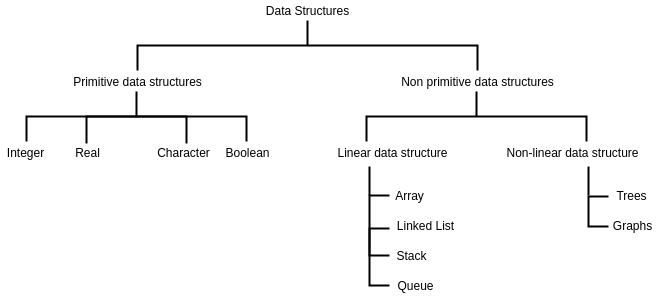
# **Categories of Data Structures**



# 1.Array

Array is a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

## 

## 1.1 Array Creation

| Assume a[n]  Print “enter no of elements”  Read n for(i=0;i<n;i++)  {  Print “ enter the element”  Read a[i] } |
| --- |

## 1.2 Array Insertion

## 

| Assume a[n],fai \\fai - Filled Array Index (index of the last element of Array) Read pos if(fai<n-1)  { Read ele for(i=fai;i>pos-1;i--)  {  a[i+1]=a[i] } a[pos-1]=ele; } |
| --- |

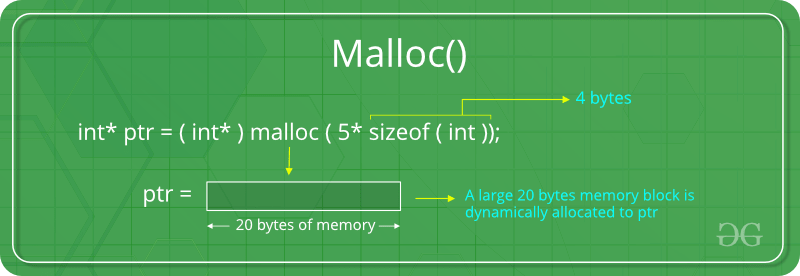
## 1.3 Array Deletion

| assume a[n], fai, pos, ele, size = n.  if( fai > -1) \\ element is present or not  {  read pos, ele.  if(pos-1<= fai) \\position is under limit  {  for(i=pos-1; i< fai; i++)  {  a[i]=a[i+1]  }  fai- -  }  else print (Invalid Position)  }  else print (No element ) |
| --- |

**1.3 Array binary search**

| assume a[10], l=0, u=9, mid, ele.  read ele \\ to be find  while(l<=u)  {  mid=(l+u)/2  if(ele>a[mid])  l=mid+1  else if(ele<a[mid])  u=mid-1  else  {  print (element found)  break  }  } |
| --- |

**1.4 How to dynamically allocate a 1D array in C?**

****

| #include <stdio.h>  #include <stdlib.h>    int main()  {     // This pointer will hold the   // base address of the block created   int\* ptr;   int n, i, sum = 0;     // Get the number of elements for the array   n = 5;   printf("Enter number of elements: %d\n", n);     // Dynamically allocate memory using malloc()   ptr = (int\*)malloc(n \* sizeof(int));     // Check if the memory has been successfully   // allocated by malloc or not   if (ptr == NULL) {   printf("Memory not allocated.\n");   exit(0);   }   else {     // Memory has been successfully allocated   printf("Memory successfully allocated using malloc.\n");     // Get the elements of the array   for (i = 0; i < n; ++i) {   ptr[i] = i + 1;   }     // Print the elements of the array   printf("The elements of the array are: ");   for (i = 0; i < n; ++i) {   printf("%d, ", ptr[i]);   }   }     return 0;  }  Output: Enter number of elements: 5 Memory successfully allocated using malloc. The elements of the array are: 1, 2, 3, 4, 5, |
| --- |

**To know more:-** [**https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/**](https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/)

## 1.4 How to dynamically allocate a 2D array in C?

Following are different ways to create a 2D array on heap (or dynamically allocate a 2D array).

In the following examples, we have considered ‘r‘ as number of rows, ‘c‘ as number of columns and we created a 2D array with r = 3, c = 4 and following values

1 2 3 4

5 6 7 8

9 10 11 12

**1) Using a single pointer:**

A simple way is to allocate memory block of size r\*c and access elements using simple pointer arithmetic.

| #include <stdio.h> #include <stdlib.h>  int main() {  int r = 3, c = 4;  int \* arr = (int \* ) malloc(r \* c \* sizeof(int));   int i, j, count = 0;  for (i = 0; i < r; i++)  for (j = 0; j < c; j++)  \*(arr + i \* c + j) = ++count;   for (i = 0; i < r; i++)  for (j = 0; j < c; j++)  printf("%d ", \*(arr + i \* c + j));   /\* Code for further processing and free the dynamically allocated memory \*/  return 0; } Output: 1 2 3 4 5 6 7 8 9 10 11 12 |
| --- |

**2) Using an array of pointers**

We can create an array of pointers of size r. After creating an array of pointers, we can dynamically allocate memory for every row.

| #include <stdio.h>  #include <stdlib.h>    int main()  {   int r = 3, c = 4, i, j, count;     int \*arr[r];   for (i=0; i<r; i++)   arr[i] = (int \*)malloc(c \* sizeof(int));     // Note that arr[i][j] is same as \*(\*(arr+i)+j)   count = 0;   for (i = 0; i < r; i++)   for (j = 0; j < c; j++)   arr[i][j] = ++count; // Or \*(\*(arr+i)+j) = ++count     for (i = 0; i < r; i++)   for (j = 0; j < c; j++)   printf("%d ", arr[i][j]);     /\* Code for further processing and free the   dynamically allocated memory \*/    return 0;  }  Output:  1 2 3 4 5 6 7 8 9 10 11 12 |
| --- |

**3) Using pointer to a pointer**

We can create an array of pointers also dynamically using a double pointer. Once we have an array of pointers allocated dynamically, we can dynamically allocate memory and for every row like method 2.

filter\_none

edit

play\_arrow

Brightness\_4

| #include <stdio.h>  #include <stdlib.h>    int main()  {   int r = 3, c = 4, i, j, count;     int \*\*arr = (int \*\*)malloc(r \* sizeof(int \*));   for (i=0; i<r; i++)   arr[i] = (int \*)malloc(c \* sizeof(int));     // Note that arr[i][j] is same as \*(\*(arr+i)+j)   count = 0;   for (i = 0; i < r; i++)   for (j = 0; j < c; j++)   arr[i][j] = ++count; // OR \*(\*(arr+i)+j) = ++count     for (i = 0; i < r; i++)   for (j = 0; j < c; j++)   printf("%d ", arr[i][j]);     /\* Code for further processing and free the   dynamically allocated memory \*/    return 0;  }  Output:  1 2 3 4 5 6 7 8 9 10 11 12 |
| --- |

**4) Using double pointer and one malloc call**

filter\_none

edit

play\_arrow

Brightness\_4

| #include<stdio.h>  #include<stdlib.h>    int main()  {   int r=3, c=4, len=0;   int \*ptr, \*\*arr;   int count = 0,i,j;     len = sizeof(int \*) \* r + sizeof(int) \* c \* r;   arr = (int \*\*)malloc(len);     // ptr is now pointing to the first element in of 2D array   ptr = (int \*)(arr + r);     // for loop to point rows pointer to appropriate location in 2D array   for(i = 0; i < r; i++)   arr[i] = (ptr + c \* i);     for (i = 0; i < r; i++)   for (j = 0; j < c; j++)   arr[i][j] = ++count; // OR \*(\*(arr+i)+j) = ++count     for (i = 0; i < r; i++)   for (j = 0; j < c; j++)   printf("%d ", arr[i][j]);     return 0;  }  Output:  1 2 3 4 5 6 7 8 9 10 11 12 |
| --- |

To know more:- <https://www.geeksforgeeks.org/dynamically-allocate-2d-array-c/>

# 

# 

# 

# 2. Linked list

## 2.1 Single Linked List

In computer science, a **linked list** is a linear collection of data elements, called nodes, each pointing to the next node by means of a pointer.

