DATA STRUCTURES

`TEACHER: Dr. Fahad Sherwani, Sir Zain`

`21k-4578 syeda samaha batool rizvi`

DR FAHAD SHERWANI `(green)`	SIR ZAIN UL HASSAN `(purple)`

▼ WEEK 1

there are two ways to study data structures

- abstract view
- implementation view

Datatypes:

data types consists of certain domain of values and operator allowed on them.

there are mainly two types of data:

- User-defined datatypes
- primitive/default datatypes

Abstract Data Types (ADT)

Entities that are definition of data and operation, but don't have implementation details

only a blue print (without specifying what is inside function and how operation is performed)

Features Of ADT

▼ Abstraction

▼ Better Conceptualization

▼ Robust

Program is robust and have ability to _______ errors

Class Activity

Encapsulation

- Process of wrapping code and data together into a single unit
- or we can say that hiding internal representation of code to prevent direct access to implementation

Inheritance

- Mechanism of reusing and extending existing class without modifying them.
- Thus producing hierarchical relationship between them.
- procedure in which one class inherits the attributes and methods of other class
- parent child relationship

Polymorphism

- It describes the situation in which something occurs in several different terms
- There can be more than one methods that have same name but different implementation
- It allows us to perform single action in different ways

▼ WEEK 1

DATA STRUCTURES:

organization of data

DATATYPES:

there are mainly two types of data

1. Primitive datatypes
    a. int
    b. float
    c. char etc.
2. User-defined datatypes
    a. classes
    b. struct

CONCEPTS IN OOP AND PF REVISION

1. classes
2. pointers (DMA)
3. templates → generic
4. iteration
5. recursion

ABSTRACT DATATYPES ADT

```
//example 1
class list{
  int * val;
public:
void create()
{
  val = new int;
  *val = 0;
}
};
```

the above class is ADT because it has all three checks.

user-defined datatype	✔️
implementation of data structures	✔️
abstraction	✔️

```
//example 2
class A{
public:
void f()
{
cout<<"hello world!!";
}
};
```

the above class is not ADT because it does not have all three checks.

user-defined datatype	✔️
implementation of data structures	❌
abstraction	✔️

💡 every class is not an ADT

ITERATION VS RECURSION

ITERATION	RECURSION
It does not require a function	It requires a function
Unrolling	Recursive calls
Does not need a stack	Always need a stack
Performance limitation	Memory limitation

ITERATION

```
for(int i=0; i<5; i++)
{
  cout<<i;
}
/*
the compiler will unroll the code.
```

UNROLLING:

`Bonus information: for clearance of unrolling concept`

The general idea of loop unrolling or loop unwinding is to replicate the code inside a loop body a number of times. The number of copies is called the loop unrolling factor. The number of iterations is divided by the loop unrolling factor.

Loop unrolling can reduce the number of loop maintenance instruction executions by the loop unrolling factor. In effect, the computations are done by the compiler rather than being done during program execution.

While loop unrolling can be beneficial, excessive unrolling degrades performance.

Since loop unrolling is a tradeoff between code size and speed, the effectiveness of loop unrolling is highly dependent on the loop unrolling *factor*
, that is, the number of times the loop is unrolled

RECURSION:

```
int f( int i)
{
  //condition
  f(i+1);
}
//compiler holds memory in the stack. If the memory is oversized, the stack will overflow. hence it has memory  limitations
```

STACK:

`Bonus information: for clearing the concept of stack`

Recursive functions use something called “the call stack.” **When a program calls a function, that function goes on top of the call stack**

<u>Last In , First Out</u>

💡 whenever we are in a scenario, where we have limited memory and performance limitation, we can neither use iteration , nor recursion .
In this case we will break recursive function into non-recursive function

▼ WEEK 2

Algorithm And Its Complexity Analysis

A good algorithm is one which takes less time to compile to be complete

▼ WEEK 2

Based On the Placement Of Recursive Calls

▼ Direct recursion

```
int f1(int p)
{
  //condition
  f1(p+1);
}
//the recursive call is placed inside a recursive function
```

▼ Indirect Recursion

```
int f1(int p)
{
  //condition
  f2(p);
}
int f2(int p)
{
  //condition
  f1(p+1);
}
//the recursive call cannot be placed inside the recursive function
//instead what we do is, we call another function, then that function call the recursive function
```

Based On Recursive Calls

▼ Head recursion

`function_start(par):`

  `base case`

  `update of parameter`

  `recursive call`

  `action`

`function_end`

recursion before action

```
int print(int num)
{
  if(num==0)
    return num;
  num--;
  cout<<print(num);
}
```

▼ Tail recursion

`function_start(par):`

  `base case`

  `update of parameter`

  `action`

  `recursive call`

`function_end`

recursion after action

```
int print(int num)
{
  if(num==0)//base case
    return num;
  num--; //modification
```

```
    cout<<num; //action
    return print(num); //recursive call
  }
```

💡 The result will be the same in both recursions
the only difference will be visible in backtracking → difference in the order of calls

💡 In the case of compiler optimization, tail recursion is efficient

Task

`implement  list ADT in C++  which can:`

- `insert new element( at the very end or beginning)`
- `remove an element( replace element  with -1)`
-

```
class list{
int a[6];
static int count;
static int i;
public:
void insert(int n)
{
  for(int j=count;j>0;j--)
  {a[j]=a[j-1];
}
a[0]=n;
count++;
}
void remove()
{
  i-=1;
delete a[i];
}
}; int list :: count=0;
int list :: i=0;
```

Rule Of Three

whenever you have copy constructor, destructor and assignment operator overloading all together in a class this is called **rule of three**

benefit: no memory leakage

```
class test{
int * var;
public:
test()
{
var= new int;
* var= 50;
}
~test()
{
delete var;
}
test(const test&o)
{
var=new int;
* var = o.var;
}
//operator overload
test operator==(const test &o)
{
var = new int;
 * var= * o.var;
return * this;
}
```

```
};
int main()
{
test ob1;
test ob2(ob1);
test ob3=ob1;
}
//destructor will delete ob2( so object 1 ki memory bhi delete  hojaye gi so when it call fo ob1 there will be no memeory to
delete this is called memory leakage.
//solution deep copy that has been done in class.
```

Dynamic Memory

```
int row=3, col=5;
int * arr= new int[row*col];
//or we can declare by this method
auto arr=new int[row][5];//auto : adjust their type automatically
// second dimension must be const or integer value but not a variable
for(int i=0; i<3;i++)
{
delete[] arr[i];
}
delete [] arr;
int ** arr=int[5];
for(i=0;i<5;i++)
{
arr[i]= new int [2];
}//declaration in a double pointer
```

Factors depending Upon Performance Of Algorithm

1. space complexity (memory)
2. time complexity (no. of steps)

Example: bigO notation:

$o(n^2)$

`we calculate complexity of algorithm, not of source code or algorithm`

SFC (Steps Frequency Count) Table

PRIMITVE STEPS: Steps which have cost

- addition
- subtraction
- division
- multiplication & other arithmetic operation
- array indexing
- function call / return

parenthesis and loop start terminate has no steps count

```
//sample algorithm
for(i=0;i<5;i++)
{
int k= a+b; //1
if(k==5)//2
{
func();//3
}
}
```

no	steps	frequency	count
1	2		
2	1		

no	steps	frequency	count
3	1		

▼ WEEK 3

Recursion And Backtracking

RECURSION:

process of solving a problem and reducing it to smaller version of itself

Recursive Definition

A definition in which something is defined in terms of smaller version of itself

Base Case

case for which solution is obtained directly

General Case

case for which solution is obtained indirectly using recursion

Recursive Function

function that calls itself

Every recursive definition must have one (or more)
base cases

General case must eventually reduce to a base case

Base case stops recursion

```
//recursive function implemnting the factoial function
int fact(int num)
{
  if(num== 0)
    return 1;
  else
    return num*fact(num-1);
}
```

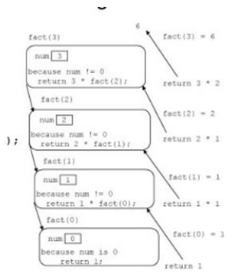


FIGURE 6-1 Execution of fact(3)

Recursive function notable comments

- Recursive function has unlimited number of copies of
  itself (logically)
- Every call to a recursive function has its own Code, set of parameters, local variables
- After completing a particular recursive call
  - Control goes back to calling environment (previous call)

- Current (recursive) call must execute completely before control goes back to the previous call
- Execution in previous call begins from point immediately following the recursive call

TYPES OF RECURSION

▼ Direct Recursion

function calls itself

▼ Indirect Recursion

function calls another function, which calls original function

▼ Tail Recursion

recursive call is the last statement

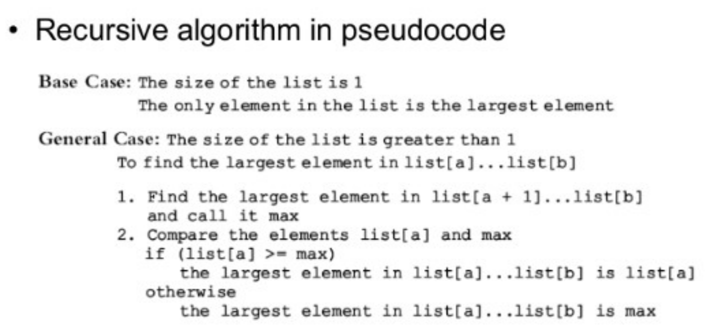
▼ Infinite Recursion

Occurs if every recursive call results in another recursive call

Executes forever (in theory)

Execution until system runs out of memory

```
//largest element in array
int largest(const int list[], int lowindex, int highindex)
{
  int max;
  if(lowindex==highindex)//size of the sublist is one
    return list[lowerindex];
else
{
  max=largest(list, lowindex+1, highindex);
if(llist[lowindex>=max)
  return list[lowindex];
else
return max;
}
}
```


- Recursive algorithm in pseudocode

```
Base Case: The size of the list is 1
           The only element in the list is the largest element

General Case: The size of the list is greater than 1
        To find the largest element in list[a]...list[b]

        1. Find the largest element in list[a + 1]...list[b]
           and call it max
        2. Compare the elements list[a] and max
           if (list[a] >= max)
              the largest element in list[a]...list[b] is list[a]
           otherwise
              the largest element in list[a]...list[b] is max
```

`fibbonacci and reversed linked list to be done her tower of hanoi`

BACKTRACKING

```
//N-Queen Problem
int main(){
in grid [4][4]={{0,0,0,0}{0,0,0,0}{0,0,0,0}{0,0,0,0}};
if(NQueen(grid, 0)
{//print queen
}
return 0;
}


bool NQueen(int grid[4][4], int col)
{
  if(col>=4)
    return true;//base case
for(int i=0;i<4;i++)//row
{
  if(check(grid,i,col))
  {
    grid[i][col]=1;
  if(NQueen(grid,col+1)
    return true;
    grid[i][col]=0;
}
}
return false;
}

bool check(int grid[4][4], int row, int col)
{
  int i,j;
  for(j=0;j<col;j++)
{//check column
if(grid[row][j])
    return false;
}
//check diagonal above
for(i=ow,j=col;i>==0 && j>=0;i--,j--)
{
  if(grid[i][j])
return false;
}
//check lower diagonal
for(i=row,j=col;i<4 && j>=0;i++,j--)
{
  if(grid[i][j])
    return false;
}
return true;
}
```

▼ WEEK 3

Recursion And Backtracking

Brute Force Approach:

Backtracking:

It is like a refined brute force

Its algorithm are straightforward methods of solving a problem that rely on sheer computing power and trying every possibilities.

