Introduction

03 November 2020 11:04 AM

Known Concepts list
- -Variables
- -Data types
  - -Primitive
  - -User defined data types

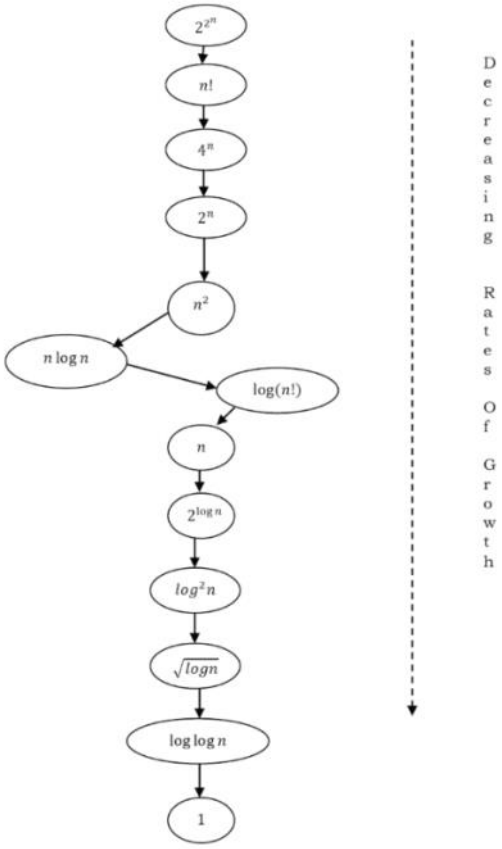
Data Structures
- -General definition
  - Data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently. Eg- array, linked list etc.
- -Types
  - -Linear : Elements are accessed in a sequential order but it is not compulsory to store all elements sequentially
    Eg-linked list, queue
  - -Non-linear : Elements of this data structure are stored/accessed in anon-linear order
    Eg-tree ,graphs

Abstract Data Types
- -General Definition
  - The definition of user defined data types including declaration of data and its operations.

Algorithms
- -General Definition
  - Step by step unambiguous instruction to solve a Given problem

Running time analysis
- -process of determining how processing time increases as the size of the problem (input size) increases.
- -The following are the common types of input sizes.
  - • Size of an array
  - • Polynomial degree
  - • Number of elements in a matrix
  - • Number of bits in the binary representation of the input
  - • Vertices and edges in a graph.

Rate of growth
- -The rate at which the running time increases as a function of input is called rate of growth.
- -comparison of deferent rates of growth:



Types of analysis

-best case

-worst case

-average case --as name suggests

Asymptotic Notation

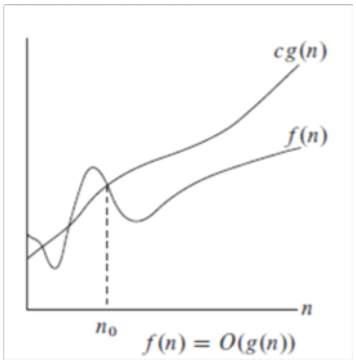
-Big-O:

-Formal definition:

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}.$$

-Non mathematical meaning: roughly it means the all the functions which are greater than equal to the function g(n) up to a multiplicative constant belongs to the set of theta of g(n).

This is also a very broad definition as it does not put a limit to tight bounds (how close should it be to g(n)) but we may use it to tightly bound f(n) . It is essentially the first part of theta n.



-Big-Omega:

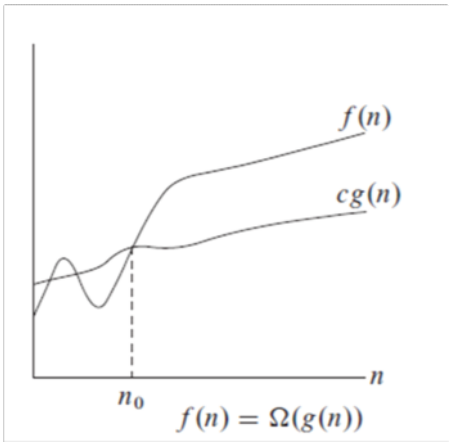
-Formal definition:

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\}.$$

-Non mathematical meaning: roughly it means the all the functions which are less than equal to the function g(n) up to a multiplicative constant belongs to the set of theta of g(n).

This is also a very broad definition as it does not put a limit to tight bounds (how close should it be to g(n)) but we

may use it to tightly bound f(n) . It is essentially the second part of theta n.

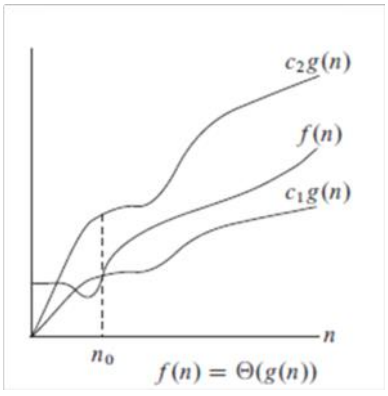


-Theta :

-Formal definition:

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}.^1$$

-Non mathematical meaning: roughly it means the all the functions which are equal to (having the same order of growth) the function g(n) up to a multiplicative constant belongs to the set of theta of g(n).
This is a very broad definition as it does not put a limit tight bounds (how close should it be to g(n)) but we may use it to tightly bound g(n).



-imp. Notes

Big-o is for worst case
Big-omega is for best case
Theta is for average case

Guidelines for asymptotic analysis:

-Loops: The running time of a loop is, at most, the running time of the statements inside the loop (including tests) multiplied by the number of iterations.
-If-then-else statements: Worst-case running time: the test, plus either the then part or the else part (whichever is the larger).
-Logarithmic complexity: An algorithm is O(logn) if it takes a constant time to cut the problem size by a fraction (usually by ½). As an example let us consider the following program:

Master theorem for divide and conquer

-

$$T(n) = aT(\frac{n}{b}) + \Theta(n^k log^p n)$$

1) If a > b^k, then $T(n) = \Theta\left(n^{log_b^a}\right)$

2) If a= b^k

   a. If $p > -1$, then $T(n) = \Theta\left(n^{log_b^a} log^{p+1} n\right)$

   b. If $p = -1$, then $T(n) = \Theta\left(n^{log_b^a} loglogn\right)$

   c. If $p < -1$, then $T(n) = \Theta\left(n^{log_b^a}\right)$

3) If $a < b^k$

   a. If $p \geq 0$, then $T(n) = \Theta(n^k log^p n)$

   b. If $p < 0$, then $T(n) = O(n^k)$

Master Theorem for subtract and conquer

-Let T(n) be a function defined on positive n, and having the property

$$T(n) = \begin{cases} c, & \text{if } n \leq 1 \\ aT(n-b) + f(n), & \text{if } n > 1 \end{cases}$$

-for some constants c, a > 0,b ≥ 0,k ≥ 0, and function f(n). If f(n) is in O(n k), then

$$T(n) = \begin{cases} O(n^k), & \text{if } a < 1 \\ O(n^{k+1}), & \text{if } a = 1 \\ O\left(n^k a^{\frac{n}{b}}\right), & \text{if } a > 1 \end{cases}$$

-Variant of Subtraction and Conquer Master Theorem
The solution to the equation T(n) = T(α n) + T((1 – α)n) + βn, where 0 < α < 1 and β > 0 are constants, is O(nlogn).

For any other type of recurrences refer to 1.26 -Narsimha

Amortized analysis

-Amortized analysis refers to determining the time-averaged running time for a sequence of operations. It is different from average case analysis, because amortized analysis does not make any assumption about the distribution of the data values, whereas average case analysis assumes the data are not “bad” (e.g., some sorting algorithms do well on average over all input orderings but very badly on certain input orderings).

-The motivation for amortized analysis is to better understand the running time of certain techniques, where standard worst case analysis provides an overly pessimistic bound. Amortized analysis generally applies to a method that consists of a sequence of operations, where the vast majority of the operations are cheap, but some of the operations are expensive. If we can show that the expensive operations are particularly rare we can change them to the cheap operations, and only bound the cheap operations.

-The general approach is to assign an artificial cost to each operation in the sequence, such that the total of the artificial costs for the sequence of operations bounds the total of the real costs for the sequence. This artificial cost is called the amortized cost of an operation. To analyze the running time, the amortized cost thus is a correct way of understanding the overall running time– but note that particular operations can still take longer so it is not a way of bounding the running time of any individual operation in the sequence.

Recursion & Backtracking

12 November 2020 09:08 AM

Known Concepts list

-Recursion

Imp Notes

-Generally, iterative solutions are more efficient than recursive solutions.

-any problem that can be solved recursively can also be solved iteratively

Backtracking

-a form of recursion where a part of problem is solved by choosing one option of several options in each recursive step until we have a set of all possible combination of options.

Points about recursion

-Global variables are not a part of recursion stack and hence is effected by every function call

-local variables are a part of recursion stack and hence is always new unaffected by any function call

- To pass data between recursive stack we need to pass it as function argument

Stacks

13 November 2020 10:38 AM

Known Concepts list
-Stack definition

Asymptotic analysis
-Array implementation

Space Complexity (for n push operations)	O(*n*)
Time Complexity of Push()	O(1)
Time Complexity of Pop()	O(1)
Time Complexity of Size()	O(1)
Time Complexity of IsEmptyStack()	O(1)
Time Complexity of IsFullStackf)	O(1)
Time Complexity of DeleteStackQ	O(1)

-Dynamic Array Implementation

Space Complexity (for *n* push operations)	O(*n*)
Time Complexity of CreateStack()	O(1)
Time Complexity of PushQ	O(1) (Average)
Time Complexity of PopQ	O(1)
Time Complexity of Top()	O(1)
Time Complexity of IsEmpryStackf)	O(1))
Time Complexity of IsFullStackf)	O(1)
Time Complexity of DeleteStackQ	O(1)

-Linked list Implementation

Space Complexity (for *n* push operations)	O(*n*)
Time Complexity of CreateStack()	O(1)
Time Complexity of Push()	O(1) (Average)
Time Complexity of Pop()	O(1)
Time Complexity of Top()	O(1)
Time Complexity of IsEmptyStack()	O(1)
Time Complexity of DeleteStack()	O(*n*)

Queue

21 November 2020 11:10 AM

Definition

A queue is an ordered list in which insertions are done at one end (rear) and deletions are done at other end (front). The first element to be inserted is the first one to be deleted. Hence, it is called First in First out (FIFO) or Last in Last out (LILO) list.

Asymptotic analysis

-simple circular array implementation

Performance: Let *n* be the number of elements in the queue:

Space Complexity (for n EnQueue operations)	O(*n*)
Time Complexity of EnQueue()	O(1)
Time Complexity of DeQueue()	O(1)
Time Complexity of IsEmptyQueue()	O(1)
Time Complexity of IsFullQueue()	O(1)
Time Complexity of QueueSize()	O(1)
Time Complexity of DeleteQueue()	O(1)

Limitations: The maximum size of the queue must be defined as prior and cannot be changed. Trying to *EnQueue* a new element into a full queue causes an implementation-specific exception.

-dynamic circular array implementation

Performance

Let *n* be the number of elements in the queue.

Space Complexity (for *n* EnQueue operations)	O(*n*)
Time Complexity of EnQueue()	O(1) (Average)
Time Complexity of DeQueue()	O(1)
Time Complexity of QueueSize()	O(1)
Time Complexity of IsEmptyQueue()	O(1)
Time Complexity of IsFullQueue()	O(1)
Time Complexity of QueueSize()	O(1)
Time Complexity of DeleteQueue()	O(1)

-linked List implementation

Performance

Let n be the number of elements in the queue, then

Space Complexity (for n EnQueue operations)	O(n)
Time Complexity of EnQueue()	O(1) (Average)
Time Complexity of DeQueue()	O(1)
Time Complexity of IsEmptyQueue()	O(1)
Time Complexity of DeleteQueue()	O(1)

Linked List

21 November 2020 11:53 AM

Array vs linked list

Array	Linked list
Constant time access to elements	O(n) time access to elements.
Pre-allocates all needed memory	Does not require pre-allocation.
Fixed size (accept for dynamic array)	Dynamic size by nature
One-block Allocation: allocates all the memory at once weather uses it or not.	One-unit allocation allocates memory as new element is added.
Inserting in the middle of array is complex	Very easy to insert an element in middle
In dynamic array we can grow or shrink arrays on the basis of how full it is but it has an overhead cost.	Grows gracefully without overhead cost.
All its elements are close to each other which is good for CPU caching	Elements are far apart connected by pointers.
Do not have extra memory cost.	Does have a memory cost of storing pointer.

Known concepts

- Singly linked list & doubly linked list
  - Insertion & deletion
    - Beginning
    - End
    - Middle
  - Deleting the whole list

Circular linked list

- A singly linked list where the tail points back to head.
- Traversing is done starting from head until we reach head back.