****

### 2.1.1 Linked list creation

| #include <stdio.h>  #include <stdlib.h>  int main()  {  struct node  {  int data;  struct node\*next;  };  struct node \*head;  head=0;  struct node \*newnode,\*temp;  int choice;  while (choice!=0)  {  printf("Enter data?(1/0)=");  scanf("%d",&choice);  newnode=(struct node\*)malloc(sizeof(struct node));  printf("\n\n Enter data=");  scanf("%d",&newnode->data);  newnode->next=0;    if(head==0)  {  head=temp=newnode;  }  else  {  temp->next=newnode;  temp=newnode;  }  }  //printing(traversal)  temp=head;  while(temp->next!=0)  {  printf("\ndata=%d",temp->data);  temp=temp->next;  }  return 0;  } |
| --- |

### 2.1.2 Linked list insertion

## 

| struct node {  int data. struct node \*ptr. }\*start == NULL;    assume \*start, \*last  t = (struct node\*) malloc( sizeof (struct node) ) .  read t -> data.  read pos.  if (pos == 1)  {  t -> ptr = start.  start = t.  }  else  {  t1 = start  for (i=2; i<pos && t1->ptr! = NULL; i++)  {  t1 = t1 -> ptr.  }  t -> ptr = t1 -> ptr.  t1 -> prt = t.  } |
| --- |

### 2.1.3 Linked list deletion

****

| struct node {  int data. struct node \*ptr. }\*start = NULL.    read pos.  if (pos == 1)  {  t = start.  start = start->ptr.  free (t).  }  else  {  t = t1 = start.  for ( i=1; i< pos; i++)  {  t1 = t.  t = t->ptr.  if (t == NULL)  {  f=1.  break.  }  }  if (f != 1)  {  t1 -> ptr = t ->ptr.  }  else  {  print (Error)  } |
| --- |

## 2.2 Circular Linked List.

In a **circularly linked list**, all nodes are **linked** in a continuous circle, without using null.The next node after the last node is the first node.

****

### 2.2.1 Circular LL creation.

| struct node {  int data. struct node \*ptr. }\*start = NULL.    declare struct node \*t, \*last.  read n.  for ( i = 1 to n )  {  t = (struct node\*) malloc( sizeof (struct node) ) .  read t -> data.  if (start == NULL)  {  start = t.  }  else  {  last -> ptr = t.  }  t -> ptr = start.  last = start.  } |
| --- |

### 

### 2.2.2 Circular LL Traversing

| struct node {  int data.  struct node \*ptr.  }\*start = NULL. 1. assume \*start. 2. declare struct node \*t. 3. for (t=start; start != NULL; t = t -> ptr) 4. { 5. print ( t -> data) 6. if (t -> ptr = start) 7. break 8. } |
| --- |

### 2.2.3 Circular LL insertion.

****

| struct node {  int data. struct node \*ptr. }\*start = NULL.  1. assume \*start, \*last. 2. declare struct node \*t, \*t1. 3. read pos. 4. t = (struct node\*) malloc( sizeof (struct node) ) . 5. read t-> data. 6. if (pos == 1) 7. { 8. t -> ptr = start. 9. start =t. 10. last -> ptr = t. 11. } 12. else 13. { 14. t1 = start. 15. for (i=2; i<pos && t1 -> ptr !=start; i++) 16. { 17. t1 = t1 -> ptr. 18. t1 -> ptr = t. 19. } } |
| --- |

### 2.2.4 Circular LL deletion

****

| struct node {  int data. struct node \*ptr. }\*start = NULL. 1. assume \*start, \*last. 2. declare struct node \*t, \*t1. 3. if (start != NULL) 4. { 5. read pos. 6. if (pos == 1) 7. { 8. t = start. 9. start = start -> ptr. 10. free (t). 11. last -> ptr = start. 12. } 13. else 14. { 15. t = start. 16. for (i=1; i < pos; i++) 17. { 18. t1 = t. 19. t = t -> ptr. 20. } 21. t1 -> ptr = t -> ptr. 22. free (t). 23. } |
| --- |

## 2.3 Doubly Linked List

A **doubly linked list** is a [linked data structure](https://en.wikipedia.org/wiki/Linked_data_structure) that consists of a set of sequentially linked [records](https://en.wikipedia.org/wiki/Record_(computer_science)) called [nodes](https://en.wikipedia.org/wiki/Node_(computer_science)). Each node contains two [fields](https://en.wikipedia.org/wiki/Field_(computer_science)), called *links*, that are [references](https://en.wikipedia.org/wiki/Reference_(computer_science)) to the previous and the next node in the sequence of nodes. The beginning and ending nodes' **previous** and **next** links, respectively, point to NULL.

### 2.3.1 Doubly linked list creation.

****

| struct node { int data; struct node \*pre, \*next; }\*start = NULL.  declare struct node \*t, \*last.  read n.  for (i = 1 to n)  {  t = (struct node\*) malloc( sizeof (struct node) )  read t -> data.  t -> next = NULL.  if ( start == NULL)  {  start = t.  t -> pre = NULL.  }  else  {  last -> next = t.  t -> pre = last.  } last = t. } |
| --- |

### 2.3.2 Doubly LL insertion.

****

| assume \*start, \*last. Declare struct node \*t, \*t1. read pos. t = (struct node\*) malloc( sizeof (struct node) ) read t -> data. If ( pos == 1) { t -> next = start. start -> pre = t. start = t. t -> pre = NULL. } else { t1 = start. for (i=2; i<pos; i++) { t1=t1->next; } t -> next = t1 -> next. t -> pre = t1. if( t1 ->next!= NULL) { (t1->next)->pre=t } t1->next=t |
| --- |

### 2.3.3 Doubly LL deletion.

| 1. assume a DLL \*start, \*last. 2. Declare struct node \*t, \*t1. 3. if( start!=NULL) 4. { 5. Read pos 6. If(pos==1) 7. { 8. t=start. 9. start=start->next 10. free(t). 11. } 12. else 13. { 14. t=t1=start. 15. for( i=1; i<pos; i++) 16. { 17. t1=t 18. t=t->next. 19. } 20. t1->next=t->next. 21. if( t->next != NULL) 22. { 23. (t->next)->pre=t1. 24. } 25. free(t) } |
| --- |

# 3. Stack

A **Stack** is an abstract data structure which is used to store data in a particular order. It is either implemented with the help of an array or a linked list. Two operations that can be performed on a **Stack** are: Push operation which inserts an element into the **stack**. Pop operation which removes the last element that was added into the **stack**.

It follows the Last In First Out(LIFO) Order.

## 3.1 Stack implementation by Array.

| Declare int stack[10],size=10,top=-1  push (int ele) {  if (top==size-1) print (overflow)   else  stack[++top]=ele. }   int pop() {  if(top==-1)  print(overflow) else  return stack[top--]  } |
| --- |

**3.2 Stack implementation by Linked list**

| struct node  {  int data  struct node \*ptr.  }\*top=NULL;   push( int ele)  {  t = (struct node\*) malloc( sizeof (struct node) )  if (t!=NULL) {   t->data=ele t->ptr=top  top= t  } else print (overflow) } |
| --- |

| int pop()  {  if(top!=NULL)  {  t=top.  top=top->ptr  ele =t->data  free(t)  return ele  } else  print underflow  } |
| --- |

## 3.2 Stack Applications

### 3.2.1 Parenthesis checking

| Declare stack[20],top=-1 // for stack implementation  void push(char ele) { stack[++top]=ele; }  char pop() { if(top==-1)  return '#'  else  return stack[top--] }  Declare FILE \*fp fp=fopen("demo.c","r") while((ch=fgetc(fp)!=EOF) { if(ch=='{') push('{') Else if (ch=='}')  {  ch1=pop();  if(ch1=='#')  Break;  } } if(ch=='#' || top!=-1) print("incorrect sequence") Else print("Correct sequence") |
| --- |

