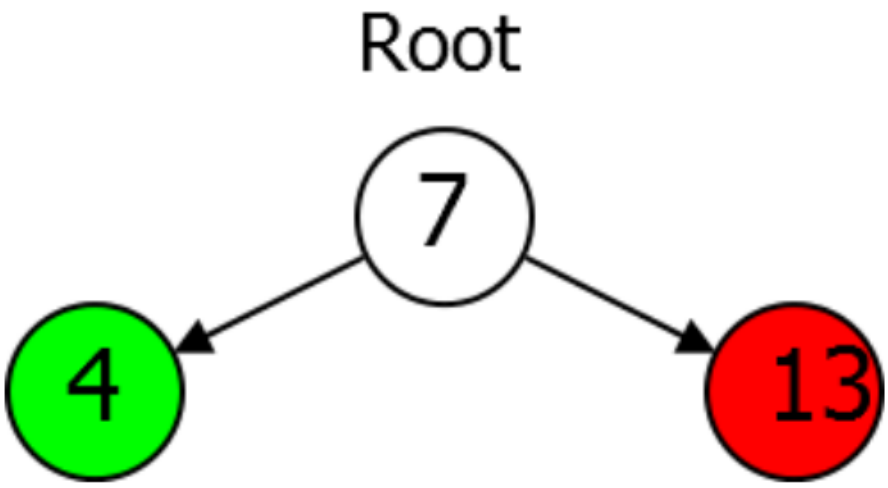
- Insert and Delete are O(n) as we need to shift the other elements to make room for the new element.

Sorted Linked-List



Binary Search Tree

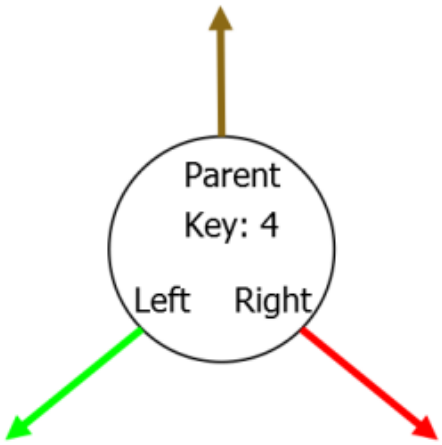
- Root node.
- Left subtree smaller keys.
- Right subtree bigger keys.



Node of a Binary Search Tree

Tree Node Data Type

- Key
- Parent
- Left Child
- Right Child



`Find(key=k, root=R) -> Node`

- Traverse the tree and look for the key at each level of traversal.
- If a certain key is missing, you can stop before reaching the null pointer to get the parent where the key(k) can be inserted as a child:

Find(k, R)

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

Find (modified)

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

`Next(node=N) -> Node`

- Find the next node to the provided node N.
- Essentially find the next available node in increasing order.

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

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

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

Range Search

Range Search

Input: Numbers x, y, root R

Output: A list of nodes with key between x and y

RangeSearch(x, y, R)

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

Insert

Insert

Input: Key k and root R

Output: Adds node with key k to the tree

Insert(k, R)

$P \leftarrow \texttt{Find}(k, R)$

Add new node with key k as child of P

Delete

Delete

Input: Node N

Output: Removes node N from the tree

Delete(N)

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

🚨 Time Complexity of all BST is O(h) where h is the height of the tree. In an unbalanced BST the depth can be n thus making the time complexity O(n).

Balanced BST - AVL Trees

- The left and the right subtree must have the same size across all subtrees.
- Time complexity of a perfectly balanced tree will be O(log(n))
- In AVL Trees:

$$\forall\ subtrees\ N\ |N.left.height - N.right.height| \leq 1$$

- The size of the tree **doubles** when the height of the tree **increases** by 2.
- An AVL will retain the indices of the array it was made from.

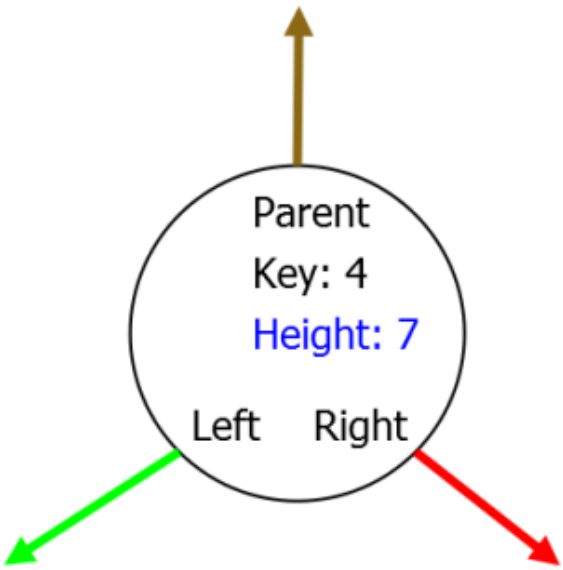
Balanced binary tree (AVL tree) animation

I made an animation of items being added to a balanced binary tree (specifically AVL tree), and rotations of the tree nodes as the tree balances itself. The number in the subscript is the height of a node (its distance from its deepest descendent leaf). The tree uses the height ...

http://www.motleytech.net/balanced-binary-tree-avl-tree-animation.html

🚨 Better to check assignment code

Nodes of AVL Trees

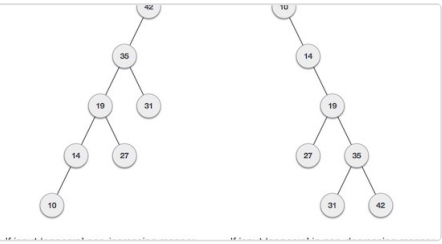


Rotation

Data Structure and Algorithms - AVL Trees

What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this − It is observed that BST's worst-case performance is closest to linear search algorithms,

https://www.tutorialspoint.com/data_structures_algorithms/avl_tree_algorithm.htm



$$\texttt{RotateRight}(X)$$

$$P \leftarrow X.\texttt{Parent}$$
$$Y \leftarrow X.\texttt{Left}$$
$$B \leftarrow Y.\texttt{Right}$$
$$Y.\texttt{Parent} \leftarrow P$$
$$P.\texttt{AppropriateChild} \leftarrow Y$$
$$X.\texttt{Parent} \leftarrow Y, Y.\texttt{Right} \leftarrow X$$
$$B.\texttt{Parent} \leftarrow X, X.\texttt{Left} \leftarrow B$$

`Rebalance(node=N)`

- You will only need to rebalance the path where the new node was inserted.
- The difference in height of the subtrees of the node being balances will never exceed 2 as we will keep on balancing the tree as it grows.

Rebalance(N)

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

AdjustHeight(N)

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

RebalanceRight(N)

$M \leftarrow N$.Left
if M.Right.Height $>$ M.Left.Height:
  RotateLeft(M)
RotateRight(N)
AdjustHeight on affected nodes

`AVLInsert(key=k, root=R)`

AVLInsert(k, R)

Insert(k, R)
$N \leftarrow$ Find(k, R)
Rebalance(N)

`AVLDelete(node=N)`

AVLDelete(N)

Delete(N)
$M \leftarrow$ Parent of node replacing N
Rebalance(M)

Insert

AVLInsert(k, R)

Insert(k, R)
$N \leftarrow$ Find(k, R)
Rebalance(N)

Delete

AVLDelete(N)

Delete(N)
$M \leftarrow$ Parent of node replacing N
Rebalance(M)

Merging in Trees*

- Complexity: O(n)

Merge

Input: Roots R_1 and R_2 of trees with all keys in R_1's tree smaller than those in R_2's

Output: The root of a new tree with all the elements of both trees

MergeWithRoot(R_1, R_2, T)

T.Left $\leftarrow R_1$
T.Right $\leftarrow R_2$
R_1.Parent $\leftarrow T$
R_2.Parent $\leftarrow T$
return T

Time $O(1)$.

Merge(R_1, R_2)

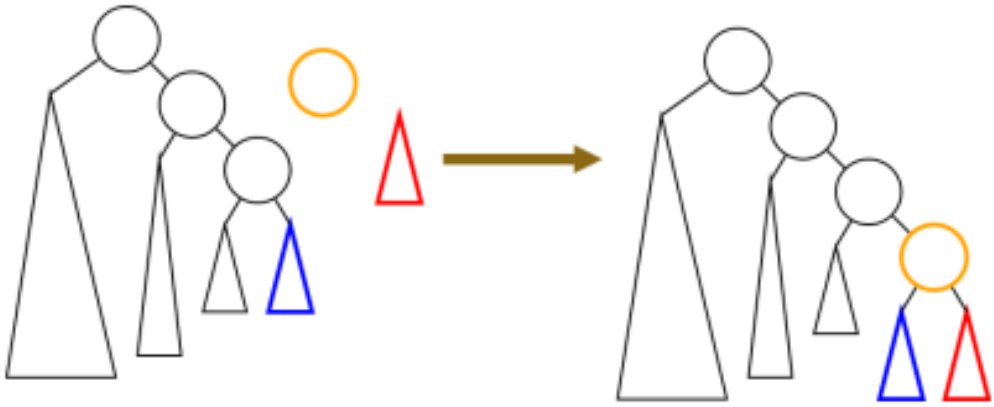
$T \leftarrow$ Find(∞, R_1)
Delete(T)
MergeWithRoot(R_1, R_2, T)
return T

Time $O(h)$.

Merge with Balance

Idea

Go down side of tree until merge with subtree of same height.



AVLTreeMergeWithRoot(R_1, R_2, T)

```
if |R1.Height − R2.Height| ≤ 1:
    MergeWithRoot(R1, R2, T)
    T.Ht ← max(R1.Height, R2.Height) + 1
    return T
```

AVLTreeMergeWithRoot(R_1, R_2, T)

```
else if R1.Height > R2.Height:
    R' ← AVLTreeMWR(R1.Right, R2, T)
    R1.Right ← R'
    R'.Parent ← R1
    Rebalance(R1)
    return root
else if R1.Height < R2.Height:
    ...
```

Split

Split

Input: Root R of a tree, key x

Output: Two trees, one with elements $\leq x$, one with elements $> x$.

🚨 Redo!

Application of Balanced BST/ AVL

`OrderStatistic(root=T, kth_value=k)`

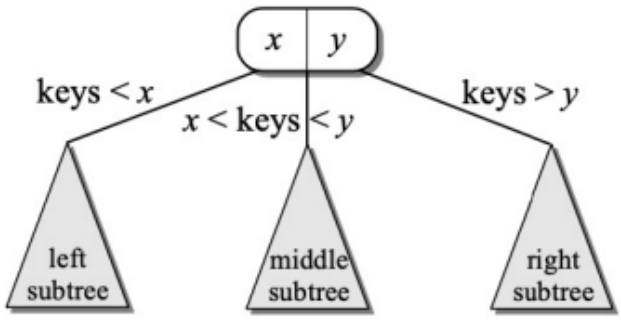
- Find the kth smallest element in the tree.

- To solve this problem we need to add an extra field of size to all the nodes that will contain the size of the subtree of the current node.
- Complexity: O(h)

```
OrderStatistic(R, k)

s ← R.Left.Size
if k = s + 1:
  return R
else if k < s + 1:
  return OrderStatistic(R.Left, k)
else if k > s + 1:
  return OrderStatistic(R.Right, k − s − 1)
```

2-3 Trees



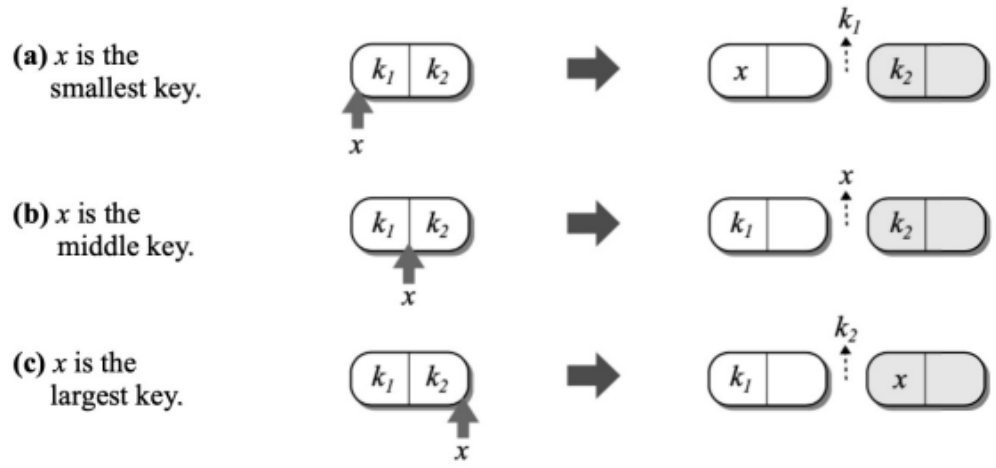
- A 2-3 tree is a search tree that is always balanced. (we dont need to perform rotations to balance it)
- Every node has capacity for:
  - one or two keys and 2 or 3 children
  - up to three children (called left, middle, and right child)
  - All leaf nodes are at the same level → this is how the balance property is maintained
  - Every internal node must contain two or three children
  - if the node has one key, it must contain two children
  - if it has two keys, it must contain three children
- Search Property: For each interior node, V: [intuitive]
  - All keys less than the first key of node V are stored in the left subtree of V
  - If two children: all keys greater than the first key of node V are stored in the middle subtree of V
  - If three children:
    - all keys greater than the first key of node V but less than the second key are stored in the middle subtree of V; and
    - all keys greater than the second key are stored in the right subtree
- 2-3 Trees are always balanced and hence have the Time Complexity of O(log n)
- 2-3 Trees are more efficient than AVL trees as AVL's rebalancing is log(n) while 2-3's rebalancing is O(1)