- eliminate choices that are not possible and proceed to recursively check only those that have potential
- Pruning: Eliminate or remove choices that are not possible

```
int main(){
in grid [4][4]={{0,0,0,0}{0,0,0,0}{0,0,0,0}{0,0,0,0}};
if(NQueen(grid, 0)
{//print queen
}
return 0;
}
bool NQueen(int grid[4][4], int col)
{
if(col>=4) return true;
for(int i=0; i<4; i++)
{
if(check(grid,i,col))
{
  grid[i][co]=1;
```

```
      if( NQueen(gid, col+1)) return true;
    gid[i][col]=0;
    }
  }
  return false;
  }

  bool check(int grid[4][4], int row, int col)
  {
    int i,j;
    for(j=0; j<col; j++)
  {
    if(grid[row][j]) etun false;
    }

  for(i=row, j=col; i>=0 && j>=0;i--, j--)
  {
    if(grid[i][j]) return fasle;
  }
  for(i=row, j=col; i<0 && j>=0;i++, j--)
  {
    if(grid[i][j]) return fasle;
  }
  return true;
  }
```

▼ WEEK 4

Array

- array is contiguous data structure.
- size has to be known at the time of compilation
- inserting an element inside an array requires shifting the data in this array

to resolve this we have linked lists

Linked Lists

to be complete

▼ WEEK 4

Linked List

Properties of a Linked List

1. successive elements are connected by pointers
2. last element points to NULL
3. can be made just as long as required
4. does not waste memory space .(but take some extra memory for pointers). I t allocates memory as list grows

Types of Linked Lists

singly linked list

circular linked list

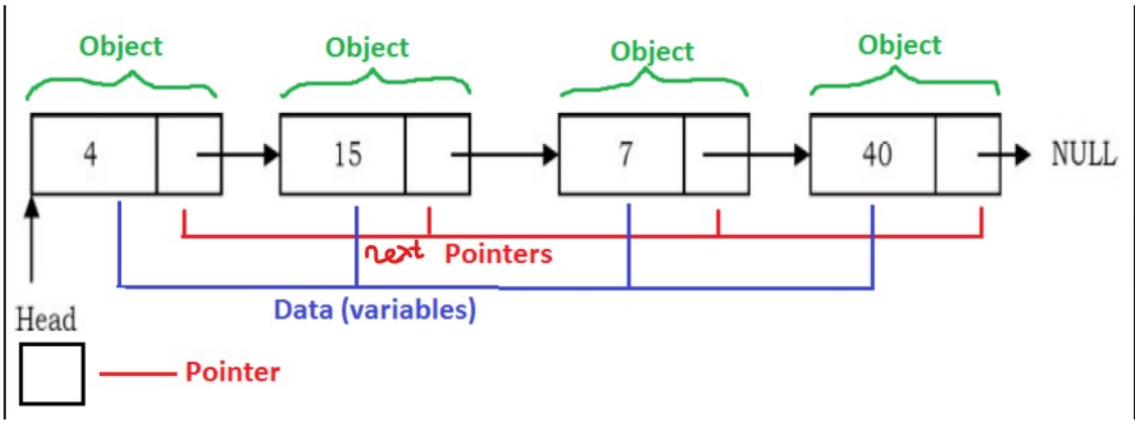
double linked list

▼ Singly linked list

also known as default linked list

- each node has next pointer to the following element

- link of the last node is null, indicating the end of list.



Basic operations in linked list

▼ Traversing

visiting each node of list exactly once

`start from next`

`follow next ptr`

`display/read contents as nodes are visited`

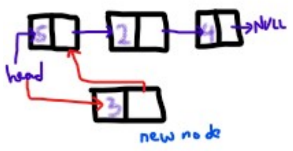
`stop when next ptr ==null`

▼ insertion

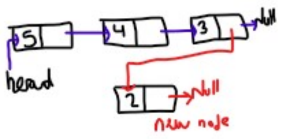
inserting a new element to the list

3 cases:

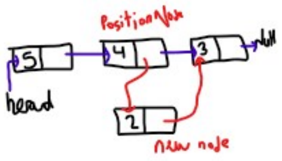
1. insert at beginning
    a. `new node is inserted before current head node`
    b. `modify next pointer`
        i. `update next ptr of new node to current head`
        ii. `update head ptr to point new node`
        iii. `traverse to last node`



2. insert at end/tail
    a. `new node’s next ptr points to null`
    b. `last node next ptr points to new node`



3. insert in middle
    a. `traverse to node where insetrion is supposed to be made`
    b. `modify two pointer`
        i. `new node points to next node of position where we want to add this`
        ii. `next ptr of position node( node before newly inserted node) points to new node`

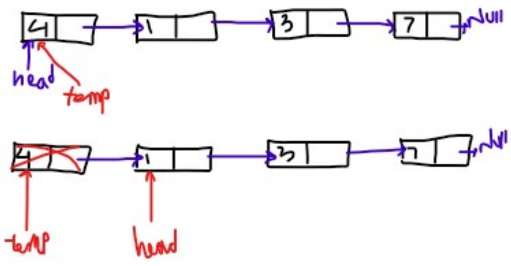


▼ deletion

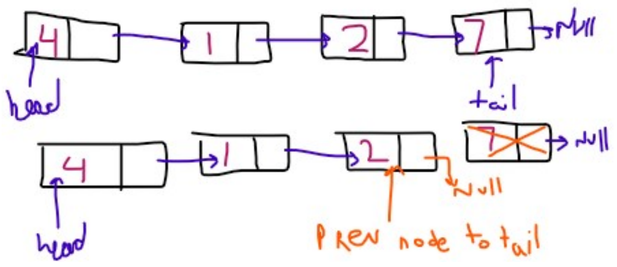
delete an existing element from list

3 cases

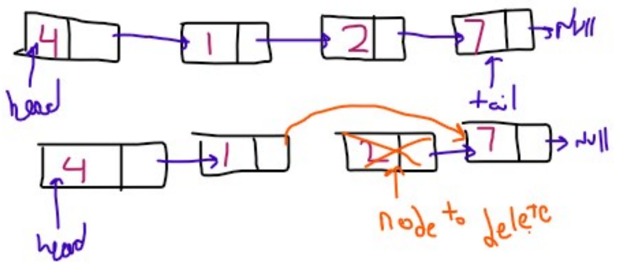
1. delete from beginning
    a. `create temp node which will pint towards head node`
    b. `move head ptr to next node and delete temp node`



2. delete from end
    a. `traverse list and maintain the previous node address too`
    b. `update previous node ptr to null`
    c. `dispose tail node`



3. delete from middle
    a. maintain previous node while traversing the list.
    b. after finding the desired node, change previous node’s next ptr to the next ptr of desired node
    c. delete currenet node



```
struct node{
  int data; //data part of node
  node * next;//next ptr of node
};
class linkedlist{
  node *head;
node * tail;
public:
  linkedlist()
{
  head=NULL;
  tail=NULL;
}
//traverse
void taverse(){
  node *temp;
  temp=head;//start from head
  while(temp!=NULL)//untill reach the end
  {
    cout<<temp->data;//visit and use node
    temp=temp->next;//shift pt to next node
  }

//insert at beginning
void insertbeg(int val)
{
  //construct new node
  node* temp=new node;
  node->data=val;

temp->next=head; //new node points to current head
head=temp;//new node designated as new head
  }

//insert last
void insertend(int val)
{
  //construct new node
  node* temp;
temp->data=val;
temp->next=NULL;//next ptr of new node to null
//check if list is empty
if(head==NULL)
{
head=temp;
tail=temp;
}
else{
tail->next=temp;//insert new node after tail
tail=tail->next;//designate new tai node
}

//insert middle
void insertmiddle(node *x, int val)
{
  //construct new node
node* p=new node;
p->data=val;
//if list is empty point to
if(head==NULL)
{
head=p;
tail=p;
}
p->next=x->next;//insert new node after node x
x->next=p;//next ptr of x to point new node
}

//delete beginning
void deletebeg(){
  //create temp node
  node*temp;
temp= head;//store head node in temp
head=head->next//point head to next of head node
delete temp;//delete temp node
}

//delete at end
void deleteend(){
node *cur;
node * prev
cur=head;
while(cur->next !=NULL)
{
  prev=cur;
  cur=cur->next;
}
prev->next=NULL;
tail=prev;
delete cur;
}

//delete middle
void deletemiddle(){
node*cur;
node *prev;
cur=head;
while(cur->data!=NULL){
  if(cur->data==7)//assume that node dat 7 is to be deleted
  break;
prev=cur;
cur=cur->next;
}
prev->next=cur->next;
delete cur;
}

};
```

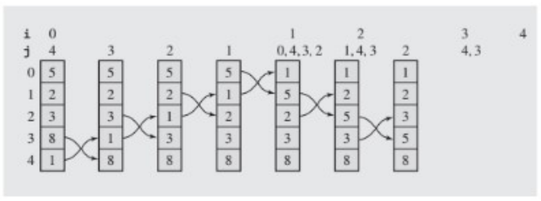
▼ WEEK 5

Bubble Sort

Algorithm

```
bubblesort(data[],n)
for i = 0 to n22
for j = n-1 down to i+1
swap elements in positions j and j-1 if they are out of order;
```

The array [5 2 3 8 1] sorted by bubble sort.



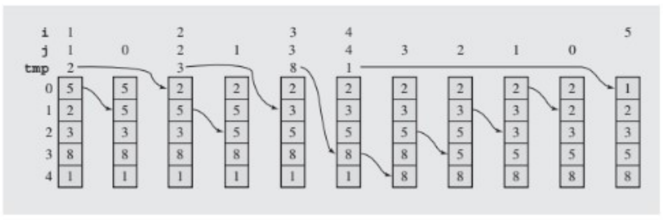
```
template<class T>
void bubblesort(T data[], int n) {
for (int i = 0; i < n-1; i++)
for (int j = n-1; j > i; --j)
if (data[j] < data[j-1])
swap(data[j],data[j-1]);
}
```

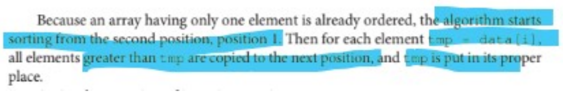
Insertion Sort

Algorithm

```
insertionsort(data[],n)
for i = 1 to n-1
move all elements data[j] greater than data[i] by one position;
place data[i] in its proper position;
```

FIGURE 9.1 The array [5 2 3 8 1] sorted by insertion sort.