## 

### 3.2.2 Solving postfix expression

| Assume a postfix expression in p[]  while(p[i]!='\0')  {  if(p[i] is equal to ('+','-','/','\*'))  {  op2=pop();  op1=pop();  switch(p[i])  {  case '+' : push(op1+op2);  break.  case '-' : push(op1-op2);  break.  case '/' : push(op1/op2);  break.  case '\*' : push(op1\*op2);  break.  } }  else  push(p[i]-'0').  i++;  }  print (result = pop()). |
| --- |

### 3.2.3 Infix to Postfix

| Declare char p[ ]; char q[ ], ch; int ind=0 read p  push('#')  for(i=0;p[i]!=0;i++)  {  if(p[i]>='0' && p[i]<='9')  q[ind++]=p[i]  else if(p[i]=='(')   push('(')  else if(p[i]==')') {  ch=pop()  while(ch!='(')  {  q[ind++]=ch  ch=pop()  }  }  else  {  op=pop()  while(prio(op)>=prio(p[i]))  {  q[ind++]=op  op=pop()  }  push(op)  push(p[i])  }  }//end of for  ch=pop()  while(ch!='#')  {  q[ind++]=ch.  ch=pop()  }  q[ind]='\0' |
| --- |

| int prio(char ele) { if(ele=='+'|| ele=='-') return 2. else if(ele=='/'||ele=='\*') return 3. Else return 1. } |
| --- |

### 3.2.4 Infix to postfix conversion by stack (Stepwise solution)

| **[(3+5\*2)+8/2]** | **Stack (pop only priority is equal and high)** | **Q(array/output)** |
| --- | --- | --- |
| [ | [ |  |
| ( | [ ( |  |
| 3 | [ ( | 3 |
| + | [ ( + | 3 |
| 5 | [ ( + | 3 5 |
| \* | [ ( + \* | 3 5 |
| 2 | [ ( + \* | 3 5 2 |
| ) | [ ( | 3 5 2 \* + |
| + | [ + | 3 5 2 \* + |
| 8 | [ + | 3 5 2 \* + 8 |
| / | [ + / | 3 5 2 \* + 8 |
| 2 | [ + / | 3 5 2 \* + 8 2 |
| ] |  | 3 5 2 \* + 8 2 / + |

**Evaluation of postfix expression :**

| **2, 3, 5, \*, +, /, -** | **stack** | **Partial Result** |
| --- | --- | --- |
| 2 | 2 |  |
| 2, 3 | 2, 3 |  |
| 2, 3, 5 | 2, 3, 5 | Op1 (opr) Op2 |
| 2, 3, \* | 2, 15 | 3\*5=15 |
| 2, 3, \*, + | 17 | 2+15=17 |
| 2, 3, \*, +,/ | 17, 1 |  |
| 2, 3, \*, +, / , - | **16** | 17-1=16 |

### 3.2.5 Infix to prefix conversion

| declare char p[], char q[], int x read p x = strlen(p) push('#') for (i = x; i > 0; i--) {  if (p[i] >= '0' && p[i] <= '9')  q[ind++] = p[i]  else if (p[i] == ')')  push(')')  else if (p[i] == '(')  ch = pop();  while (ch != ')') {  q[ind++] = ch  ch = pop()  }  else {  op = pop()  while (prio(op) > prio(p[i])) {  q[ind++] = op  op = pop()  }  push(op)  push(p[i]) } } //end of for ch = pop() while (ch != '#') {  q[ind++] = ch.  ch = pop() } |
| --- |

| int prio(char ele) { if(ele=='+'|| ele=='-') return 2. else if(ele=='/'||ele=='\*') return 3. else return 1. } |
| --- |

**Infix to Prefix conversion by Stack (Stepwise solution)**

| **a+b-c\*d+(e-f)**  **(right to left)** | **Stack (Pop only priority is high)** | **Q(array/output)** |
| --- | --- | --- |
| ) | # ) |  |
| f | # ) | f |
| - | # ) - | f |
| e | # ) - | f e |
| ( | # | f e - |
| + | # + | f e - |
| d | # + | f e - d |
| \* | # + \* | f e - d |
| c | # + \* | f e - d c |
| - | # + - | f e - d c \* |
| b | # + - | f e – d c \* b |
| + | # + - + | f e – d c \* b |
| A | # + - + | f e – d c \* b a |
|  |  | f e – d c \* b a + - + |

Reverse the expression **+, -, +, a, b, \*, c, d, –, e, f**

### 3.2.6 Evaluation of prefix.

| -, +,2,3,\*,/,8,2,3 (R to L) | Stack | Result |
| --- | --- | --- |
| 3 | 3 |  |
| 2 | 3, 2 |  |
| 8 | 3, 2, 8 |  |
| / | 3, 4 | 8/2=4 |
| \* | 12 | 4\*3=12 |
| 3 | 12, 3 |  |
| 2 | 12, 3, 2 |  |
| + | 12, 5 | 3+2=5 |
| - | -7 | 5-12=-7 |

### 3.2.7 Tower of Hanoi.

****

| TOH(n, A, B, C) // ( S, M, D) { if(n==1) { Print("move %c disk from %c to %c", n, A, C) Return } TOH( n-1, A, C, B) // (S, D, M) Print( "move %c disk from %c to %c", n, A, C) TOH(n-1, B, A, C) //( M, S, D) } |
| --- |

# 

# 

# 

# 4.Queue

## 4.1 Implementation of Queue by Array.

| define queue[10], front=-1, rear=-1, size=10;  insertionQA(int ele) { if(rear==size-1) { print(Overflow) } else if(rear==-1) { rear=front=0 queue[rear]=ele }  else  {  rear=rear+1;  queue[rear]=ele  }  }//end of insertionQA  int deletionQA() { if(front==-1) { print(overflow) } else if(front==rear)  {  ele=queue[front] front=rear=-1  return ele  } else  {  ele=queue[front] front=front+1  return ele }}  printQA() { if(front==-1) { print(underflow) } else { for(i=front;i<=rear;i++) Print(queue[i]) } } |
| --- |

## 4.2 Circular Queue.

| declare int cq[], front=-1,rear=-1,size=10; insertionCQA(int ele) { if (front==(rear+1)%size)) or  //if (front==0&&rear==size-1||front==rear+1)  print(Overflow) else if(rear==-1) { front=rear=0 cq[rear]=ele } else { rear=(rear+1)%size cq[rear]=ele }  }  DeletionQA() { if(front==-1) print(overflow) else if (front==rear) front=rear=-1 else front=(front+1)%size }  print() {  for(i=front;front!=-1;i=(i+1)%size) { print(cq[i]) if(i==rear) break;  }  } |
| --- |

| OR //alternate for print()  print() { if(front<=rear) { for(i=front;i<=rear;i++) print(cq[i]) } else { for(i=front;i<=size-1;i++) print(cq[i]) for(i=0;i<=rear;i++) print(cq[i]) } } |
| --- |

## 4.3 Queue- Linked List

| struct node { int data struct node \*ptr }\*front=NULL,\*rear=NULL  insertionQLL(int ele) { struct node \*t t=malloc() if(t!=NULL) { t->data=ele t->ptr=NULL if(rear==NULL) front=rear=t else { rear->ptr=t rear=t } }  deletionQLL() { if(front==NULL) print(underflow) else if(front==rear) { free (front) front=rear=NULL } else { t=front front=front->ptr free(t) } }  printQLL { for(t=front;t!=NULL;t=t->ptr) print(t->data) } |
| --- |

## 4.4 Priority Queue by linked list

| struct node { int data, pri struct node \*ptr }\*front=NULL;  insertionPQLL(int ele, int priority) { t= malloc; if(t!=NULL)  { t->data=ele t->pri=priority  if(front==NULL) { front=t; t->ptr=NULL; } else { if(t->pri < front->pri) { t->ptr=front front=t } else { for(t2=front; t->pri >= t2->pri && t2!=NULL; ) { t1=t2 t2=t2->ptr } t->ptr=t1->ptr t1->ptr=t; }//end of inner else }// end of outer else }// end of if } |
| --- |