Insertion

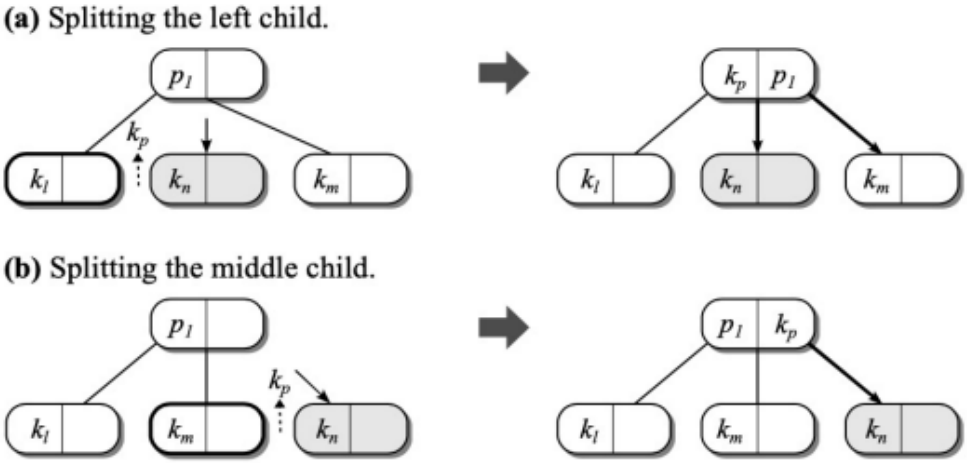
B-Tree Visualization

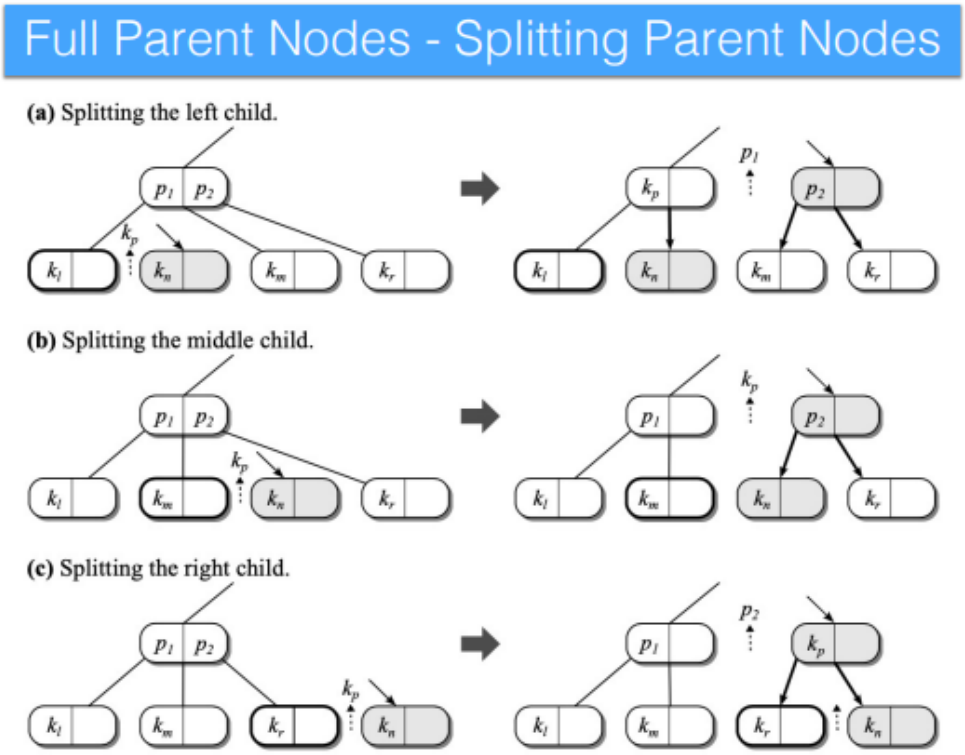
https://www.cs.usfca.edu/~galles/visualization/BTree.html

- Promotion CheatSheet



Q2: Where to place the promoted key and how to adjust child pointers?





Memory and Trees

- A node of a tree stores:
  - Address of left child (4-8 bytes)
  - Address of right child (4-8 bytes)
  - Address of parent (4-8 bytes) (optional)
  - Key (key used to index the data) (≥ 4 bytes) (usually an integer)
  - Data/value (multiple bytes of data)
- Thus a single node inside the tree can take up to 1Kbs of space or even more. Storing a large tree with 1Kb per node will require multiple Gigabytes of storage space. To solve this memory crisis we make use of B+ trees.

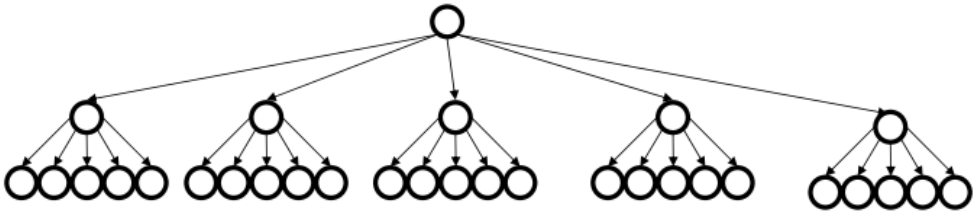
Caching

- OS caches frequently accessed data into faster storage mediums like L1 an L2 caches to reduce memory access times.
- OS uses temporal and spatial locality to speed up access.
  - Temporal Locality - Recently accessed memory space is likely to be accessed again and so the OS will cache it to faster memory.

    - Spatial Locality - Memory spaces nearby a recently accessed memory space are likely to be accessed and so the OS will cache them to faster memory. (Arrays are faster because when we access a single element the rest of the array is cached so further access become faster)
- Nodes inside a tree are located at random places in the memory so caching a tree in an efficient way becomes a difficult task.

M-aray Search Tree

- Suppose we use a search tree with M children/node
    - We use an array to store children in **sorted order**
    - Choose M so that it fits into a disk block (1 access for an array) (caching possible)



$$\text{Height of M-aray} = O(log_m n)$$

$$\text{Worst case running time} = O(log_2 m * log_m n)$$
$$\text{binray search * traversal}$$

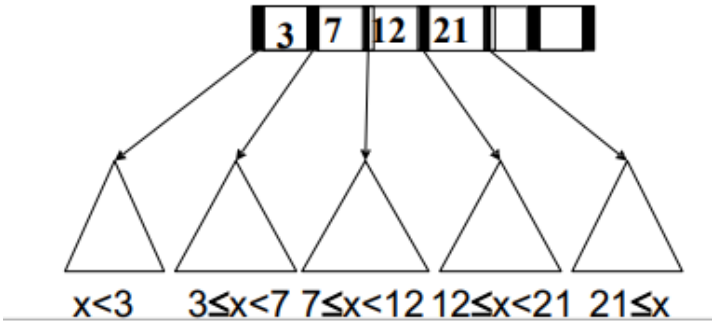
Benefits of M-aray Search Tree

- Smaller height
    - Path length reduces as we increase M
- Potential improvements in the running time of operations
    - Smaller height means potentially less number of nodes to traverse
    - Storing children nodes in an array exploits memory locality (good for caches)

B/B+ Trees

- B+ trees are use by databases and OS to store obnoxiously large datasets.
- B+ trees are designed in such a way that they only load the data required into the memory.

- B+ trees have two types of nodes: internal nodes (signposts) & leaf nodes (i.e., data nodes)
  - Each internal node has room for up to **M-1** (sorted) keys & **M** pointers
  - Each leaf node has room for **L** key-value pairs, sorted by key
- Different from BST in that we don't store data in internal nodes but `find` is still an easy root-to-leaf recursive algorithm:
  - At each internal node do binary search on (up to) M-1 keys to find the branch to take
  - At the leaf do binary search on (up to) L data items



Calculating M and L

- The value of M depends on the size of a disk block. Disk block is the largest size of memory that a disk controller can read in 1 I/O operation. The value of M is set to the size of the disk block so that the max amount of keys can be read in 1 disk operation.
- Same is the case with the value of L. The internal nodes contains pointers only (8 bytes) so its they all can have a consistent number. The leaf nodes on the other hand contain data which can be more than or less than 8 bytes.

Calculating L

L is the number of data records that can be stored in each leaf. Since we want to do just one disk access per leaf, this is the same as the number of data records per disk block.

Since a disk block is 4096 and a data record is 1024, we choose L = ⌊4096 / 1024⌋ = 4 data records per leaf.

Each interior node contains M pointers and M-1 keys. To maximize M (and therefore keep the tree flat and wide) and yet do just one disk access per interior node, we have the following relationship

$$4M + 8(M - 1) <= 4096$$
$$12M <= 4104$$
$$M <= 342$$

So choose the largest possible M (making tree as shallow as possible) of 342.

B+ Tree Rules

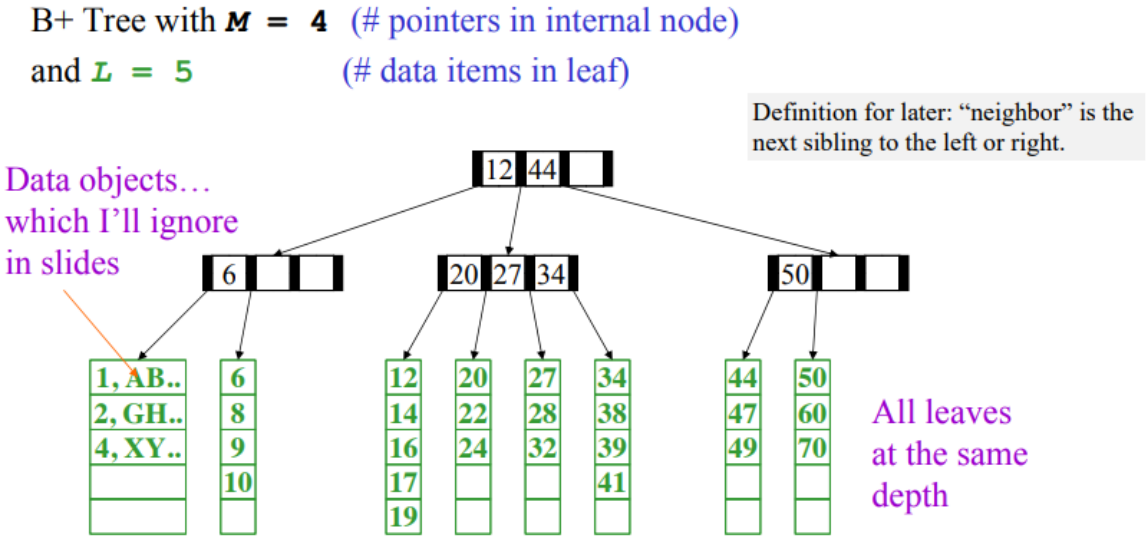
- Internal nodes
  - store up $ceil(\frac{M}{2}) - 1$ to $M - 1$ keys

    - have between $ceil(\frac{M}{2})$ and M children - this allows two half filled nodes to be joined to make a single full node.
- Leaf nodes
    - store data and all leaf nodes are at the same depth
    - contain between $ceil(\frac{L}{2})$ and L data items, stored in **sorted order**
- Root node (special)
    - has between 1 and $M-1$ keys
    - has between 2 and M children (or root could be a leaf)

The half full (M/2) property of the internal and leaf nodes ensure balance.

- All leaf nodes are at the same level.
- B-Trees grow and shrink from root instead of the leaves. (just like 2-3 trees)

B+ Tree Example



`search` Complexity:

- Find the correct subnode at every signpost: $O(log_2 M)$
    - From the M subnodes do a binary search to find which node we need to traverse down from.

    - (we need to do this on every subnode as all the data is stored at the leaves always)
- Go through the depth of the tree: $O(log_M N)$
- Find the object in the leaf: $O(log_2 L)$
    - From the L nodes in the leaf perform a binary search to find the correct object
- **Total Time Complexity for search:**

$$O(log_2 L + log_M N * log_2 M)$$

Disk Friendliness

- Many keys are stored in one node so all brought to memory/cache in one disk access
- Internal nodes contain only keys
    - Much of tree structure can be loaded into memory irrespective of data object size
    - Data resides on the disk

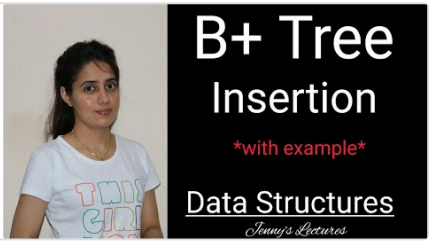
https://s3-us-west-2.amazonaws.com/secure.notion-static.com/065aa566-e47d-456b-be82-cd9a36f1d730/Lec12-CS202-Spring2022.pdf

Insertion

5.29 B+ tree insertion | B+ tree creation example | Data structure

Learn how to insert data in b+ tree of order 4. Step by step instructions showing the insertion process in b+ treeProperties of B-tree:https://www.youtube.co...

https://www.youtube.com/watch?v=DqcZLulVJ0M



Insertion Complexity

- Find correct leaf: $O(log_M * log_M N)$ - [binary search on each node along the path and traversal]
- Insert at leaf: $O(log_2 L + L)$ - [binary search + move elements by one spot]
- Split Leaf: $O(L)$ - [creating a new leaf and copying items]
- Split parent all the way up to root: $O(M * log_M N)$ - [O(M) for splitting nodes and traversal to]

$$O(L + M * log_M N)$$

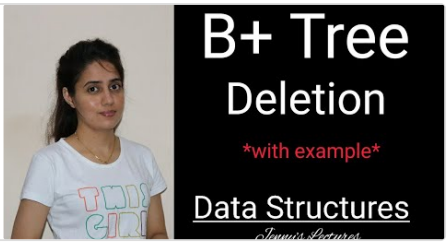
- The complexity is fine as **splits** which require more computational time are not very common.

Deletion

5.30 B+ tree deletion| with example |Data structure

Discussed all the cases of deleting a key from b+ tree with example. Step by step instructions showing how to delete data from b+ tree.B+ tree insertion: ht...

https://www.youtube.com/watch?v=pGOdeCpuwpI



Deletion Complexity:

- Find correct leaf: $O(log_M * log_M N)$ - [binary search on each node along the path and traversal]
- Remove at leaf: $O(log_2 L + L)$ - [binary search + move elements by one spot]
- Adopt from or merge with neighbor: $O(L)$
- Adopt or merge all the way up to the root: $O(M * log_M N)$

$$O(L + M * log_M N)$$

- The complexity is fine as **merges** which require more computational time are not very common.

Insert

- Find correct leaf: $O(\log_2 M \log_M n)$
- Insert in leaf: $O(L)$
- Split leaf: $O(L)$
- Split parents all the way up to root: $O(M \log_M n)$

Delete

- Find correct leaf: $O(\log_2 M \log_M n)$
- Remove from leaf: $O(L)$
- Adopt/merge from/with neighbor leaf: $O(L)$
- Adopt or merge all the way up to root: $O(M \log_M n)$

🚨 B+ and B trees are the same. The only difference is that B+ trees stores data only in the leaf nodes and have a property called L.

Hash Tables

Load factor of a hash table

$$\alpha = \frac{n}{m}$$
$$n\text{: length of the dataset}$$
$$m\text{: size of the hash table array}$$

- Time complexity of hash-tables in terms of load-factor is: $\theta(1 + \alpha)$
- Our goal is to keep the load-factor closer to 1 so that the time complexity of our table is $O(1)$
- As long as $m = \theta(n)$, then $\theta(1 + \frac{n}{m}) = \theta(1)$
- We can bound the load-factor to $\theta(n)$ by **increasing m** when required

Closed Addressing

- Use a hash function to calculate an index to which a key is mapped to.

Direct Addressing

- Direct addressing is the simplest form of hashtable.
- Keys in form of integers are used as indices of arrays to store data.
- Non-numeric keys can not be used as hashing them might cause collisions.
- Non-numeric keys and collisions can be solved by using:
  - Separate Chaining
  - Liner Probing

Separate Chaining

- Declare an array of a certain size `m`.
- Hash the key to an integer value, ranging from 0 to `m` so that it can used as an index.
- Store the object into the array at the location specified by the hashed value of the key.