Because an array having only one element is already ordered, the algorithm starts sorting from the second position, position 1. Then for each element tmp = data[i], all elements greater than tmp are copied to the next position, and tmp is put in its proper place.

```
template<class T>
void insertionsort(T data[], int n) {
for (int i = 1,j; i < n; i++) {
T tmp = data[i];
for (j = i; j > 0 && tmp < data[j-1]; j--)
data[j] = data[j-1];
data[j] = tmp;
}
}
```

▼ WEEK 5

Advantages of Singly Linked List

1. does not waste memory
2. can grow and shrink during program execution
3. can be made as long as required
4. “on-demand” memory allocation

Disadvantages of Singly Linked List

1. insertion or deletion at middle or end require traversal
2. searching is slower than arrays

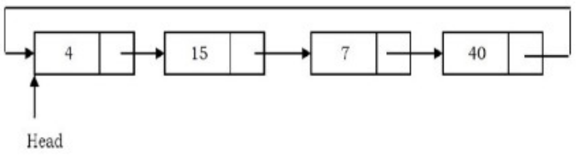
3. not suitable for caching (due to no concept of proximity allocation)

▼ circular linked list

a node which do not have any end is know as circular linked list

next pointer of tail node points to head node

create a loop like structure



▼ insertion

1. at end
    a. `create new node , next ptr of new node points to itself`
    b. `update next ptr of new node to head node`
    c. `taverse the list to tail`
    d. `next ptr of previous node pooints to new node`
2. at beginning
    a. `create new node and keep it’s next pointer pointing to iteself`
    b. `update next ptr of new node pointing to head`
    c. `traverse to tail`
    d. `point tail’s next ptr to new node`
    e. `set new node as head node`
3. at middle

    same as insertion in middle of singly linked list

▼ deletion

1. at beginning
    a. `traverse to tail node`
    b. `tail node is prev of head node . and we wanto to delet head node`
    c. `crate temp node pointing to head`
    d. `update tail nodes next ptr to next node of head`
    e. `delete head node`
    f. `make next node of head node as a new head`

2. at end

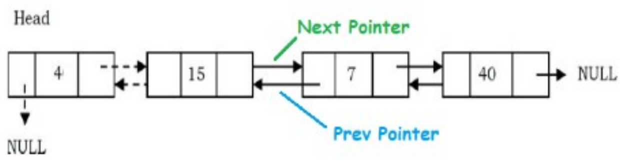
    same as delete from middle in singly linked list
3. at middle

    same as delete from middle in singly linked list

▼ double linked list

also known as two-way linked list(navigate in both directions)

each node has pointer to its successor node as well as predecessor (previous) ptr



▼ insertion

1. at beginning
    a. `update next ptr of new node to current head node`
    b. `make prev ptr of new node NULL`
    c. `update head prev ptr  to point new node as head`
2. at end
    a. `next ptr of new node to NULL`
    b. `prev ptr points to tail node`
    c. `update next ptr of tail to new node`
3. at middle
    a. `next ptr of new node points to next node of position node`
    b. `prev ptr of new node points to position node`
    c. `next ptr of position node points to new node`
    d. `prev ptr of new node’s next node points to new node`

▼ deletion

1. at beginning
    a. `create temp node and point to head node`
    b. `move head nodes next ptr to next node`
    c. `change prev ptr of head node to null`
    d. `delete temp node`
2. at end
    a. `traverse the list and maintain prev node’s address too`
    b. ` we will have two ptr`
        i. `one pointing tail node`
        ii. `another pointing node before tail`
    c. `update next ptr of previuos node to null`
    d. `delete tail node`
3. at middle
    a. `maintain prev node while trversing the list`
    b. `on locating node to be delete,`
        i. `change prev nodes next ptr to next ptr of current node(next node of the deleting node)`

        ii. `change prev ptr of next node to prev node(curr → prev)`

    c. `delete current node`

Advantages of Doubly Linked List

- we can delete a node even if we don't have the previous node's address
    - since each node has prev ptr and can traverse backward
- less expensive travsal

Disadvantages of Doubly Linked List

- each node requires an extra pointer, more memory
- insertion and deletion of a node takes longer due to more pointers

---

Sorting Algorithms

an operation that arranges the elements of a list in a certain order [either ascending or descending]

generally categorized on following parameters

- number of comparisons
- number of swaps
- stability
- online
- in place
- no of shifts

▼ Bubble Sort

working:

it works by iterating the input array from first element to last

comparing each pair and swapping if needed

continues its iteration until no more swaps are needed

time complexity: o(n^2)

Algorithm:

```
begin bubble sort(list)
for all elements of lsit
if list[i]>list[i+1]
    swap(list[i], list[i+1])
    end if
end for
return list
end bubblesort
```

```
void bubblesort(int array[], int size)
{
  for(int step=0; step<size; step++)
{
  for(int i=0;i<size-step;i++)
  {
    if(array[i]>array[i+1])
    {
      int temp=array[i];
      array[i]=array[i+1];
      array[i+1]=temp;
    }
  }
  }
}
```

Bubble sort Algorithm with flag

algorithm

```
begin bubblesort(list)
swapped=false
for all elements of list
    if list[i]>list[i+1]
        swap(list[i], list[i+1])
        swapped =true
    end if
    if(swappped)
    break //breaks outer loop
end for
return list
end bubblesort
```

Analysis

positives:

- algorithm can detect if list is already sorted
- in-place
- stable

best case complexity can be improved from o(n^2) to o(n) by using flags to check swaps(for already sorted list)

▼ Insertion Sort

working:

each iteration removes an element from input data

inserts it into the correct position in the list being sorted

process repeats until all input elements have gone through

time complexity: o(n^2)

Algorithm

```
void insertsort(int array[], int size)
{
  for( int i=1; i<size;i++)
  {
    int key=array[i];
    int j=i-1;
    while(key<array[j]&&j>=0)
    {
      array[j+1]=array[j];
      --j;
    }
    array[j+1]=key;
}
}
```

Analysis:

positives:

- adaptive like bubble sort(detect sorted input)
- stable

- in-place
- online(can sort new inputs)

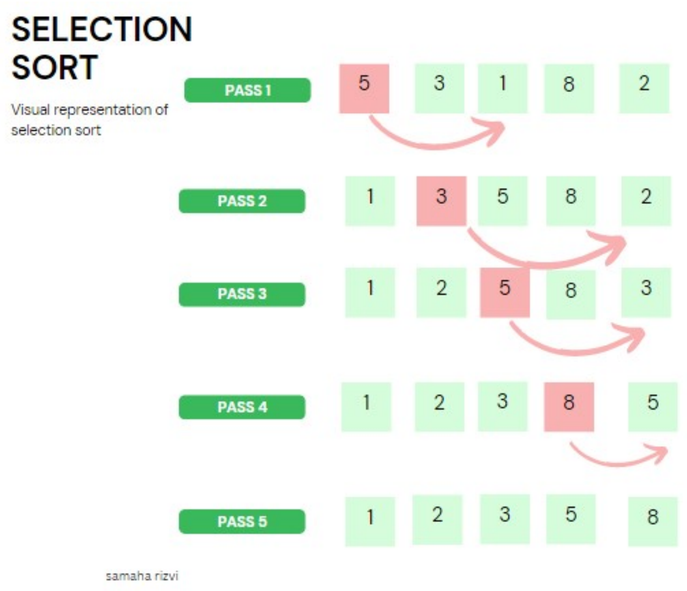
negatives:

only works for small input size

▼ WEEK 7

SELECTION SORT

Selection sort is an attempt to localize the **exchanges of array elements** by finding a **misplaced element** first and **putting it in its final place**. The element with the **lowest value** is selected and **exchanged** with the element in the **first position.**



Algorithm

```
selectionsort(data[],n)
for i = 0 to n-2
select the smallest element among data[i], . . . , data[n-1];
swap it with data[i];
```

```
//code from book
template<class T>
void selectionsort(T data[], int n) {
for (int i = 0,j,least; i < n-1; i++) {
for (j = i+1, least = i; j < n; j++)
if (data[j] < data[least])
least = j;
swap(data[least],data[i]);
}
}
```

```
// Selection sort in C++

#include <iostream>
using namespace std;

// function to swap the the position of two elements
void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}
```

```
// function to print an array
void printArray(int array[], int size) {
  for (int i = 0; i < size; i++) {
    cout << array[i] << " ";
  }
  cout << endl;
}

void selectionSort(int array[], int size) {
  for (int step = 0; step < size - 1; step++) {
    int min_idx = step;
    for (int i = step + 1; i < size; i++) {

      // To sort in descending order, change > to < in this line.
      // Select the minimum element in each loop.
      if (array[i] < array[min_idx])
        min_idx = i;
    }

    // put min at the correct position
    swap(&array[min_idx], &array[step]);
  }
}

// driver code
int main() {
  int data[] = {20, 12, 10, 15, 2,-11};
  int size = sizeof(data) / sizeof(data[0]);
  selectionSort(data, size);
  cout << "Sorted array in Acsending Order:\n";
  printArray(data, size);
}
```

COMB SORT

an improvement over bubble sort

preprocessing the data by
comparing elements step positions away from one another, which is an idea behind
the comb sort. In each pass, step becomes smaller until it is equal to 1

The idea is that large elements are moved toward the end
of the array before proper sorting begins. Here is an implementation:

```
//book code
template<class T>
void combsort(T data[], const int n) {
int step = n, j, k;
while ((step = int(step/1.3)) > 1) // phase 1
for (j = n-1; j >= step; j--) {
k = j-step;
if (data[j] < data[k])
swap(data[j],data[k]);
}
bool again = true;
for (int i = 0; i < n-1 && again; i++) // phase 2
for (j = n-1, again = false; j > i; --j)
if (data[j] < data[j-1]) {
swap(data[j],data[j-1]);
again = true;
}
```

```
// C++ implementation of Comb Sort
#include<bits/stdc++.h>
using namespace std;

// To find gap between elements
int getNextGap(int gap)
{
  // Shrink gap by Shrink factor
  gap = (gap*10)/13;

  if (gap < 1)
    return 1;
  return gap;
}
```

```
// Function to sort a[0..n-1] using Comb Sort
void combSort(int a[], int n)
{
  // Initialize gap
  int gap = n;

  // Initialize swapped as true to make sure that
  // loop runs
  bool swapped = true;

  // Keep running while gap is more than 1 and last
  // iteration caused a swap
  while (gap != 1 || swapped == true)
  {
    // Find next gap
    gap = getNextGap(gap);

    // Initialize swapped as false so that we can
    // check if swap happened or not
    swapped = false;

    // Compare all elements with current gap
    for (int i=0; i<n-gap; i++)
    {
      if (a[i] > a[i+gap])
      {
        swap(a[i], a[i+gap]);
        swapped = true;
      }
    }
  }
}

// Driver program
int main()
{
  int a[] = {8, 4, 1, 56, 3, -44, 23, -6, 28, 0};
  int n = sizeof(a)/sizeof(a[0]);

  combSort(a, n);

  printf("Sorted array: \n");
  for (int i=0; i<n; i++)
    printf("%d ", a[i]);

  return 0;
}
```

SHELL SORT

efficient sorting element

improvement over insertion sort

of Shell sort is an ingenious division of the array data into several subarrays.

The trick is that elements spaced farther apart are compared first, then the elements closer to each other are compared, and so on, until adjacent elements are compared on the last pass. The original array is logically subdivided into subarrays by picking every ith element as part of one subarray.

The core of Shell sort is to divide an array into subarrays by taking elements h positions apart. Three features of this algorithm vary from one implementation to another:

1. The sequence of increments
2. A simple sorting algorithm applied in all passes except the last
3. A simple sorting algorithm applied only in the last pass, for 1-sort

```
//book code
template<class T>
void Shellsort(T data[], int n) {
register int i, j, hCnt, h;
int increments[20], k;
// create an appropriate number of increments h
for (h = 1, i = 0; h < n; i++) {
increments[i] = h;
h = 3*h + 1;
}
// loop on the number of different increments h
for (i--; i >= 0; i--) {
h = increments[i];
// loop on the number of subarrays h-sorted in ith pass
for (hCnt = h; hCnt < 2*h; hCnt++) {
// insertion sort for subarray containing every hth element of
for (j = hCnt; j < n; ) { // array data
T tmp = data[j];
k = j;
while (k-h >= 0 && tmp < data[k-h]) {
data[k] = data[k-h];
k -= h;
}
data[k] = tmp;
j += h;
}
}
}
}
```

```
// C++ implementation of Shell Sort
#include <iostream>
using namespace std;

/* function to sort arr using shellSort */
int shellSort(int arr[], int n)
{
  // Start with a big gap, then reduce the gap
  for (int gap = n/2; gap > 0; gap /= 2)
  {
    // Do a gapped insertion sort for this gap size.
    // The first gap elements a[0..gap-1] are already in gapped order
    // keep adding one more element until the entire array is
    // gap sorted
    for (int i = gap; i < n; i += 1)
    {
      // add a[i] to the elements that have been gap sorted
      // save a[i] in temp and make a hole at position i
      int temp = arr[i];

      // shift earlier gap-sorted elements up until the correct
      // location for a[i] is found
      int j;
      for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
        arr[j] = arr[j - gap];

      // put temp (the original a[i]) in its correct location
      arr[j] = temp;
    }
  }
  return 0;
}

void printArray(int arr[], int n)
{
  for (int i=0; i<n; i++)
    cout << arr[i] << " ";
}

int main()
{
  int arr[] = {12, 34, 54, 2, 3}, i;
  int n = sizeof(arr)/sizeof(arr[0]);

  cout << "Array before sorting: \n";
  printArray(arr, n);

  shellSort(arr, n);

  cout << "\nArray after sorting: \n";
  printArray(arr, n);

  return 0;
}
```