## 4.5 Priority using array:

| Declare mq[3][5], fr[3][2]={-1,-1,-1,-1,-1,-1}; rsize=3, csize=5;  InsertionPQA(in ele, int priority) { if(fr[priority][0]==(fr[priority][1]+1)%csize) Print (Overflow) else if(fr[priority][0]==-1) { fr[priority][0]=fr[priority][1]=0 mq[priority][0]=ele; } else { fr[priority][1]=(fr[priority][1]+1)%csize mq[priority][fr[priority][1]]=ele;  }  }  deletionPQA() { for(i=0;i<rsize;i++)  {  if(fr[i][0]!=-1)  {  if(fr[i][0]==fr[i][1])  {  print(mq[priority][fr[priority][0]])  fr[priority][0]=fr[priority][1]=-1  break;  }  else  {  print(mq[i][fr[i][0]])  fr[i][0]=(fr[i][0]+1)%csize  break;  }  }  if(i==rsize)  print(“underflow”);  } |
| --- |

# 

# 

# 

# 

# 

# 

# 

# 

# 5.Trees

## 5.1 Binary Tree

### 5.1.1 Binary Tree Creation :

| struct node  {  int data;  struct node \*lc,\*rc;  } \*root=NULL;  struct node \* Binarytreecreation() {  Print “Enter data otherwise -1”  Read data  if( data!=-1)  {  t=malloc()  t->data=data;  Print “Enter left child of %d”,data  t->lc=binarytreecreation()  Print “Enter right child of %d”,data  t->rc=binarytreecreation()  return t; }  else  return NULL;  }  main()  {  root=binarytreecreation();  preorder(root);  } |
| --- |

### 5.1.2 Pre Order:

| // using recursion  preorder(struct node \*t)  {  if(t!=NULL)  {  Print:- t->data;  preorder(t->lc);  preorder(t->rc); }  } |
| --- |

| Ptr is pointing binary tree(Root) //using stack preorder(struct node \*ptr)  {  While(ptr!=NULL) { Print(ptr->data) If(ptr->rc!=NULL) push(ptr->rc) else if(ptr->lc!=NULL) ptr=ptr->lc; else ptr=pop() }  } |
| --- |

### 5.1.3 Inorder:

| //using recursion  inorder(struct node \* t)  {  if(t!=NULL)  {  inorder(t->lc);  Print:- t->data;  inorder(t->rc);  }  } |
| --- |

| // using stack  again: While(ptr != NULL) { push(ptr) ptr=ptr->lc } ptr=pop() while(ptr!=NULL) { Print(ptr->data) If(ptr->rc!=NULL) { ptr=ptr->rc goto again } Else ptr=pop() } |
| --- |

### 5.1.4 Postorder

| postorder(struct node \* t) //using recursion  {  if(t!=NULL)  {  postorder(t->lc);  postorder(t->rc);  Print:- t->data;  }  } |
| --- |



## 5.2 Binary Search Tree

### 5.2.1 Searching in BST

| search(struct node \*root1, int ele) { If(root1==NULL) Print(Not Found) Else if(root1->data==ele) Print(Found) Else if(ele>root1->data) search(root1->rc,ele) Else if(ele<root1->data) search(root1->lc,ele) } |
| --- |

### 5.2.2 BST Insertion and Creation.

| struct node { int data struct node \*lc,\*rc; };  struct node \*insertion(struct node\* root1, int ele) { If(root1==NULL) { struct node \*t=malloc() t->data=ele t->lc=t->rc=NULL  return t  } else if(ele>root1->data) root1->rc=insertion(root1->rc,ele) else if(ele<root1->data) root1->lc=insertion(root1->lc,ele)  return root1 } main() { struct node \*root =NULL read ele root=insertion(root,ele) } |
| --- |

### 

### 5.2.3 Deletion in BST

| struct node\* deletion(struct node\* root1, int ele) { If(root1==NULL) return root1; else if(ele >root1->data) root1->rc=deletion(root1->rc,ele) else if(ele<root1->data) root1->lc=deletion(root1->lc,ele) else { If(root1->lc==NULL) { struct node \*temp=root1->rc free root1 return temp } If(root1->rc==NULL) struct node \*temp=root1->lc free root1 return temp } struct node \*temp=minvalue(root1->rc) root1->data=temp->data root1->rc=deletion(root1->rc,temp->data) } //end of else return root1 }  struct node \*minvalue(struct node \*t) { while(t->lc!=NULL)  { t=t->lc }  return root1 } |
| --- |
|  |

## 5.3 AVL Tree

### 5.3.1 AVL Tree Insertion

| struct node { int data struct node \*lc, \*rc int ht } |
| --- |

| int height(struct node \*t) {  if(t==NULL) return 0; else  {  int lh, rh;  lh=height(t->lc);  rh=height(t->rc);  if(lh>rh) return (lh+1); else return (rh+1);  } } |
| --- |

| int balfact(struct node \*t) { int lh, rh  if(t==NULL)  return 0  if(t->lc==NULL)  lh=0  else  lh=t->lc->ht  if(t->rc==NULL)  rh=0  else  rh=t->rc->ht  return (lh-rh) } |
| --- |

| struct node \* insertion(struct node \* t, int ele)  {  If(t == NULL)   {  t = malloc()  t - > data = ele  t - > lc = t - > rc = NULL  }  else if (ele > t - > data)   {  t - > rc = insertion(t - > rc, ele)  if (balfact(t) == -2)   {  if(ele > t - > rc - > data)  t = RRrotation(t)  else  t = RLrotation(t)  }  }  else if (ele < t - > data)   {  t - > lc = insertion(t - > lc, ele)  if (balfact(t) == 2)   {  if(ele < t - > lc - > data)  t = LLrotation(t)  else  t = LRrotation(t)  }  t - > ht = height(t)  return t  } } |
| --- |

| LLrotation(struct node \*t) { t=RightRotate(t) return t } |
| --- |

| RRrotation(struct node \*t) { t=LeftRotate(t) return t } |
| --- |

| LeftRotate(\*a) { struct node \*b=a->rc a->rc=b->lc b->lc=a a->ht=height(a) b->ht=height(b) return b } |
| --- |

| RightRotate(\*a) { struct node \*b=a->lc a->lc=b->rc b->rc=a a->ht=height(a) b->ht=height(b) return b } |
| --- |

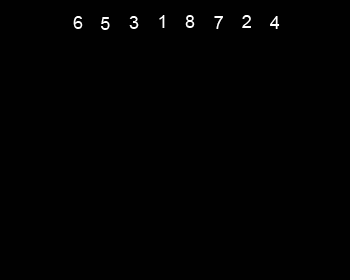
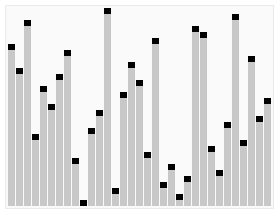
| RLrotation(struct node \*t) { t->rc=RightRotate(t->rc) t=LeftRotate(t) return t } |
| --- |

| LRrotation(struct node \*t) { t->lc=LeftRotate(t->lc) t=RightRotate(t) return t } |
| --- |