- If the index is already filled, use a **linked list** to **chain objects together**.
- The expected length of a chain is equal to the load factor of the hashtable. (read ahead)

Time Complexity of Chaining

- Get, Set, Update: $\theta(a+1)$ - where c is the length of the longest linked lists
- Memory Complexity: $\theta(n+m)$ - where n is the size total size of our data and m is the size of the master array of the hash table.
- When the load factor of chained hashtable reaches 1.5 we double m (size of the array), this decreases the average size of the linked lists.

Open Addressing

- Unlike chaining, no separate data structure is used for storing items in Open Addressing.
- All items are stored within the table, one item per slot → m >= n (number of items to be stored
  should less than the table size).
- Hash function now specifies the order of slots to try for a key (for insert, delete, lookup) not just one slot.

Linear probing (an implementation of open addressing)

- (Keep probing each index in a linear fashion till you find an empty index)
- Hash the key with the following function and **if the location is free**, store in the array:

$$h(key) = key \,\%\, M$$
$$key\text{: hashed value}$$
$$M\text{: being the size of the array}$$

- If the location is filled, use the following hash function to find and store the value on the next free location (we are essentially adding 1 to the original hash every time we encounter a filled location):

$$h(home, i) = (home + i) \,\%\, M$$
$$home = h(key) = key \,\%\, M$$
$$i = 1 \text{ if first slot was filled or } i = 2 \text{ if the second slot was filled too ...}$$

Search

How do we do search on this hash table?

- Compute the hash of the key and continue incrementing i until we find the key
- If you find an empty slot, then stop (which indicates that the key is not in the table)

Deletion

- We use a deletion property to parse the empty indexes properly.

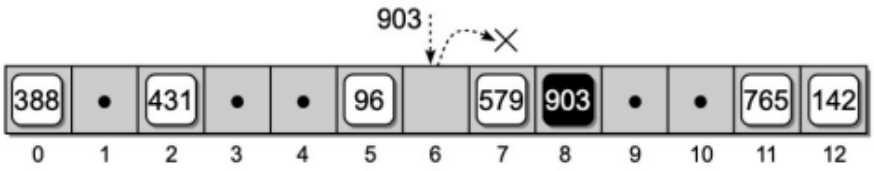


Figure 11.9: Incorrect deletion from the hash table.

- Insert a flag "deleted" (which takes the value 0 or 1)
  - For insert, treat "deleted=1" as "none" and do the insertion
  - For search, treat "deleted=1" as "continue looking head"

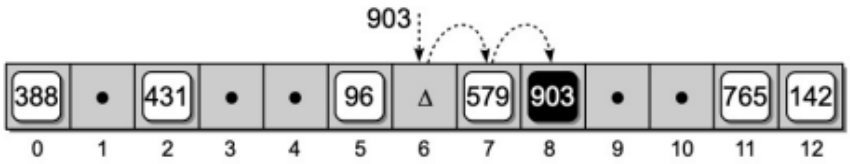


Figure 11.10: The correct way to delete a key from the hash table.

Downsides of Open Addressing (without cluster control)

- Clustering (consecutive groups of occupying slots):
- Colliding items lump together, causing future collisions which then cause a longer sequence of probes

- This slows down the running time of operations

Counteracting Clustering

Spread out the hash values, instead of them being next to each other

Step-size

$$h(home, i, c) = (home + i * c) \% M$$
$$c\text{: pre-defined step size}$$

Quadratic Probing

$$h(home, i) = (home + i^2) \% M$$

Problem with Step-size and Quadratic hashing

- Two keys that map to the same value will have the same hashes even after multiple probing attempts.
  - e.g. 648 has the sequence 11, 12, 2, 7, 1;
  - 388 which maps to the original hash 11 will also have the same sequence of 11, 12, 2, 7 ,1

Double hashing (best approach)

$$h(home, i, key) = (home + i * hp(key)) \% M$$
$$hp(key)\text{: secondary hash function}$$
$$key\text{: the value being hashed}$$

- Secondary hash function must conform to the following rules:
  - The output should be > 0, else multiple hash attempts will result in the same original hash
  - The output should be less than the length of the array (< M)
  - Example function:

$$hp(key) = 1 + key \% 8$$
$$where\ 8 < M$$

Resizing the hash table (open addressed)

- As the table fills up, collisions become more common.

- A hash table on average works best when its not more than three quarters full. (keep load factor between 1/2 to 2/3)
- We will double the size of the hash table once it reaches a certain threshold, and **re-compute all the hashes.**

Time Complexity

- Load factor

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

- Number of probes needed on average

$$num\ probes = \frac{1}{1-\alpha}$$

- Time Complexity on average:

$$\theta(1)$$
$$\text{provided that } \frac{1}{2} < \alpha < \frac{2}{3}$$

Open Addressing	Closed Addressing
the keys may have been stored in an open slot different from the one to which it originally mapped	the key is contained within the entry to which it mapped

Open Addressing vs Chaining

- Open Addressing yields better cache performance (better memory usage, no pointers needed)
- Chaining is less sensitive to hash functions and the load factor
- Experiments and average-case analysis suggest that for best performance keep
  - a < 0.5 for the open-addressing schemes
  - a < 0.9 for chaining

Hash Function

- A hash function is a composition of two functions

$$hash(key) = c(hc(key))$$

    - hashcode (hc): maps non-integer keys to integers
    - compression (c): maps integers to integers in a smaller range (0,1,…,m-1)
- The advantage of separating the hash function into two such components is that the hash code portion of that computation is independent of a specific hash table size
- This allows the development of a general hash code for each object that can be used for a hash table of any size; only the compression function depends upon the table size

Hashcode

- Integer-cast:
    - cast the key into an integer
    - if the key is more than 32-bit in terms of integer, just consider the higher or lower order 32-bits
    - Problem: moderately similar keys will cause collisions e.g. 123657, 123679
- Component sums:
    - Partition the bits of the key into components of fixed lengths and we sum the components ignoring overflows
    - Problem: Highly similar keys will cause collisions, e.g. "stop", "tops", "pots", and "spot”
- Polynomial Accumulation
    - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16, or 32) and use a fixed value z (hashcode base) to evaluate the hash

$$p(z) = a_0 + a_1 z + a_2 z^2 + ... + a_{n-1} z^{n-1}$$

    - Hash code base (z) is a small prime number.

Compression Function

MOD function:

$$c(k) = |i| \,\%\, m$$

- If we take m to be a prime number, then this hash function helps “spread out” the distribution of hashed values

MAD method

- solves collisions
- The use of prime number reduces collisions (this is because of the nature of the prime numbers)

The Multiple Add and Divide (MAD) Method

- A more sophisticated compression function, which helps eliminate repeated patterns in a set of integer keys is

$$h(k) = [(ai + b) mod\ p]\ mod\ N$$

Where N is the size of the hash table, p is a prime number larger than N, and a and b are integers chosen at random from the interval [0, p - 1], with $a > 0$.

Double Hashing and MAD

Double Hashing	MAD
Avoids clustering while probing when a collision occurs.	Prevents collision from the get go.

Set

- **A set is just like a map, but we store the keys only.**

Priority Queues

- A priority queue is a generalization of a queue where each element is assigned a priority and elements come out in order by priority.

Definition

Priority queue is an abstract data type supporting the following main operations:

- `Insert`(p) adds a new element with priority p
- `ExtractMax`() extracts an element with maximum priority

- `Remove`(it) removes an element pointed by an iterator it
- `GetMax`() returns an element with maximum priority (without changing the set of elements)
- `ChangePriority`(it, p) changes the priority of an element pointed by it to p

Binary Heap (Priority Queue)

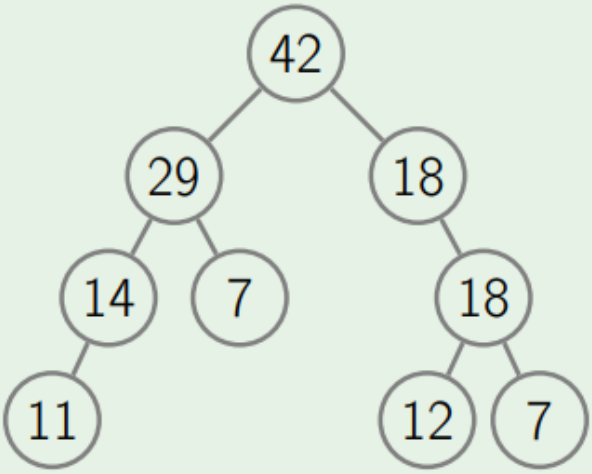
- Binary heaps is a way to implement priority queues.
- They have a few special restrictions, in addition to the usual binary tree:
  - Must be complete (also known as shape property). All levels are filled except the last level which is filled from left to right
  - Ordering of keys must obey the heap-order property
  - Max-heap: a parent's key is always ≥ both its children's keys

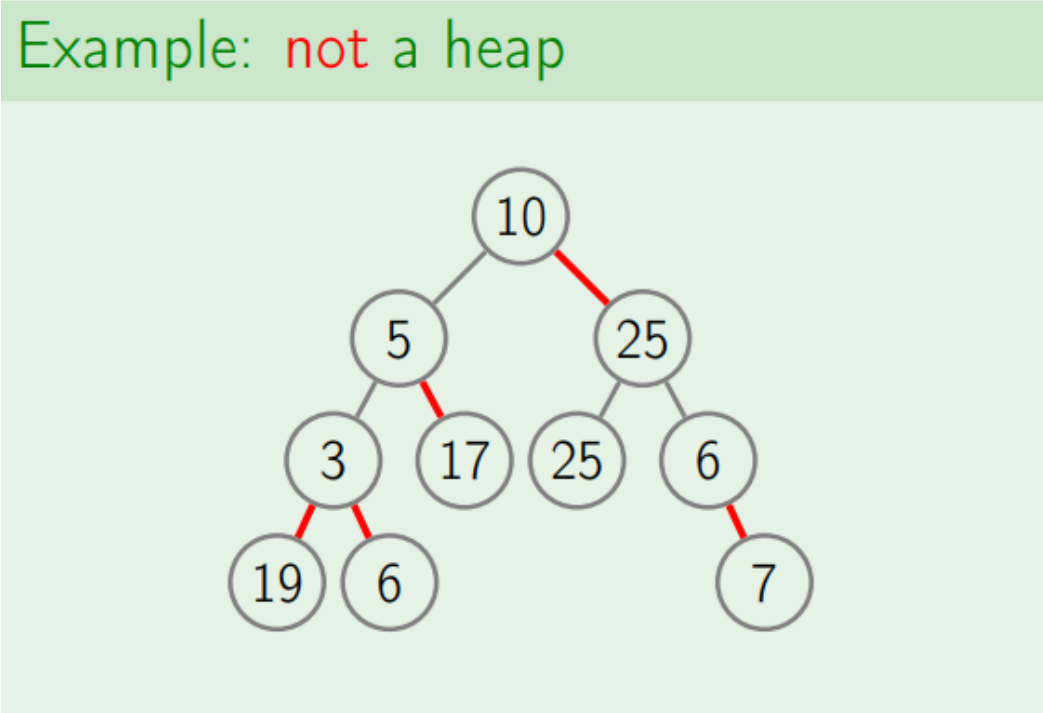
- Min-heap: version: a parent's key is always ≤ both its children's keys

Definition

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

Example: heap





Insert

- Add the element at **any leaf from the left** and shift up if necessary.
- In the array implementation, add the new element at the greatest empty index in the array.

Delete (while keeping heap balanced)

- Replace the key to delete with the right most leaf from the tree. Shift down appropriately.
- In the array implementation replace with the greatest filled index in the array.

Time Complexity:

- `getMax()` or `getMin()`: $O(1)$
- (Unbalances) All other operations: $O(h)$
- (Balanced/Complete) All other operations: $O(logn)$

Complete Tree

- A tree is complete if all its levels are filled except possibly the last, which is filled from left to right.
- Complete trees help keep the total height of the tree low.

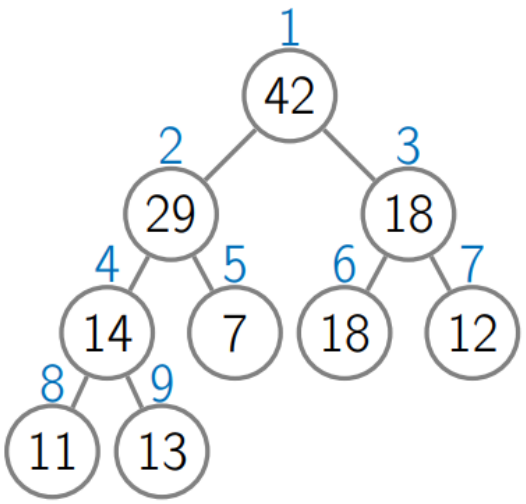
- Complete trees can also be represented as an array to improve time complexity through spatial locality

Why completeness ensures h = log n?

- A heap T storing n entries has height $h = \log n$.
- From the fact that T is complete, we know that
  - the number of nodes in levels 0 through $h-1$ of T is precisely $1+2+4+\cdots+2^{h-1} = 2^h - 1$, and
  - the number of nodes in level h is at least 1 and at most 2^h.
- Therefore:
  $n \geq 2^h - 1 + 1 = 2^h$ and $n \leq 2^h - 1 + 2^h = 2^{h+1} - 1$

- By taking the logarithm of both sides of inequality $2^h \leq n$, we see that height $h \leq \log n$.
- By rearranging terms and taking the logarithm of both sides of inequality $n \leq 2^{h+1} - 1$, we see that $\log(n+1) - 1 \leq h$. Since h is an integer, these two inequalities imply that $h = \text{floor}(\log n)$.

Storing Binary Heap as an Array



$$\text{parent}(i) = \left\lfloor \frac{i}{2} \right\rfloor$$

$$\text{leftchild}(i) = 2i$$

$$\text{rightchild}(i) = 2i + 1$$

- i is the index.
- Check Slides for explanation

Advantages

- Fast: At max O(log n)

- Space Efficient: We dont need to store the tree; we can just use an array to store the tree as using the formulas discussed earlier.

Sorting (contents)

Bubble Sort

Insertion Sort

Selection Sort

Merge Sort

Quick Sort

Radix Sort

Lossless Compression

- Compression where no information is lost after compressing.

Fixed Length Encoding

- Each character is represented using the same number of bits

Variable Length Encoding

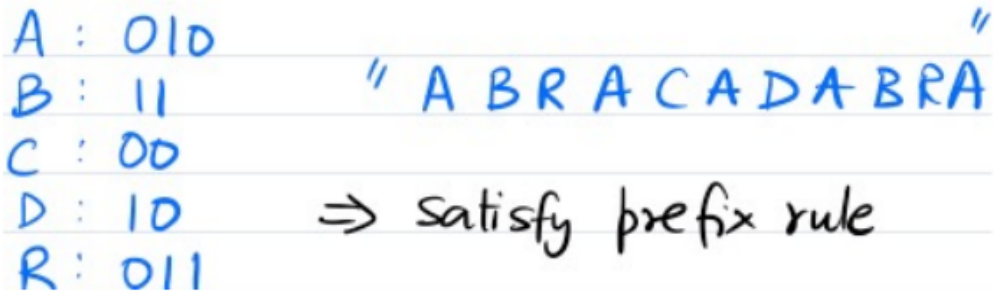
- Uses binary codes of different length
- This is a a lot better at saving space
- VLE helps by using fewer bits to represent characters that are used more often. Thus improving compression on average.
- With VLE we run into the ambiguity problem:

What can we do to prevent such ambiguities?