▼ WEEK 7

//searching algorithms

//selection sort

//merge sort

Quick Sort

- divide and conquer approach
- break major problem/array into sub arrays , solve them and combine them

Algorithm

- `pivot: can be any element of array`
    - `best practice is rightmost or leftmost element`
- `compare from right to left`
- `from right scan for smaller element`
- `from left scan for larger element`
- `left ptr ==right ptr → call end`
- `pivot is on its correct position`

Code

```
int quick(int arr[],int low, int high)
{
  if(low<high)
    {
      int pivot=partition(arr,low,high);
      quick(arr,low,piv-1);
      quick(arr,piv+1,high);
      }
  int partition(int arr[],int low, int high)
{
    int pivot=arr[high];
int i=low-1;
for(int j=low;j<high;j++)
{
if(arr[j]<=pivot)
{
  i++;
swap(arr[j],arr[i]);
}
}
swap(arr[i+1],arr[high]);
return (i+1);
}
```

▼ WEEK 8

quick sort

`pivot→ can be any number`

`it can be `

- `random`
- `last`
- `first`
- `medium`

`if stack ≤ pivot →start ++`

`else if end > pivot →end  - -`

`when start has crossed end ,swap with pivot` complexity o(n^2)

```
/* C++ implementation of QuickSort */
#include <bits/stdc++.h>
using namespace std;

// A utility function to swap two elements
void swap(int* a, int* b)
{
  int t = *a;
  *a = *b;
  *b = t;
}

/* 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 left of pivot and all greater elements to right
of pivot */
int partition(int arr[], int low, int high)
{
  int pivot = arr[high]; // pivot
  int i
    = (low
    - 1); // Index of smaller element and indicates
        // the right position of pivot found so far

  for (int 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], &arr[j]);
    }
  }
  swap(&arr[i + 1], &arr[high]);
  return (i + 1);
}

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high)
{
  if (low < high) {
    /* pi is partitioning index, arr[p] is now
    at right place */
    int pi = partition(arr, low, high);

    // Separately sort elements before
    // partition and after partition
    quickSort(arr, low, pi - 1);
    quickSort(arr, pi + 1, high);
  }
}

/* Function to print an array */
void printArray(int arr[], int size)
{
  int i;
  for (i = 0; i < size; i++)
    cout << arr[i] << " ";
  cout << endl;
}

// Driver Code
int main()
{
  int arr[] = { 10, 7, 8, 9, 1, 5 };
  int n = sizeof(arr) / sizeof(arr[0]);
  quickSort(arr, 0, n - 1);
  cout << "Sorted array: \n";
  printArray(arr, n);
  return 0;
}
```

Merge Sort

`reduces bigO complexity of quick sort`

`divide array into subarrays`

`q=(p+r)/2`

`l=p - q`

`R= q+1 - r`

`n1=q- (P+1)`

`n2=1-q`

`l=[1—n1 +1]`

`R=[1—n2+1]`

`if(L[i]<R[j])`

`{`

`a[k]=l[i]`

`i++;}`

`else{`

`a[k]=R[j]`

`j++}`

```
// C++ program for Merge Sort
#include <iostream>
using namespace std;

// Merges two subarrays of array[].
// First subarray is arr[begin..mid]
// Second subarray is arr[mid+1..end]
void merge(int array[], int const left, int const mid,
    int const right)
{
  auto const subArrayOne = mid - left + 1;
  auto const subArrayTwo = right - mid;

  // Create temp arrays
  auto *leftArray = new int[subArrayOne],
    *rightArray = new int[subArrayTwo];

  // Copy data to temp arrays leftArray[] and rightArray[]
  for (auto i = 0; i < subArrayOne; i++)
    leftArray[i] = array[left + i];
  for (auto j = 0; j < subArrayTwo; j++)
    rightArray[j] = array[mid + 1 + j];

  auto indexOfSubArrayOne
    = 0, // Initial index of first sub-array
    indexOfSubArrayTwo
    = 0; // Initial index of second sub-array
  int indexOfMergedArray
    = left; // Initial index of merged array

  // Merge the temp arrays back into array[left..right]
  while (indexOfSubArrayOne < subArrayOne
    && indexOfSubArrayTwo < subArrayTwo) {
    if (leftArray[indexOfSubArrayOne]
      <= rightArray[indexOfSubArrayTwo]) {
      array[indexOfMergedArray]
        = leftArray[indexOfSubArrayOne];
      indexOfSubArrayOne++;
    }
    else {
      array[indexOfMergedArray]
        = rightArray[indexOfSubArrayTwo];
      indexOfSubArrayTwo++;
    }
    indexOfMergedArray++;
  }
  // Copy the remaining elements of
  // left[], if there are any
  while (indexOfSubArrayOne < subArrayOne) {
    array[indexOfMergedArray]
      = leftArray[indexOfSubArrayOne];
    indexOfSubArrayOne++;
    indexOfMergedArray++;
  }
  // Copy the remaining elements of
  // right[], if there are any
  while (indexOfSubArrayTwo < subArrayTwo) {
    array[indexOfMergedArray]
      = rightArray[indexOfSubArrayTwo];
    indexOfSubArrayTwo++;
    indexOfMergedArray++;
  }
  delete[] leftArray;
  delete[] rightArray;
}

// begin is for left index and end is
// right index of the sub-array
// of arr to be sorted */
void mergeSort(int array[], int const begin, int const end)
{
  if (begin >= end)
    return; // Returns recursively

  auto mid = begin + (end - begin) / 2;
  mergeSort(array, begin, mid);
  mergeSort(array, mid + 1, end);
  merge(array, begin, mid, end);
}

// UTILITY FUNCTIONS
// Function to print an array
void printArray(int A[], int size)
{
  for (auto i = 0; i < size; i++)
    cout << A[i] << " ";
}

// Driver code
int main()
{
  int arr[] = { 12, 11, 13, 5, 6, 7 };
  auto arr_size = sizeof(arr) / sizeof(arr[0]);

  cout << "Given array is \n";
  printArray(arr, arr_size);

  mergeSort(arr, 0, arr_size - 1);

  cout << "\nSorted array is \n";
  printArray(arr, arr_size);
  return 0;
}
```

Radix Sort

`improved version of bubble sort`

`compare elemnts in term of place value`

`first compare units, then tens, then hundereds and so on`

`its code it little bit complex, but concept wise it is very simple`

```
// C++ implementation of Radix Sort

#include <iostream>
using namespace std;

// A utility function to get maximum value in arr[]
int getMax(int arr[], int n)
{
  int mx = arr[0];
  for (int i = 1; i < n; i++)
    if (arr[i] > mx)
      mx = arr[i];
  return mx;
}

// A function to do counting sort of arr[] according to
// the digit represented by exp.
void countSort(int arr[], int n, int exp)
{
  int output[n]; // output array
  int i, count[10] = { 0 };

  // Store count of occurrences in count[]
  for (i = 0; i < n; i++)
    count[(arr[i] / exp) % 10]++;

  // Change count[i] so that count[i] now contains actual
  // position of this digit in output[]
  for (i = 1; i < 10; i++)
    count[i] += count[i - 1];
  // Build the output array
  for (i = n - 1; i >= 0; i--) {
    output[count[(arr[i] / exp) % 10] - 1] = arr[i];
    count[(arr[i] / exp) % 10]--;
  }

  // Copy the output array to arr[], so that arr[] now
  // contains sorted numbers according to current digit
  for (i = 0; i < n; i++)
    arr[i] = output[i];
}

// The main function to that sorts arr[] of size n using
// Radix Sort
void radixsort(int arr[], int n)
{
  // Find the maximum number to know number of digits
  int m = getMax(arr, n);

  // Do counting sort for every digit. Note that instead
  // of passing digit number, exp is passed. exp is 10^i
  // where i is current digit number
  for (int exp = 1; m / exp > 0; exp *= 10)
    countSort(arr, n, exp);
}

// A utility function to print an array
void print(int arr[], int n)
{
  for (int i = 0; i < n; i++)
    cout << arr[i] << " ";
}

// Driver Code
int main()
{
  int arr[] = { 170, 45, 75, 90, 802, 24, 2, 66 };
  int n = sizeof(arr) / sizeof(arr[0]);

  // Function Call
  radixsort(arr, n);
  print(arr, n);
  return 0;
}
```

STACKS

Last In First Out (LIFO)

Push() and Pop() through top

```
#include<iostream>
using namespace std;
int MAX=10;
int arr[10],top=-1;
class stack{
  public:
    stack(){
    }
    void push(int val){
      if(top>=MAX-1){
        cout<<"\nStack overflow";
        return;
      }
      else{
        top++;
        arr[top]=val;
      }
    }
    void pop(){
      if(top<=-1){
        cout<<"\nStack is empty!!!\n";
        return;
      }
      else{
        cout<<"\nElement popped!!\n";
        top--;
      }
    }
    void popmid(){
      if(top!=-1){
      int mid=top/2;
      for(int i=mid;i<top;i++){
        int temp=arr[i];
        arr[i]=arr[i+1];
        arr[i+1]=temp;
      }
      pop();
      }
      else{
        cout<<"\nStack is empty!!";
      }

    }
    void display(){
      if(top==-1){
        cout<<"\nStack is empty!!\n";
      }
      else{
        cout<<"\nDisplay stack!!\n";
        for(int i=top;i>=0;i--){
          cout<<arr[i];
          cout<<endl;
        }
      }
    }
};
int main(int argc, char const *argv[]){
  stack ob1;
  ob1.push(3);
  ob1.push(4);
  ob1.push(5);
  ob1.push(6);
  ob1.push(7);
  ob1.display();
  ob1.popmid();
  ob1.display();
  return 0;
}
```

```
#include <iostream>
#include <queue>
using namespace std;

class Stack {

    queue<int> primary_queue, secondary_queue;

    public:
        void push(int element){

            // enqueue in secondary_queue
            secondary_queue.push(element);

            // add elements of primary_queue to secondary_queue
            while(!primary_queue.empty()){
                secondary_queue.push(primary_queue.front());
                primary_queue.pop();
            }

            // swapping the queues
            queue<int> temp_queue = primary_queue;
            primary_queue = secondary_queue;
            secondary_queue = temp_queue;
        }

        void pop(){
            if(primary_queue.empty()){
                return;
            } else {
                primary_queue.pop();
            }
        }

        int top(){
            if(primary_queue.empty()){
                return -1;
            } else {
                return primary_queue.front();
            }
        }

        void displayStack()
        {
            queue<int> temp_queue = primary_queue;

            while(!temp_queue.empty()){
                cout<<temp_queue.front()<<" ";
                temp_queue.pop();
            }
            cout<<"\n";

        }
};

int main(){

    Stack s;

    s.push(1);
    s.push(2);
    s.push(3);
    s.push(4);

    s.displayStack();

    cout<<"Top: "<<s.top()<<"\n";

    s.pop();

    s.displayStack();

    return 0;

}
```