Topics left

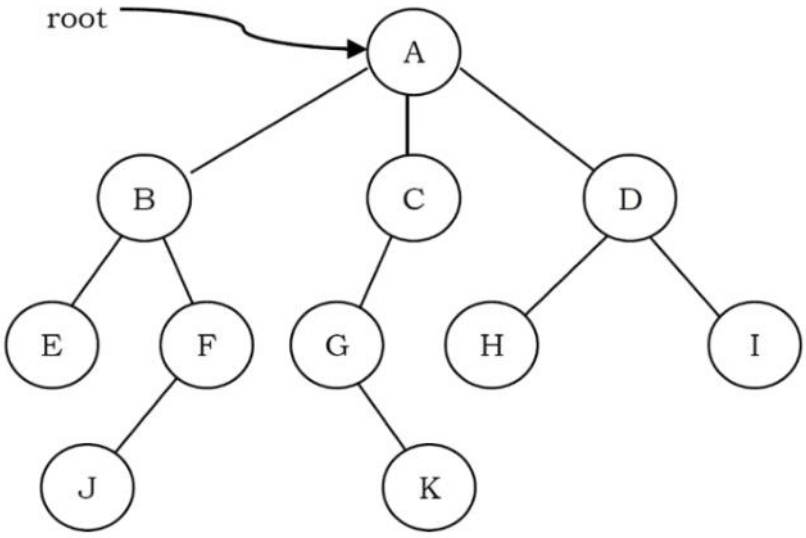
- 3.9 3.10 3.11

Trees

21 November 2020 01:21 PM

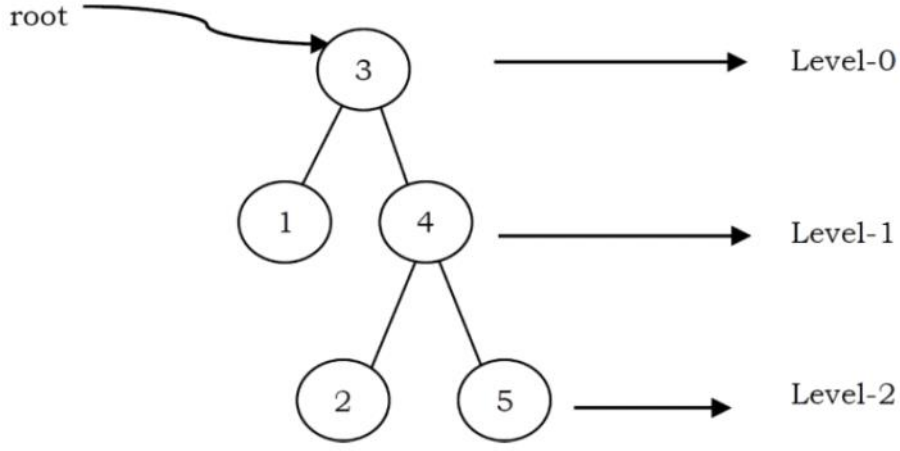
Known concepts

definition

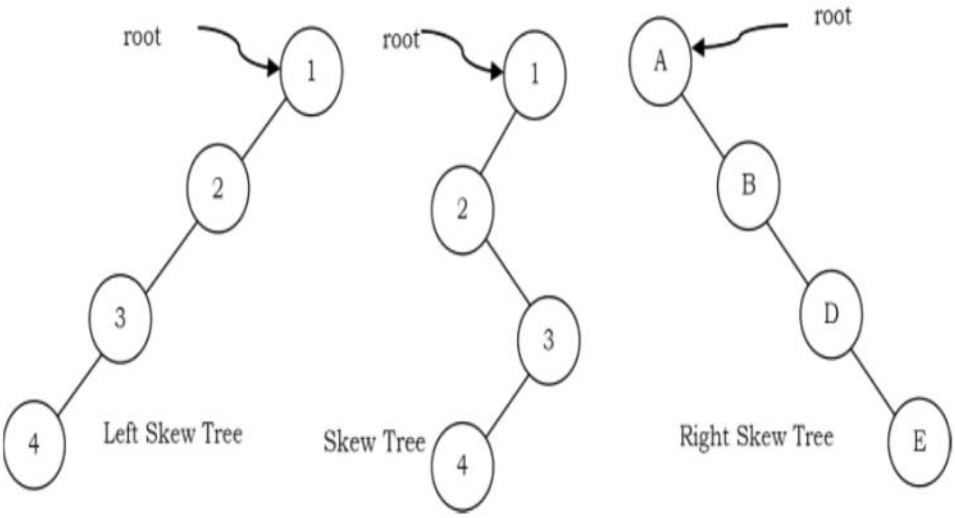


Terminology

- **Root :** The root of a tree is the node with no parents. There can be at most one root node in a tree (node A in the above example).
- **Edge :** An edge refers to the link from parent to child (all links in the figure).
- **Leaf :** A node with no children is called leaf node (E,J,K,H and I).
- **Siblings :** Children of same parent are called siblings (B,C,D are siblings of A, and E,F are the siblings of B).
- **Parent :**
- **Ancestor :** A node p is an ancestor of node q if there exists a path from root to q and p appears on the path. (For example, A,C and G are the ancestors of K).
- **Descendent :** The node q is called a descendant of p. **i**f there exists a path from root to q and p appears on the path. (For example K is the descendent of A,C,G).
- **Depth :**
  - -The depth of a node is the length of the path from the root to the node (depth of G is 2, A – C – G).
  - -depth of the tree is the maximum depth among all the nodes
- **Level :** The set of all nodes at a given depth is called the level of the tree (B, C and D are the same level). The root node is at level zero.



- **Height :**
  - -The height of a node is the length of the path from that node to the deepest node.
  - -The height of a tree is the length of the path from the root to the deepest node in the tree.
  - -A (rooted) tree with only one node (the root) has a height of zero. The height of B is 2 (B – F – J).
  - -Height of the tree is the maximum height among all the nodes in the tree
- **Size of node :** The size of a node is the number of descendants it has including itself (the size of the subtree C is 3).
- **Skew trees :** If every node in a tree has only one child (except leaf nodes) then we call such trees skew trees.
  - **Left skew trees :** If every node has only left child then we call them left skew trees.
  - **Right skew trees :** if every node has only right child then we call them right skew trees.



- An **internal node** is a **node** which carries at least one child

Other points

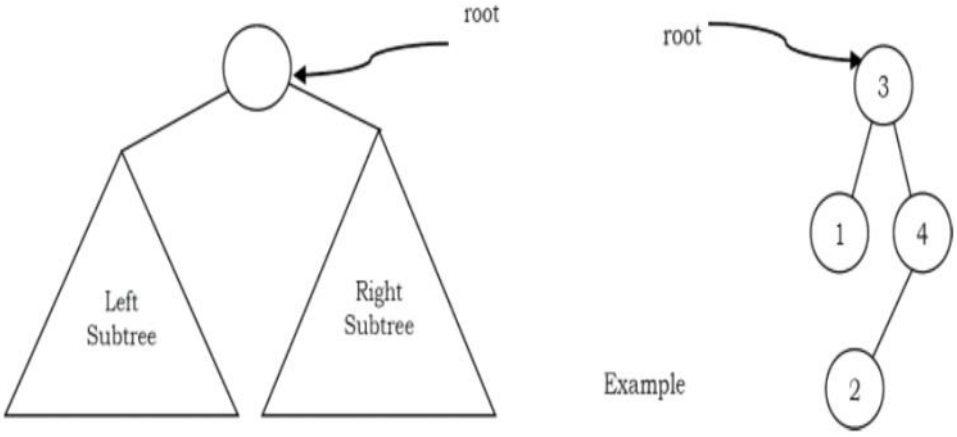
- -For a given tree, depth and height returns the same value. But for individual nodes we may get
- different results.

Binary trees

-definition

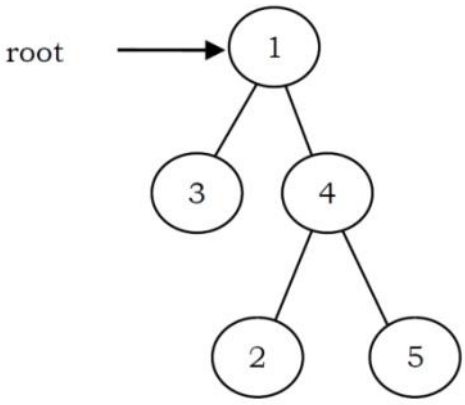
A tree is called binary tree if each node has zero child, one child or two children. Empty tree is also a valid binary tree. We can visualize a binary tree as consisting of a root and two disjoint binary trees, called the left and right subtrees of the root.

Generic Binary Tree

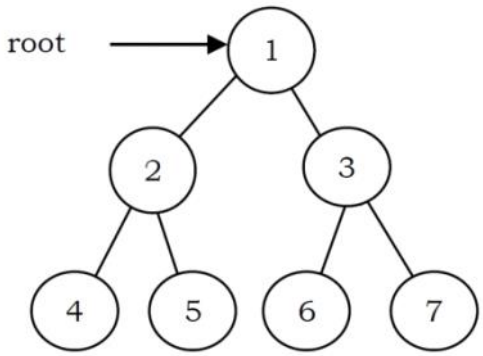


-Types

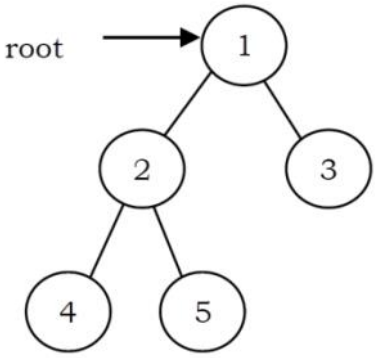
- **Strict Binary Tree :**
  A binary tree is called strict binary tree if each node has exactly two children or no children.



- **Full Binary Tree :**
  A binary tree is called full binary tree if each node has exactly two children and all leaf nodes are at the same level.



- **Complete Binary Tree :**
  A Binary Tree is a complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible .



Properties of binary tree

- The number of nodes n in a full binary tree is $2^{h+1} - 1$.
- At each level we have 2^l $nodes\ where\ l\ is\ the\ no\ of\ levels$
  The number of nodes n in a complete binary tree is between 2^h (minimum) and $2^{h+1} - 1$ (maximum).
- The number of leaf nodes in a full binary tree is 2^l .
- The number of NULL links (wasted pointers) in a complete binary tree of n nodes is n + 1.

Binary tree traversals

If we denote current data as D and the data(node) to the left as L and the node to the right as L, then the order in which the nodes are traversed are as follows.
1.LDR
2.RDL
3.DLR
4.DRL
5.RLD
6.LRD
The first two are called In order traversal
Next two are called Pre order Traversal
Next two are called Post order Traversal
These are called so based on how D is traversed

Pre order Traversal

Recursive

```
void PreOrder(struct BinaryTreeNode *root){
    if(root) {
        printf("%d",root→data);
        PreOrder(root→left);
        PreOrder (root→right);
    }
}
```

Time Complexity: O(n). Space Complexity: O(n).

Non - Recursive

```
void PreOrderNonRecursive(struct BinaryTreeNode *root){
    struct Stack *S = CreateStack();
    while(1) {
        while(root) {
            //Process current node
            printf("%d",root→data);
            Push(S,root);
            //If left subtree exists, add to stack
            root = root→left;
        }
        if(IsEmptyStack(S))
            break;
        root = Pop(S);
        //Indicates completion of left subtree and current node, now go to right subtree
        root = root→right;
    }
    DeleteStack(S);
}
```

Time Complexity: O(n). Space Complexity: O(n).

In order Traversal

Recursive

```
void InOrder(struct BinaryTreeNode *root){
    if(root) {
        InOrder(root→left);
        printf("%d",root→data);
        InOrder(root→right);
    }
}
```

Time Complexity: O(n). Space Complexity: O(n).

Non-Recursive

```
void InOrderNonRecursive(struct BinaryTreeNode *root){
    struct Stack *S = CreateStack();
    while(1) {
        while(root) {
            Push(S,root);
            //Got left subtree and keep on adding to stack
            root = root→left;
        }
        if(IsEmptyStack(S))
            break;
        root = Pop(S);
        printf("%d", root→data);   //After popping, process the current node
        //Indicates completion of left subtree and current node, now go to right subtree
        root = root→right;
    }
    DeleteStack(S);
}
```

Time Complexity: O(n). Space Complexity: O(n).

Post order Traversal

Recursive

```
void PostOrder(struct BinaryTreeNode *root){
    if(root)        {
            PostOrder(root→left);
            PostOrder(root→right);
            printf("%d",root→data);
        }
}
```

Time Complexity: O(n). Space Complexity: O(n).

Non recursive

```
void PostOrderNonRecursive(struct BinaryTreeNode *root) {
   struct SimpleArrayStack *S = CreateStack();
   struct BinaryTreeNode *previous = NULL;
   do{
      while (root!=NULL){
         Push(S, root);
         root = root->left;
      }
      while(root == NULL && !IsEmptyStack(S)){
         root = Top(S);
         if(root->right == NULL || root->right == previous){
            printf("%d ", root->data);
            Pop(S);
            previous = root;
            root = NULL;
         }
         else
            root = root->right;
      }
   }while(!IsEmptyStack(S));
}
```

Time Complexity: O(n). Space Complexity: O(n).

Level order Traversal

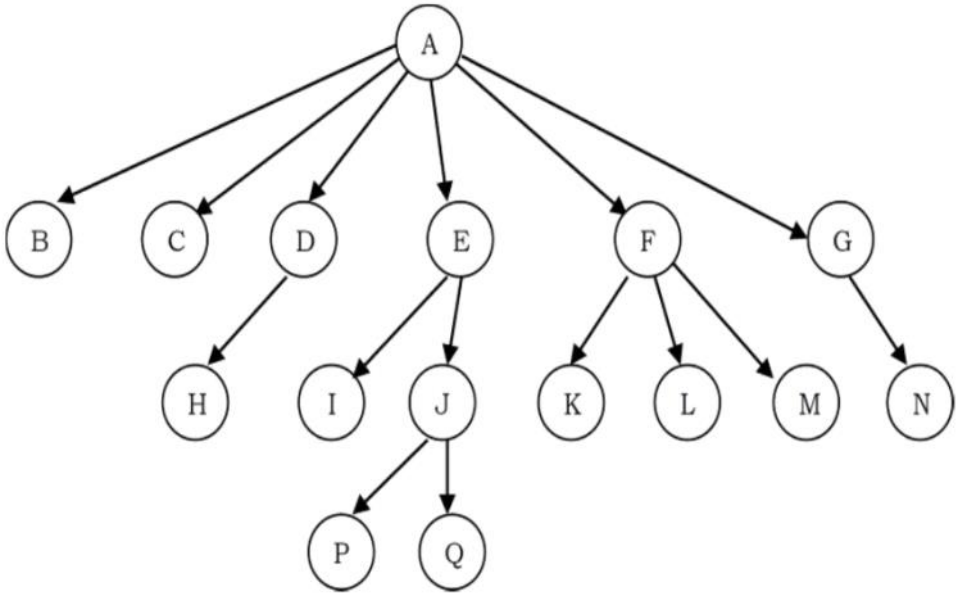
When nodes of first level are traversed then next and so on

```
void LevelOrder(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    struct Queue *Q = CreateQueue();
    if(!root)
        return;
    EnQueue(Q,root);
    while(!IsEmptyQueue(Q)) {
        temp = DeQueue(Q);
        //Process current node
        printf("%d", temp→data);
        if(temp→left)
            EnQueue(Q, temp→left);
        if(temp→right)
            EnQueue(Q, temp→right);
    }
    DeleteQueue(Q);
}
```

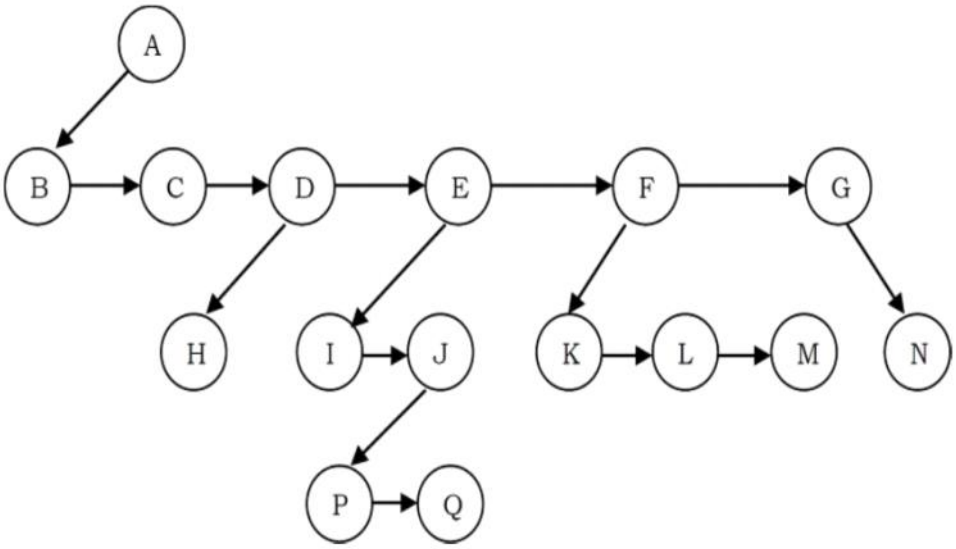
Time Complexity: O(n). Space Complexity: O(n). Since, in the worst case, all the nodes on the entire last level could be in the queue simultaneously.