🚨 c is the prefix of a

How to solve ambiguities?

- To prevent ambiguities in VLE, each encoding must satisfy the prefix rule: “no code is a prefix of another code”.
- Even with these restrictions, VLE provides significant savings.



VLE: 29 bits
ASCII: 11 x 8 = 88 bits
Unicode: 11 x 16 = 178 bits

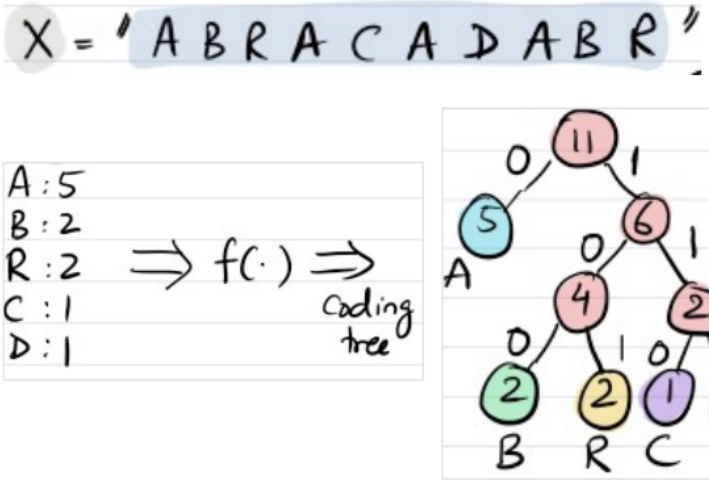
Huffman Encoding

- A way to calculate prefix safe variable length encoding

Huffman Coding | Greedy Algo-3 - GeeksforGeeks

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding

https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/



- Code for a char c: trace the path from the root to the leaf
- Char c is stored in the external node
- Each internal node stores the frequency of all external nodes in the subtree rooted at v

Constructing the Huffman Coding Tree

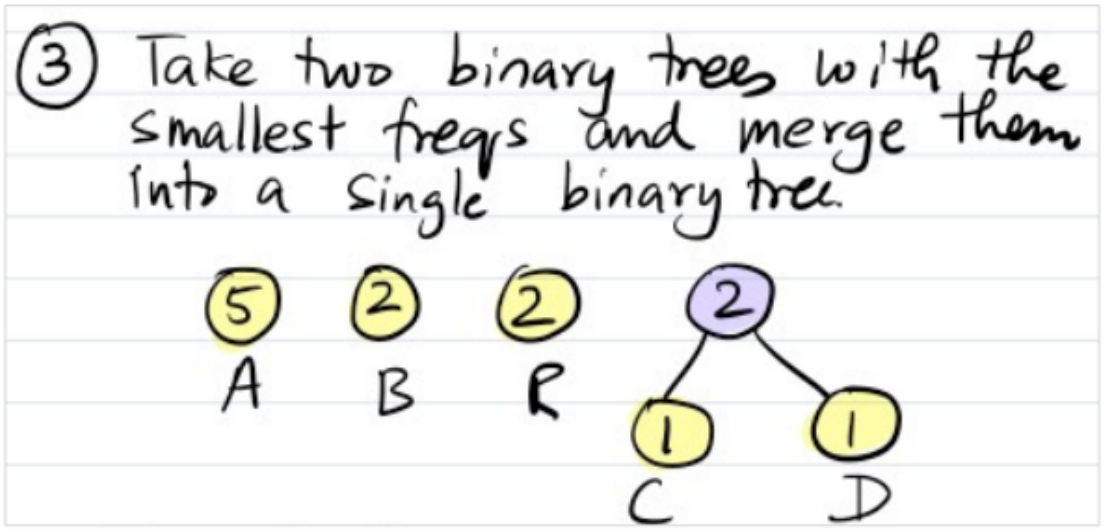
- X: ABRACADABRA

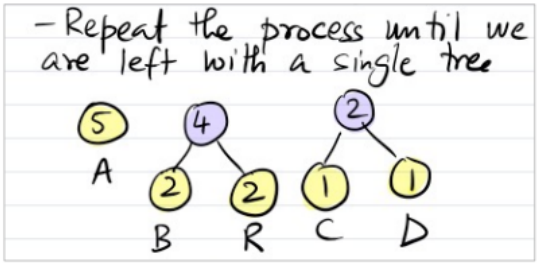
① Find the frequency of all chars in X

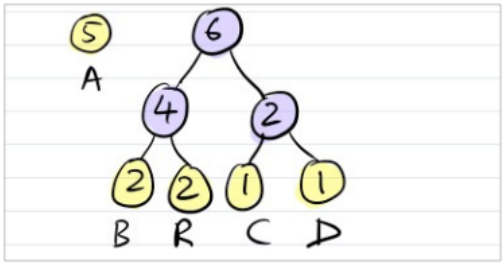
A	B	R	C	D
5	2	2	1	1

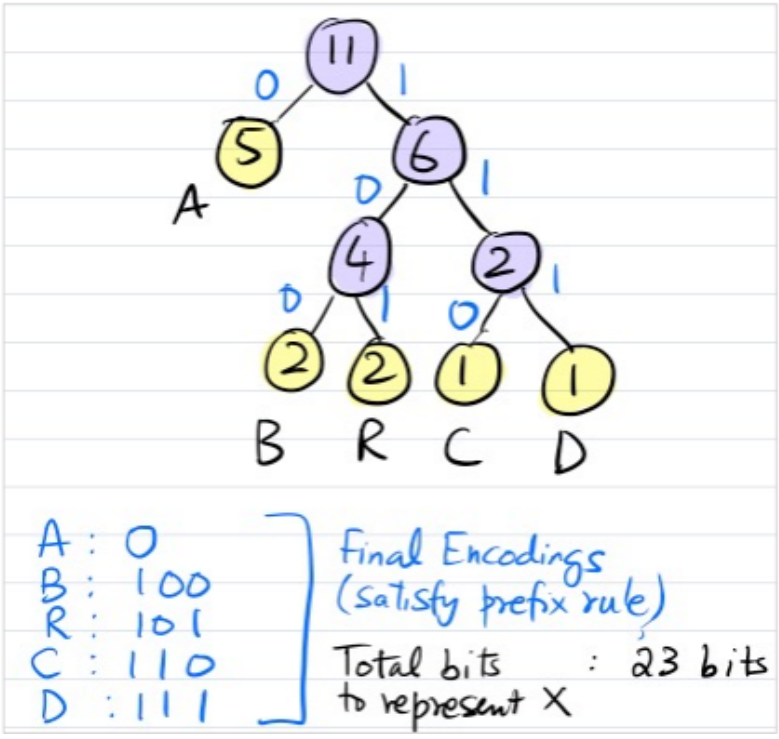
② Create d single-node binary trees where d is the # of distinct chars of string X

(5)	(2)	(2)	(1)	(1)
A	B	R	C	D

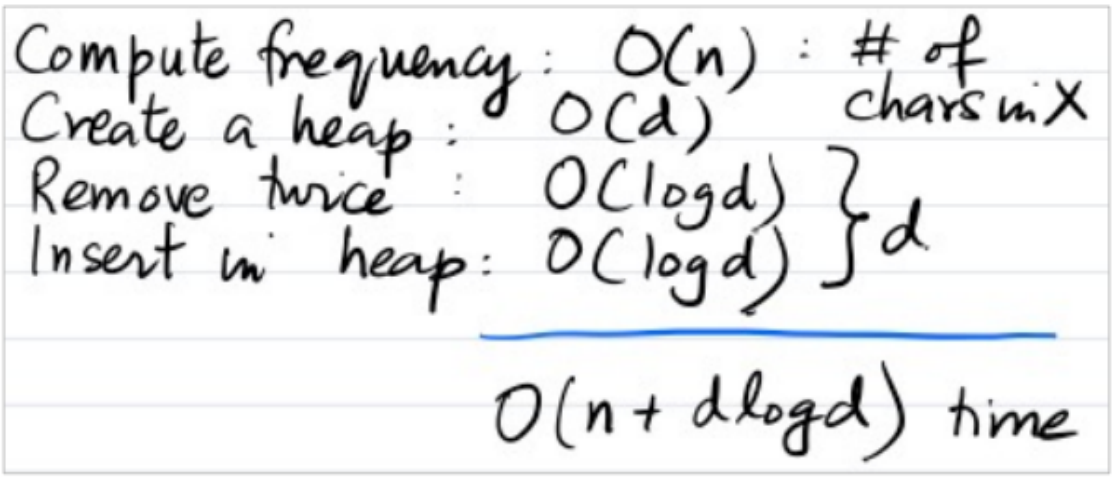






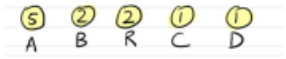


Time Complexity



Algorithm

Algorithm Huffman(X):
Input: String X of length n with d distinct characters
Output: Coding tree for X
Compute the frequency $f(c)$ of each character c of X.
Initialize a priority queue Q.
for each character c in X **do**
Create a single-node binary tree T storing c.
Insert T into Q with key $f(c)$.
while len(Q) > 1 **do**
$(f_1, T_1) = Q$.remove_min()
$(f_2, T_2) = Q$.remove_min()
Create a new binary tree T with left subtree T_1 and right subtree T_2.
Insert T into Q with key $f_1 + f_2$.
$(f, T) = Q$.remove_min()
return tree T



Encoding

- Encoding is just done by traversing the tree and building the bit sequence.

Decoding

- It is essential that we use the same tree to do both encoding and decoding of files.
- Huffman coding is not necessarily unique
  - However, each Huffman tree creates a unique encoding of a file

- Need to ensure that the decoding algorithm generates the exact same tree, so that we can get back the file

- Some options
  - Send the encoding table along with the compressed file
  - Send characters and their frequencies (but you need to ensure that you will always get the exact same tree given the character frequencies!)

> 🚨 Note: There won't be any compression if all characters occur with the same frequency and all characters in the character set are used.

Some Remarks

- For every possible input file, a unique code is generated just for that file and is sent along with the compressed file.
- Huffman coding requires 2 passes on the data/text (building a table & encoding the file)
- Storage for the 'coding table' is needed

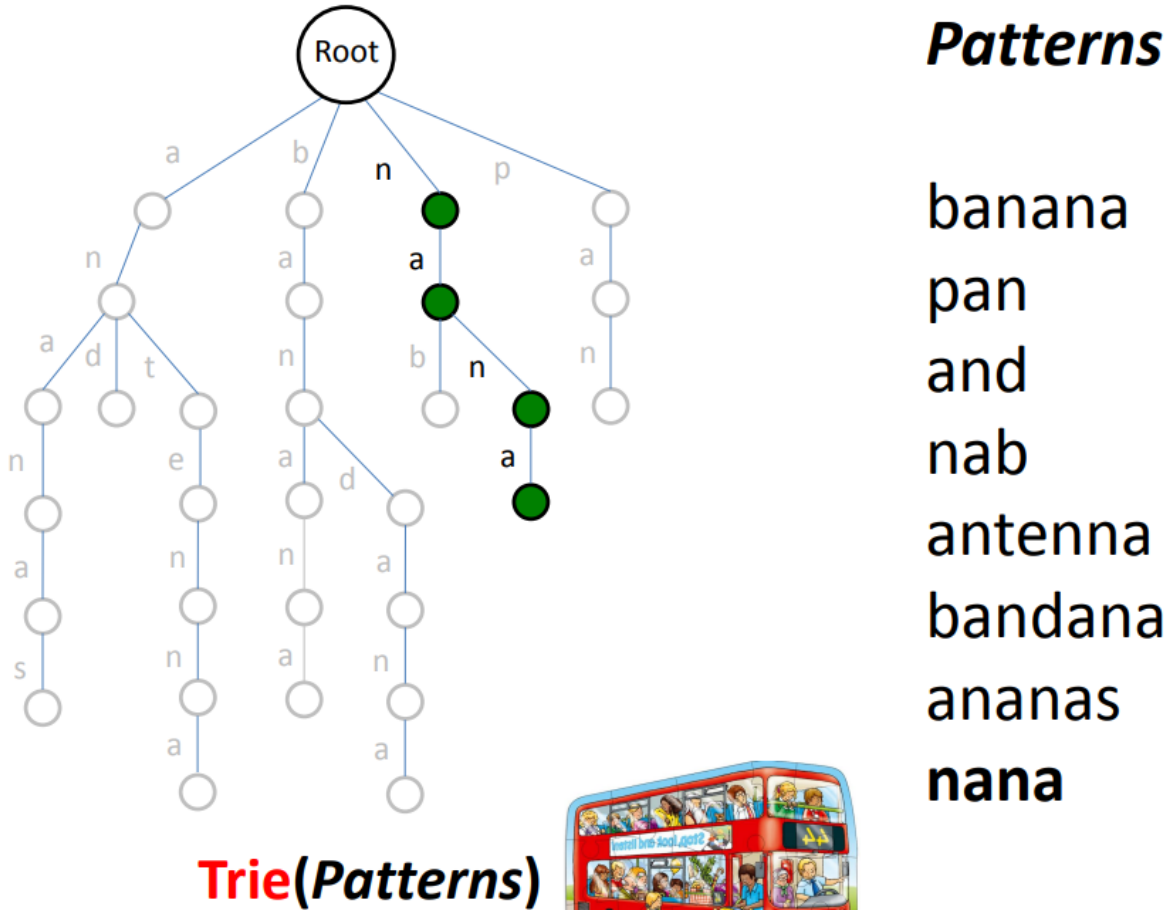
Pattern Matching [REDO]

Brute Force Algorithms:

- Iterate over the data and for each pattern to search, iterate the length of the element and repeat if pattern doesn't match.
- Time Complexity:
  - Single Pattern: $O(\text{len of data} \times \text{len of pattern})$
  - Multiple Patterns: $O(\text{len of data} \times \text{total len of all patterns})$

Trie

- Tries are used to search for patterns in a large amount of data efficiently.
- A trie is a tree where each node stores a char/letter except the root node.
- The values stored in the trie are patterns we need to search for.