### 5.3.2 AVL Tree Deletion

| deletion( \* t, ele)   {  if (t == NULL)   {  print "Not Found!"  return NULL;  }    if (ele > t - > data)   {  t - > rc = deletion(t - > rc = ele);  if (balfact(t) == 2)   {  if (balfact(t - > lc) >= 0)  t = LLrotation(t);  else  t = LRrotation(t);  }  }   else if (ele < t - > data)   {  t - > lc = deletion(t - > lc, ele)  if (balfact(t) == -2) {  if (balfact(t - > lc) <= 0)  RRrotation(t);  else  RLrotation(t);  }    if (t - > rc != NULL)   {  \* temp = t - > rc;  while (temp - > lc = NULL)  temp = temp - > lc;  t - > data = temp - > data;  t - > rc = deletion(t - > rc, temp - > data);  if (balfact(t) == 2)   {  if (balfact(t - > lc) >= 0)  t = LLrotation(t);  else  t = LRrotation(t);  }   else   {  return (t - > lc)  }  t - > ht = height(t - > lc)  return t;  }  } |
| --- |

## 5.4 Heap sorting (Max Heap)



### 

### 

### 5.4.1 Heap Insertion

| Max heap   1. Assume a[max] \\n=last index of max heap tree 2. n=n+1, ptr=n, ele=a[n] 3. While (ptr>0) 4. { 5. par=(ptr-1)/2 6. if(ele=<a[par]) 7. { 8. a[ptr]=ele; 9. Return; 10. } 11. a[ptr]=a[par] 12. ptr=par; 13. } 14. a[ptr]=ele; 15. exit |
| --- |

### 5.4.2 Heap Deletion

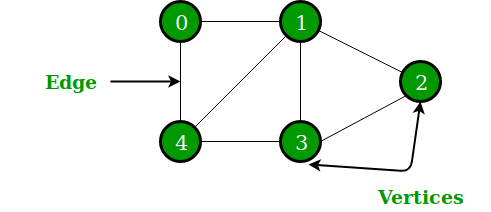
| Max heap   1. Assume a[max] 2. ele=a[n],n=n-1, ptr=0, left=1, right=2 3. While (right<=n) 4. { 5. if(ele>=a[left] and ele>=a[right] ) 6. { 7. a[ptr]=ele; 8. Return; 9. } 10. else if (a[left]<a[right]) 11. { 12. a[ptr]=a[right] 13. ptr=right; 14. } 15. else if (a[left]>a[right]) 16. { 17. a[ptr]=a[left] 18. ptr=left; 19. } 20. left=(2\*ptr)+1; 21. right=left+1; 22. } 23. if(left==n and ele<=a[left]) 24. a[ptr]=ele; 25. else 26. { 27. a[ptr]=a[left] 28. a[left]=ele; 29. } 30. exit |
| --- |

6. Graph

**Source:-** <https://www.geeksforgeeks.org/graph-and-its-representations/>

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as,

***A Graph consists of a finite set of vertices(or nodes) and set of Edges which connect a pair of nodes.***



In the above Graph, the set of vertices V = {0,1,2,3,4} and the set of edges E = {01, 12, 23, 34, 04, 14, 13}.

Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, on Facebook, each person is represented by a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale etc.

# 6.1 Graph and its representations

Graph is a data structure that consists of the following two components:

**1.** A finite set of vertices also called as nodes.

**2.** A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not the same as (v, u) in the case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

Graphs are used to represent many real-life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, on Facebook, each person is represented by a vertex(or node). Each node is a structure and contains information like person id, name, gender and locale. See thisfor more applications of graph.

Following is an example of an undirected graph with 5 vertices.

graph_representation1

Following two are the most commonly used representations of a graph.

**1.** Adjacency Matrix

**2.** Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

**6.1.1 Adjacency Matrix:**

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

The adjacency matrix for the above example graph is:

Adjacency Matrix Representation

*Pros:* Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex ‘u’ to vertex ‘v’ are efficient and can be done O(1).

*Cons:* Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.

Please see [this](https://ide.geeksforgeeks.org/9je5j6jJ13) for a sample Python implementation of adjacency matrix.

**6.1.2 Adjacency List:**

An array of lists is used. Size of the array is equal to the number of vertices. Let the array be array[]. An entry array[i] represents the list of vertices adjacent to the ***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is adjacency list representation of the above graph.

Adjacency List Representation of Graph

[Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.](https://practice.geeksforgeeks.org/problems/print-adjacency-list/0)

Note that in the below implementation, we use dynamic arrays (vector in C++/ArrayList in Java) to represent adjacency lists instead of linked lists. The vector implementation has advantages of cache friendliness.

| // A simple representation of graphs using STL  #include<bits/stdc++.h>  using namespace std;   // A utility function to add an edge in an  // undirected graph.  void addEdge(vector<int> adj[], int u, int v)  {   adj[u].push\_back(v);   adj[v].push\_back(u);  }   // A utility function to print the adjacency list  // representation of graph  void printGraph(vector<int> adj[], int V)  {   for (int v = 0; v < V; ++v)   {   cout << "\n Adjacency list of vertex "  << v << "\n head ";   for (auto x : adj[v])   cout << "-> " << x;   printf("\n");   }  }   // Driver code  int main()  {   int V = 5;   vector<int> adj[V];   addEdge(adj, 0, 1);   addEdge(adj, 0, 4);   addEdge(adj, 1, 2);   addEdge(adj, 1, 3);   addEdge(adj, 1, 4);   addEdge(adj, 2, 3);   addEdge(adj, 3, 4);   printGraph(adj, V);   return 0;  } |
| --- |

# 

# 

# 6.2 Graph traversing

## 6.2.1 Breadth First Search (BFS )

[Breadth First Traversal (or Search)](http://en.wikipedia.org/wiki/Breadth-first_search) for a graph is similar to Breadth First Traversal of a tree (See method 2 of [this post](https://www.geeksforgeeks.org/level-order-tree-traversal/)). The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don’t mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is 2, 0, 3, 1.



[Recommended: Please solve it on “*PRACTICE* ” first, before moving on to the solution.](https://practice.geeksforgeeks.org/problems/bfs-traversal-of-graph/1)

Following are the implementations of simple Breadth First Traversal from a given source.

The implementation uses [adjacency list representation](http://en.wikipedia.org/wiki/Adjacency_list) of graphs. [STL](http://en.wikipedia.org/wiki/Standard_Template_Library)‘s [list container](http://www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html#LIST) is used to store lists of adjacent nodes and queue of nodes needed for BFS traversal.

| BFS Algorithm:-  BFS (G, s) //Where G is a graph and s is the source node  let Q be queue.  Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.  mark s as visited.  while ( Q is not empty)  //Removing that vertex from queue,whose neighbour will be visited now  v = Q.dequeue( )  //processing all the neighbours of v   for all neighbours w of v in Graph G  if w is not visited   Q.enqueue( w )//Stores w in Q to further visit its neighbour  mark w as visited. |
| --- |

| vector <int> v[10] ; //Vector for maintaining adjacency list explained above  int level[10]; //To determine the level of each node  bool vis[10]; //Mark the node if visited   void bfs(int s) {  queue <int> q;  q.push(s);  level[ s ] = 0 ; //Setting the level of the source node as 0  vis[ s ] = true;  while(!q.empty())  {  int p = q.front();  q.pop();  for(int i = 0;i < v[ p ].size() ; i++)  {  if(vis[ v[ p ][ i ] ] == false)  {  //Setting the level of each node with an increment in the level of parent node  level[ v[ p ][ i ] ] = level[ p ]+1;   q.push(v[ p ][ i ]);  vis[ v[ p ][ i ] ] = true;  }  }  }  } |
| --- |

## 

## 6.2.2 Depth First Search (DFS)

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:

Pick a starting node and push all its adjacent nodes into a stack.

Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.

Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

| Pseudocode  DFS-iterative (G, s): //Where G is a graph and s is source vertex  let S be stack  S.push( s ) //Inserting s in stack   mark s as visited.  while ( S is not empty):  //Pop a vertex from stack to visit next  v = S.top( )  S.pop( )  //Push all the neighbours of v in stack that are not visited   for all neighbours w of v in Graph G:  if w is not visited :  S.push( w )   mark w as visited |
| --- |

| DFS-recursive(G, s):  mark s as visited  for all neighbours w of s in Graph G:  if w is not visited:  DFS-recursive(G, w) |
| --- |

# 6.3 Minimum Spanning Tree

## 6.3.1 Prim's Algorithm

Prim's algorithm is a [minimum spanning tree](https://www.programiz.com/dsa/spanning-tree-and-minimum-spanning-tree#minimum-spanning) algorithm that takes a graph as input and finds the subset of the edges of that graph which

* form a tree that includes every vertex
* has the minimum sum of weights among all the trees that can be formed from the graph

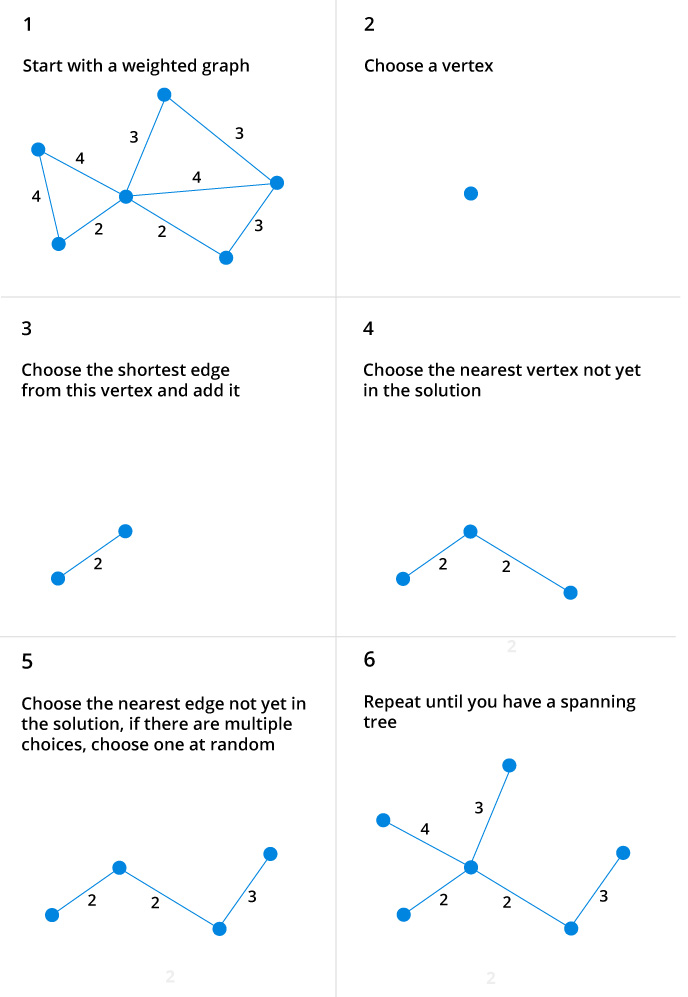
**How Prim's algorithm works**

It falls under a class of algorithms called [greedy algorithms](http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Greedy/greedyIntro.htm) which find the local optimum in the hopes of finding a global optimum.

We start from one vertex and keep adding edges with the lowest weight until we reach our goal.

The steps for implementing Prim's algorithm are as follows:

1. Initialize the minimum spanning tree with a vertex chosen at random.
2. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
3. Keep repeating step 2 until we get a minimum spanning tree

****

Prim's Algorithm pseudocode

The pseudocode for prim's algorithm shows how we create two sets of vertices U and V-U. U contains the list of vertices that have been visited and V-U the list of vertices that haven't. One by one, we move vertices from set V-U to set U by connecting the least weight edge.

1. T = ∅;
2. U = { 1 };
3. while (U ≠ V)
4. let (u, v) be the lowest cost edge such that u ∈ U and v ∈ V - U;
5. T = T ∪ {(u, v)}
6. U = U ∪ {v}

Prim's Algorithm Implementation in C++

The program below implements Prim's algorithm in C++. Although [adjacency matrix](https://www.programiz.com/dsa/graph-adjacency-matrix) representation of graph is used, this algorithm can also be implemented using [Adjacency List](https://www.programiz.com/dsa/graph-adjacency-list) to improve its efficiency.

1. #include <iostream>
2. #include <cstring>
3. using namespace std;
4. #define INF 9999999
5. // number of vertices in graph
6. #define V 5
7. // create a 2d array of size 5x5
8. //for adjacency matrix to represent graph
9. int G[V][V] = {
10. {0, 9, 75, 0, 0},
11. {9, 0, 95, 19, 42},
12. {75, 95, 0, 51, 66},
13. {0, 19, 51, 0, 31},
14. {0, 42, 66, 31, 0}
15. };
16. int main () {
17. int no\_edge; // number of edge
18. // create a array to track selected vertex
19. // selected will become true otherwise false
20. int selected[V];
21. // set selected false initially
22. memset (selected, false, sizeof (selected));
23. // set number of edge to 0
24. no\_edge = 0;
25. // the number of edge in minimum spanning tree will be
26. // always less than (V -1), where V is the number of vertices in
27. //graph
28. // choose 0th vertex and make it true
29. selected[0] = true;
30. int x; // row number
31. int y; // col number
32. // print for edge and weight
33. cout << "Edge" << " : " << "Weight";
34. cout << endl;
35. while (no\_edge < V - 1) {
36. //For every vertex in the set S, find the all adjacent vertices
37. // , calculate the distance from the vertex selected at step 1.
38. // if the vertex is already in the set S, discard it otherwise
39. //choose another vertex nearest to selected vertex at step 1.
40. int min = INF;
41. x = 0;
42. y = 0;
43. for (int i = 0; i < V; i++) {
44. if (selected[i]) {
45. for (int j = 0; j < V; j++) {
46. if (!selected[j] && G[i][j]) { // not in selected and there is an edge
47. if (min > G[i][j]) {
48. min = G[i][j];
49. x = i;
50. y = j;
51. }
53. }
54. }
55. }
56. }
57. cout << x << " - " << y << " : " << G[x][y];
58. cout << endl;
59. selected[y] = true;
60. no\_edge++;
61. }
62. return 0;
63. }

On running the above code, we get the output as:

Edge : Weight

0 - 1 : 9

1 - 3 : 19

3 - 4 : 31

3 - 2 : 51

**Prim's vs Kruskal's Algorithm**

[Kruskal's algorithm](https://www.programiz.com/dsa/kruskal-algorithm) is another popular minimum spanning tree algorithm that uses a different logic to find the MST of a graph. Instead of starting from a vertex, Kruskal's algorithm sorts all the edges from low weight to high and keeps adding the lowest edges, ignoring those edges that create a cycle.

## 6.3.2 Kruskal's Algorithm

Kruskal's algorithm is a [minimum spanning tree](https://www.programiz.com/dsa/spanning-tree-and-minimum-spanning-tree#minimum-spanning) algorithm that takes a graph as input and finds the subset of the edges of that graph which

* form a tree that includes every vertex
* has the minimum sum of weights among all the trees that can be formed from the graph