Generic Trees (N-ary Trees)

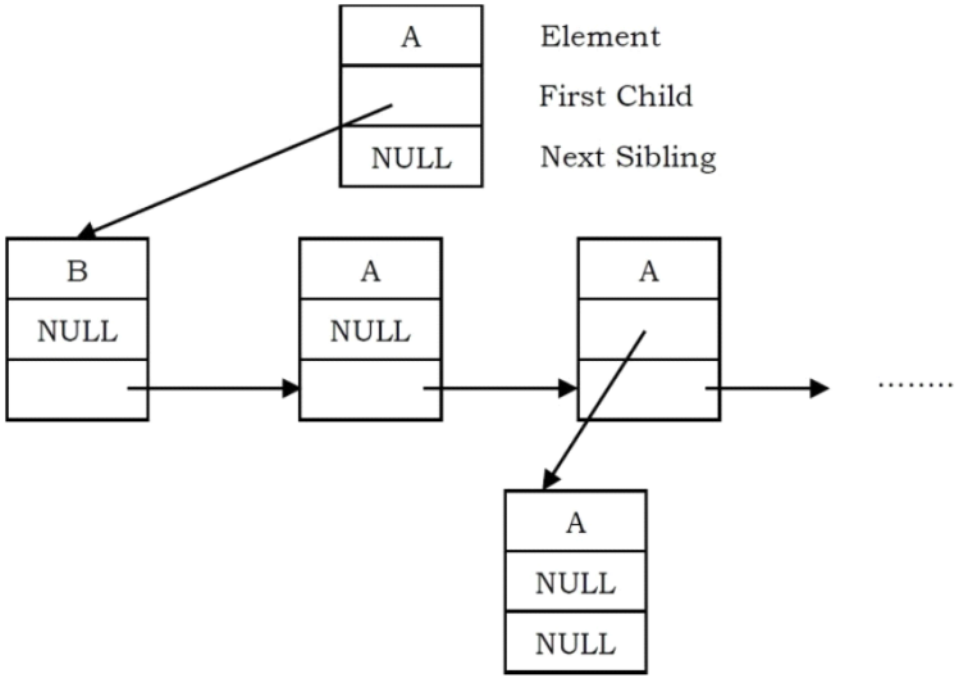
These type of tree can be represented as



We here consider a general form of tree in which we don't have a constant no. Of children at each node
Such tree can be represented as First child / Next sibling form .



Or



This in turn effectively is a binary tree with first node holding a n-ary tree and second one holding a LL of Nodes.

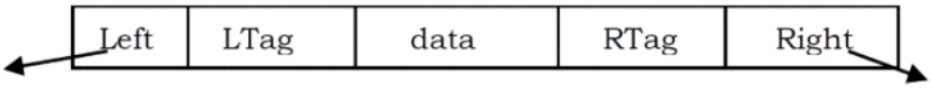
Threaded binary tree

All the previous traversal algorithms require at least one of either queue or linked list and we also see that a lot of point ers (of leaf) remain null which is a wastage of space.
Here we use those pointers to improve our traversing algorithm without using extra space.
In threaded binary trees we store information regarding which node to traverse next in those free pointers so that we don't require any new stack or LL for that.

Based on which null pointer contains those info we classify threaded binary trees as follows

- If we store predecessor information in NULL left pointers only, then we can call such binary trees left threaded binary trees.
- If we store successor information in NULL right pointers only, then we can call such binary trees right threaded binary trees.
- If we store predecessor information in NULL left pointers and successor information in NULL right pointers, then we can call such binary trees fully threaded binary trees or simply threaded binary trees.

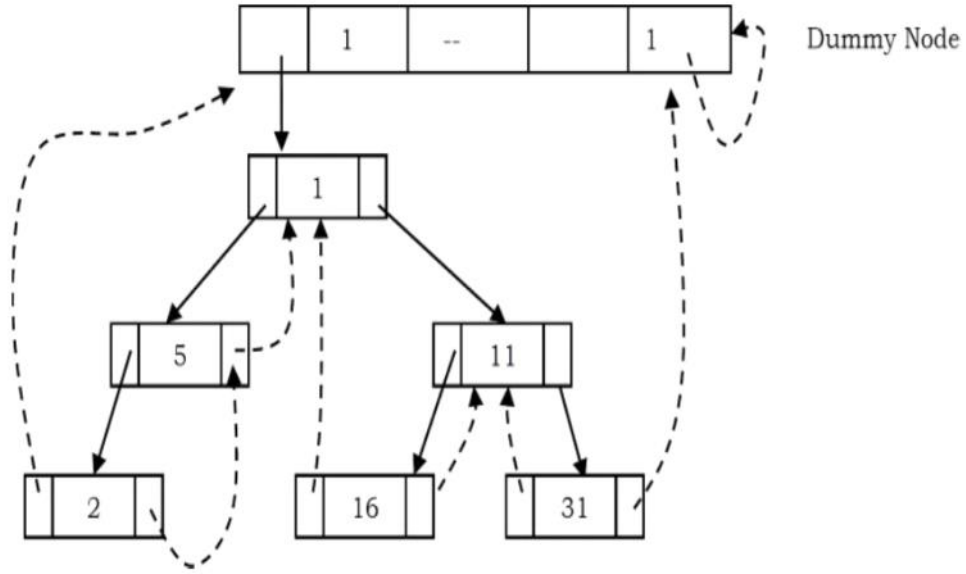
To differentiate between weather the Pointers contain children or a reference to successor or predecessor 2 more variables a re used usually called Ltag and Rtag
Described as follows:



	Regular Binary Trees	Threaded Binary Trees
if LTag == 0	NULL	left points to the in-order predecessor
if LTag == 1	left points to the left child	left points to left child
if RTag == 0	NULL	right points to the in-order successor
if RTag == 1	right points to the right child	right points to the right child

Note: Similarly, we can define preorder/postorder differences as well.

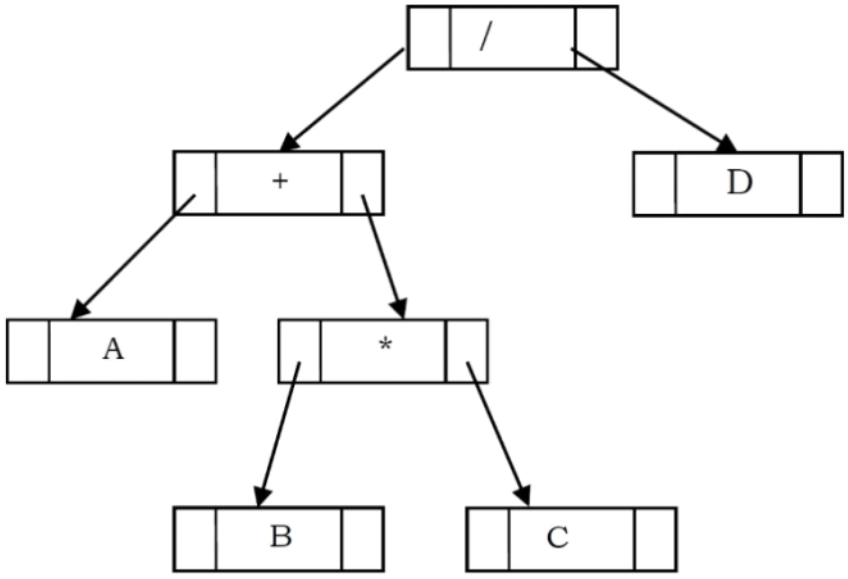
But the leftmost and rightmost node are still empty and does not points to anything so they are made to point a dummy node the rightmost node of this dummy node points to the whole tree as shown



This is an example of in order threaded binary tree

Expression trees

A tree representing expression is called expression tree.
In expression trees, leaf nodes are operands and non-leaf nodes are operators
But this is only for binary operations
For unary operation only one child will be present
For ternary operations three children will be present
The figure below shows a simple expression tree for (A + B * C) / D.



Binary search trees (BST)

A binary tree in which All the left subtree elements should be less than root element and all the right subtree elements should be greater than the root element also both right and left sub trees must also be binary search trees .

Important Notes on Binary Search Trees

- Since root data is always between left subtree data and right subtree data, performing inorder traversal on binary search tree produces a sorted list.
- While solving problems on binary search trees, first we process left subtree, then root data, and finally we process right subtree. This means, depending on the problem, only the intermediate step (processing root data) changes and we do not touch the first and third steps.
- If we are searching for an element and if the left subtree root data is less than the element we want to search, then skip it. The same is the case with the right subtree.. Because of this, binary search trees take less time for searching an element than regular binary trees. In other words, the binary search trees consider either left or right subtrees for searching an element but not both.
- The basic operations that can be performed on binary search tree (BST) are insertion of element, deletion of element, and searching for an element. While performing these operations on BST the height of the tree gets changed each time. Hence there exists variations in time complexities of best case, average case, and worst case.
- The basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with node n, such operations runs in O(lgn) worst-case time. If the tree is a linear chain of n nodes (skew-tree), however, the same operations takes O(n) worst-case time.

Search in BST

Recursive

```
struct BinarySearchTreeNode *Find(struct BinarySearchTreeNode *root, int data ){
    if( root == NULL )
        return NULL;
    if( data < root→data )
        return Find(root→left, data);
    else if( data > root→data )
        return( Find( root→right, data );
    return root;
}
```

Time Complexity: O(n), in worst case (when BST is a skew tree). Space Complexity: O(n), for recursive stack.

Non Recursive

```
struct BinarySearchTreeNode *Find(struct BinarySearchTreeNode *root, int data ){
    if( root == NULL )
        return NULL;
    while (root) {
        if(data == root→data)
            return root;
        else if(data > root→data)
            root = root→right;
        else root = root→left;
    }
    return NULL;
}
```

Time Complexity: O(n). Space Complexity: O(1).

Minimum element in BST

recursive

```
struct BinarySearchTreeNode *FindMin(struct BinarySearchTreeNode *root){
    if(root == NULL)
        return NULL;
    else  if( root→left == NULL )
        return root;
    else
        return FindMin( root→left );
}
```

Time Complexity: O(n), in worst case (when BST is a *left skew* tree).
Space Complexity: O(n), for recursive stack.

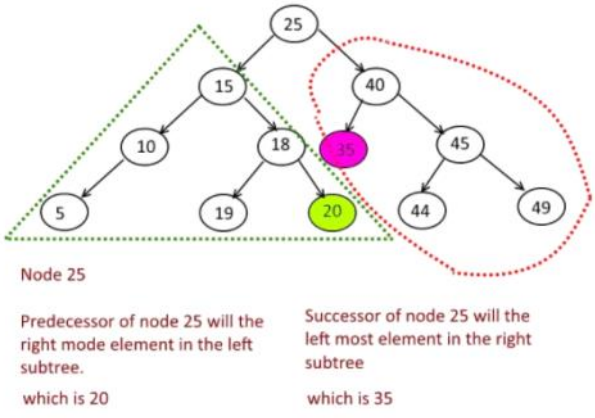
Non recursive

```
struct BinarySearchTreeNode *FindMax(struct BinarySearchTreeNode * root ) {
    if( root == NULL )
        return NULL;
    while( root→right != NULL )
        root = root→right;
    return root;
}
```

Time Complexity: O(n). Space Complexity: O(1).

In order Predecessor and Successor

When you do the in order traversal of a binary tree, the neighbours of given node are called Predecessor(the node lies behind of given node) and Successor (the node lies ahead of given node).



Inserting in BST

```
struct BinarySearchTreeNode *Insert(struct BinarySearchTreeNode *root, int data) {
    if( root == NULL ) {
        root = (struct BinarySearchTreeNode *) malloc(sizeof(struct  BinarySearchTreeNode));
        if( root == NULL ) {
            printf("Memory Error");
            return;
        }
        else {
            root→data = data;
            root→left = root→right = NULL;
         }
    }
    else {
        if( data < root→data )
            root→left = Insert(root→left,  data);
        else if( data > root→data )
            root→right = Insert(root→right,  data);
    }

    return root;
}
```

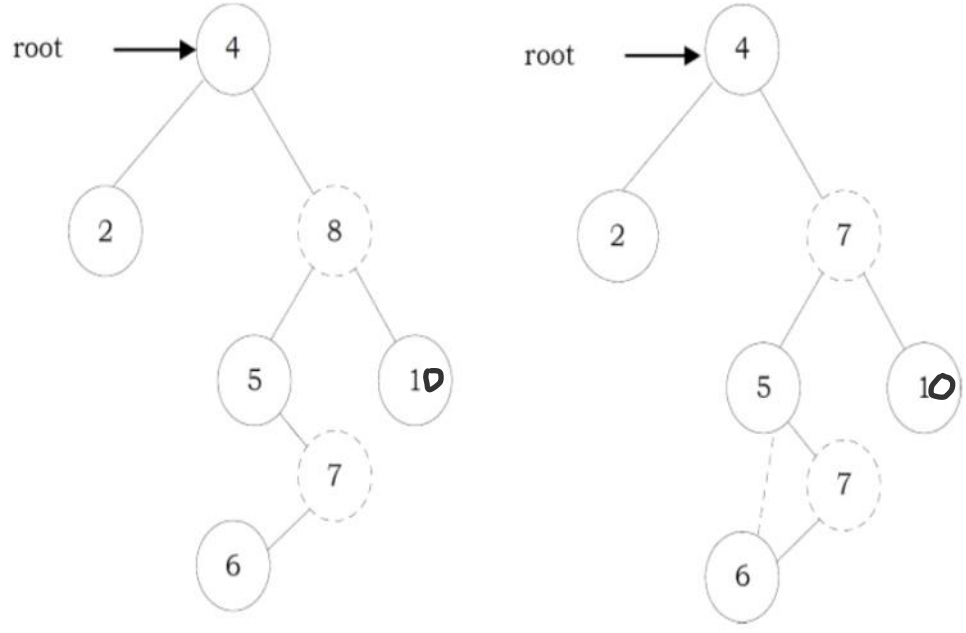
Note: In the above code, after inserting an element in subtrees, the tree is returned to its parent. As a result, the complete tree will get updated.

Time Complexity:O(n).
Space Complexity:O(n), for recursive stack. For iterative version, space complexity is O(1).

Deleting an element in BST

If the element to be deleted has both children: The general strategy is to replace the key of this node with the largest element of the left subtree and recursively delete that node (which is now empty).

The largest node in the left subtree cannot have a right child, so the second delete is an easy one.



Algo:

```
struct BinarySearchTreeNode *Delete(struct BinarySearchTreeNode *root, int data) {
    struct BinarySearchTreeNode *temp;
    if( root == NULL )
        printf("Element not there in tree");
    else if(data < root→data)
            root→left = Delete(root→left, data);
    else if(data > root→data )
        root→right = Delete(root→right, data);
    else {
        //Found element
        if( root→left && root→right ) {
            /* Replace with largest in left subtree */
            temp = FindMax( root→left );
            root→data = temp→data;
            root→left = Delete(root→left, root→data);
        }
        else {
                /* One child */
                temp = root;
                if( root→left == NULL )
                    root = root→right;
                if( root→right == NULL )
                    root = root→left;
                free( temp );
        }
    }
    return root;
}
```

Time Complexity: O(n). Space Complexity: O(n) for recursive stack. For iterative version, space complexity is O(1).

Balanced BST

When trees are skew the worst case complexity becomes O(n)
To solve this problem a full balanced trees are used whose worst case complexity is O(log(n))
These trees are also called HB(k) or height balanced trees with a balance factor of k, where k is the difference between left subtree height and right subtree height
HB(0) is called a full balanced tree.

AVL trees

In HB(k), if k = 1 (if balance factor is one), such a binary search tree is called an AVL tree. That means an AVL tree is a binary search tree with a balance condition: the difference between left subtree height and right subt ree height is at most 1.