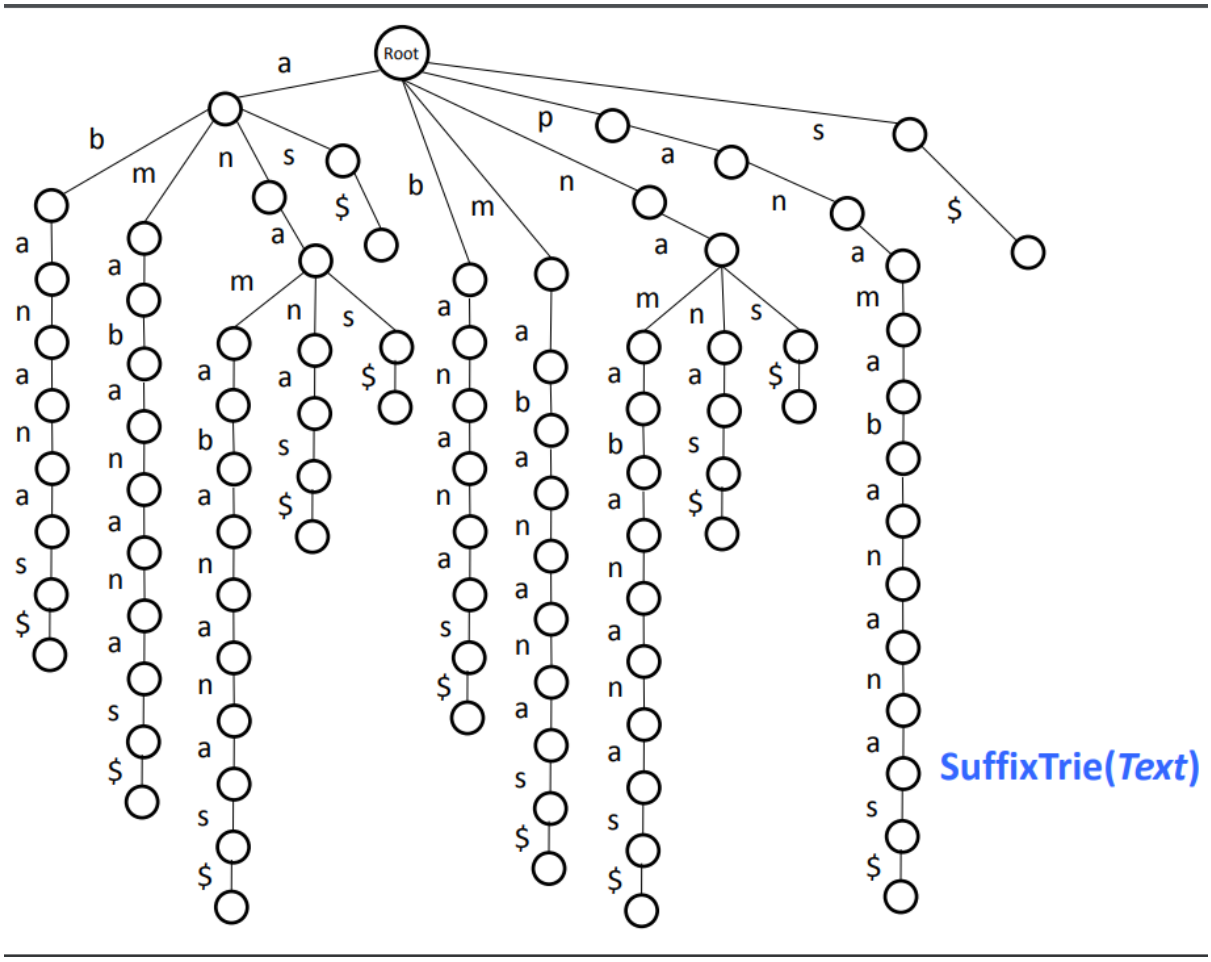
- For each character walk down the Trie. If we reach the leaf then a pattern match is found and we move on to the next char.
- If we the pattern changes before reaching a leaf, we move onto the next character.

Complexities

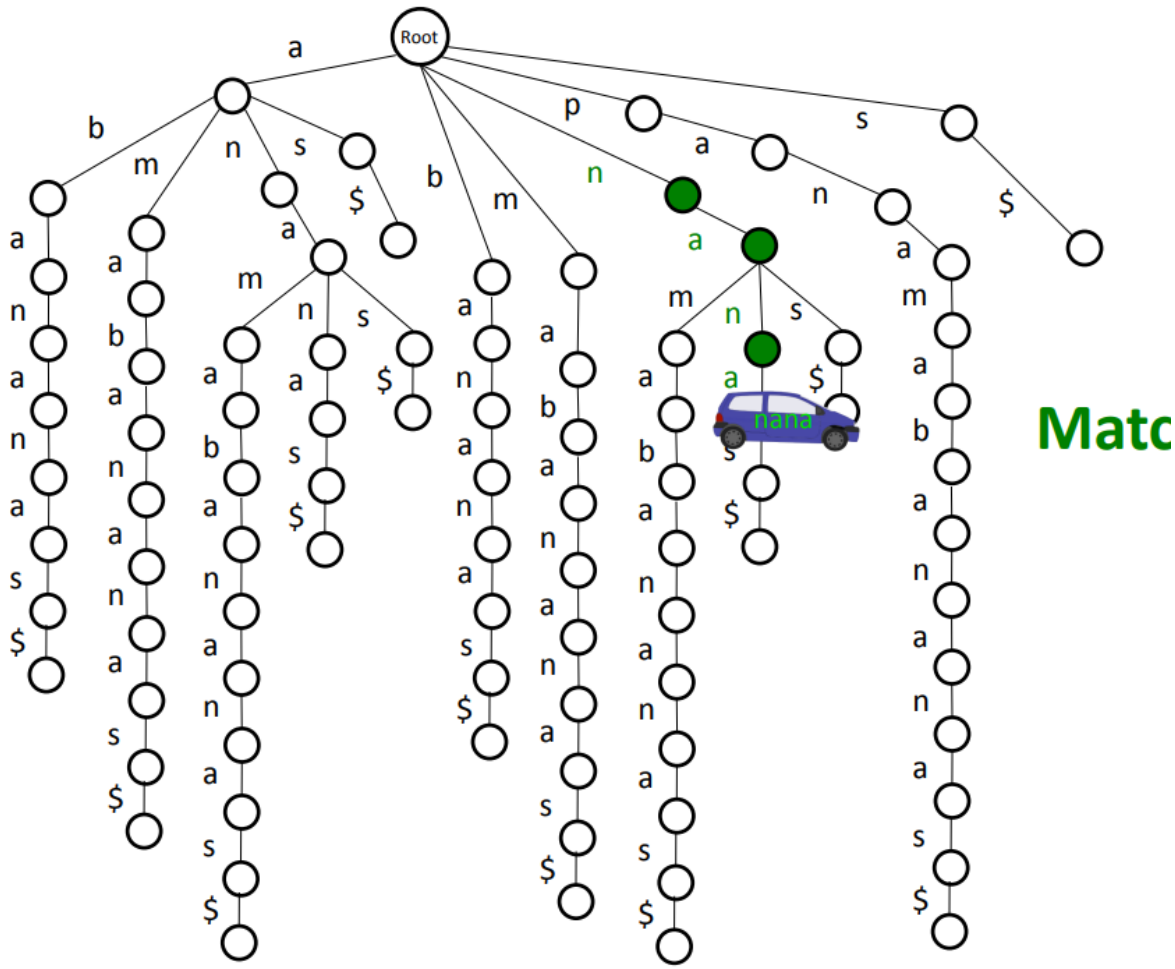
- Time Complexity: $O(\text{len of data} \times \text{length of longest pattern})$
- Memory Complexity: $O(\text{len of patterns})$

Suffix Trie

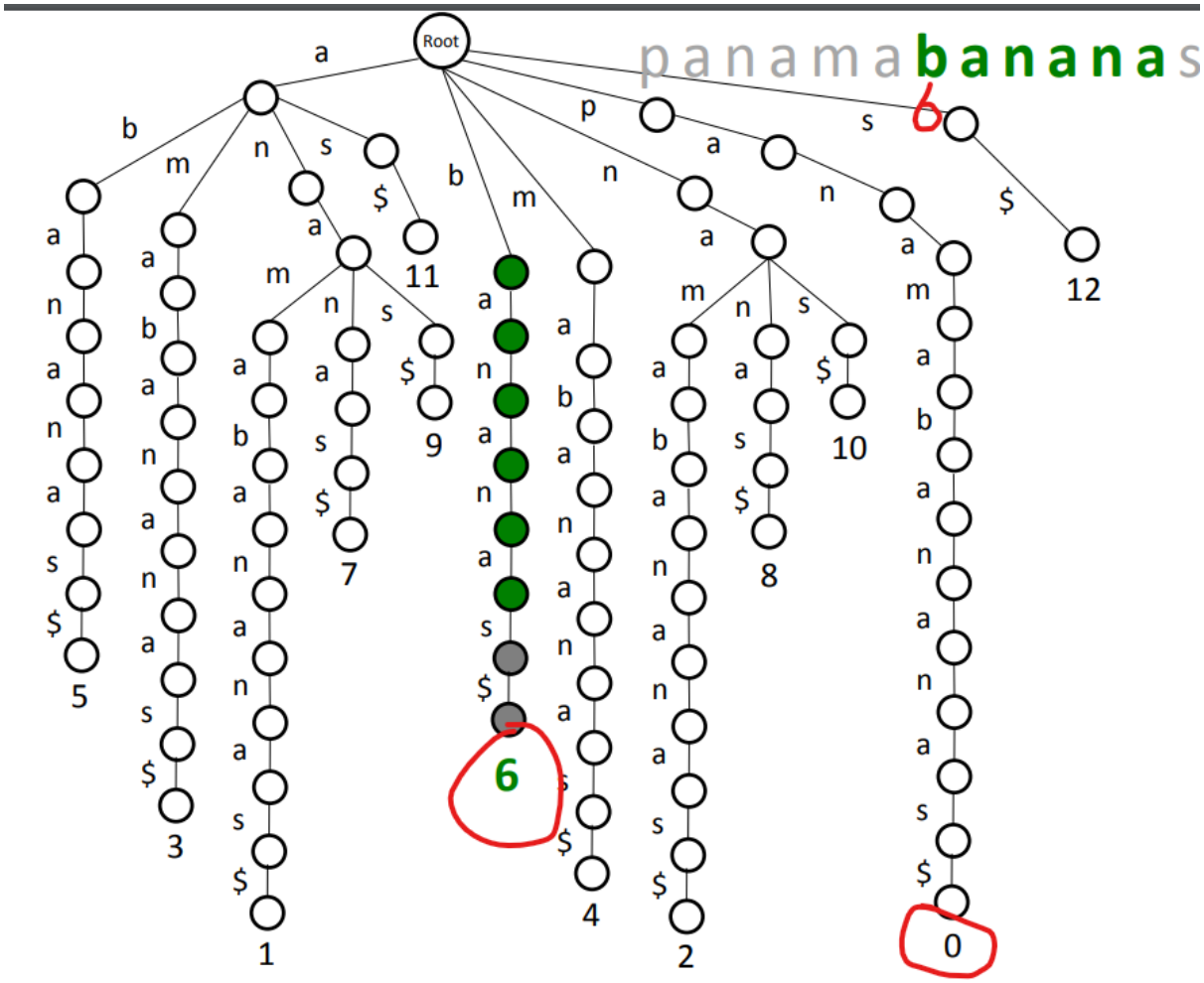
- Instead of the patterns we use all possible suffixes from a dataset to generate a trie.
- We put a $ at the end of each suffix.
- Suffix Trie of panamabananas:



- For each pattern we drive down the trie and see if the whole pattern can be spelled before reaching the end.



- We can tell where the match occurred by looking adding extra info to the suffix trie.

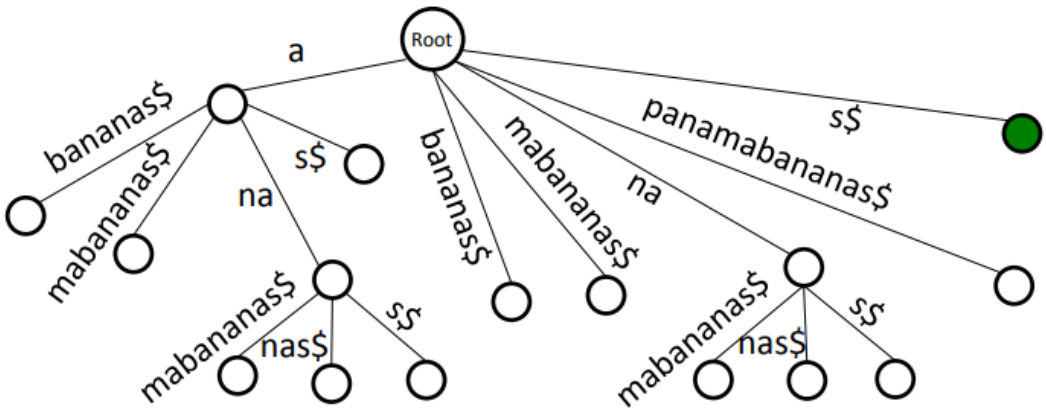


Complexities

- Time Complexity: O(length of text * len of longest pattern)
- Memory Complexity: $O(n^2)$

Compressed Suffix Trie

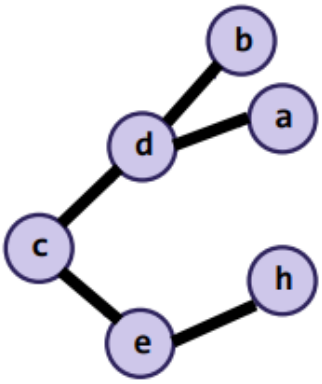
- A trie with its leaves concatenated.



SuffixTree(*Text*)

Graph

- A graph consists of two sets, V and E → G(V, E)
- V: set of vertices
- E: set of edges



V={a,b,c,d,e,h}
E={(c,d),(d,b),(d,a),
(c,e),(e,h)}

Directed and Undirected Graph

- Directed graphs (or digraphs):
  - Every edge e is directed from some vertex v to another vertex w (you cannot go from w to v via this edge)
  - e=(v,w) is an ordered pair (v is the source/origin, w is the destination)
- Undirected graphs:

- Edges do not have directions and every edge e=(v,w) is an unordered pair (you can travel in both directions)
- e=(v,w)=(w,v)

Path

Path

- A path from vertex a to b is a sequence of vertices that can be followed starting from a to reach b
- A simple path repeats no vertices, except the first might also be the last
- Example, one path from P to K: {P, Q, I, L, K}

Path Length

- Path length is the number edges in the path.
- Distance is the length of the shortest path

Neighbor

- Two vertices connected directly by an edge.

Reachability, Connectedness

Reachable

- Vertex Y is reachable from B if a path exists from Y to V.

Connected

- A graph is connected if every vertex is reachable from every other vertex. Connectedness is used to describe an **undirected graph**.

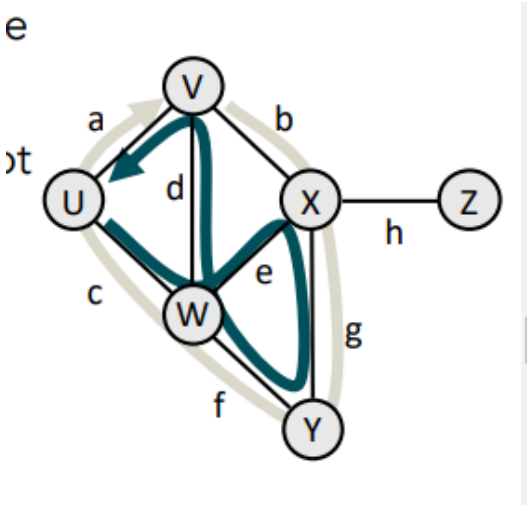
Strongly Connected

- A cyclic directed graph where all vertices are reachable to each other.