How Kruskal's algorithm works

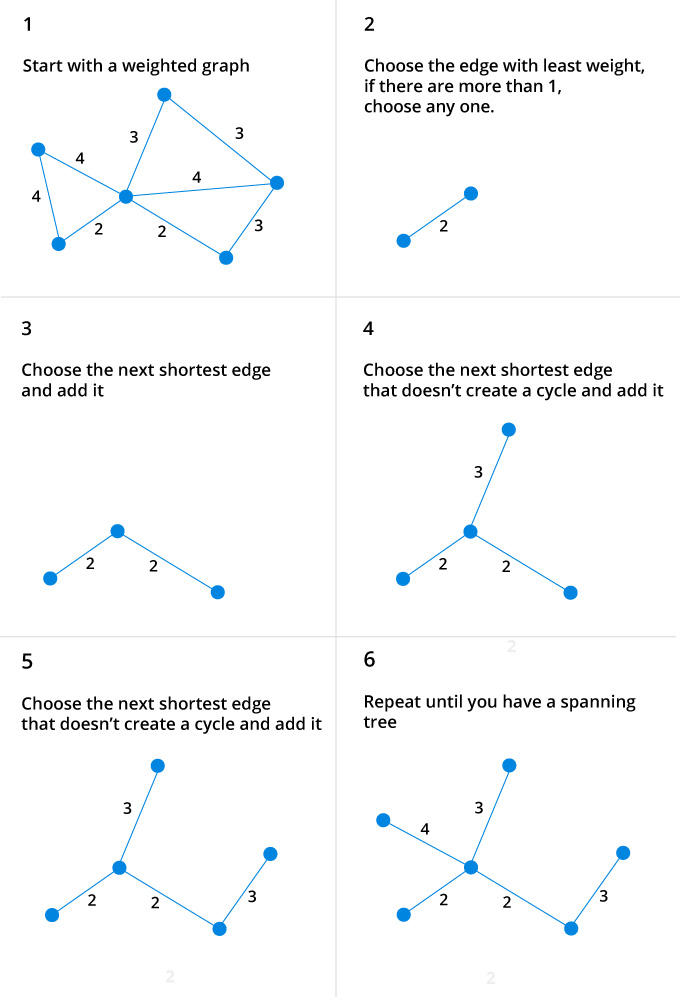
It falls under a class of algorithms called [greedy algorithms](http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Greedy/greedyIntro.htm) which find the local optimum in the hopes of finding a global optimum.

We start from the edges with the lowest weight and keep adding edges until we reach our goal.

The steps for implementing Kruskal's algorithm are as follows:

1. Sort all the edges from low weight to high
2. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
3. Keep adding edges until we reach all vertices.

Example of Kruskal's algorithm

****

Kruskal Algorithm Pseudocode

Any minimum spanning tree algorithm revolves around checking if adding an edge creates a loop or not.

The most common way to find this out is an algorithm called [Union FInd](https://www.cs.duke.edu/courses/cps100e/fall09/notes/UnionFind.pdf). The Union-Find algorithm divides the vertices into clusters and allows us to check if two vertices belong to the same cluster or not and hence decide whether adding an edge creates a cycle.

1. KRUSKAL(G):
2. A = ∅
3. For each vertex v ∈ G.V:
4. MAKE-SET(v)
5. For each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v):
6. if FIND-SET(u) ≠ FIND-SET(v):
7. A = A ∪ {(u, v)}
8. UNION(u, v)
9. return A

Kruskal's Algorithm Implementation in C++

Here is the code for C++ implementation in C++. We use standard template libraries to make our work easier and code cleaner.

1. #include <iostream>
2. #include <vector>
3. #include <algorithm>
4. using namespace std;
5. #define edge pair<int,int>
6. class Graph {
7. private:
8. vector<pair<int, edge>> G; // graph
9. vector<pair<int, edge>> T; // mst
10. int \*parent;
11. int V; // number of vertices/nodes in graph
12. public:
13. Graph(int V);
14. void AddWeightedEdge(int u, int v, int w);
15. int find\_set(int i);
16. void union\_set(int u, int v);
17. void kruskal();
18. void print();
19. };
20. Graph::Graph(int V) {
21. parent = new int[V];
22. //i 0 1 2 3 4 5
23. //parent[i] 0 1 2 3 4 5
24. for (int i = 0; i < V; i++)
25. parent[i] = i;
26. G.clear();
27. T.clear();
28. }
29. void Graph::AddWeightedEdge(int u, int v, int w) {
30. G.push\_back(make\_pair(w, edge(u, v)));
31. }
32. int Graph::find\_set(int i) {
33. // If i is the parent of itself
34. if (i == parent[i])
35. return i;
36. else
37. // Else if i is not the parent of itself
38. // Then i is not the representative of his set,
39. // so we recursively call Find on its parent
40. return find\_set(parent[i]);
41. }
42. void Graph::union\_set(int u, int v) {
43. parent[u] = parent[v];
44. }
45. void Graph::kruskal() {
46. int i, uRep, vRep;
47. sort(G.begin(), G.end()); // increasing weight
48. for (i = 0; i < G.size(); i++) {
49. uRep = find\_set(G[i].second.first);
50. vRep = find\_set(G[i].second.second);
51. if (uRep != vRep) {
52. T.push\_back(G[i]); // add to tree
53. union\_set(uRep, vRep);
54. }
55. }
56. }
57. void Graph::print() {
58. cout << "Edge :" << " Weight" << endl;
59. for (int i = 0; i < T.size(); i++) {
60. cout << T[i].second.first << " - " << T[i].second.second << " : "
61. << T[i].first;
62. cout << endl;
63. }
64. }
65. int main() {
66. Graph g(6);
67. g.AddWeightedEdge(0, 1, 4);
68. g.AddWeightedEdge(0, 2, 4);
69. g.AddWeightedEdge(1, 2, 2);
70. g.AddWeightedEdge(1, 0, 4);
71. g.AddWeightedEdge(2, 0, 4);
72. g.AddWeightedEdge(2, 1, 2);
73. g.AddWeightedEdge(2, 3, 3);
74. g.AddWeightedEdge(2, 5, 2);
75. g.AddWeightedEdge(2, 4, 4);
76. g.AddWeightedEdge(3, 2, 3);
77. g.AddWeightedEdge(3, 4, 3);
78. g.AddWeightedEdge(4, 2, 4);
79. g.AddWeightedEdge(4, 3, 3);
80. g.AddWeightedEdge(5, 2, 2);
81. g.AddWeightedEdge(5, 4, 3);
82. g.kruskal();
83. g.print();
84. return 0;
85. }

When we run the program, we get output as

Edge : Weight

1 - 2 : 2

2 - 5 : 2

2 - 3 : 3

3 - 4 : 3

0 - 1 : 4

Kruskal's vs Prim's Algorithm

[Prim's algorithm](https://www.programiz.com/dsa/prim-algorithm) is another popular minimum spanning tree algorithm that uses a different logic to find the MST of a graph. Instead of starting from an edge, Prim's algorithm starts from a vertex and keeps adding lowest-weight edges which aren't in the tree, until all vertices have been covered.

7. Sorting and Searching

# 7.1 Sorting

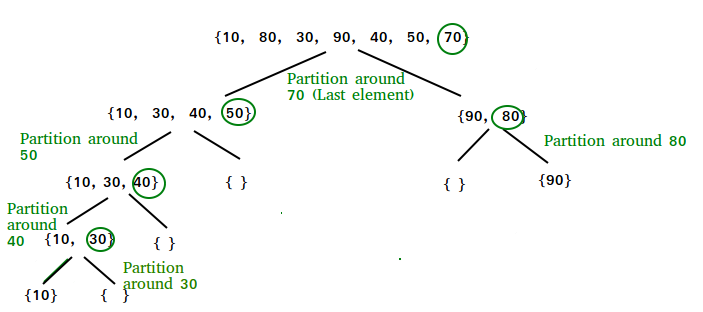
## 7.1.1 Quick Sort

Source:- <https://www.geeksforgeeks.org/quick-sort/>

Like [Merge Sort](http://quiz.geeksforgeeks.org/merge-sort/), QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.