Queue

First In First Out

enqueue() → from rear end

dequeue() → from front end

```
#include<iostream>
using namespace std;
int MAX=10;
int arr[10],front=-1,rear=-1;
class queue{
    public:
  queue(){
  }
  void enQueue(int val){
    if(rear==MAX-1){
      cout<<"\nQueue overflow!!!!\n";
      return;
    }
    if(front==-1){
      rear=0;
      front=0;
    }
    else{
      rear+=1;
    }
    arr[rear]=val;
  }
  void deQueue(){
    if(front==-1){
      cout<<"\nQueue is empty!!!\n";
      return;
    }
    else{
      cout<<"\nElement deleted!!\n";
      front++;
    }
  }
  void reverse(){
    int i,j;
    for(i=front,j=rear;i<j;i++,j--){
      int temp=arr[i];
      arr[i]=arr[j];
      arr[j]=temp;
    }
  }
  void display(){
    if(front==-1){
      cout<<"\nQueue is empty!!\n";
    }
      else{
        cout<<"\nDisplay Queue\n";
        for(int i=front;i<=rear;i++){
          cout<<arr[i]<<" ";
        }
        cout<<endl;
      }

  }
};
int main(int argc, char const *argv[]){
  queue ob1;
  ob1.enQueue(1);
  ob1.enQueue(2);
  ob1.enQueue(3);
  ob1.enQueue(4);
  ob1.enQueue(5);
  ob1.enQueue(6);
  ob1.enQueue(7);
  ob1.enQueue(8);
  ob1.enQueue(9);
  ob1.enQueue(10);
  ob1.display();
  ob1.reverse();
  cout<<"\nReversed queue";
  ob1.display();
  return 0;
}
```

stack using queues

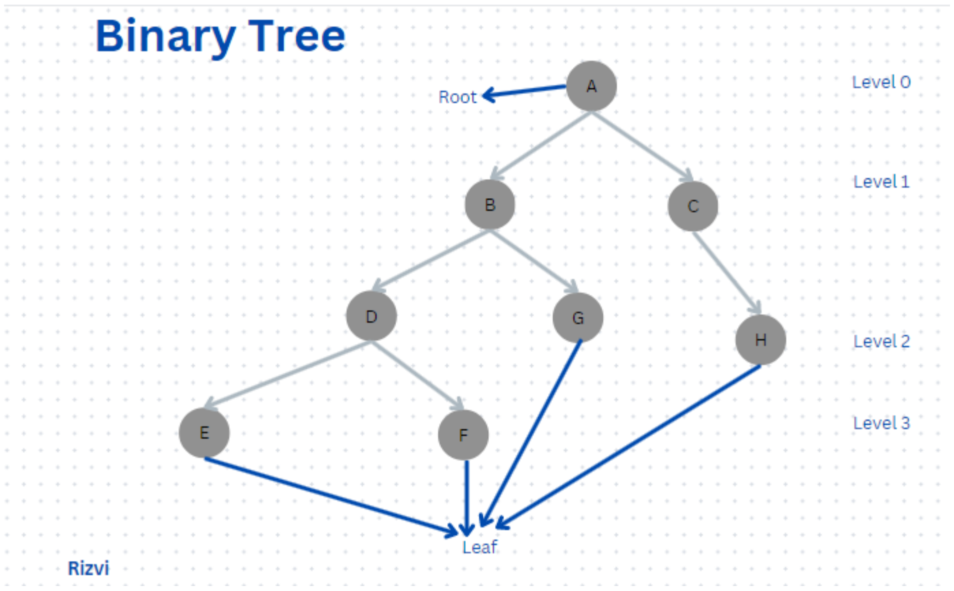
```
#include<iostream>
using namespace std;
int MAX=10;
int ar[10],front=-1,rear=-1;
class queue{

    public:
  int size;
  queue(){
    front=-1;
    rear=-1;
    ar[10]={0};
  }
  void enQueue(int val){
    if(rear==MAX-1){
      cout<<"\nQueue overflow!!!!\n";
      return;
    }
    if(front==-1){
      rear=0;
      front=0;
    }
    else{
      rear+=1;
    }
    ar[rear]=val;
    size=rear;
  }
  int deQueue(){
    int topop;
    if(front==-1){
      cout<<"\nQueue is empty!!!\n";
      return 0;
    }
    else{
      cout<<"\nElement deleted!!\n";
      topop=ar[front];
      front++;
    }
    return topop;
  }
  bool isempty(){
    if(front==-1){
      return 1;
    }
  }
  void reverse(){
    int i,j;
    for(i=front,j=rear;i<j;i++,j--){
      int temp=ar[i];
      ar[i]=ar[j];
      ar[j]=temp;
    }
  }
  void display(){
    if(front==-1){
      cout<<"\nQueue is empty!!\n";
    }
      else{
        cout<<"\nDisplay Queue\n";
        for(int i=front;i<=rear;i++){
          cout<<ar[i]<<" ";
        }
        cout<<endl;
      }

  }
};
void stackpush(queue q1, queue q2,int val){
  if(q1.isempty()){
    q1.enQueue(val);
  }
  else{
    for(int i=0;i<q1.size;i++){
      q2.enQueue(q1.deQueue());
    }
    q1.enQueue(val);
    for(int i=0;i<q1.size;i++){
      q1.enQueue(q2.deQueue());
    }
  }
}
int main(int argc, char const *argv[]){
  queue q1;
  q1.enQueue(5);
  q1.enQueue(6);
  q1.display();
  queue q2;
  stackpush(q1,q2,4);
  q1.display();
  return 0;
}
```

▼ WEEK 9

TREES

Binary Tree

- non-linear data structure
- two child nodes
- each node has unique path
- no child can have two parents



FULL TREE

- every node has two child and all leaf nodes are on same level
- ‘h’ represents height of tree
- max nodes at a level= 2^L
- total nodes of full tree = 2^l1 + 2^l2 +…….2^ln
- leftmost → smallest
- rightmost → largest

Complete Tree

- leaf nodes must on extreme left
- rest is same full tree

Infix/ Postfix/ Prefix expressions

Infix [x+y]

- operators are written in between elements

Postfix [xy+]

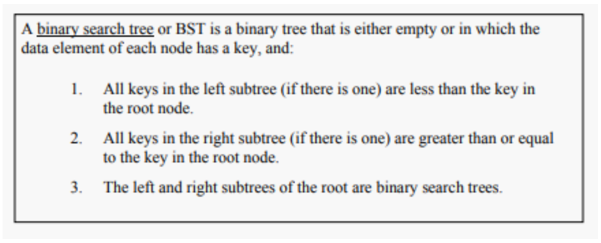
- Reverse Polish notation
- operators are on right

Prefix [+xy]

- operators are at beginning

▼ WEEK 10

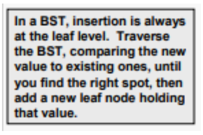
Binary Search Tree


A binary search tree or BST is a binary tree that is either empty or in which the data element of each node has a key, and:

1. All keys in the left subtree (if there is one) are less than the key in the root node.
2. All keys in the right subtree (if there is one) are greater than or equal to the key in the root node.
3. The left and right subtrees of the root are binary search trees.

https://courses.cs.vt.edu/cs2604/spring04/Notes/C03.BinarySearchTrees.pdf

Insertion

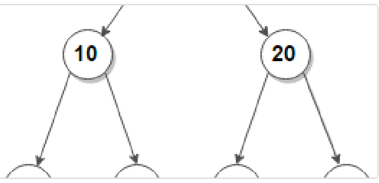

In a BST, insertion is always at the leaf level. Traverse the BST, comparing the new value to existing ones, until you find the right spot, then add a new leaf node holding that value.

```
// C++ program to demonstrate insertion
// in a BST recursively.
#include <iostream>
using namespace std;

class BST {
  int data;
  BST *left, *right;

public:
  // Default constructor.
  BST();

  // Parameterized constructor.
  BST(int);

  // Insert function.
  BST* Insert(BST*, int);

  // Inorder traversal.
  void Inorder(BST*);
};

// Default Constructor definition.
BST ::BST()
  : data(0)
  , left(NULL)
  , right(NULL)
{
}

// Parameterized Constructor definition.
BST ::BST(int value)
{
  data = value;
  left = right = NULL;
}

// Insert function definition.
BST* BST ::Insert(BST* root, int value)
{
  if (!root) {
    // Insert the first node, if root is NULL.
    return new BST(value);
  }

  // Insert data.
  if (value > root->data) {
    // Insert right node data, if the 'value'
    // to be inserted is greater than 'root' node data.

    // Process right nodes.
    root->right = Insert(root->right, value);
  }
  else if (value < root->data){
    // Insert left node data, if the 'value'
    // to be inserted is smaller than 'root' node data.

    // Process left nodes.
    root->left = Insert(root->left, value);
  }

  // Return 'root' node, after insertion.
  return root;
}

// Inorder traversal function.
// This gives data in sorted order.
void BST ::Inorder(BST* root)
{
  if (!root) {
    return;
  }
  Inorder(root->left);
  cout << root->data << endl;
  Inorder(root->right);
}

// Driver code
int main()
{
  BST b, *root = NULL;
  root = b.Insert(root, 50);
  b.Insert(root, 30);
  b.Insert(root, 20);
  b.Insert(root, 40);
  b.Insert(root, 70);
  b.Insert(root, 60);
  b.Insert(root, 80);

  b.Inorder(root);
  return 0;
}

// This code is contributed by pkthapa
```

Insertion in a BST - Iterative and Recursive Solution | Techie Delight

A Binary Search Tree (BST) is a rooted binary tree, whose nodes each store a key (and optionally, an associated value), and each has two distinguished subtrees, commonly denoted left and right. The tree should satisfy the BST property, which states that each node's key must be greater than all keys

https://www.techiedelight.com/insertion-in-bst/



Deletion

- CASE 1: No Child

    Simply remove it from the tree.

- CASE 2: 1 Child

    Copy the child to the node and delete the child

- CASE 3: 2 Child

    Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor.

```
if root==NULL return rppt
if key < root→data, root→lef=(delteenode(root→left, key)
if key > root→data, root→right=(root→right, key)
else
    if root is leaf , return null
    only left child, return left child
    only right child, return right child
    has two child, than set the root as inorder successor, and recursion to delete the node
    return
```

```
// C++ program to demonstrate
// delete operation in binary
// search tree
#include <bits/stdc++.h>
using namespace std;

struct node {
  int key;
  struct node *left, *right;
};

// A utility function to create a new BST node
struct node* newNode(int item)
{
  struct node* temp
    = (struct node*)malloc(sizeof(struct node));
  temp->key = item;
  temp->left = temp->right = NULL;
  return temp;
}

// A utility function to do
// inorder traversal of BST
void inorder(struct node* root)
{
  if (root != NULL) {
    inorder(root->left);
    cout << root->key <<" ";
    inorder(root->right);
  }
}

/* A utility function to
insert a new node with given key in
* BST */
struct node* insert(struct node* node, int key)
{
  /* If the tree is empty, return a new node */
  if (node == NULL)
    return newNode(key);

  /* Otherwise, recur down the tree */
  if (key < node->key)
    node->left = insert(node->left, key);
  else
    node->right = insert(node->right, key);

  /* return the (unchanged) node pointer */
  return node;
}

/* Given a non-empty binary search tree, return the node
with minimum key value found in that tree. Note that the
entire tree does not need to be searched. */
struct node* minValueNode(struct node* node)
{
  struct node* current = node;

  /* loop down to find the leftmost leaf */
  while (current && current->left != NULL)
    current = current->left;

  return current;
}

/* Given a binary search tree and a key, this function
deletes the key and returns the new root */
struct node* deleteNode(struct node* root, int key)
{
  // base case
  if (root == NULL)
    return root;

  // If the key to be deleted is
  // smaller than the root's
  // key, then it lies in left subtree
  if (key < root->key)
    root->left = deleteNode(root->left, key);

  // If the key to be deleted is
  // greater than the root's
  // key, then it lies in right subtree
  else if (key > root->key)
    root->right = deleteNode(root->right, key);
  // if key is same as root's key, then This is the node
  // to be deleted
  else {
    // node has no child
    if (root->left == NULL and root->right == NULL)
      return NULL;

    // node with only one child or no child
    else if (root->left == NULL) {
      struct node* temp = root->right;
      free(root);
      return temp;
    }
    else if (root->right == NULL) {
      struct node* temp = root->left;
      free(root);
      return temp;
    }

    // node with two children: Get the inorder successor
    // (smallest in the right subtree)
    struct node* temp = minValueNode(root->right);

    // Copy the inorder successor's content to this node
    root->key = temp->key;

    // Delete the inorder successor
    root->right = deleteNode(root->right, temp->key);
  }
  return root;
}

// Driver Code
int main()
{
  /* Let us create following BST
      50
    /  \
    30   70
    / \ / \
  20 40 60 80 */
  struct node* root = NULL;
  root = insert(root, 50);
  root = insert(root, 30);
  root = insert(root, 20);
  root = insert(root, 40);
  root = insert(root, 70);
  root = insert(root, 60);
  root = insert(root, 80);

  cout << "Inorder traversal of the given tree \n";
  inorder(root);

  cout << "\nDelete 20\n";
  root = deleteNode(root, 20);
  cout << "Inorder traversal of the modified tree \n";
  inorder(root);

  cout << "\nDelete 30\n";
  root = deleteNode(root, 30);
  cout << "Inorder traversal of the modified tree \n";
  inorder(root);

  cout << "\nDelete 50\n";
  root = deleteNode(root, 50);
  cout << "Inorder traversal of the modified tree \n";
  inorder(root);

  return 0;
}

// This code is contributed by shivanisinghss2110
```