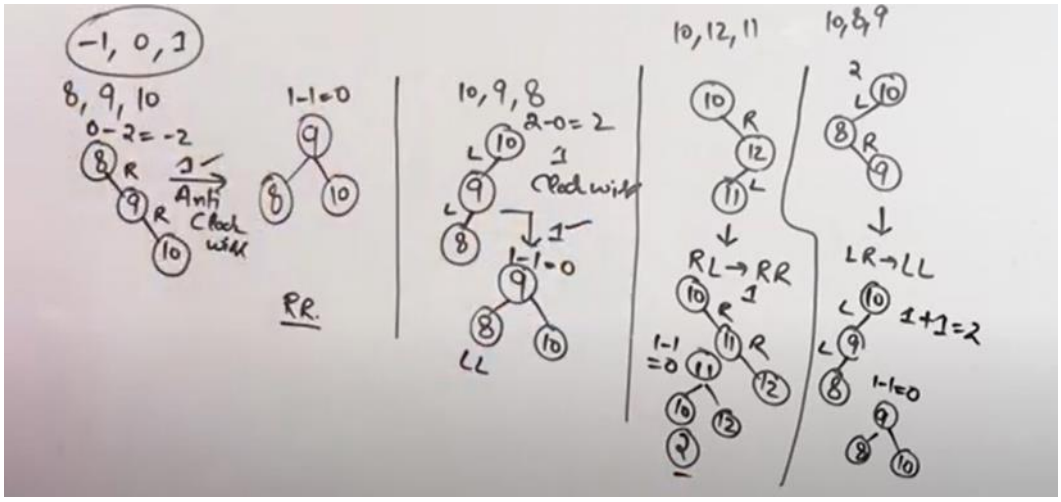
Height of such trees is in order of logn i.e O(logn) n is nodes

AVL tree is similar to a BST but at every insertion we need to maintain its property
We do that by rotations

If we continue to balance AVL tree at the time of any violations we will have a balance factor (height of left sub tree - height of right sub tree) of at most 2 or -2

Following are the four cases of imbalances

1. An insertion into the left subtree of the left child of X. [LL]
2. An insertion into the right subtree of the left child of X. [LR]
3. An insertion into the left subtree of the right child of X. [RL]
4. An insertion into the right subtree of the right child of X. [RR]


-1, 0, 1
8, 9, 10
0-2=-2
1
Anti Clock wise
1-1=0
RR.
10, 9, 8
2-0=2
1
Clock wise
1-1=0
LL
10, 12, 11
RL → RR
1
1-1 =0
10, 8, 9
2
LR → LL
1+1=2
1-1=0

Priority Queues & Heaps

18 December 2020 07:57 AM

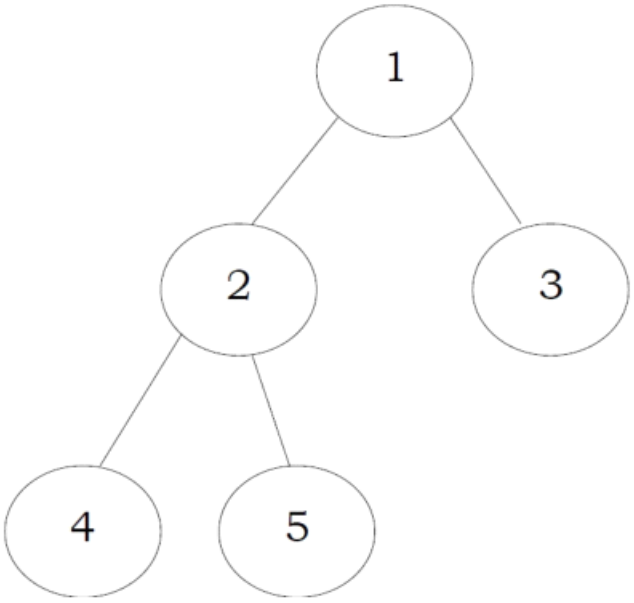
Priority queue is kind of queue where elements are extracted in a sequence not necessary to the sequence of insertion but rather they are extracted in either ascending order or descending order

These operations are similar to Enqueue and Dequeue operations Except that in case of daqing dequeus either the maximum or the minimum element

if it dequeus smallest element discord ascending priority queue otherwise descending priority queue

Heap

A heap is a tree with some special properties. The basic requirement of a heap is that the value of
a node must be $\geq$ (or $\leq$) than the values of its children. This is called heap property. A heap also
has the additional property that all leaves should be at h or h – 1 levels (where h is the height of
the tree) for some h > 0 (complete binary trees). That means heap should form a complete binary
tree (as shown below).



Types

- Min heap : The value of a node must be less than or equal to the values of its children
- Max heap : The value of a node must be greater than or equal to the values of its children

Representing heaps

Heaps can be represented as an array or a binary tree
Representing heap as an array means that though operations are performed as if it were a tree but the keys are stored in an array, storing becomes simple that way

Index calculation

Parent = (i-1)/2

Left child= 2*i +1

Right child = 2*i+2

heapifying an element

hey depending on the elements position dis compared either to its parent or its children
depending on whether it is maximum heap or minimum heap the elements is either
swapped up or down until the heap condition is satisfied
when it is swapped up it is called percolate up
when it is swap down it's called percolate down

Inserting an element

After inserting an element in a binary heap it may not satisfy Hey property
so after insertion at any of the leaves he is compared to its parents to check whether it
satisfies heap property it's not it is swapped and is done so until the heap property is satisfied
Complexity O(log(n)) n- node
This is also called percolate up

Deleting an element

Only deleting the root element is supported in a binary
heap to delete the root element one of the leaf notes is
copied to the root and the leaf element is destroyed then
percolate down is called
Complexity O(log(n)) n- node

Important Observation

hey children are maintaining the heap property
so in case of leaf node there's no children and hence they're always heapified
so we don't need to check leaf nodes

Heap sort

One main application of heap ADT is sorting (heap sort). The heap sort algorithm inserts all elements (from an unsorted array) into a heap, then removes them from the root of a heap until the heap is empty.

Disjoint Sets

19 December 2020 10:43 AM

As the name suggest disjoint sets are the data types in which elements as in mathematical sets are he presents and order is disregarded

The main operation of disjoint sets is

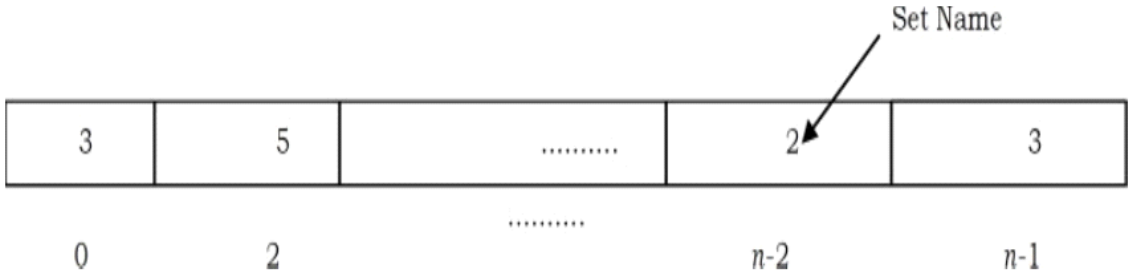
- Make set- takes an element and makes a set out of it
- Find - takes an element and find is set name
- Union - takes two element and combine them under one set name

This data structure is implemented in two ways

- Quick find-but slow union
- Quick union but slow find

Quick find

The whole data structure is implemented in array the index number is the data and the name of the set is stored in that index as shown.



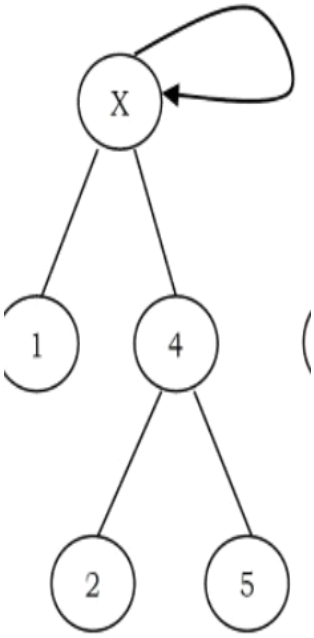
Here the find operation works in O(1) time

The union operation works in O(n) time (for example zero union 2 would mean changing all the threes to 5)

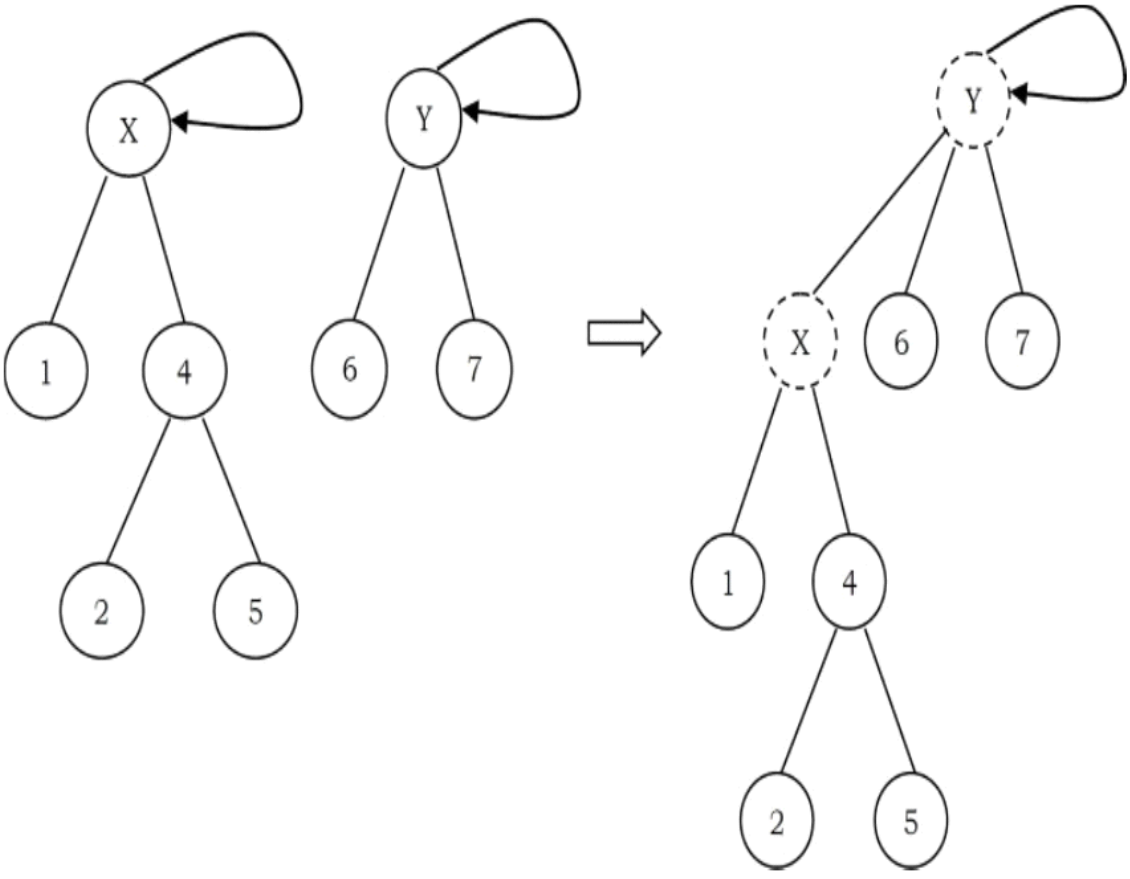
Fast Union

implementation one

the set is represented as a tree with the root as set name



Union operation



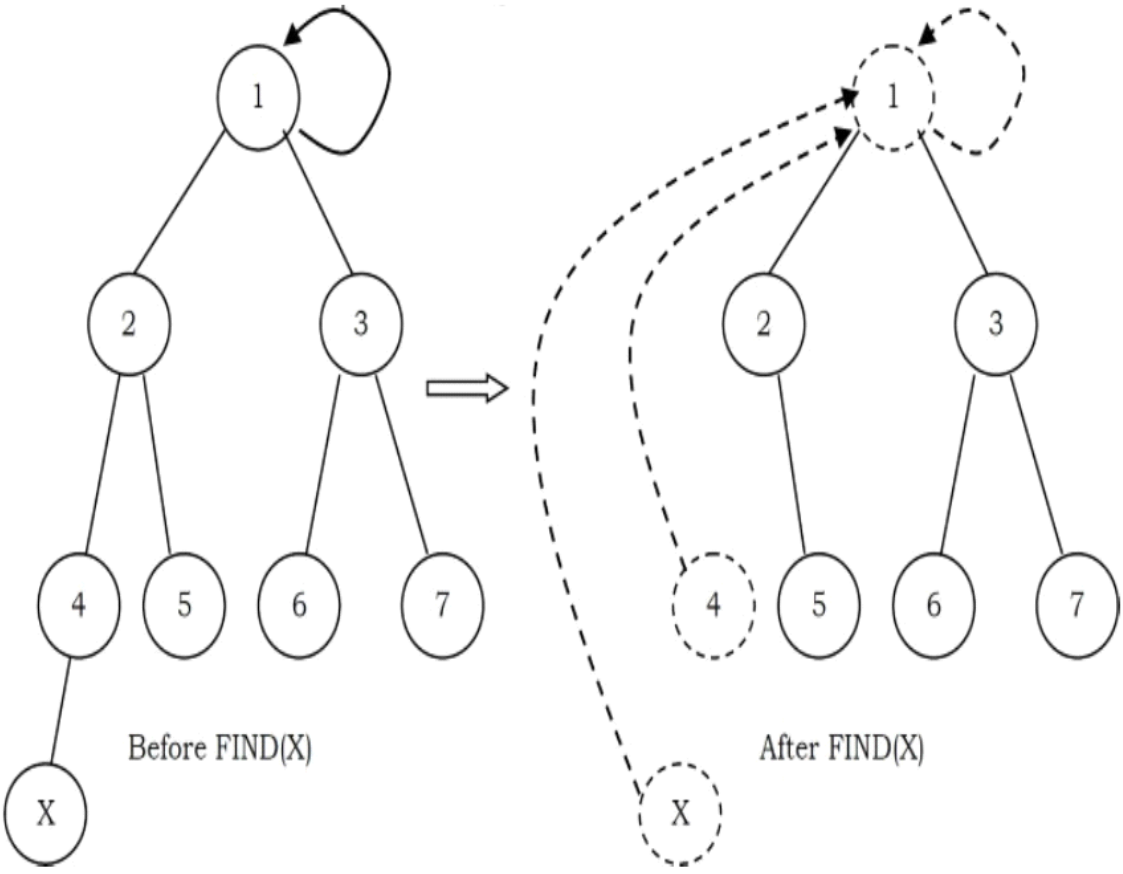
Complexity of union operation is O(1)
for a find operation one has to traverse from the node to the root node
complexity of find operation O(h) where h is tree height
To improve the complexity of find operation we usually

union by size (weight) - Make the smaller tree a subtree of the larger tree (the depth of any node is never more than logn)

union by height (rank) - Make the tree with less height a subtree of the tree with more height
(if we take the UNION of two trees of the same height, the height of the UNION is one larger than the common height, and otherwise equal to the max of the two heights.)

Path compression

With path compression the only change to the FIND function is that S[X] is made equal to the value returned by FIND. That means, after the root of the set is found recursively, X is made to point directly to it. This happen recursively to every node on the path to the root.



8.10 Summary

Performing *m* union-find operations on a set of *n* objects.

Algorithm	Worst-case time
Quick-Find	*mn*
Quick-Union	*mn*
Quick-Union by Size/Height	$n + m \log n$
Path compression	$n + m \log n$
Quick-Union by Size/Height + Path Compression	$(m + n) \log n$

Graphs

22 December 2020 08:32 AM

Intro

9.2 Glossary

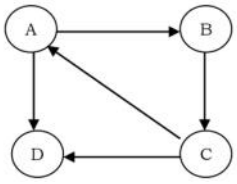
Graph: A graph is a pair (V, E), where V is a set of nodes, called *vertices*, and £ is a collection of pairs of vertices, called *edges*.