**Pseudo Code for recursive QuickSort function :**

| /\* low --> Starting index, high --> Ending index \*/ quickSort(arr[], low, high) {  if (low < high)  {  /\* pi is partitioning index, arr[pi] is now  at right place \*/  pi = partition(arr, low, high);   quickSort(arr, low, pi - 1); // Before pi  quickSort(arr, pi + 1, high); // After pi  } } |
| --- |

**Partition Algorithm**

There can be many ways to do partition, following pseudo code adopts the method given in CLRS book. The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

| /\* low --> Starting index, high --> Ending index \*/ quickSort(arr[], low, high) {  if (low < high)  { /\* pi is partitioning index, arr[pi] is now at right place \*/  pi = partition(arr, low, high);   quickSort(arr, low, pi - 1); // Before pi  quickSort(arr, pi + 1, high); // After pi  } } |
| --- |

**Pseudo code for partition()**

| /\* This function takes last element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot) to the left of pivot and all greater elements to right of pivot \*/  partition (arr[], low, high) {  // pivot (Element to be placed at right position)  pivot = arr[high];     i = (low - 1) // Index of smaller element   for (j = low; j <= high- 1; j++)  {  // If current element is smaller than the pivot  if (arr[j] < pivot)  {  i++; // increment index of smaller element  swap arr[i] and arr[j]  }  }  swap arr[i + 1] and arr[high])  return (i + 1) } |
| --- |

**Illustration of partition() :**

arr[] = {10, 80, 30, 90, 40, 50, 70}

Indexes: 0 1 2 3 4 5 6

low = 0, high = 6, pivot = arr[h] = 70

Initialize index of smaller element, **i = -1**

Traverse elements from j = low to high-1

**j = 0** : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

**i = 0**

arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j

// are same

**j = 1** : Since arr[j] > pivot, do nothing

// No change in i and arr[]

**j = 2** : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

**i = 1**

arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

**j = 3** : Since arr[j] > pivot, do nothing

// No change in i and arr[]

**j = 4** : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

**i = 2**

arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped

**j = 5** : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]

**i = 3**

arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

We come out of loop because j is now equal to high-1.

**Finally we place pivot at correct position by swapping**

**arr[i+1] and arr[high] (or pivot)**

arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

Now 70 is at its correct place. All elements smaller than

70 are before it and all elements greater than 70 are after

It.

## 7.1.2 Merge sort

Like [QuickSort](https://www.geeksforgeeks.org/quick-sort/), Merge Sort is a [Divide and Conquer](https://www.geeksforgeeks.org/divide-and-conquer-introduction/) algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted subarrays into one. See following C implementation for details.

| MergeSort(arr[], l, r) If r > l  1. Find the middle point to divide the array into two halves:   middle m = (l+r)/2  2. Call mergeSort for first half:   Call mergeSort(arr, l, m)  3. Call mergeSort for second half:  Call mergeSort(arr, m+1, r)  4. Merge the two halves sorted in step 2 and 3:  Call merge(arr, l, m, r) |
| --- |

The following diagram from [wikipedia](http://en.wikipedia.org/wiki/File:Merge_sort_algorithm_diagram.svg) shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided into two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.



| void mergeSort(int arr[], int l, int r) {  if (l < r)  { // Same as (l+r)/2, but avoids overflow for  // large l and h  int m = l+(r-l)/2;    // Sort first and second halves  mergeSort(arr, l, m);  mergeSort(arr, m+1, r);   merge(arr, l, m, r);  } } |
| --- |

| void merge(int arr[], int l, int m, int r) {  int i, j, k;  int n1 = m - l + 1;  int n2 = r - m;    /\* create temp arrays \*/  int L[n1], R[n2];    /\* Copy data to temp arrays L[] and R[] \*/  for (i = 0; i < n1; i++)  L[i] = arr[l + i];  for (j = 0; j < n2; j++)  R[j] = arr[m + 1+ j];    /\* Merge the temp arrays back into arr[l..r]\*/  i = 0; // Initial index of first subarray  j = 0; // Initial index of second subarray  k = l; // Initial index of merged subarray  while (i < n1 && j < n2)  {  if (L[i] <= R[j])  {  arr[k] = L[i];  i++;  }  else  {  arr[k] = R[j];  j++;  }  k++;  }    /\* Copy the remaining elements of L[], if there are any \*/  while (i < n1)  {  arr[k] = L[i];  i++;  k++;  }  /\* Copy the remaining elements of R[], if there are any \*/  while (j < n2)  {  arr[k] = R[j];  j++;  k++;  } }   /\* l is for left index and r is right index of the  sub-array of arr to be sorted \*/ |
| --- |

8. The C++ Standard Template Library (STL)

Source:- <https://www.geeksforgeeks.org/the-c-standard-template-library-stl/>

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms, and iterators. It is a generalized library and so, its components are parameterized. A working knowledge of [template classes](https://www.geeksforgeeks.org/templates-cpp/) is a prerequisite for working with STL.

**STL has four components**

* Algorithms
* Containers
* Functions
* Iterators

**Algorithms**

The header algorithm defines a collection of functions especially designed to be used on ranges of elements.They act on containers and provide means for various operations for the contents of the containers.

* Algorithm
  + [Sorting](http://quiz.geeksforgeeks.org/sort-algorithms-the-c-standard-template-library-stl/)
  + [Searching](http://quiz.geeksforgeeks.org/binary-search-algorithms-the-c-standard-template-library-stl/)
  + [Important STL Algorithms](https://www.geeksforgeeks.org/c-magicians-stl-algorithms/)
  + [Useful Array algorithms](https://www.geeksforgeeks.org/useful-array-algorithms-in-c-stl/)
  + [Partition Operations](https://www.geeksforgeeks.org/stdpartition-in-c-stl/)
* Numeric
  + [valarray class](https://www.geeksforgeeks.org/std-valarray-class-c/)

**Containers**

Containers or container classes store objects and data. There are in total seven standard “first-class” container classes and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.

* Sequence Containers: implement data structures which can be accessed in a sequential manner.
  + [vector](http://quiz.geeksforgeeks.org/vector-sequence-containers-the-c-standard-template-library-stl-set-1/)
  + [list](http://quiz.geeksforgeeks.org/list-sequence-containers-the-c-standard-template-library-stl/)
  + [deque](http://quiz.geeksforgeeks.org/deque-sequence-containers-the-c-standard-template-library-stl/)
  + [arrays](https://www.geeksforgeeks.org/array-class-c/)
  + [forward\_list](https://www.geeksforgeeks.org/forward-list-c-set-1-introduction-important-functions/)( Introduced in C++11)
* Container Adaptors : provide a different interface for sequential containers.
  + [queue](http://quiz.geeksforgeeks.org/queue-container-adaptors-the-c-standard-template-library-stl/)
  + [priority\_queue](http://quiz.geeksforgeeks.org/priority-queue-container-adaptors-the-c-standard-template-library-stl/)
  + [stack](http://quiz.geeksforgeeks.org/stack-container-adaptors-the-c-standard-template-library-stl/)
* Associative Containers : implement sorted data structures that can be quickly searched (O(log n) complexity).
  + [set](http://quiz.geeksforgeeks.org/set-associative-containers-the-c-standard-template-library-stl/)
  + [multiset](http://quiz.geeksforgeeks.org/multiset-associative-containers-the-c-standard-template-library-stl/)
  + [map](http://quiz.geeksforgeeks.org/map-associative-containers-the-c-standard-template-library-stl/)
  + [multimap](http://quiz.geeksforgeeks.org/multimap-associative-containers-the-c-standard-template-library-stl/)
* Unordered Associative Containers : implement unordered data structures that can be quickly searched
  + [unordered\_set](https://www.geeksforgeeks.org/unordered_set-in-cpp-stl/) (Introduced in C++11)
  + [unordered\_multiset](https://www.geeksforgeeks.org/unordered_multiset-and-its-uses/) (Introduced in C++11)
  + [unordered\_map](https://www.geeksforgeeks.org/unordered_map-in-cpp-stl/) (Introduced in C++11)
  + [unordered\_multimap](https://www.geeksforgeeks.org/unordered_multimap-and-its-application/) (Introduced in C++11)

**Functions**

The STL includes classes that overload the function call operator. Instances of such classes are called function objects or functors. Functors allow the working of the associated function to be customized with the help of parameters to be passed.

* [Functors](https://www.geeksforgeeks.org/functors-in-cpp/)

**Iterators**

As the name suggests, iterators are used for working upon a sequence of values. They are the major feature that allow generality in STL.

* [Iterators](https://www.geeksforgeeks.org/iterators-c-stl/)

**Utility Library**

Defined under <utility header>

* [pair](http://quiz.geeksforgeeks.org/pair-simple-containers-the-c-standard-template-library-stl/)