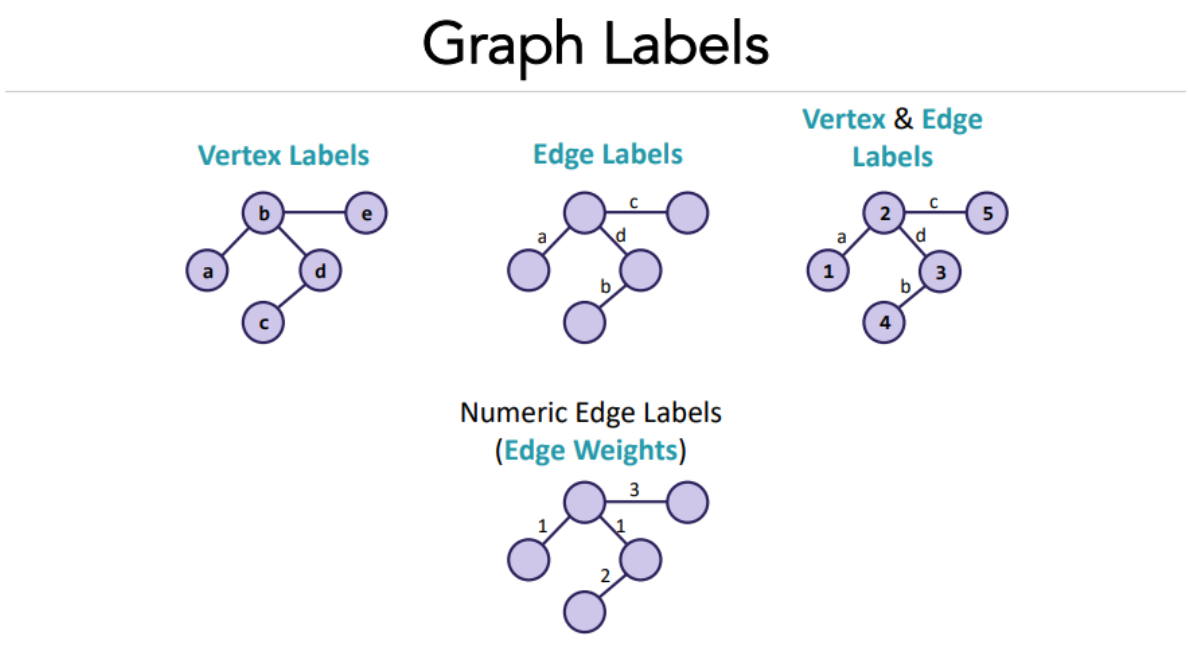
Complete

- If every vertex has a direct edge to every other vertex.

Cycles



- A **cycle** is a path whose first and last vertices are the same
  - Ex: {V, X, Y, W, U, V} and {U, W, V, U}
- A **simple cycle** is a cycle with no repeated vertices (except the starting vertex)
  - Ex: {V, X, W, V, X, W, V} is NOT a simple cycle
- Note: For undirected graphs, we require that edges in a cycle be distinct
  - Why? The path {V, X, V} should NOT be considered a cycle because (V, X) and (X, V) are the same edge whereas in a digraph, they are different so it makes sense to call them a cycle in case of latter
- **Acyclic Graph:** A graph without any cycles.



Vertex Degree

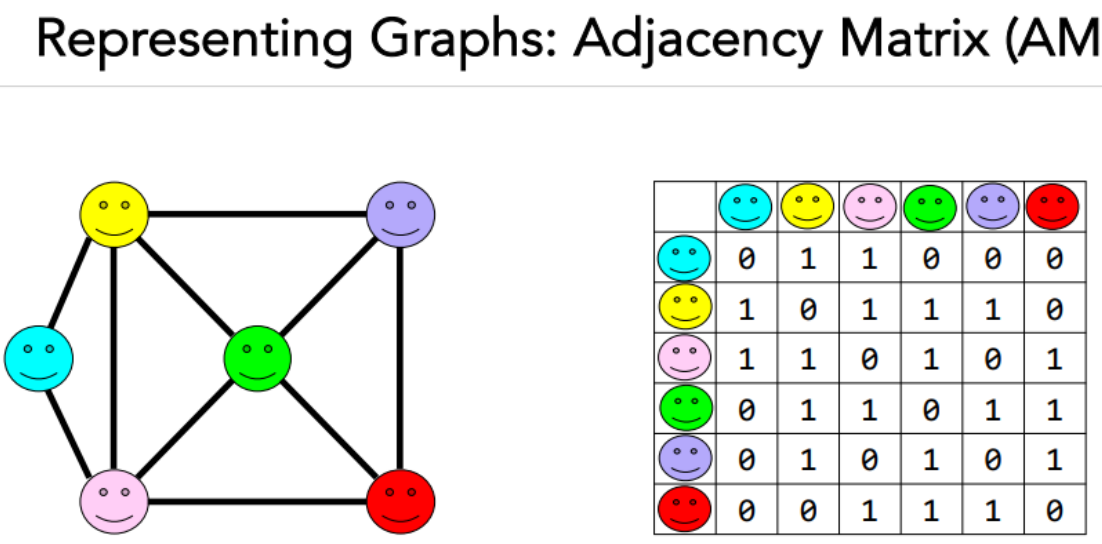
- Degree is the number of edges incident on a vertex.
- For Diagraphs:
  - In-degree: number of edges directed towards a vertex
  - out-degree: number of edges directed away from a vertex

Weighted Graph

- Weight is the cost associated with a given edge

Adjacency Matrix - Graph as Array

```
Given a graph G=(V,E), its Adjacency Matrix
(M) is a |V| x |V| matrix (2D array) where,
    M[i][j] = 1 if (i,j) is an edge in G
    M[i][j] = 0 if (i,j) otherwise
```



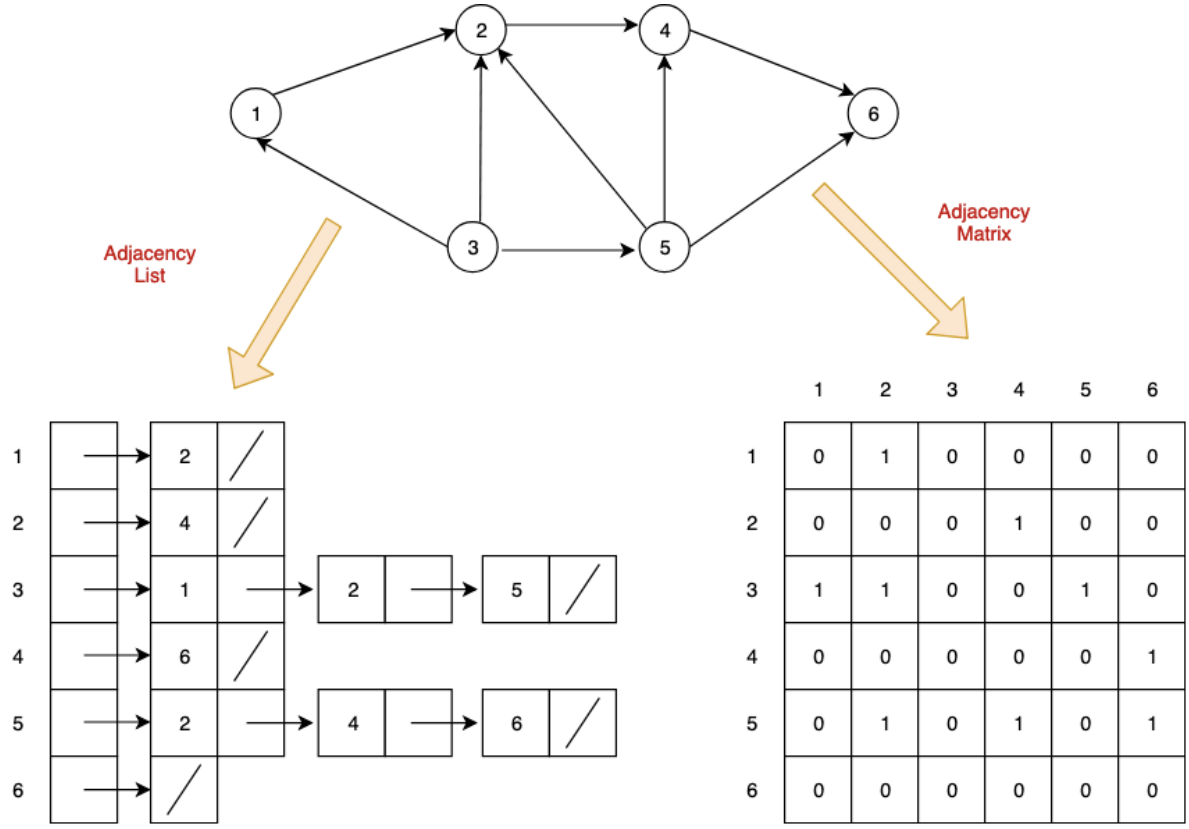
Properties

- AMs are symmetric (along the diagonal) for undirected graphs
- Supports fast lookups

- Good for **dense** graphs.
- Memory Complexity: $O(|V|^2)$

Adjacency List (like a dict) - Graph as a Linked List

- Store a linked list for each vertex containing the neighboring vertices.
- Slighting slower than Adjacency Matrix
- Good for **sparsely** populated graphs.
- Memory Complexity: $O(|V| + |E|)$



Graph Traversal

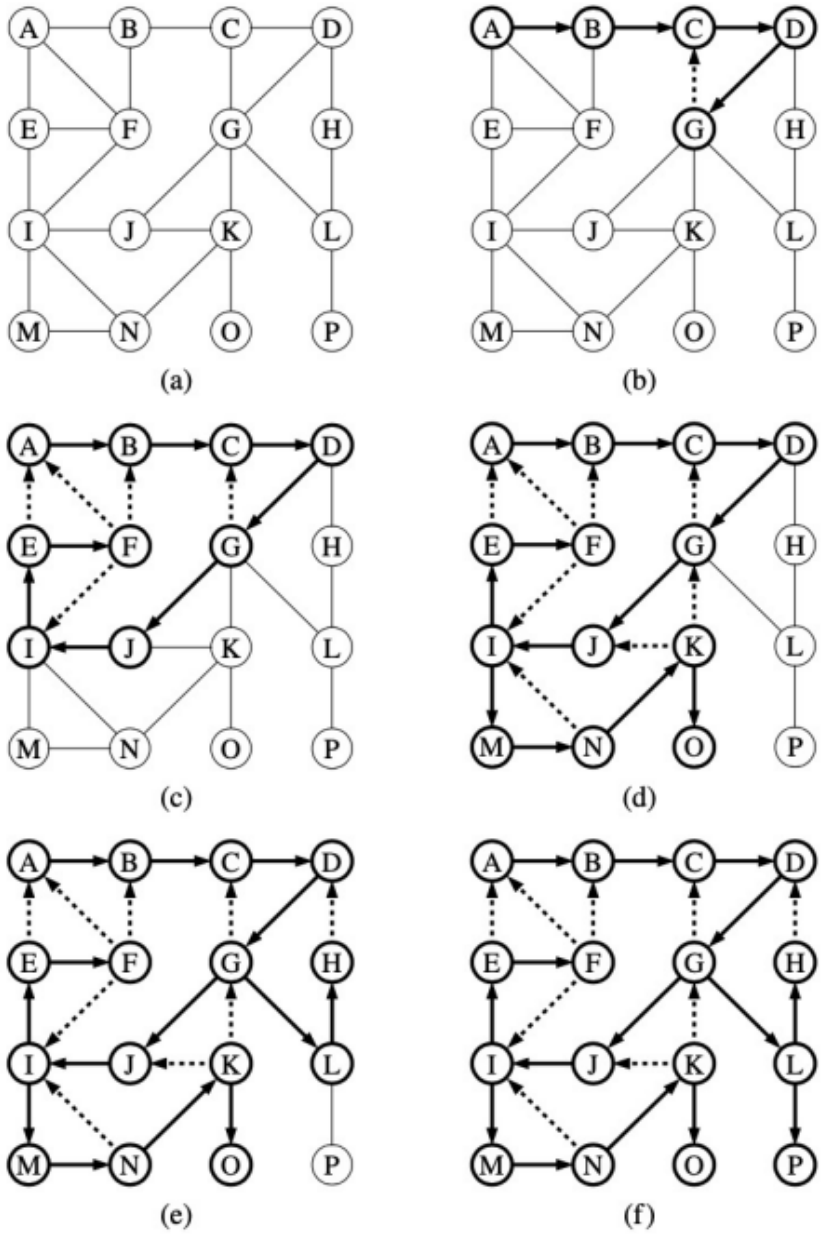
Depth First Search

Algorithm DFS(G,u): {We assume u has already been marked as visited}

Input: A graph G and a vertex u of G

Output: A collection of vertices reachable from u, with their discovery edges

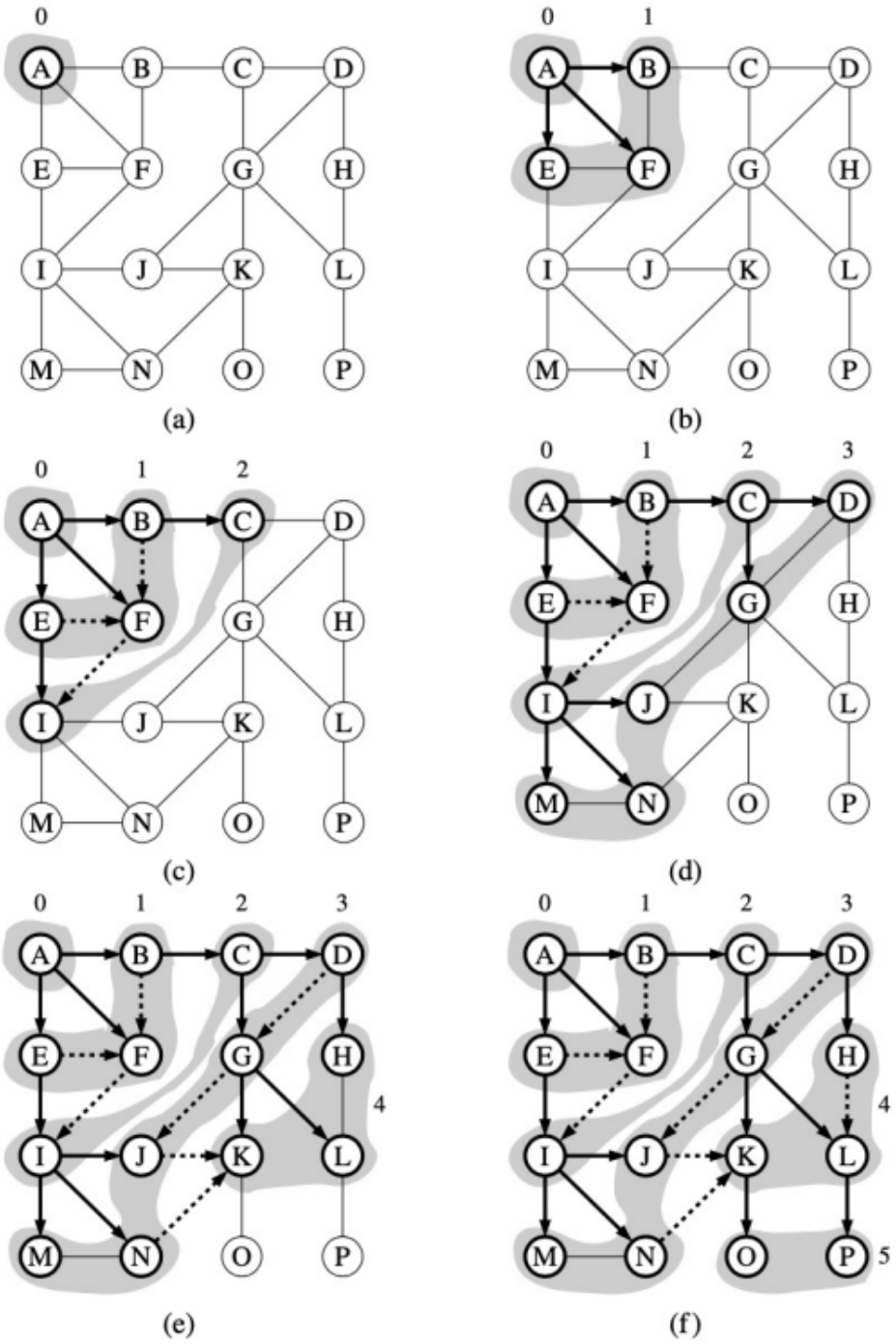
for each outgoing edge e = (u, v) of u **do**
  if vertex v has not been visited **then**
    Mark vertex v as visited (via edge e).
    Recursively call DFS(G,v).