- *Vertices* and *edges* are positions and store elements

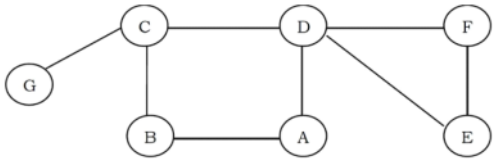
- Definitions that we use:
  - *Directed edge:*
    - ordered pair of vertices (u, v)
    - first vertex u is the origin
    - second vertex v is the destination
    - Example: one-way road traffic

  - *Undirected edge:*
    - unordered pair of vertices (u, v)
    - Example: railway lines

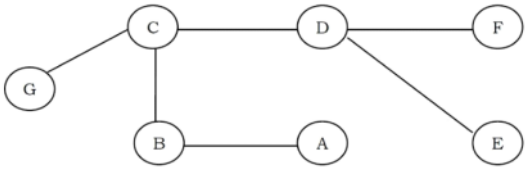
  - *Directed graph:*
    - all the edges are directed
    - Example: route network



  - *Undirected graph:*
    - all the edges are undirected
    - Example: flight network



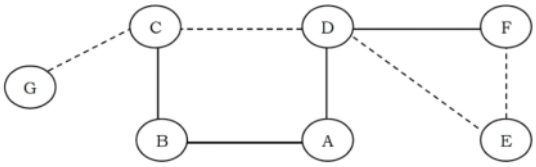
- When an edge connects two vertices, the vertices are said to be adjacent to each other and the edge is incident on both vertices.
- A graph with no cycles is called a *tree*. A tree is an acyclic connected graph.



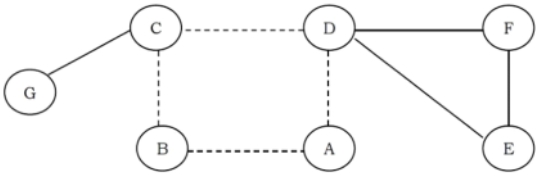
- A self loop is an edge that connects a vertex to itself.

- Two edges are parallel if they connect the same pair of vertices.

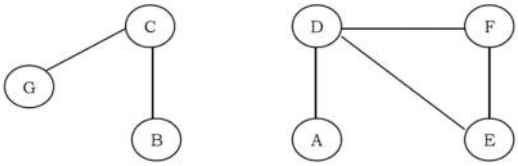
- The *D*egree of a vertex is the number of edges incident on it.
- A subgraph is a subset of a graph's edges (with associated vertices) that form a graph.
- A path in a graph is a sequence of adjacent vertices. Simple path is a path with no repeated vertices. In the graph below, the dotted lines represent a path from *G* to *E*.



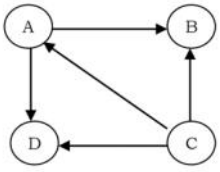
- A cycle is a path where the first and last vertices are the same. A simple cycle is a cycle with no repeated vertices or edges (except the first and last vertices).



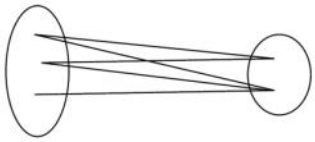
- We say that one vertex is connected to another if there is a path that contains both of them.
- A graph is connected if there is a path from *every* vertex to every other vertex.
- If a graph is not connected then it consists of a set of connected components.



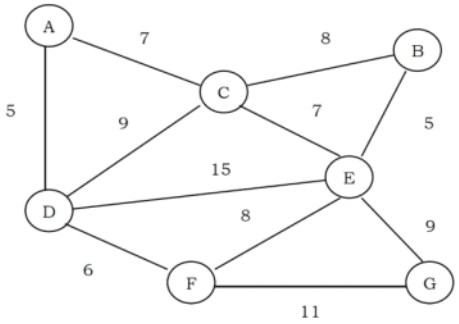
- A *directed* acyclic graph [DAG] is a directed graph with no cycles.



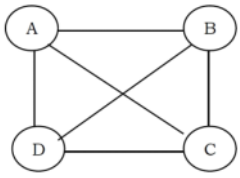
- A forest is a disjoint set of trees.
- A spanning tree of a connected graph is a subgraph that contains all of that graph's vertices and is a single tree. A spanning forest of a graph is the union of spanning trees of its connected components.
- A bipartite graph is a graph whose vertices can be divided into two sets such that all edges connect a vertex in one set with a vertex in the other set.



- In *weighted graphs* integers (*weights*) are assigned to each edge to represent (distances or costs).



- Graphs with all edges present are called *complete* graphs.

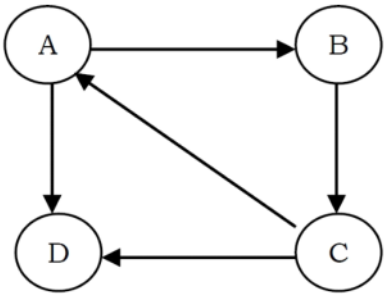


- Graphs with relatively few edges (generally if it edges < $|V| \log |V|$) are called *sparse graphs*.
- Graphs with relatively few of the possible edges missing are called *dense*.
- Directed weighted graphs are sometimes called *network*.
- We will denote the number of vertices in a given graph by $|V|$, and the number of edges by $|E|$. Note that E can range anywhere from 0 to $|V|(|V| - 1)/2$ (in undirected graph). This is because each node can connect to every other node.

Graph representation

Adjacency matrix

in this representation we use a matrix of v x v where v is the number of vertices
the value of the matrix is boolean
M[a][b] is set to one if a is connected to b and 0 otherwise
In an undirected graph if a is connected to b then B is also said to be connected to a
hence M[a][b]=M[b][a]=1
but in a directed graph if a is connected to b
Then only M[a][b]=1 to Represent the connection between a to b and not otherwise
Eg:

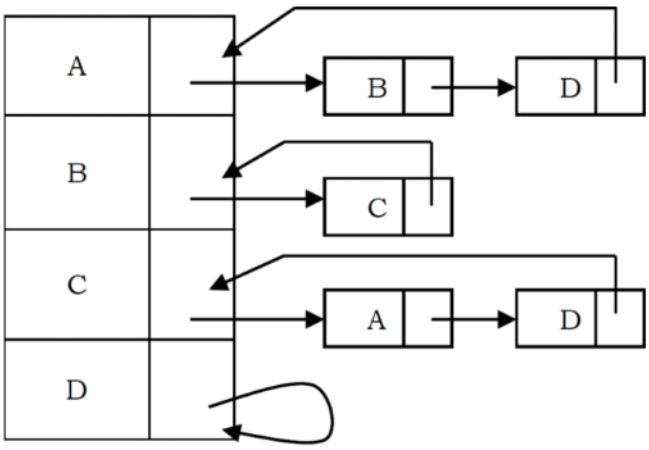


The adjacency matrix for this graph can be given as:

	A	B	C	D
A	0	1	0	1
B	0	0	1	0
C	1	0	0	1
D	0	0	0	0

Adjacency list

In this representation all the vertices connected to a particular vertex is represented by a list called adjacency list
hey there are as many list as the number of vertices
For the above graph this is the linked list implementation of adjacency list



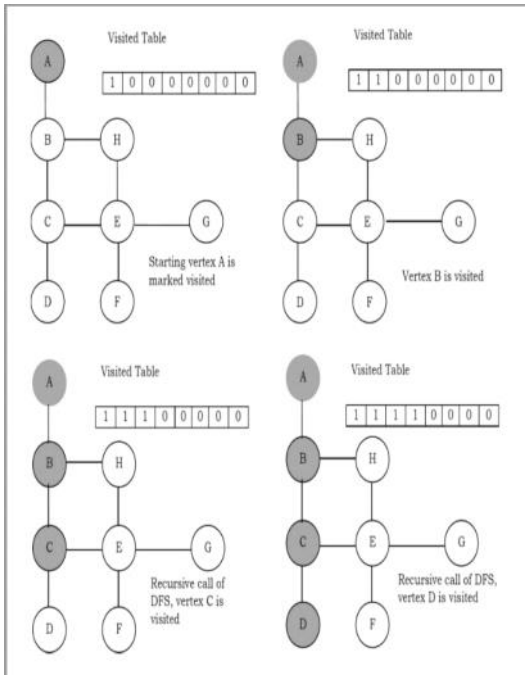
Adjacency set

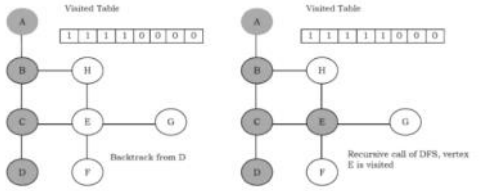
it is similar to adjacency list but instead of linked list disjoint sets are used

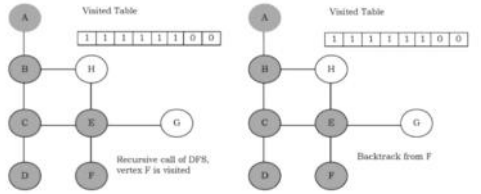
Representation	Space	Checking edge between v and w?	Iterate over edges incident to v?
List of edges	E	E	E
Adj Matrix	V^2	1	V
Adj List	$E+V$	$Degree(v)$	$Degree(v)$
Adj Set	$E+V$	$log(Degree(v))$	$Degree(v)$

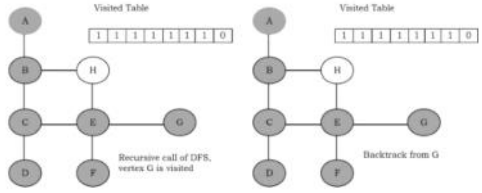
Graph traversals

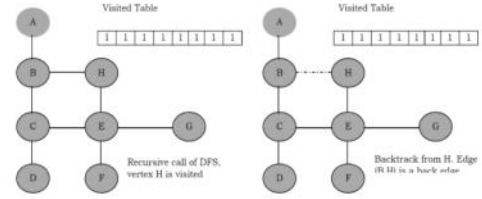
DFS

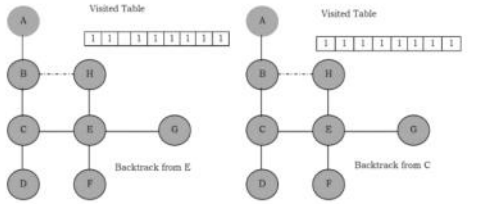


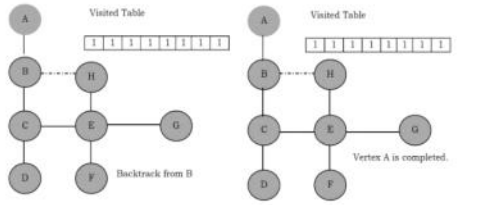

Visited Table
1 1 1 1 0 0 0 0
A
B
H
C
E
G
D
F
Backtrack from D
Visited Table
1 1 1 1 1 0 0 0
Recursive call of DFS, vertex E is visited


Visited Table
1 1 1 1 1 1 0 0
Recursive call of DFS, vertex F is visited
Visited Table
1 1 1 1 1 1 0 0
Backtrack from F

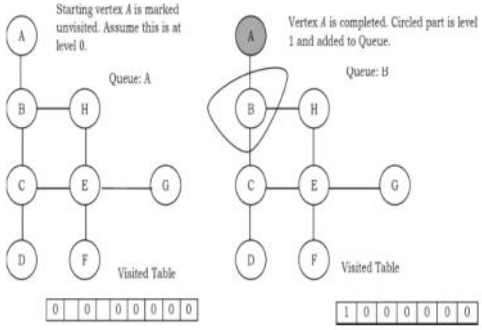

Visited Table
1 1 1 1 1 1 1 0
Recursive call of DFS, vertex G is visited
Visited Table
1 1 1 1 1 1 1 0
Backtrack from G

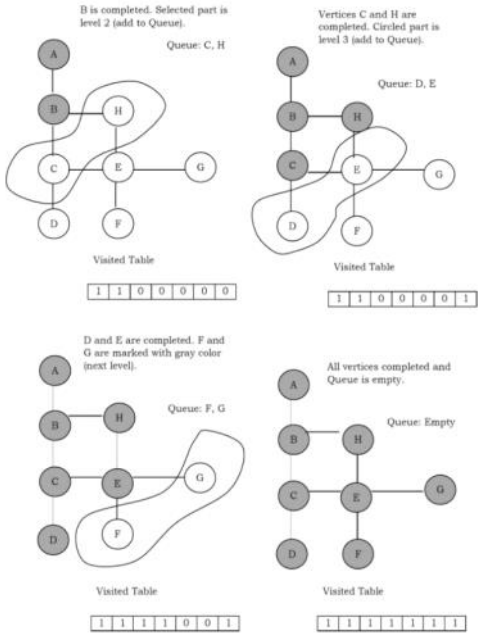

Visited Table
1 1 1 1 1 1 1 1
Recursive call of DFS, vertex H is visited
Visited Table
1 1 1 1 1 1 1 1
Backtrack from H. Edge


Visited Table
1 1 1 1 1 1 1 1
Backtrack from E
Visited Table
1 1 1 1 1 1 1 1
Backtrack from C


Visited Table
1 1 1 1 1 1 1 1
Backtrack from B
Visited Table
1 1 1 1 1 1 1 1
Vertex A is completed.

BFS

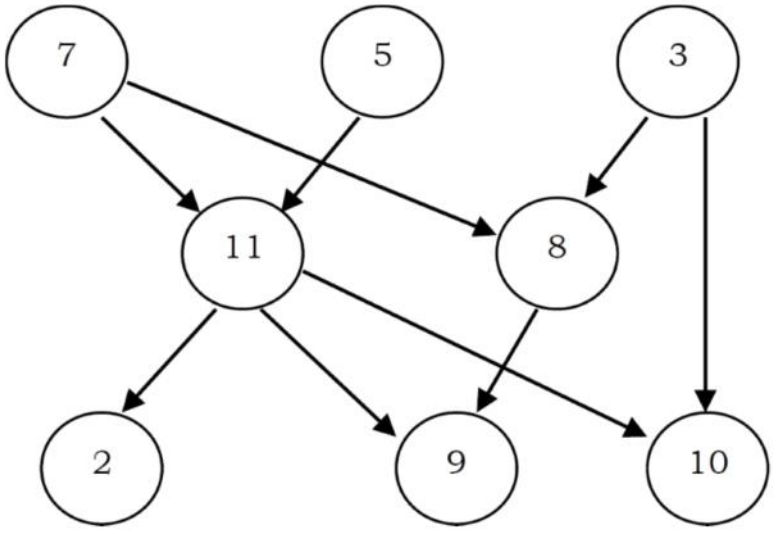

Starting vertex A is marked unvisited. Assume this is at level 0.
Queue: A
Visited Table
0 0 0 0 0 0 0
Vertex A is completed. Circled part is level 1 and added to Queue.
Queue: B
Visited Table
1 0 0 0 0 0 0



Topological Sort

Topological sort is an ordering of vertices in a directed acyclic graph [DAG] in which each node comes before all nodes to which it has outgoing edges.

Eg:



7, 5, 3, 11, 8, 2, 9, 10 and 3, 5, 7, 8, 11, 2, 9, 10 are both topological orderings.

Algo

While nodes remain

Push those nodes in any order who does not have incoming edges

Delete them from graph //when you delete them you expose new nodes who does not have incoming edges

Shortest path Algorithms

There are three versions of it

Shortest path in unweighted Graph - a special case Dijkstra's Algorithm

Shortest path in weighted Graph - Dijkstra's Algorithm

Shortest path in weighted graph with negative edges - Bellman Ford Algorithm

First one

Algo

Create a queue

create a 3 x v array where v = no. Of vertex

Initialize it like this:

Vertex	Distance[v]	Previous vertex which gave Distance[v]
A	-1	-
B	-1	-
C	0	-
D	-1	-
E	-1	-
F	-1	-
G	-1	-

Enqueue the node from which distance is to be found

While the queue is not empty

V=deque the queue

For each w adjacent to v

If distance[w]==-1

Distance[w]=Distance[v]+1
Path[w]=v
Enqueue w

At the end delete the queue

Second one (Dijkstra's)

Algo

Create a Priority queue
create a 3 x v array where v = no. Of vertex
Initialize it like this:

Vertex	Distance[v]	Previous vertex which gave Distance[v]
A	-1	-
B	-1	-
C	0	-
D	-1	-
E	-1	-
F	-1	-
G	-1	-

Enqueue the node from which distance is to be found //distance is the priority
While the priority queue is not empty
V=delete min from queue
For each w adjacent to v
If distance[w]==-1
Distance[w]=Distance[v]+1
Path[w]=v
Enqueue w
Id distance[w]> new distance
Distance[w]=new distance
Update the priority of the vertex
Path[w]=v

At the end delete the queue

Why Dijkstra's fail with negative edge

In using The priority queue and selecting the node hey with minimum priority Dijkstra hey already assumes that there will beno negative edge because it assumes that having an extra edge in the path will only increase the distance.