▼ WEEK 12

AVL Trees (Balanced BST)

`balancing factor= height of left subtree- height of right subtree`

`if balancing factor ==0 || 1 ||-1 , tree is balanced, else`

`we have to perform rotations to make it balanced`

`4 cases`

- `left left case`
- `left right case`
- `right right case`
- `right left case`

`for left left, and right  right case, single rotation`

`for left right , and right left case, double rotation`

```
// AVL tree implementation in C++

#include <iostream>
using namespace std;

class Node {
   public:
  int key;
  Node *left;
  Node *right;
  int height;
};

int max(int a, int b);

// Calculate height
int height(Node *N) {
  if (N == NULL)
    return 0;
  return N->height;
}

int max(int a, int b) {
  return (a > b) ? a : b;
}

// New node creation
Node *newNode(int key) {
  Node *node = new Node();
  node->key = key;
  node->left = NULL;
  node->right = NULL;
  node->height = 1;
  return (node);
}

// Rotate right
Node *rightRotate(Node *y) {
  Node *x = y->left;
  Node *T2 = x->right;
  x->right = y;
  y->left = T2;
  y->height = max(height(y->left),
          height(y->right)) +
        1;
  x->height = max(height(x->left),
          height(x->right)) +
        1;
  return x;
}

// Rotate left
Node *leftRotate(Node *x) {
  Node *y = x->right;
  Node *T2 = y->left;
  y->left = x;
  x->right = T2;
  x->height = max(height(x->left),
          height(x->right)) +
        1;
  y->height = max(height(y->left),
          height(y->right)) +
        1;
  return y;
}

// Get the balance factor of each node
int getBalanceFactor(Node *N) {
  if (N == NULL)
    return 0;
  return height(N->left) -
```

```
       height(N->right);
}

// Insert a node
Node *insertNode(Node *node, int key) {
  // Find the correct postion and insert the node
  if (node == NULL)
    return (newNode(key));
  if (key < node->key)
    node->left = insertNode(node->left, key);
  else if (key > node->key)
    node->right = insertNode(node->right, key);
  else
    return node;

  // Update the balance factor of each node and
  // balance the tree
  node->height = 1 + max(height(node->left),
             height(node->right));
  int balanceFactor = getBalanceFactor(node);
  if (balanceFactor > 1) {
    if (key < node->left->key) {
      return rightRotate(node);
    } else if (key > node->left->key) {
      node->left = leftRotate(node->left);
      return rightRotate(node);
    }
  }
  if (balanceFactor < -1) {
    if (key > node->right->key) {
      return leftRotate(node);
    } else if (key < node->right->key) {
      node->right = rightRotate(node->right);
      return leftRotate(node);
    }
  }
  return node;
}

// Node with minimum value
Node *nodeWithMimumValue(Node *node) {
  Node *current = node;
  while (current->left != NULL)
    current = current->left;
  return current;
}

// Delete a node
Node *deleteNode(Node *root, int key) {
  // Find the node and delete it
  if (root == NULL)
    return root;
  if (key < root->key)
    root->left = deleteNode(root->left, key);
  else if (key > root->key)
    root->right = deleteNode(root->right, key);
  else {
    if ((root->left == NULL) ||
      (root->right == NULL)) {
      Node *temp = root->left ? root->left : root->right;
      if (temp == NULL) {
        temp = root;
        root = NULL;
      } else
        *root = *temp;
      free(temp);
    } else {
      Node *temp = nodeWithMimumValue(root->right);
      root->key = temp->key;
      root->right = deleteNode(root->right,
                   temp->key);
    }
  }

  if (root == NULL)
    return root;

  // Update the balance factor of each node and
  // balance the tree
  root->height = 1 + max(height(root->left),
             height(root->right));
  int balanceFactor = getBalanceFactor(root);
  if (balanceFactor > 1) {
    if (getBalanceFactor(root->left) >= 0) {
      return rightRotate(root);
    } else {
      root->left = leftRotate(root->left);
      return rightRotate(root);
    }
  }
  if (balanceFactor < -1) {
    if (getBalanceFactor(root->right) <= 0) {
      return leftRotate(root);
    } else {
      root->right = rightRotate(root->right);
      return leftRotate(root);
    }
  }
  return root;
}

// Print the tree
void printTree(Node *root, string indent, bool last) {
  if (root != nullptr) {
    cout << indent;
    if (last) {
      cout << "R----";
      indent += "   ";
    } else {
      cout << "L----";
      indent += "|  ";
    }
    cout << root->key << endl;
    printTree(root->left, indent, false);
    printTree(root->right, indent, true);
  }
}

int main() {
  Node *root = NULL;
  root = insertNode(root, 33);
  root = insertNode(root, 13);
  root = insertNode(root, 53);
  root = insertNode(root, 9);
  root = insertNode(root, 21);
  root = insertNode(root, 61);
  root = insertNode(root, 8);
  root = insertNode(root, 11);
  printTree(root, "", true);
  root = deleteNode(root, 13);
  cout << "After deleting " << endl;
  printTree(root, "", true);
}
```

▼ WEEK 14

Priority Queues / Heap

priority queue is the queue that is implemented via heap

heap is the tree data structures

heap is always a complete tree

`max heap= parent[i] ≥ a[i]`

`min heap= parent[i] ≤ a[i]`

heapify

the process of creating a heap data structure from binary tree

`create a complete binary tree`

`start from the first index of non-leaf node whose index is given by n/2-1`

`set current element as the largest`

`left child is 2i+1, right child is 2i+2`

`if the left child is greater than i, set the left child as the largest`

`if the right child is greater than i, set the right child as the largest.`

`swap largest with  i`

`repeat steps 3-7 until subtrees are  also heapified`

`for min heap, the parent must be smaller than both child`

Insertion

`trickle up method`

`insert new element at the end of the tree`

`heapify the tree`

Deletion

`trickle down method`

`select the element to be deleted`

`swap with the last element`

`remove the last element`

`heapify the tree`

```
// Max-Heap data structure in C++

#include <iostream>
#include <vector>
using namespace std;

void swap(int *a, int *b)
{
  int temp = *b;
  *b = *a;
  *a = temp;
}
void heapify(vector<int> &hT, int i)
{
  int size = hT.size();
  int largest = i;
  int l = 2 * i + 1;
  int r = 2 * i + 2;
  if (l < size && hT[l] > hT[largest])
    largest = l;
  if (r < size && hT[r] > hT[largest])
    largest = r;

  if (largest != i)
  {
    swap(&hT[i], &hT[largest]);
    heapify(hT, largest);
  }
}
void insert(vector<int> &hT, int newNum)
{
  int size = hT.size();
  if (size == 0)
  {
    hT.push_back(newNum);
  }
  else
  {
    hT.push_back(newNum);
    for (int i = size / 2 - 1; i >= 0; i--)
    {
      heapify(hT, i);
    }
  }
}
void deleteNode(vector<int> &hT, int num)
{
  int size = hT.size();
  int i;
  for (i = 0; i < size; i++)
  {
    if (num == hT[i])
      break;
  }
  swap(&hT[i], &hT[size - 1]);

  hT.pop_back();
  for (int i = size / 2 - 1; i >= 0; i--)
  {
    heapify(hT, i);
  }
}
void printArray(vector<int> &hT)
{
  for (int i = 0; i < hT.size(); ++i)
    cout << hT[i] << " ";
  cout << "\n";
}

int main()
{
  vector<int> heapTree;

  insert(heapTree, 3);
  insert(heapTree, 4);
  insert(heapTree, 9);
  insert(heapTree, 5);
  insert(heapTree, 2);

  cout << "Max-Heap array: ";
  printArray(heapTree);

  deleteNode(heapTree, 4);

  cout << "After deleting an element: ";

  printArray(heapTree);
}
```

▼ WEEK 15

Hash Table

stores the elements in key-value pairs

time complexity: big O(1)

most common hash functions

- k mod (10)
- k mod n
- mid square method
- folding method

whenever two elements are at the same place collision happens

The collision can be avoided by:

- open hashing
    - chaining
- close hashing
    - linear probing
    - quadratic probing
    - double hashing

Chaining

if the hash function produces the same index for multiple elements, these elements are stored in the same index via a doubly-linked list

```
#include <iostream>
const int T_S = 200;
using namespace std;
struct HashTableEntry {
   int v, k;
   HashTableEntry *n;
   HashTableEntry *p;
   HashTableEntry(int k, int v) {
      this->k = k;
      this->v = v;
      this->n = NULL;
   }
};
class HashMapTable {
   public:
      HashTableEntry **ht, **top;
      HashMapTable() {
         ht = new HashTableEntry*[T_S];
         for (int i = 0; i < T_S; i++)
            ht[i] = NULL;
      }
      int HashFunc(int key) {
         return key % T_S;
      }
      void Insert(int k, int v) {
         int hash_v = HashFunc(k);
         HashTableEntry* p = NULL;
         HashTableEntry* en = ht[hash_v];
         while (en!= NULL) {
            p = en;
            en = en->n;
         }
         if (en == NULL) {
            en = new HashTableEntry(k, v);
            if (p == NULL) {
               ht[hash_v] = en;
            } else {
               p->n = en;
            }
         } else {
            en->v = v;
         }
      }
      void Remove(int k) {
         int hash_v = HashFunc(k);
         HashTableEntry* en = ht[hash_v];
         HashTableEntry* p = NULL;
         if (en == NULL || en->k != k) {
            cout<<"No Element found at key "<<k<<endl;
            return;
         }
         while (en->n != NULL) {
            p = en;
            en = en->n;
         }
         if (p != NULL) {
            p->n = en->n;
         }
         delete en;
         cout<<"Element Deleted"<<endl;
      }
      void SearchKey(int k) {
         int hash_v = HashFunc(k);
         bool flag = false;
         HashTableEntry* en = ht[hash_v];
         if (en != NULL) {
            while (en != NULL) {
               if (en->k == k) {
                  flag = true;
               }
               if (flag) {
                  cout<<"Element found at key "<<k<<": ";
                  cout<<en->v<<endl;
               }
               en = en->n;
            }
         }
         if (!flag)
            cout<<"No Element found at key "<<k<<endl;
      }
      ~HashMapTable() {
         delete [] ht;
      }
};
int main() {
   HashMapTable hash;
   int k, v;
   int c;
   while (1) {
      cout<<"1.Insert element into the table"<<endl;
      cout<<"2.Search element from the key"<<endl;
      cout<<"3.Delete element at a key"<<endl;
      cout<<"4.Exit"<<endl;
      cout<<"Enter your choice: ";
      cin>>c;
      switch(c) {
         case 1:
            cout<<"Enter element to be inserted: ";
            cin>>v;
            cout<<"Enter key at which element to be inserted: ";
            cin>>k;
            hash.Insert(k, v);
         break;
         case 2:
            cout<<"Enter key of the element to be searched: ";
            cin>>k;
            hash.SearchKey(k);
         break;
         case 3:
            cout<<"Enter key of the element to be deleted: ";
            cin>>k;
            hash.Remove(k);
         break;
         case 4:
            exit(1);
         default:
            cout<<"\nEnter correct option\n";
      }
   }
   return 0;
}
```