Breadth First Search

```
def BFS(g, s, discovered):
  """Perform BFS of the undiscovered portion of Graph g starting at Vertex s.

  discovered is a dictionary mapping each vertex to the edge that was used to
  discover it during the BFS (s should be mapped to None prior to the call).
  Newly discovered vertices will be added to the dictionary as a result.
  """
  level = [s]                       # first level includes only s
  while len(level) > 0:
    next_level = [ ]                # prepare to gather newly found vertices
    for u in level:
      for e in g.incident_edges(u): # for every outgoing edge from u
        v = e.opposite(u)
        if v not in discovered:     # v is an unvisited vertex
          discovered[v] = e         # e is the tree edge that discovered v
          next_level.append(v)      # v will be further considered in next pass
    level = next_level              # relabel 'next' level to become current
```



Cycle Detection

- Assign 3 states to each vertex:
  - unvisited
  - inProgress
  - done
- initially all nodes are univisited

- a vertex is in inProgress state when its neighbors are being explored
- when a vertex is fully explored we mark it done.
- A cycle is detected when in DFS we encounter an **inProgress** node**.**

Spanning Tree

- A tree like structure in a graph which **connects all the vertices** **without any cycles**.
- Minimum spanning tree is a spanning tree with the **least total weight** out of all the possible spanning trees.
- Number of edges in a spanning tree: $|V| - 1$

Kruskal's Algorithm

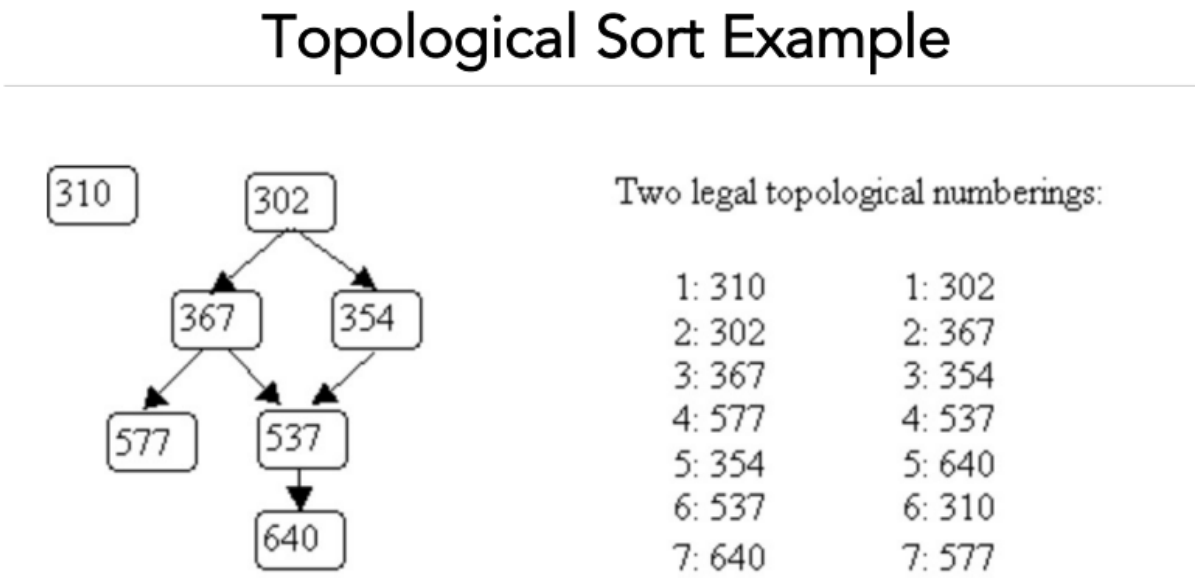
Kruskal's Algorithm

```
(1) Create a new graph T with same vertices as
  G but no edges yet
(2) Make a list of all edges in G
(3) Sort edges by weight (lowest to highest)
(4) Loop through edges in sorted order
      For each edge (u,w)
        if u & w are not connected by a path then
        we connect them i.e., add (u,w) to T
```

- AL representation
  - O(|V|) // Creating a graph T with |V| vertices but not edges
  - O(|E|) // Creating a list of edges
  - O(|E|log|E|) // Sorting edges
  - O(|E|) * O(|V|+|E|) // Checking if (u,v) are already connected in T
  - **Total: O(|E||V|+|E|²)**

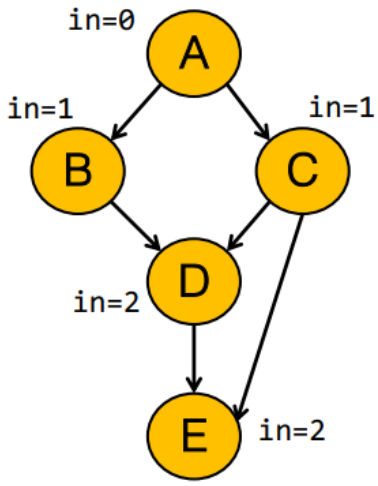
Topological Sort

- Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge a b, vertex a comes before b in the ordering.



Topological sort is only possible in Directed Acyclic Graph.
There needs to exist a vertex which are 0 as its out-degree i.e. sink node, in order to make Topological Sort possible.

In-degree Method



- Compute the in-degree of each node
- Choose a vertex with in=0 and put in the sorted sequence
- Remove in=0 node from G and recompute in-degree

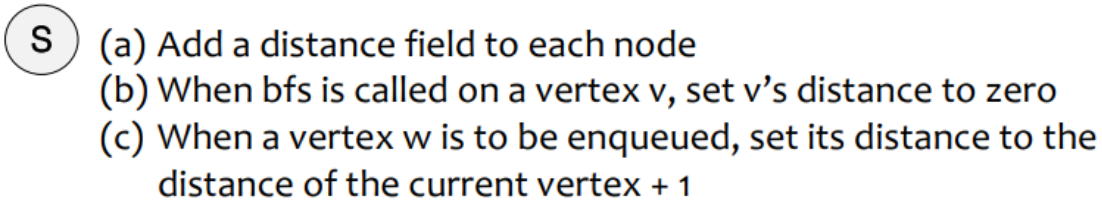
- Output: $S = \{A, B, C, D, E\}$

> 🚨 The remove and recalculate is important. Try to perform the algorithm without remove and recalculate with the edge (B, D) missing, to figure out why.

DFS method

> Topological Sort | CodePath Guides Cliffnotes
>
> Aside from DFS and BFS, the most common graph concept that interviews will test is topological sorting. Topological sorting produces a linear ordering of nodes in a directed graph such that the direction of edges is respected.
>
> https://guides.codepath.com/compsci/Topological-Sort/

BFS - Shortest Path Unweighted



> 🚨 Check Slides for examples

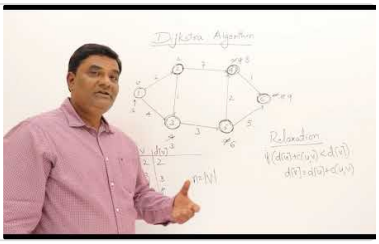
Dijkstar - Shortest Path Weighted

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

3.6 Dijkstra Algorithm - Single Source Shortest Path - Greedy Method

Dijkstra Algorithm for Single Source Shortest PathProcedureExamplesTime ComplexityDrawbacksPATREON : https://www.patreon.com/bePatron?u=20475192Courses on Ud...

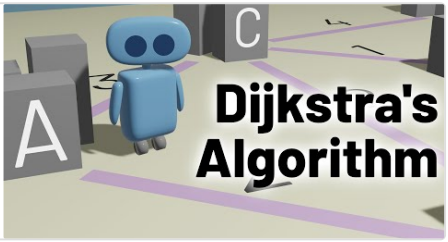
https://www.youtube.com/watch?v=XB4MIexjvY0&t=2s



How Dijkstra's Algorithm Works

Dijkstra's Algorithm allows us to find the shortest path between two vertices in a graph. Here, we explore the intuition behind the algorithm - what informat...

https://www.youtube.com/watch?v=EFg3u_E6eHU&t=425s



Complexity

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

Kruskall vs Dijstra

- Dijkstra finds the shortest path from a vertex to all other vertices.
- Kruskall finds a graph (which is also a tree) with all vertices connected such

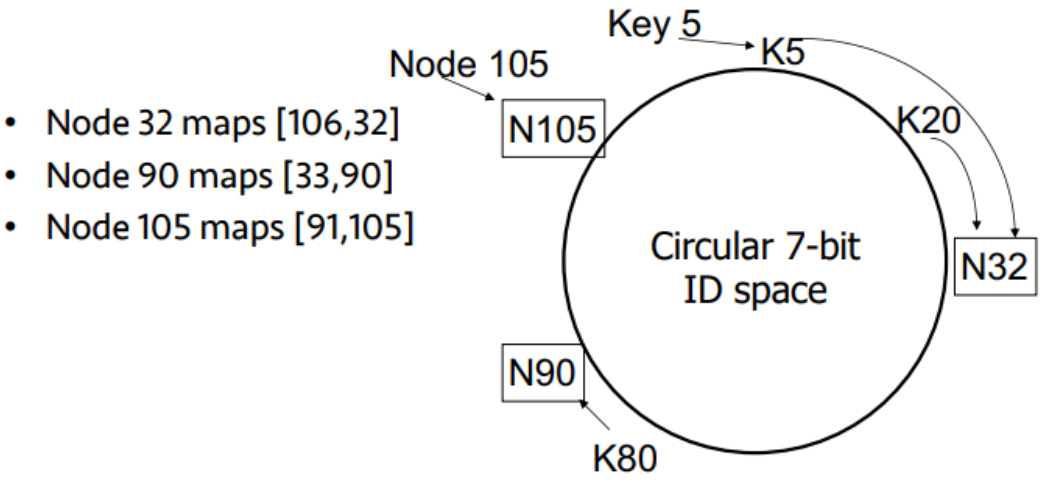
- Shortest Possible Path is used to send packet such that the time taken is minimum.

that the total edge weight is the least.

- Minimum Spanning Trees gives us the configuration to lay down network lines such that all nodes are connected and the total cost of the network lines is minimum.

Distributed Hash Table

- A distributed hash table data structure stores the (key, value) pairs distributed across multiple machines.
- It supports the following operations.
  - put(key, value): inserts the (key, value) pair
  - get(key): returns the value
  - delete(key): deletes the (key, value) pair
  - Expected time for operations: O(1)
- Many applications (e.g., Facebook) require storing of billions of (key, value) pairs. IF we use a single hashtable and a computer we wont be able to fit all the values and it will also be prone to complete data loss on failures.



A key is stored at its successor: node with next higher ID

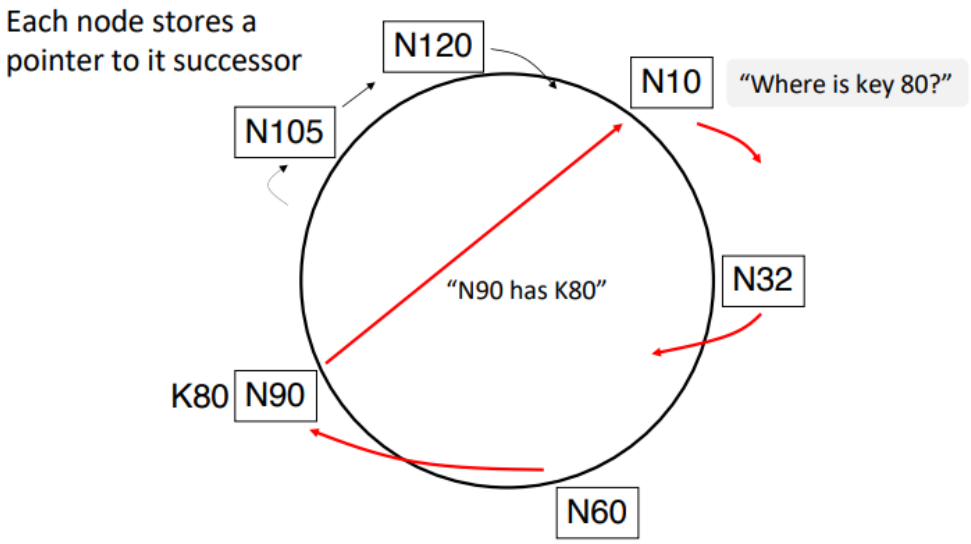
Consistent Hashing

- Usually hash function depend on the size of the array storing the hashes.
- In consistent a hash table is used which does not depend on the size of the array and in our case, the number of severs or nodes.
- To implement DHT an *arbitrary circle* is used. You could imagine it as a an array going from 0 to an constant (INT_MAX). Fixing the size of the array lets us implement consistent hashing.