Third one (Bellman-Ford)

Algo

Create a queue
create a 3 x v array where v = no. Of vertex
Initialize it like this:

Vertex	Distance[v]	Previous vertex which gave Distance[v]
A	-1	-
B	-1	-
C	0	-
D	-1	-
E	-1	-
F	-1	-
G	-1	-

//assume -1 above is + ifinity
Repeat |v| -1 times:
Re initialize the queue
Enqueue the node from which distance is to be found
While the priority queue is not empty
V=dequeue
For each w adjacent to v
If distance[w]> new distance
Distance[w]=new distance
Path[w]=v
Enqueue(w)
At the end delete the queue

Spanning tree

The Spanning tree of a graph is a subgraph that contains all the vertices and is also a tree. A graph may have many spanning trees.

Minimum spanning tree

The spanning tree of a graph formed with minimum weights. (the shortest way in a map)

Algo

Prim's

Implementation is equal to Dijkstra's but instead of returning a table we should return tree

Kruskal's

S=fie //initialize a set to be empty
Make |v| set for |V| nodes
Sort edges by increasing weights
For each edge (u,v)
If find(u)!= Find(V)
S=s U {u,v}
Union(u,v)

Stack

29 December 2020 07:00 AM

Expression evaluation

- Infix - A+B
- Prefix - +AB -polish
- Postfix - AB+ -reverse polish
  - With use of prefix and postfix brackets can be eliminated

Introduction

23 February 2021 03:02 AM

Sorting : an algorithm that arranges the list of elements in a particular order (either ascending or descending), giving a permutation of input as the output.

Sorting one of the most important algorithms as it reduces the complexity of other problems on account of the data being sorted.

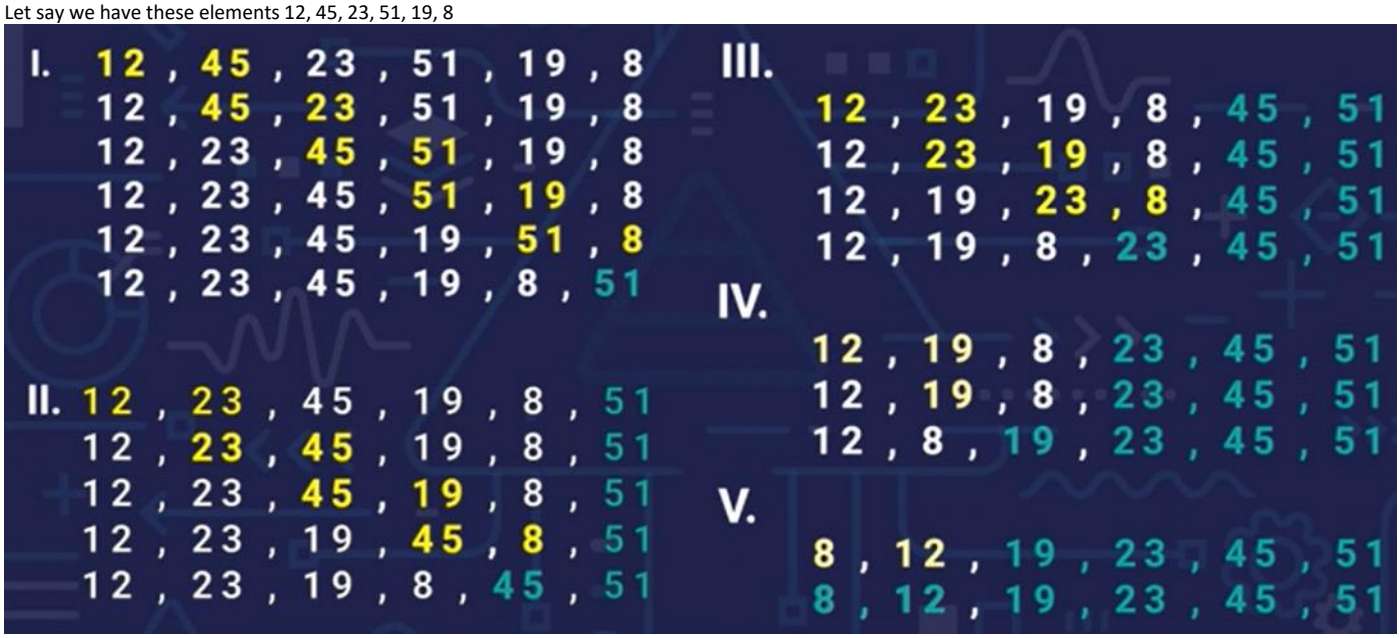
Classification of sorting algorithms

Based on		Remarks
Number of comparisons		
	O(nlogn)	
	$O(n^2)$	
Number of swaps(inversions)		
Memory usage		
	In place	Needs O(1) or O(log(n)) space complexity
Recursion		
	Recursive	
	Non-recursive	
Stability		
	stable	Same element does not change place with respect to their keys.
	unstable	Same element may change place with respect to their keys.
Adaptability		Some algorithms take advantage of the degree of pre-sorted-ness these are called adaptive.
Internal/external		
	Internal	Uses the main memory exclusively during sort; high speed
	external	Uses external memory eg hard drive etc.

Bubble Sort

23 February 2021 01:58 PM

- Repeatedly swap two adjacent element if they are in wrong order.
- Let say we have these elements 12, 45, 23, 51, 19, 8



- In the ith iteration we check the first n - i elements.
- At any time the first part (specifically n -i) of array is unsorted and the last part that is (i elements) are sorted.
- The sorted part is increasing as iteration increases.

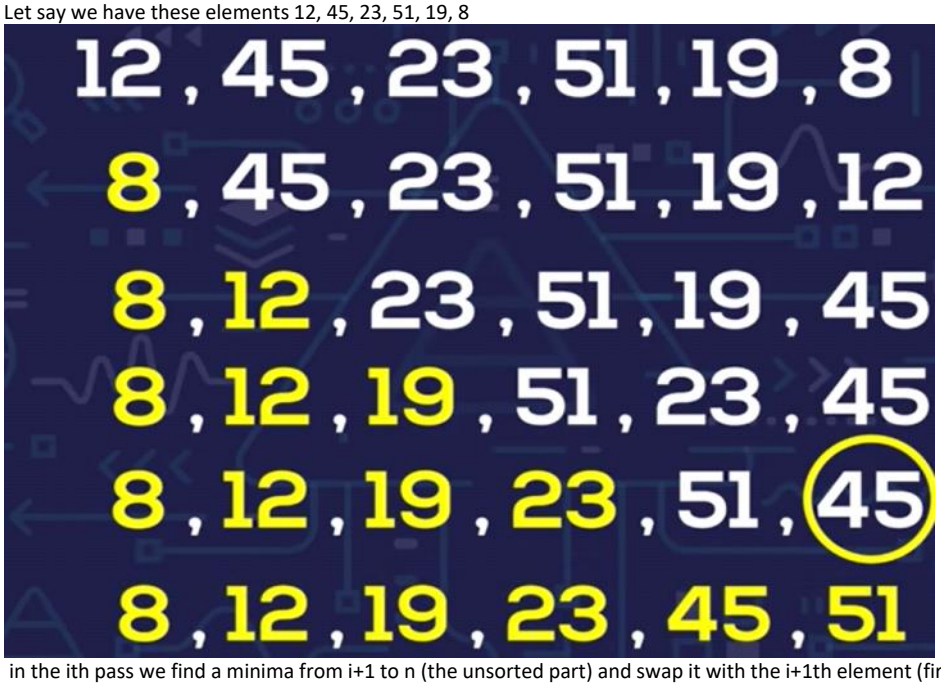
Code:

```
for (size_t i = 0; i < data size -1; i++)// the loop goes data size -1
times
//since we start from 0 and we need to iterate until the second last
//data, last data dont have any thing to compare with  actually
compared
    {
        for (size_t j = 0; j < data size -i; j++)
        {
            if(data[j]>data[j+1]){
                Swap the two ;
            }
        }
    }
```

Selection sort

24 February 2021 07:02 PM

- Find the minimum element in the unsorted part of array and swap it with the beginning of the unsorted part.
- Let say we have these elements 12, 45, 23, 51, 19, 8



- in the ith pass we find a minima from i+1 to n (the unsorted part) and swap it with the i+1th element (first element of unsorted part)
- The element till i will be sorted and rest unsorted.

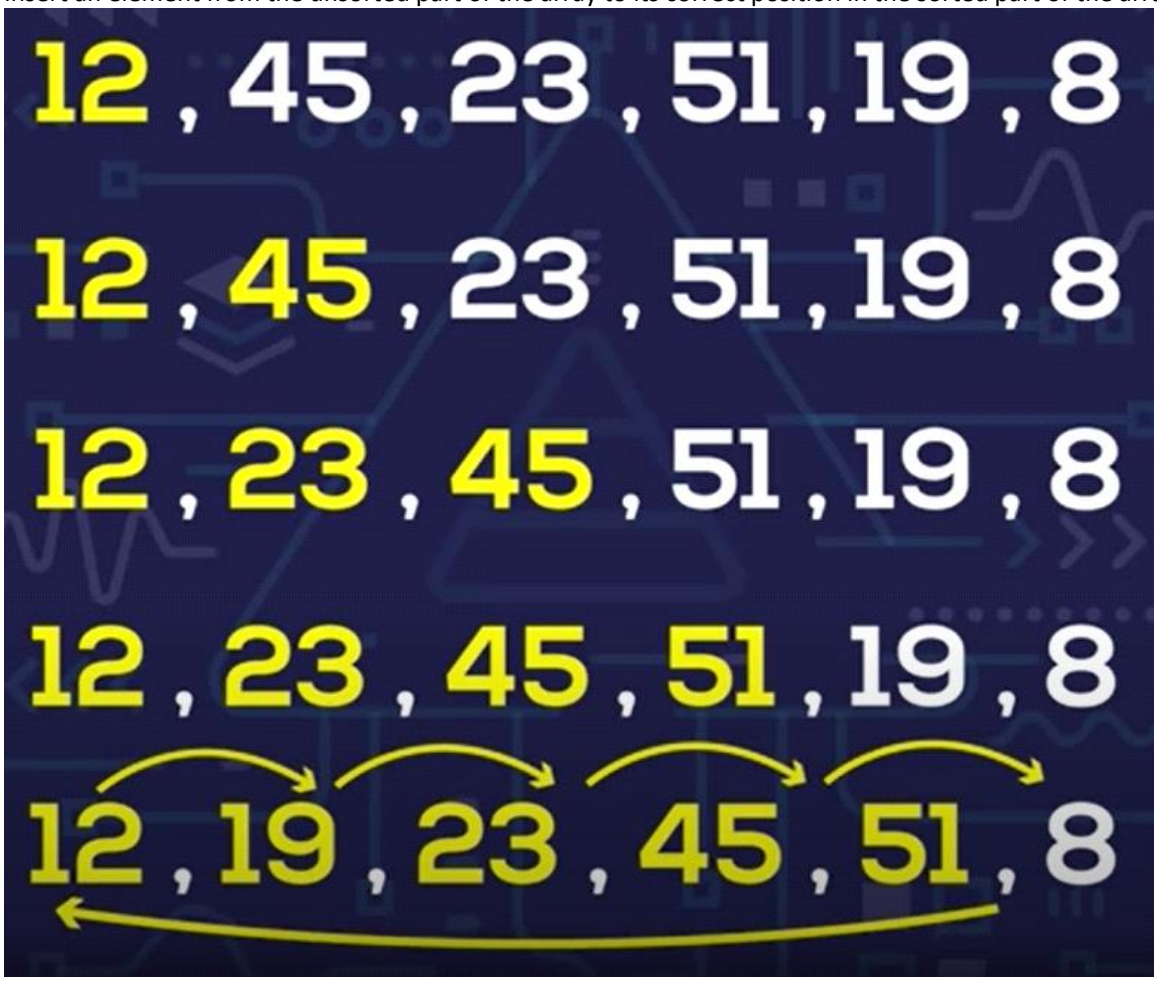
Code:

```
for (size_t i = 0; i < data size-1; i++)//this loop goes data size -1 times because ate the end we find minima from amongst at least one element array
    {
        //find minima
        min=i;
        for (size_t j = i+1; j < data size; j++)
        {
            if(data[j]<data[min]){
                min=j;
            }
        }
        //swap
        temp=data[i];
        data[i]=data[min];
        data[min]=temp;
        print(data);
    }
```

Insertion Sort

21 January 2021 11:12 AM

- Insert an element from the unsorted part of the array to its correct position in the sorted part of the array.



- Grab the first element from unsorted part Go back from the last of sorted part until the grabbed element is larger than the element in sorted part
  - While going advance the position of each element one step
  - At the point where the element in sorted part is less than grabbed one place it there.