Linear Probing

`Collision is resolved by checking next slot`

`h(k, i) = (h’ (k) + i) mod m`

`If a collision occurs at h(k, 0)`
`, then h(k, 1)`
` is checked. In this way, the value of i`
` is incremented linearly`

```
#include <bits/stdc++.h>
using namespace std;

// template for generic type
template <typename K, typename V>

// Hashnode class
class HashNode {
public:
  V value;
  K key;

  // Constructor of hashnode
  HashNode(K key, V value)
  {
    this->value = value;
    this->key = key;
  }
};

// template for generic type
template <typename K, typename V>

// Our own Hashmap class
class HashMap {
  // hash element array
  HashNode<K, V>** arr;
  int capacity;
  // current size
  int size;
  // dummy node
  HashNode<K, V>* dummy;

public:
  HashMap()
  {
    // Initial capacity of hash array
    capacity = 20;
    size = 0;
    arr = new HashNode<K, V>*[capacity];

    // Initialise all elements of array as NULL
    for (int i = 0; i < capacity; i++)
      arr[i] = NULL;

    // dummy node with value and key -1
    dummy = new HashNode<K, V>(-1, -1);
  }
  // This implements hash function to find index
  // for a key
  int hashCode(K key)
  {
    return key % capacity;
  }

  // Function to add key value pair
  void insertNode(K key, V value)
  {
    HashNode<K, V>* temp = new HashNode<K, V>(key, value);

    // Apply hash function to find index for given key
    int hashIndex = hashCode(key);

    // find next free space
    while (arr[hashIndex] != NULL
      && arr[hashIndex]->key != key
      && arr[hashIndex]->key != -1) {
      hashIndex++;
      hashIndex %= capacity;
    }

    // if new node to be inserted
    // increase the current size
    if (arr[hashIndex] == NULL
      || arr[hashIndex]->key == -1)
      size++;
    arr[hashIndex] = temp;
  }

  // Function to delete a key value pair
  V deleteNode(int key)
  {
    // Apply hash function
    // to find index for given key
    int hashIndex = hashCode(key);

    // finding the node with given key
    while (arr[hashIndex] != NULL) {
      // if node found
      if (arr[hashIndex]->key == key) {
        HashNode<K, V>* temp = arr[hashIndex];

        // Insert dummy node here for further use
        arr[hashIndex] = dummy;

        // Reduce size
        size--;
        return temp->value;
      }
      hashIndex++;
      hashIndex %= capacity;
    }

    // If not found return null
    return NULL;
  }

  // Function to search the value for a given key
  V get(int key)
  {
    // Apply hash function to find index for given key
    int hashIndex = hashCode(key);
    int counter = 0;

    // finding the node with given key
    while (arr[hashIndex] != NULL) { // int counter =0; // BUG!

      if (counter++ > capacity) // to avoid infinite loop
        return NULL;

      // if node found return its value
      if (arr[hashIndex]->key == key)
        return arr[hashIndex]->value;
      hashIndex++;
      hashIndex %= capacity;
    }

    // If not found return null
    return NULL;
  }

  // Return current size
  int sizeofMap()
  {
    return size;
  }
  // Return true if size is 0
  bool isEmpty()
  {
    return size == 0;
  }

  // Function to display the stored key value pairs
  void display()
  {
    for (int i = 0; i < capacity; i++) {
      if (arr[i] != NULL && arr[i]->key != -1)
        cout << "key = " << arr[i]->key
          << " value = "
          << arr[i]->value << endl;
    }
  }
};

// Driver method to test map class
int main()
{
  HashMap<int, int>* h = new HashMap<int, int>;
  h->insertNode(1, 1);
  h->insertNode(2, 2);
  h->insertNode(2, 3);
  h->display();
  cout << h->sizeofMap() << endl;
  cout << h->deleteNode(2) << endl;
  cout << h->sizeofMap() << endl;
  cout << h->isEmpty() << endl;
  cout << h->get(2);

  return 0;
}
```

Quadratic Probing

work in the same way as linear probing but the spacing between the slots is slightly increases

`h(k, i) = (h'(k) + c1i + c2i2) mod m`

```
// C++ implementation of
// the Quadratic Probing
#include <bits/stdc++.h>
using namespace std;

// Function to print an array
void printArray(int arr[], int n)
{
  // Iterating and printing the array
  for (int i = 0; i < n; i++) {
    cout << arr[i] << " ";
  }
}

// Function to implement the
// quadratic probing
void hashing(int table[], int tsize, int arr[], int N)
{
  // Iterating through the array
  for (int i = 0; i < N; i++) {
    // Computing the hash value
    int hv = arr[i] % tsize;

    // Insert in the table if there
    // is no collision
    if (table[hv] == -1)
      table[hv] = arr[i];
    else {
      // If there is a collision
      // iterating through all
      // possible quadratic values
      for (int j = 0; j < tsize; j++) {
        // Computing the new hash value
        int t = (hv + j * j) % tsize;
        if (table[t] == -1) {
          // Break the loop after
          // inserting the value
          // in the table
          table[t] = arr[i];
          break;
        }
      }
    }
  }
  printArray(table, N);
}

// Driver code
int main()
{
  int arr[] = { 50, 700, 76, 85, 92, 73, 101 };
  int N = 7;

  // Size of the hash table
  int L = 7;
  int hash_table[7];

  // Initializing the hash table
  for (int i = 0; i < L; i++) {
    hash_table[i] = -1;
  }

  // Function call
  hashing(hash_table, L, arr, N);
  return 0;
}
```

Double hashing

After applying hash function h(k), if a collision occurs, another hash function is generated to determine the next slot.

`h(k, i) = (h1(k) + ih2(k)) mod m`

```
/*
** Handling of collision via open addressing
** Method for Probing: Double Hashing
*/

#include <iostream>
#include <vector>
#include <bitset>
using namespace std;
#define MAX_SIZE 10000001ll

class doubleHash {

  int TABLE_SIZE, keysPresent, PRIME;
  vector<int> hashTable;
  bitset<MAX_SIZE> isPrime;

  /* Function to set sieve of Eratosthenes. */
  void __setSieve(){
    isPrime[0] = isPrime[1] = 1;
    for(long long i = 2; i*i <= MAX_SIZE; i++)
      if(isPrime[i] == 0)
        for(long long j = i*i; j <= MAX_SIZE; j += i)
          isPrime[j] = 1;

  }

  int inline hash1(int value){
    return value%TABLE_SIZE;
  }

  int inline hash2(int value){
    return PRIME - (value%PRIME);
  }

  bool inline isFull(){
    return (TABLE_SIZE == keysPresent);
  }

  public:

  doubleHash(int n){
    __setSieve();
    TABLE_SIZE = n;

    /* Find the largest prime number smaller than hash table's size. */
    PRIME = TABLE_SIZE - 1;
    while(isPrime[PRIME] == 1)
      PRIME--;
    keysPresent = 0;

    /* Fill the hash table with -1 (empty entries). */
    for(int i = 0; i < TABLE_SIZE; i++)
      hashTable.push_back(-1);
  }

  void __printPrime(long long n){
    for(long long i = 0; i <= n; i++)
      if(isPrime[i] == 0)
        cout<<i<<", ";
    cout<<endl;
  }

  /* Function to insert value in hash table */
  void insert(int value){

    if(value == -1 || value == -2){
      cout<<("ERROR : -1 and -2 can't be inserted in the table\n");
    }

    if(isFull()){
      cout<<("ERROR : Hash Table Full\n");
      return;
    }

    int probe = hash1(value), offset = hash2(value); // in linear probing offset = 1;

    while(hashTable[probe] != -1){
      if(-2 == hashTable[probe])
        break;                 // insert at deleted element's location
      probe = (probe+offset) % TABLE_SIZE;
    }

    hashTable[probe] = value;
    keysPresent += 1;
  }

  void erase(int value){
    /* Return if element is not present */
    if(!search(value))
      return;

    int probe = hash1(value), offset = hash2(value);

    while(hashTable[probe] != -1)
      if(hashTable[probe] == value){
        hashTable[probe] = -2;     // mark element as deleted (rather than unvisited(-1)).
        keysPresent--;
        return;
      }
      else
        probe = (probe + offset) % TABLE_SIZE;

  }

  bool search(int value){
    int probe = hash1(value), offset = hash2(value), initialPos = probe;
    bool firstItr = true;

    while(1){
      if(hashTable[probe] == -1)         // Stop search if -1 is encountered.
        break;
      else if(hashTable[probe] == value)     // Stop search after finding the element.
        return true;
      else if(probe == initialPos && !firstItr) // Stop search if one complete traversal of hash table is completed.
        return false;
      else
        probe = ((probe + offset) % TABLE_SIZE); // if none of the above cases occur then update the index and check at it.

      firstItr = false;
    }
    return false;
  }

  /* Function to display the hash table. */
  void print(){
    for(int i = 0; i < TABLE_SIZE; i++)
      cout<<hashTable[i]<<", ";
    cout<<"\n";
  }

};
```

```
int main(){
  doubleHash myHash(13); // creates an empty hash table of size 13

  /* Inserts random element in the hash table */

  int insertions[] = {115, 12, 87, 66, 123},
    n1 = sizeof(insertions)/sizeof(insertions[0]);

  for(int i = 0; i < n1; i++)
    myHash.insert(insertions[i]);

  cout<< "Status of hash table after initial insertions : "; myHash.print();


  /*
  ** Searches for random element in the hash table,
  ** and prints them if found.
  */

  int queries[] = {1, 12, 2, 3, 69, 88, 115},
    n2 = sizeof(queries)/sizeof(queries[0]);

  cout<<"\n"<<"Search operation after insertion : \n";

  for(int i = 0; i < n2; i++)
    if(myHash.search(queries[i]))
      cout<<queries[i]<<" present\n";


  /* Deletes random element from the hash table. */

  int deletions[] = {123, 87, 66},
    n3 = sizeof(deletions)/sizeof(deletions[0]);

  for(int i = 0; i < n3; i++)
    myHash.erase(deletions[i]);

  cout<< "Status of hash table after deleting elements : "; myHash.print();

  return 0;
}
```

▼ WEEK 16

Graphs

- Directed
  - edges in graph have no direction
- Undirected
  - edges in graph have directions
  - also known as digraph

💡 trees are special cases of graph which have no loops , and unique path to every node

- Complete Graph
  - graph in which every vertex is connected to every other vertex

💡 no. of edges in complete directed graph is : N*(N-1)
no. of edges in complete undirected graph is : N*(N-1)/2

- Weighted Graph
  - each edge of graph carries certain value/cost

Graph Using 2D-Array (adjacency Matrix)

- good for dense graph
- connectivity can be tested quickly

```
#include<iostream>
#include<iomanip>

using namespace std;

// A function to print the matrix.
void PrintMat(int **mat, int n)
{
  int i, j;

  cout<<"\n\n"<<setw(4)<<"";
  for(i = 0; i < n; i++)
    cout<<setw(3)<<"("<<i+1<<")";
  cout<<"\n\n";

  // Print 1 if the corresponding vertexes are connected otherwise 0.
  for(i = 0; i < n; i++)
  {
    cout<<setw(3)<<"("<<i+1<<")";
    for(j = 0; j < n; j++)
    {
      cout<<setw(4)<<mat[i][j];
    }
    cout<<"\n\n";
  }
}

int main()
{
  int i, v, e, j, v1, v2;

  // take the input of the number of edges.
  cout<<"Enter the number of vertexes of the graph: ";
  cin>>v;

  // Dynamically declare graph.
  int **graph;
  graph = new int*[v];

  for(i = 0; i < v; i++)
  {
    graph[i] = new int[v];
    for(j = 0; j < v; j++)graph[i][j] = 0;
  }

  cout<<"\nEnter the number of edges of the graph: ";
  cin>>e;

  // Take the input of the adjacent vertex pairs of the given graph.
  for(i = 0; i < e; i++)
  {
    cout<<"\nEnter the vertex pair for edge "<<i+1;
    cout<<"\nV(1): ";
    cin>>v1;
    cout<<"V(2): ";
    cin>>v2;

    graph[v1-1][v2-1] = 1;
    graph[v2-1][v1-1] = 1;
  }

  // Print the 2D array representation of the graph.
  PrintMat(graph, v);
}
```