Handling Failures

- Each node knows IP addresses of next r nodes
- In an event of a failure each item is replicated at next nodes

Lookup

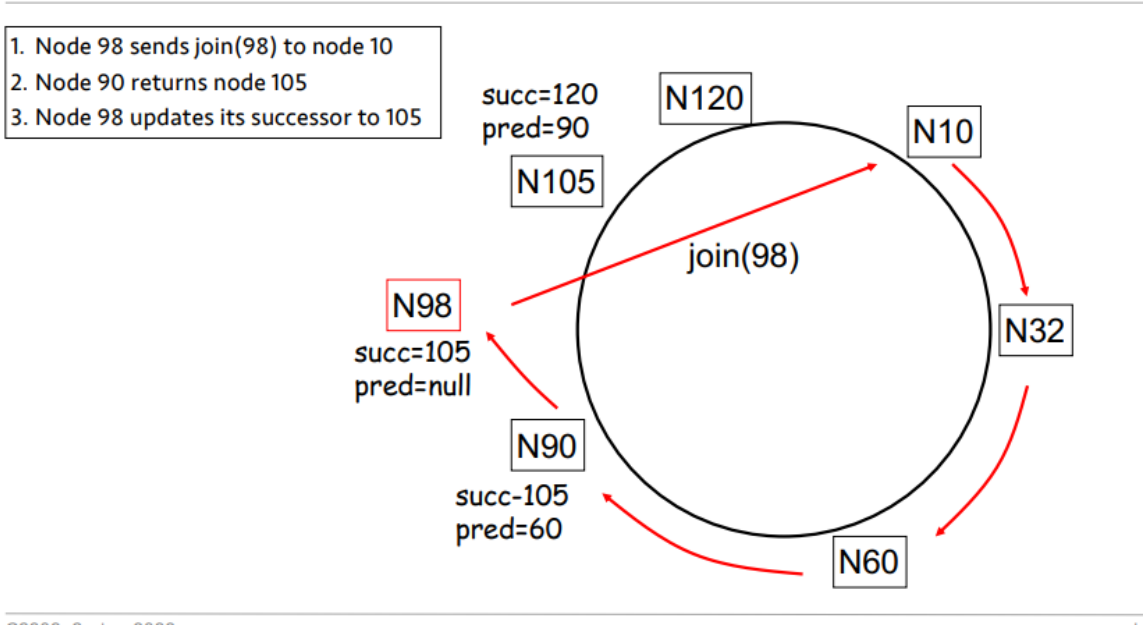


- Upon receiving a request, the node checks if the required key is in its domain. If not, it forwards the request to the next node which does the same. Until the key is found.

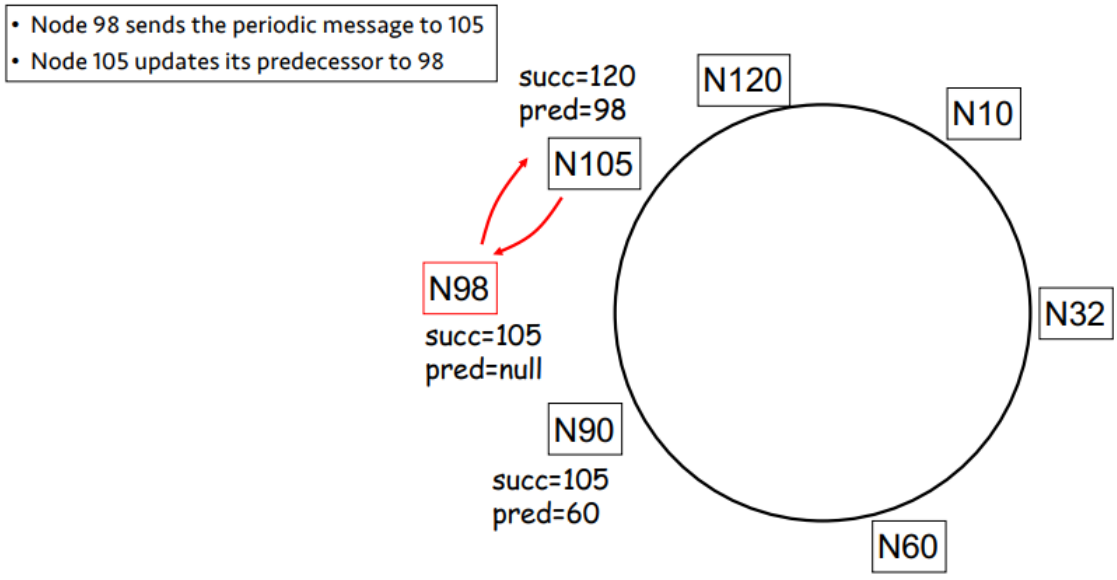
Node Joining

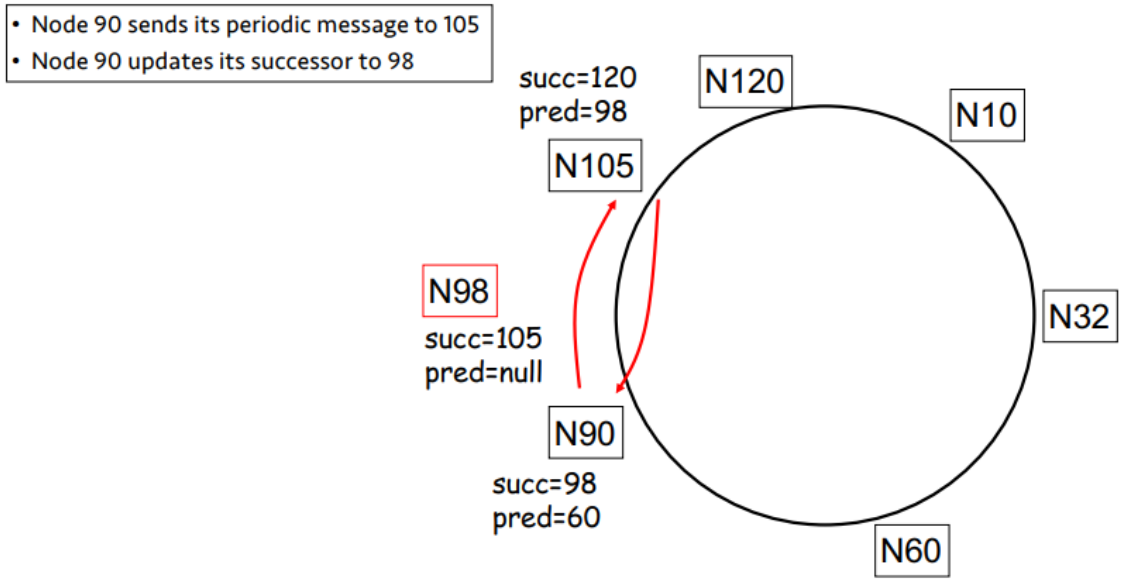
- Suppose C is the new node, who becomes the predecessor of node B
- Each node A periodically sends a message to its successor B asking "who is your predecessor (or successor)?"
- B returns its predecessor C to A
- When A receives the reply it checks if C is between A and B
  - if TRUE, A updates its successor to C

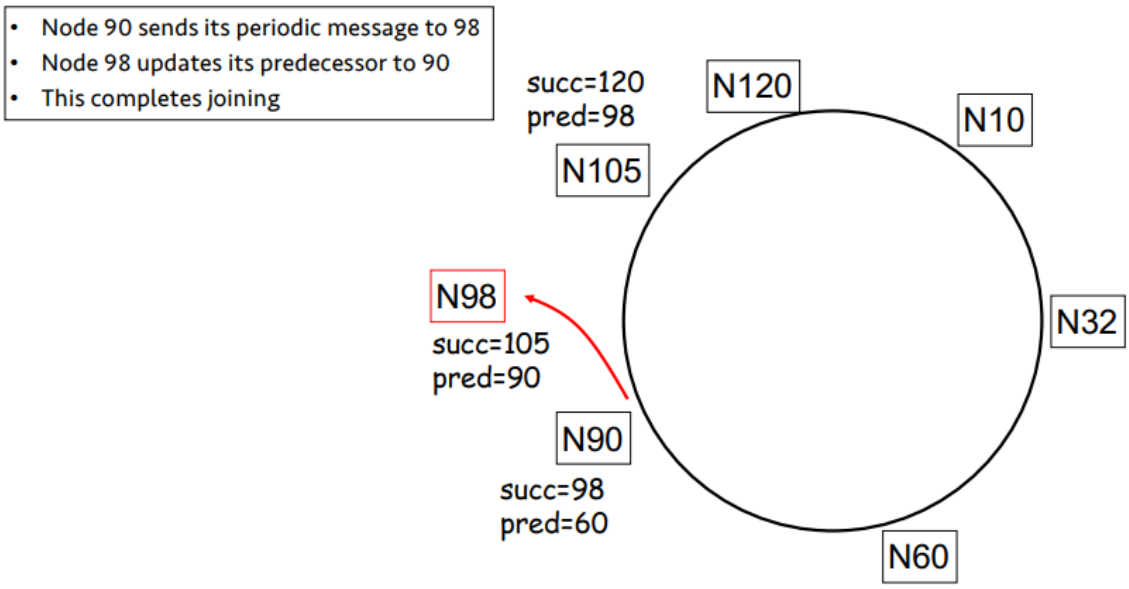
- if FALSE, A doesn't do anything



N10 actually tells N98 who its successor is.
N90 is checking if N98 is between N90 and N105 if so itll return the correct successor







Bloom Filter

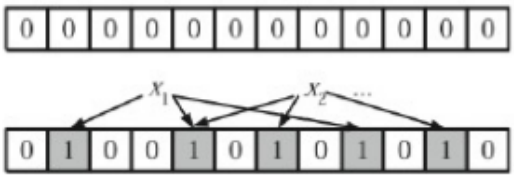
- A space-efficient **probabilistic** data structure used to test whether an element is a member of a set.
- Represent a set $S = \{x_1, x_2, ..., x_n\}$ using an array of m bits. I.e. the data will be stored in an array of size m where each element can have either a 1 or a 0.
- Uses k independent hash functions: $[h_1, h_2, ..., h_k]$, where each hash function has an output range of 1 - m.
  – hash functions have output range {1,..,m}

Insertion

```
for every x in S:
  for every hash function h_i(i=1 to k):
    set h_i(x)th bit of m to 1
```

$h_1(x_1)=2$
$h_2(x_1)=5$
$h_3(x_1)=9$

$h_1(x_2)=5$
$h_2(x_2)=7$
$h_3(x_2)=11$

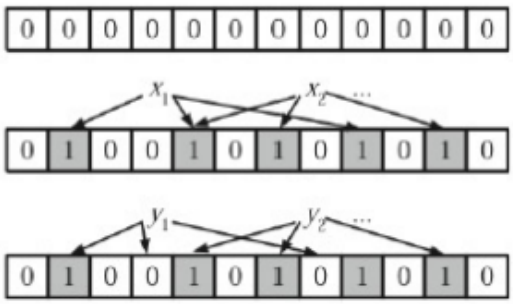


Checking Membership

- To check membership of y, calculate hashes using all the hash function from h=(h_1 to h_k)
- if any bit is 0 → y is not in the set
- if all bits are 1 → y is in set

$h_1(y_1)=2$
$h_2(y_1)=4$
$h_3(y_1)=8$

$h_1(y_2)=5$
$h_2(y_2)=7$
$h_3(y_2)=11$



🚨 In the example above, y_1 is not in the set

False Positives/Negatives

- False Positive (FP)
  - When the new item to be searched hashes to slots which are already 1 due to the hashes of other items

- False Negative (FN):
  - “You claim an element is NOT in S when it is”
  - False negatives are not possible as removal of elements are not allowed.

How likely are false positives?

- The probability that a hash function of an item maps to a specific slot is 1/m
- What is the probability that a specific hash misses a specific slot? 1-1/m
- What is the probability that k hashes miss a specific slot? $(1-1/m)^k$
- What is the probability that all hashes of n items miss a specific slot? $p = (1-1/m)^{nk} \approx e^{-kn/m}$

p = the probability that for all the items in the set, all the hash functions will fail to set a specific bit to 1.
1-p = the probability that for at least one item, one hash function sets a specific bit to 1. (thus causing a false positive)
(1-p)^k = as we need all the corresponding bits from the k hash function to be 1 to declare a false positive, we multiply the probability k times.

- What does (1-p) represent?
  - (1-p) = prob that the corresponding bit of a specific hash of y = 1
- What does $(1-p)^k$ represent?
  - $(1–p)^k$ = prob all corresponding bits of all hashes of y = 1
- What is the probability of false positives?
  - $(1–p)^k$ = i.e. $(1-(1-1/m)^{nk})^k \approx (1 – e^{-kn/m})^k$

Caching with Bloom Filter

- The servers of Akamai Technologies, a content delivery provider, use Bloom filters to prevent "one-hit wonders" from being stored in its disk caches.

- One-hit-wonders are web objects requested by users just once
- Using a Bloom filter to detect the second request for a web object and caching that object only on its second request prevents one-hit wonders from entering the disk cache, significantly reducing disk workload and increasing disk cache hit rates.

Parallel Algorithms

- Parallel computing is the ability to run multiple computations (tasks) at the same time
- Benefits of parallel computing:
  - Solve problems in shorter time
  - Energy efficiency
- Parallel computing requires a different way of organizing algorithms than sequential computing:
  - Two computations can be performed in parallel only if they do not depend on each other
  - Consequence: Identify dependencies between different computations when designing parallel algorithms

Parallelism vs Concurrency

- If parallel computations need access to shared resources, then concurrency needs to be managed i,e, parallelism and concurrency are different.

Analyzing Parallel Algorithms

- Involves two measures: Work and Span
- Work: total number of primitive operations performed by an algorithm
  - If running on a sequential machine, it corresponds to the sequential time

    - Ideal: Divide work evenly across processors (e.g., If we had W work and P processors, then even division would give each process W/P work)
    - Thus, total time taken would be W/P (perfect speedup)
    - However, perfect speedups are not always possible, e.g., in a fully sequential algorithms
- If a task depends on another task, we have to complete them in order. Span enables analyzing the extent to which an algorithm can be divided among processors
    - Span of an algorithm corresponds to the longest sequence of dependences in the computation
    - It can be thought of as the time an algorithm would take if we had an unlimited number of processors on an ideal machine

- In the diagram above C1 to C3 are dependent and the rest are independent. The span of the algorithm is the time taken to execute C1 to C3.

Why Work and Span

- They give us a good sense of efficiency (Work) and scalability (Span)
- Predict the time of an algorithm on any number of processors by assuming a “reasonable” runtime scheduler
- Predict the largest number of processors for which we can get close to perfect speedup
- Suppose there is work W and span S, then running time is at most:
    - T = (W-S)/P + S on P processors:
    - If the first term dominates —> getting close to perfect speedup
- In practice we have P processors. How well can we do?
    - We cannot do better than O(S)
    - We cannot do better than O(W/P)

Calculating Work and Span

- Suppose there are two subcomputations A and B that we would like to execute
  - W_1 and S_1 are the work and span of A
  - W_2 and S_2 are the work and span of B
  - 1 is the cost of combining the computation results

	work (W)	**span** (S)
Sequential composition	$1 + W_1 + W_2$	$1 + S_1 + S_2$
Parallel composition	$1 + W_1 + W_2$	$1 + \max(S_1, S_2)$