Code:

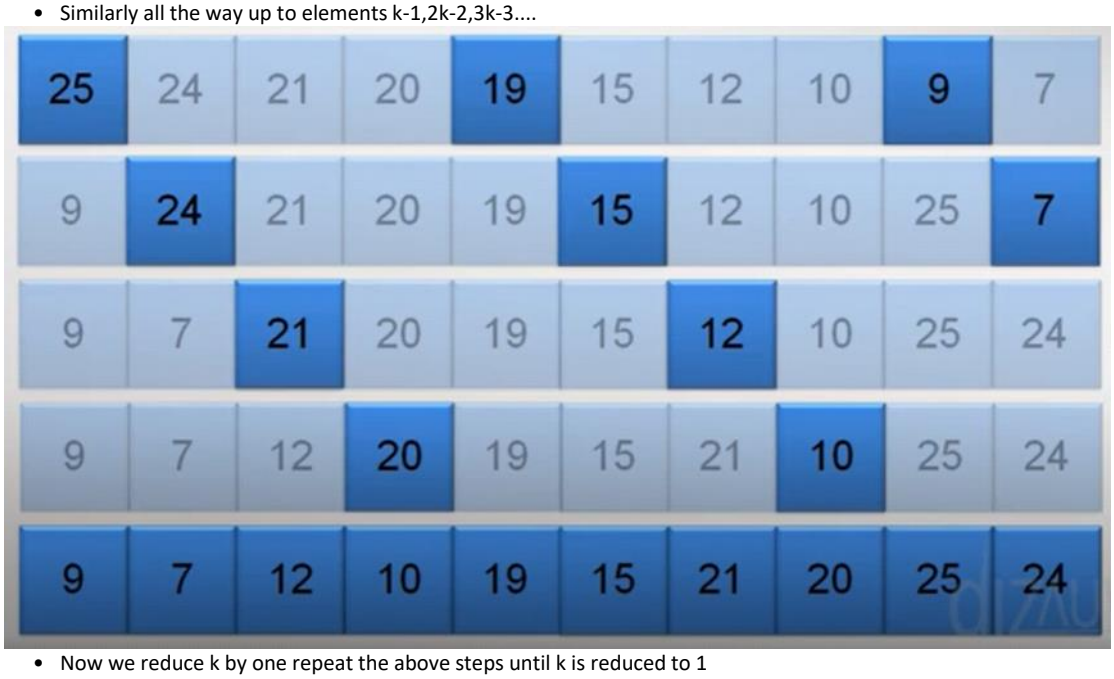
```
for (size_t i = 1; i < data size; i++)
    {
        temp=data[i];//grabbing part
        int j;
        for (j = i-1; j >=0; j--)
        {
            if(data[j]>temp){//shifting elements ahead
                data[j+1]=data[j];
            }else{//got the position
                break;
            }
```

```
        }
        data[j+1]=temp;
    }
```

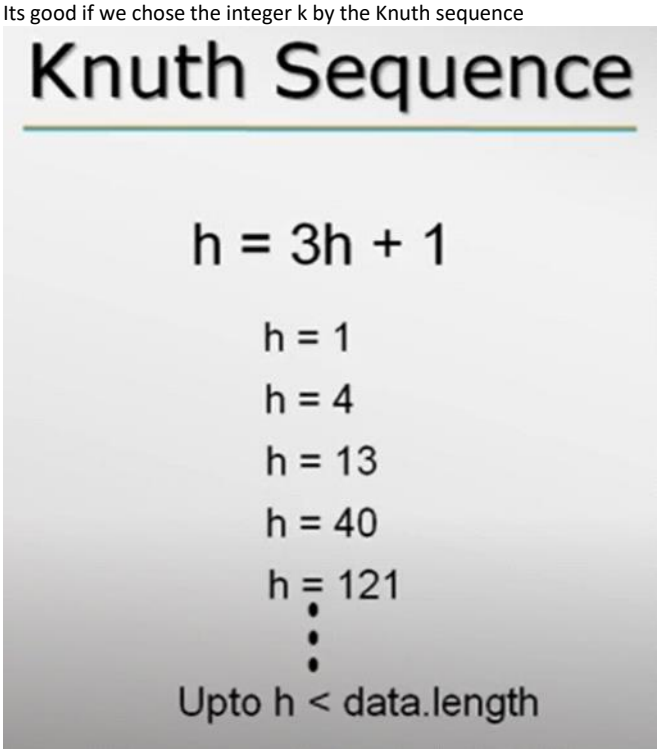
Shell sort

25 February 2021 04:21 PM

- If in a given array of elements the smaller elements are far away from the beginning the insertion sort performs less efficiently as a lot of element shifting takes place the shell sort solves that.
- Here we select an integer say k
- Then we consider elements 0, k, 2k,3k... Independently and sort them with insertion sort
- Then we repeat the same with elements 1, k+1,2k+1,...
- Similarly all the way up to elements k-1,2k-2,3k-3....



- Now we reduce k by one repeat the above steps until k is reduced to 1
- At this point the array is completely sorted.
- Its good if we chose the integer k by the Knuth sequence



- So if we have array size 100 we choose k as 40

Code:

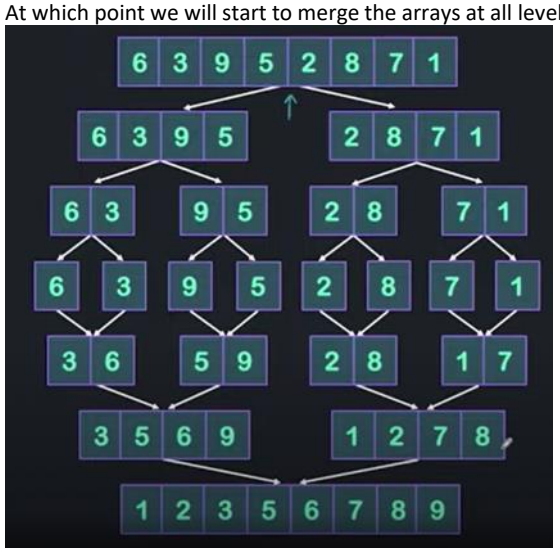
```
while (k != 0) {// k is the assumed integer decrementing k here
        for (size_t m = 0; m < k; m++) {//m is the first element of
//array considered in each pass
```

```
            for (size_t i = m; i < n; i += k) {
                temp = data[i];
                int j;
                for (j = i - k; j >= 0; j -= k) {
                    if (data[j] > temp) {
                        data[j + k] = data[j];
                    } else {
                        break;
                    }
                }
                data[j + k] = temp;
            }
        }
        k--;
        print(data);
    }
```

Merge Sort

21 January 2021 11:54 AM

- This algorithm is based on divide and conquer technique
- Her we recursively divide the array until there is one element left
- At which point we will start to merge the arrays at all levels of recursion



- We take two sorted arrays and form one sorted array in merge procedure as follows

- Compare the two pointed values which ever is smaller put in original array and increment the pointer

Code:

```
void merge(vector<int> &data, int l, int mid, int r) {
    int a = 0;
    int b = 0;
    int i=l;
    vector<int> temp(data.begin() + l, data.begin() + mid+1),
        temp1(data.begin() + mid + 1, data.begin() + r+1);
    while (a <temp.size()&&b<temp1.size()) {
        if (temp[a] < temp1[b]) {
            data[i]=temp[a];
            a++;
        } else {
            data[i]=temp1[b];
            b++;
```

```
            }
            i++;
        }
        if (a < temp.size()) {// if some elements are still left all
//elements of temp1 are copied to main array
            while (a <temp.size()) {
                data[i]=temp[a];
                a++;
                i++;
            }
        }
        if (b <temp1.size()) {// if some elements are still left all
//elements of temp are copied to main array
            while (b <temp1.size()) {
                data[i]=temp1[b];
                b++;
                i++;
            }
        }
}
void mergeSort(vector<int> &data, int l, int r) {
    if (l < r) {
        int mid = (l + r) / 2;
        mergeSort(data, l, mid);
        mergeSort(data, mid + 1, r);
        merge(data, l, mid, r);
    }
}

Then just call
mergeSort(data, 0, data.size() - 1);
From main
```

Heap sort

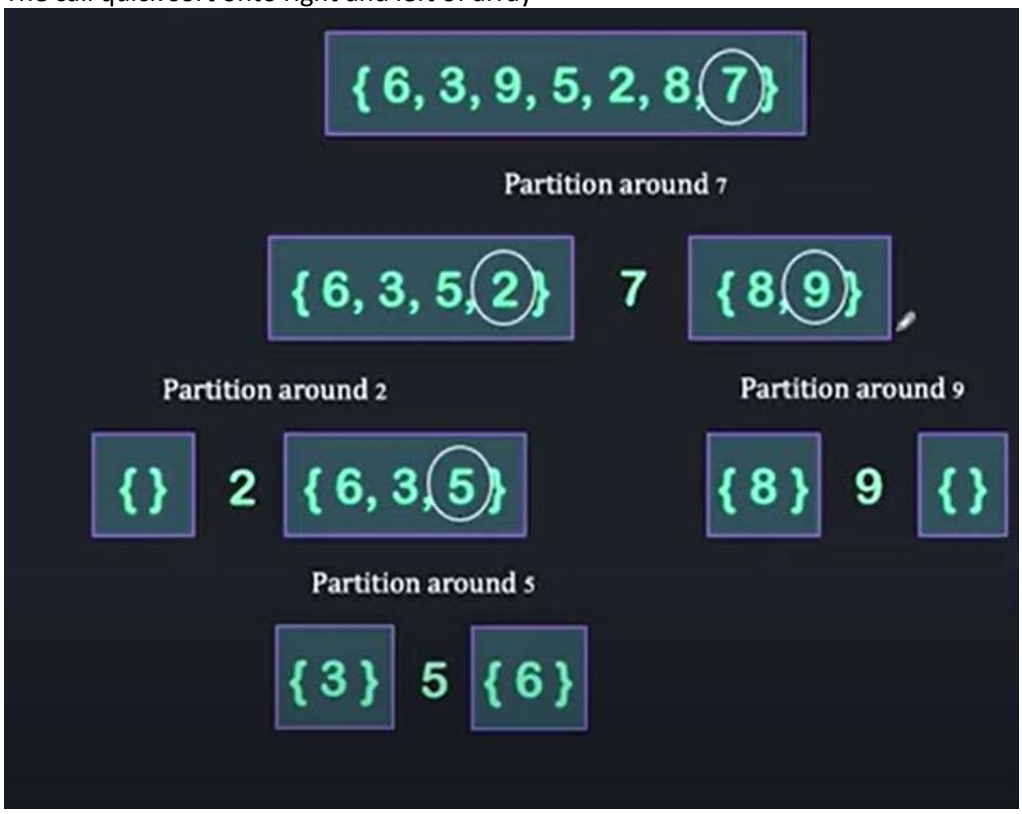
25 February 2021 11:29 PM

- Make a priority queue
- Take all elements one by one

Quick sort

25 February 2021 11:30 PM

- This algorithm is based on divide and conquer
- We randomly chose a pivot
- Then place the pivot at right position leaving the element to left and right of it (smaller and greater respectively than pivot) un sorted.
- The call quick sort onto right and left of array



- The partition function here is of great significance and a bit tricky
  - The partition function is responsible for
    - choosing pivot
    - Placing it at correct position
    - Returning that position

    Here we always chose the last element of array as pivot

```
Partition(arr[], l, r){

   pivot = arr[r];
   i = l - 1
   for j=l to r-1
        if arr[j] < pivot
             i++;
             swap(i,j)

   swap(i+1, r)
   return i+1

}
```

  - Here we are actually accumulating the elements larger than pivot at the end do it over the data for better

understanding

Code:

```
int partition(vector<int> &data, int l, int r) {
    int pi = data[r];
    int i = l - 1, temp;
    for (size_t j = l; j < r; j++) {
        if (data[j] < pi) {
            i++;
            temp = data[i];
            data[i] = data[j];
            data[j] = temp;
        }
    }
    i++;
    temp = data[i];
    data[i] = data[r];
    data[r] = temp;
    return i;
}
void quickSort(vector<int> &data, int l, int r) {
    if (l < r) {
        int pi_ind = partition(data, l, r);
        quickSort(data, l, pi_ind - 1);
        quickSort(data, pi_ind + 1, r);
    }
}
```

Then just call

```
quickSort(data, 0, data.size() - 1);
```

From main

Tree sort

26 February 2021 12:04 AM

- Insert all elements in BST
- Do an inorder traversal

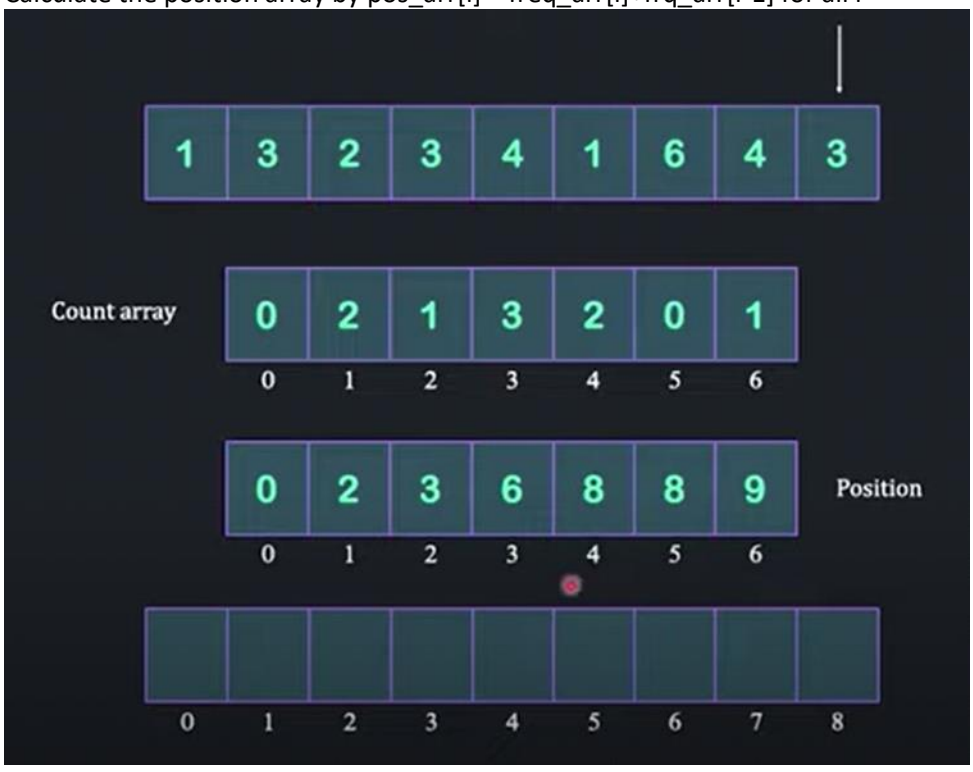
Comparison

26 February 2021 12:05 AM

Name	Average Case	Worst Case	Auxiliary Memory	Is Stable?	Other Notes
Bubble	$O(n^2)$	$O(n^2)$	1	yes	Small code
Selection	$O(n^2)$	$O(n^2)$	1	no	Stability depends on the implementation.
Insertion	$O(n^2)$	$O(n^2)$	1	yes	Average case is also $O(n + d)$, where d is the number of inversions.
Shell	-	$O(nlog^2n)$	1	no	
Merge sort	$O(nlogn)$	$O(nlogn)$	depends	yes	
Heap sort	$O(nlogn)$	$O(nlogn)$	1	no	
Quick sort	$O(nlogn)$	$O(n^2)$	$O(logn)$	depends	Can be implemented as a stable sort depending on how the pivot is handled.
Tree sort	$O(nlogn)$	$O(n^2)$	$O(n)$	depends	Can be implemented as a stable sort.

Count sort

26 February 2021 12:21 AM

- Store the frequency of every element in the array
- Calculate the position array by pos_arr[i] = freq_arr[i]+frq_arr[i-1] for all i
- 

  Travers the unsorted array.
- decrement the respective value at position array .
- store the value at decremented position in final array.

Code

```
vector<int> pos(max+1,0);
    for (size_t i = 0; i < n; i++)
    {
        pos[data[i]]++;
    }
    for (size_t i = 1; i <= max; i++)
    {
        pos[i]=pos[i-1]+pos[i];
    }
    vector<int> res(n,0);
    for (size_t i = 0; i < n; i++)
    {
        pos[data[i]]--;
        res[pos[data[i]]]=data[i];
    }
```

Bucket sort

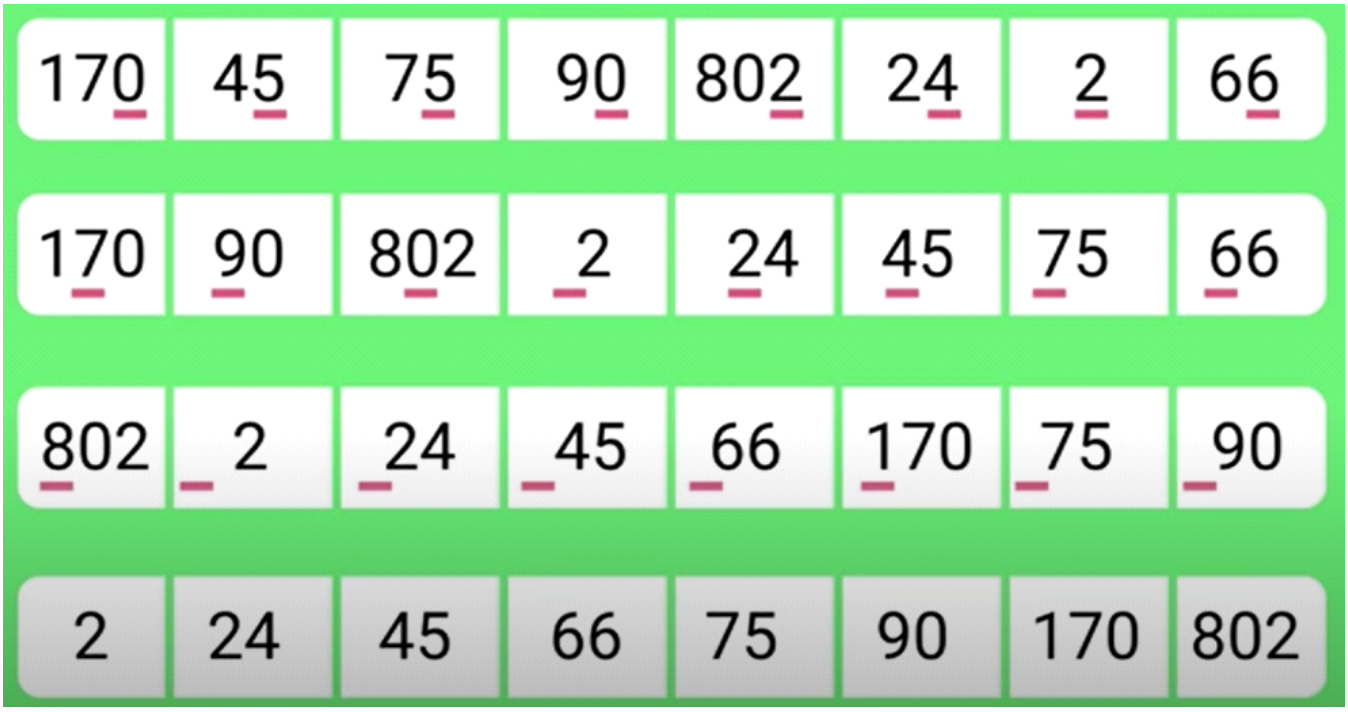
26 February 2021 12:43 AM

- First divide the elements into smaller buckets
    - Lets say elements with value falling from 0-9 goes bucket 1, from 10-19 goes to bucket 2 and so on.
    - Here 10 is divider
- Use some sorting algorithm to sort these buckets and the join them.
- The divider is chosen considering the range of elements to be sorted
- The element belong to which bucket is calculated mathematically by floor(element/divider).

Radix sort

26 February 2021 12:54 AM

- The algorithm compares the LS digit of all numbers in decimal format
- Sorts it according to LS digit
- Then the it compares the digit before LSB and sorts according to that and so on until MS digit is reached



- Repeat no of digits times
  - put the number with kth current digit to kth bucket
  - Repeat no. of bucket times
    - Take the number put first(from bottom) in buckets i then next from bucket I
    - And place them in array

Topological sort

26 February 2021 03:10 AM

In graphs

External sorting

26 February 2021 03:10 AM

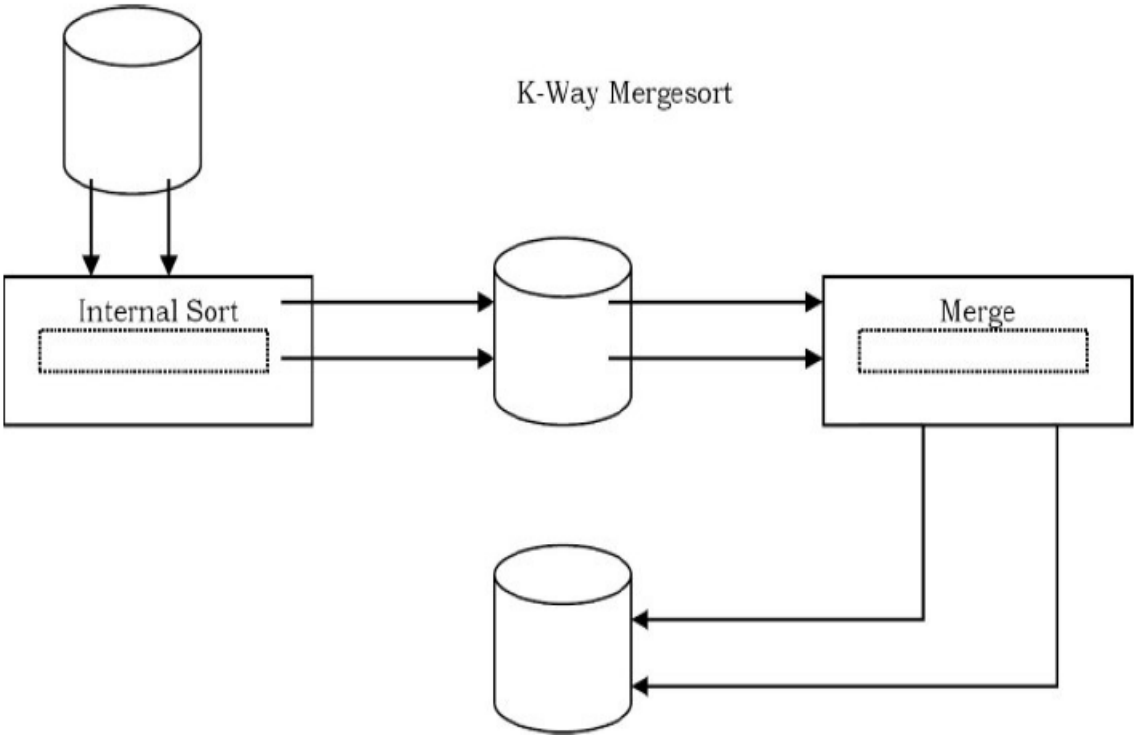
External sorting is a generic term for a class of sorting algorithms that can handle massive amounts of data.

Simple External Merge sort

A number of records from each tape are read into main memory, sorted using an internal sort, and
then output to the tape. For the sake of clarity, let us assume that 900 megabytes of data needs to
be sorted using only 100 megabytes of RAM.

1. Read 100MB of the data into main memory and sort by some conventional method (let us say Quick sort).
2. Write the sorted data to disk.
3. Repeat steps 1 and 2 until all of the data is sorted in chunks of 100MB. Now we need to merge them into one single sorted output file.
4. Read the first 10MB of each sorted chunk (call them input buffers) in main memory (90MB total) and allocate the remaining 10MB for output buffer.
5. Perform a 9-way Mergesort and store the result in the output buffer. If the output buffer is full, write it to the final sorted file. If any of the 9 input buffers gets empty, fill it with the next 10MB of its associated 100MB sorted chunk; or if there is no more data in the sorted chunk, mark it as exhausted and do not use it for merging. The

This can be generalized in k way a follows

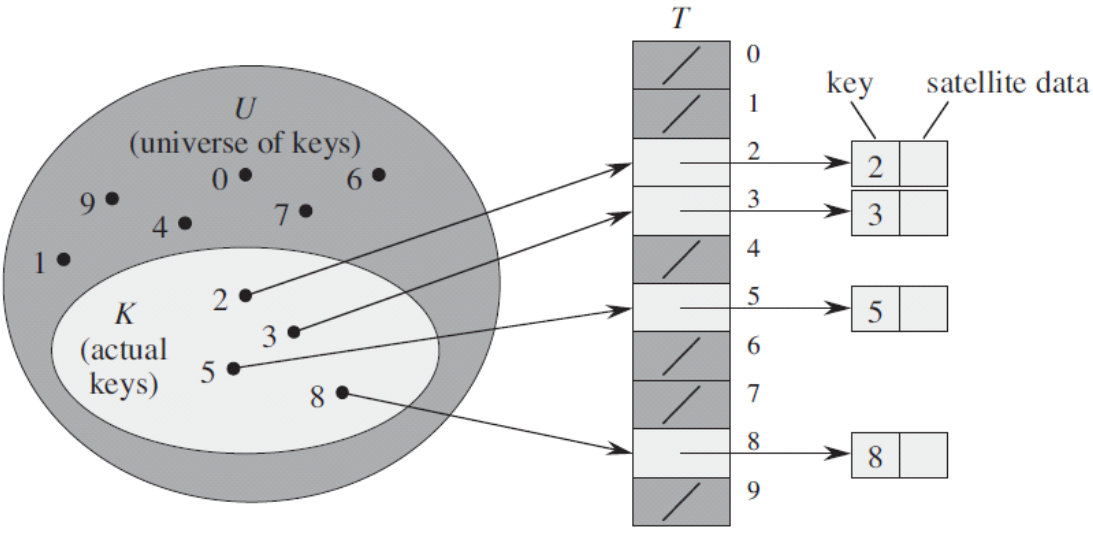


Basix

26 February 2021 09:27 AM

Why do we need it

- In an array we can access, store, delete elements in constant time.
- We just need to call index of array to do so.
- There memory address is directly calculated by mathematical function like {ref_address + index*sizeof(object)}
- The problem with array is we have fixed indexing
    - So if we want to store the count of occurrence of a set of numbers.
    - We need to create an array of size maximum of the numbers in the set.(because we need count[max])
    - Hence wasting a lot of space.(imagine the set has 5 elements {1,50,1,50,10000})
- This problem is solved by hashing.
- What hashing does is calculates the index based on the element
    - So in an array we would be calling count[10000]
    - In hash map/hash table we would be calling count[h(10000)]
    - This function 'h' is hash function and is responsible for calculating index.
        - This function takes into account the no of keys etc. and takes considerably small space.
- Hashing is efficient when possible no of keys (elements to be stored) are a lot more and the number of key actually stored is less.

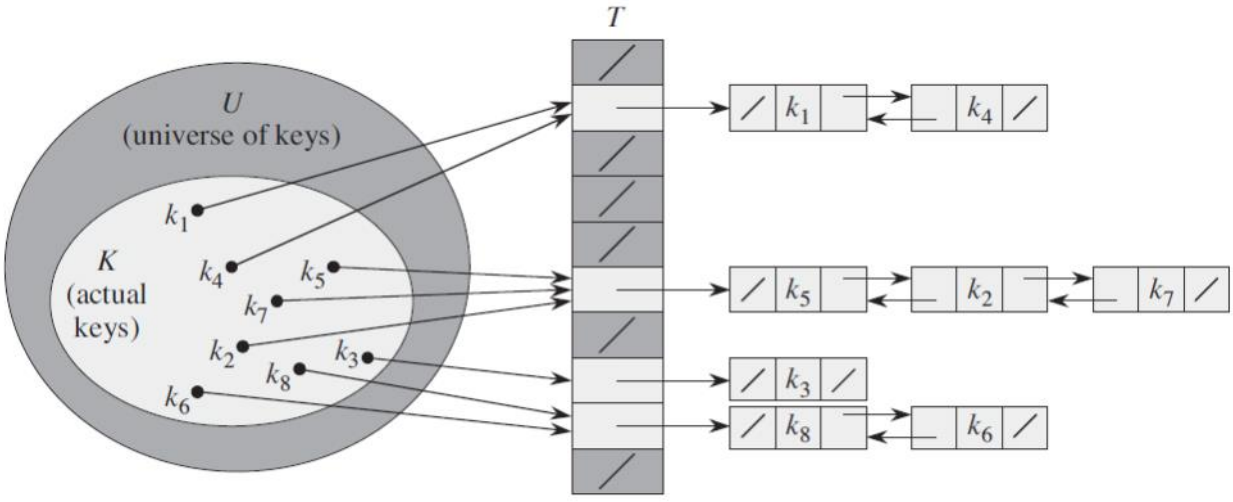


- Hashing proves good even when keys are more in number.

Terminology

26 February 2021 03:00 PM

- Hash table/map - a data structure which implements hashing, basically it is a generalisation of array where we directly store the value in the index of 'key', in a hash table we store the value in the index of 'h(key)'.
- Hash function - it is a mathematical function which spits out the index of memory location when given the input of key, h(key)=index
- Collision- an event where the hash function gives the same value of index with 2 or more different inputs of keys.
- Chaining - a technique of collision resolution where the keys with same index are stored as a linked list.



Components of hashing

The following are the main components of hashing concepts

- Hash table - defined above
- Hash functions - devoted another section hash functions
- Collisions - defined above
- Collision resolution techniques - devoted another section collision resolution techniques

Hash functions

26 February 2021 03:13 PM

- The goals of a hash function are as follows. (properties of good hash function)
  - Minimize collision
  - Be easy and quick to compute
  - Distribute key values evenly in the hash table
  - Use all the information provided in the key
  - Have a high load factor for a given set of keys
- a hash function that maps each item into a unique slot is referred to as a perfect hash function.
- Perfect hash functions are not practically feasible.
- Load factor = $\frac{total\ numer\ of\ keys\ hashed}{the\ number\ of\ index\ in\ hash\ table} = \frac{number\ of\ elements\ in\ hash\ table}{hash\ table\ size}$
- This load factor gives a degree of uniformity of distribution of keys over the index

Collision resolution techniques

26 February 2021 03:40 PM

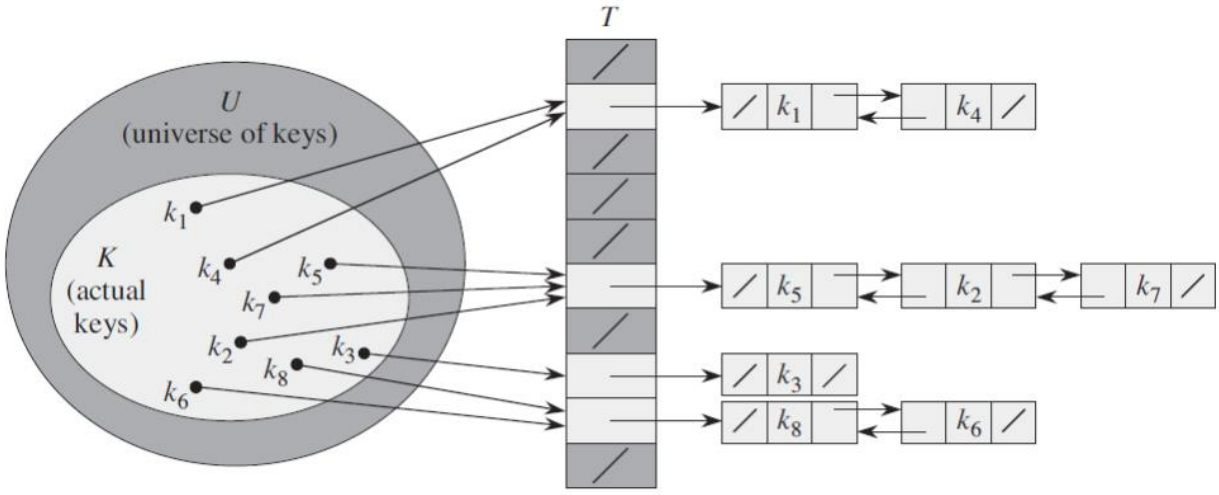
The most popular collision resolution techniques are

- Direct Chaining: An array of linked list application
  - Separate chaining
- Open Addressing: Array-based implementation
  - Linear probing (linear search)
  - Quadratic probing (nonlinear search)
  - Double hashing (use two hash functions)

Separate chaining

26 February 2021 03:43 PM

When two or more records hash to the same location, these records are constituted into a singly-linked list
called a chain.



Open addressing

26 February 2021 03:45 PM

Intro

10 March 2021 11:05 AM

Tasks

17 December 2020 07:51 AM

Implement threaded trees
Postfix -> expression tree

Implement all adt

Left outs:
Xor tree

Address calculation

12 March 2021 06:32 PM

Assume a two dimensional matrix whose row index starts at r_f and ends at r_l
And whose column index starts at c_f and ends at c_l

M[(rf-rl+1),(cf-cl+1)]

Address of M[i,j]

If row major is

(Cl-cf+1)(i-rf)+(j-cf)

If column major

(rl-rf+1)(j-cf)+(i-rf)