Graph Using Linked-List (adjacency List)

- good for sparse graph
- adjacent vertices can be found quickly

```
#include<iostream>

using namespace std;

int main()
```

```
{
  int i, v, e, j, count;

  // take the input of the number of vertex and edges.
  cout<<"Enter the number of vertexes of the graph: ";
  cin>>v;
  cout<<"\nEnter the number of edges of the graph: ";
  cin>>e;
  int edge[e][2];

  // Take the input of the adjacent vertex pairs of the given graph.
  for(i = 0; i < e; i++)
  {
    cout<<"\nEnter the vertex pair for edge "<<i+1;
    cout<<"\nV(1): ";
    cin>>edge[i][0];
    cout<<"V(2): ";
    cin>>edge[i][1];
  }

  // Print the adjacency list representation of the graph.
  cout<<"\n\nThe adjacency list representation for the given graph: ";
  for(i = 0; i < v; i++)
  {
    count = 0;
    // For each vertex print, its adjacent vertex.
    cout<<"\n\t"<<i+1<<"-> { ";
    for(j = 0; j < e; j++)
    {
      if(edge[j][0] == i+1)
      {
        cout<<edge[j][1]<<"  ";
        count++;
      }
      else if(edge[j][1] == i+1)
      {
        cout<<edge[j][0]<<"  ";
        count++;
      }
      else if(j == e-1 && count == 0)
        cout<<"Isolated Vertex!";
    }
    cout<<" }";
  }
}
```

Searching in Graph

Breadth-First Search (BFS)

implemented via queue

look at all adjacent paths at same depth before going further in depth

```
template<class VertexType>

void BreadthFirtsSearch(GraphType<VertexType> graph,
VertexType startVertex, VertexType endVertex);

{

QueType<VertexType> queue;

QueType<VertexType> vertexQ;//

bool found = false;

VertexType vertex;

VertexType item;

graph.ClearMarks();

queue.Enqueue(startVertex);

do {

queue.Dequeue(vertex);

if(vertex == endVertex)
found = true;

else {

if(!graph.IsMarked(vertex)) {

graph.MarkVertex(vertex);

graph.GetToVertices(vertex, vertexQ);

while(!vertxQ.IsEmpty()) {

vertexQ.Dequeue(item);

if(!graph.IsMarked(item))

queue.Enqueue(item);

}

}

}

} while (!queue.IsEmpty() && !found);

if(!found)

cout << "Path not found" << endl;

}
```

Depth-First Search (DFS)

implemented via stack

traverse to depth first

```
template <class ItemType>

void DepthFirstSearch(GraphType<VertexType> graph,
VertexType startVertex, VertexType endVertex)

{

StackType<VertexType> stack;

QueType<VertexType> vertexQ;

bool found = false;

VertexType vertex;

VertexType item;

graph.ClearMarks();

stack.Push(startVertex);

do {

stack.Pop(vertex);

if(vertex == endVertex)

found = true;


else {

if(!graph.IsMarked(vertex)) {

graph.MarkVertex(vertex);

graph.GetToVertices(vertex, vertexQ);

while(!vertexQ.IsEmpty()) {
```

```
vertexQ.Dequeue(item);

if(!graph.IsMarked(item))

stack.Push(item);

}

}

} while(!stack.IsEmpty() && !found);

if(!found)

cout << "Path not found" << endl;

}


template<class VertexType>

void GraphType<VertexType>::GetToVertices(VertexType vertex,

QueTye<VertexType>& adjvertexQ)

{

int fromIndex;

int toIndex;

fromIndex = IndexIs(vertices, vertex);

for(toIndex = 0; toIndex < numVertices; toIndex++)

if(edges[fromIndex][toIndex] != NULL_EDGE)

adjvertexQ.Enqueue(vertices[toIndex]);

}
```

Single Source Shortest Path Problem

Dijkstra’s Algorithm

```
// C++ program for Dijkstra's single source shortest path
// algorithm. The program is for adjacency matrix
// representation of the graph
#include <iostream>
using namespace std;
#include <limits.h>

// Number of vertices in the graph
#define V 9

// A utility function to find the vertex with minimum
// distance value, from the set of vertices not yet included
// in shortest path tree
int minDistance(int dist[], bool sptSet[])
{

  // Initialize min value
  int min = INT_MAX, min_index;

  for (int v = 0; v < V; v++)
    if (sptSet[v] == false && dist[v] <= min)
      min = dist[v], min_index = v;

  return min_index;
}

// A utility function to print the constructed distance
// array
void printSolution(int dist[])
{
  cout << "Vertex \t Distance from Source" << endl;
  for (int i = 0; i < V; i++)
    cout << i << " \t\t\t\t" << dist[i] << endl;
}
```

```
// Function that implements Dijkstra's single source
// shortest path algorithm for a graph represented using
// adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
  int dist[V]; // The output array. dist[i] will hold the
        // shortest
  // distance from src to i

  bool sptSet[V]; // sptSet[i] will be true if vertex i is
          // included in shortest
  // path tree or shortest distance from src to i is
  // finalized

  // Initialize all distances as INFINITE and stpSet[] as
  // false
  for (int i = 0; i < V; i++)
    dist[i] = INT_MAX, sptSet[i] = false;

  // Distance of source vertex from itself is always 0
  dist[src] = 0;

  // Find shortest path for all vertices
  for (int count = 0; count < V - 1; count++) {
    // Pick the minimum distance vertex from the set of
    // vertices not yet processed. u is always equal to
    // src in the first iteration.
    int u = minDistance(dist, sptSet);

    // Mark the picked vertex as processed
    sptSet[u] = true;

    // Update dist value of the adjacent vertices of the
    // picked vertex.
    for (int v = 0; v < V; v++)

      // Update dist[v] only if is not in sptSet,
      // there is an edge from u to v, and total
      // weight of path from src to v through u is
      // smaller than current value of dist[v]
      if (!sptSet[v] && graph[u][v]
        && dist[u] != INT_MAX
        && dist[u] + graph[u][v] < dist[v])
        dist[v] = dist[u] + graph[u][v];
  }

  // print the constructed distance array
  printSolution(dist);
}

// driver's code
int main()
{

  /* Let us create the example graph discussed above */
  int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
            { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
            { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
            { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
            { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
            { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
            { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
            { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
            { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

  // Function call
  dijkstra(graph, 0);

  return 0;
}

// This code is contributed by shivanisinghss2110
```

Bellman-Ford Algorithm

Floyd Warshall Algorithm

```
// C++ Program for Floyd Warshall Algorithm
#include <bits/stdc++.h>
using namespace std;
```

```
// Number of vertices in the graph
#define V 4

/* Define Infinite as a large enough
value.This value will be used for
vertices not connected to each other */
#define INF 99999

// A function to print the solution matrix
void printSolution(int dist[][V]);

// Solves the all-pairs shortest path
// problem using Floyd Warshall algorithm
void floydWarshall(int dist[][V])
{

  int i, j, k;

  /* Add all vertices one by one to
  the set of intermediate vertices.
  ---> Before start of an iteration,
  we have shortest distances between all
  pairs of vertices such that the
  shortest distances consider only the
  vertices in set {0, 1, 2, .. k-1} as
  intermediate vertices.
  ----> After the end of an iteration,
  vertex no. k is added to the set of
  intermediate vertices and the set becomes {0, 1, 2, ..
  k} */
  for (k = 0; k < V; k++) {
    // Pick all vertices as source one by one
    for (i = 0; i < V; i++) {
      // Pick all vertices as destination for the
      // above picked source
      for (j = 0; j < V; j++) {
        // If vertex k is on the shortest path from
        // i to j, then update the value of
        // dist[i][j]
        if (dist[i][j] > (dist[i][k] + dist[k][j])
          && (dist[k][j] != INF
            && dist[i][k] != INF))
          dist[i][j] = dist[i][k] + dist[k][j];
      }
    }
  }

  // Print the shortest distance matrix
  printSolution(dist);
}

/* A utility function to print solution */
void printSolution(int dist[][V])
{
  cout << "The following matrix shows the shortest "
      "distances"
      " between every pair of vertices \n";
  for (int i = 0; i < V; i++) {
    for (int j = 0; j < V; j++) {
      if (dist[i][j] == INF)
        cout << "INF"
          << " ";
      else
        cout << dist[i][j] << " ";
    }
    cout << endl;
  }
}

// Driver's code
int main()
{
  /* Let us create the following weighted graph
      10
  (0)------->(3)
    |  /|\
  5 |  |
    |  | 1
  \|/  |
  (1)------->(2)
      3  */
  int graph[V][V] = { { 0, 5, INF, 10 },
            { INF, 0, 3, INF },
            { INF, INF, 0, 1 },
            { INF, INF, INF, 0 } };
  // Function call
  floydWarshall(graph);
  return 0;
}
```

Greedy Algorithms For Minimum Spanning Tree

Prims Algorithm

```
// A C++ program for Prim's Minimum
// Spanning Tree (MST) algorithm. The program is
// for adjacency matrix representation of the graph
#include <bits/stdc++.h>
using namespace std;

// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST
int minKey(int key[], bool mstSet[])
{
  // Initialize min value
  int min = INT_MAX, min_index;

  for (int v = 0; v < V; v++)
    if (mstSet[v] == false && key[v] < min)
      min = key[v], min_index = v;

  return min_index;
}

// A utility function to print the
// constructed MST stored in parent[]
void printMST(int parent[], int graph[V][V])
{
  cout << "Edge \tWeight\n";
  for (int i = 1; i < V; i++)
    cout << parent[i] << " - " << i << " \t"
      << graph[i][parent[i]] << " \n";
}

// Function to construct and print MST for
// a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
  // Array to store constructed MST
  int parent[V];

  // Key values used to pick minimum weight edge in cut
  int key[V];

  // To represent set of vertices included in MST
  bool mstSet[V];

  // Initialize all keys as INFINITE
  for (int i = 0; i < V; i++)
    key[i] = INT_MAX, mstSet[i] = false;

  // Always include first 1st vertex in MST.
  // Make key 0 so that this vertex is picked as first
  // vertex.
  key[0] = 0;
  parent[0] = -1; // First node is always root of MST

  // The MST will have V vertices
  for (int count = 0; count < V - 1; count++) {
    // Pick the minimum key vertex from the
    // set of vertices not yet included in MST
    int u = minKey(key, mstSet);

    // Add the picked vertex to the MST Set
    mstSet[u] = true;

    // Update key value and parent index of
    // the adjacent vertices of the picked vertex.
    // Consider only those vertices which are not
    // yet included in MST
    for (int v = 0; v < V; v++)

      // graph[u][v] is non zero only for adjacent
      // vertices of m mstSet[v] is false for vertices
      // not yet included in MST Update the key only
      // if graph[u][v] is smaller than key[v]
      if (graph[u][v] && mstSet[v] == false
        && graph[u][v] < key[v])
        parent[v] = u, key[v] = graph[u][v];
  }

  // print the constructed MST
  printMST(parent, graph);
}

// Driver's code
int main()
{
  /* Let us create the following graph
    2 3
  (0)--(1)--(2)
  | / \ |
  6| 8/ \5 |7
  | / \ |
  (3)-------(4)
      9  */
  int graph[V][V] = { { 0, 2, 0, 6, 0 },
            { 2, 0, 3, 8, 5 },
            { 0, 3, 0, 0, 7 },
            { 6, 8, 0, 0, 9 },
            { 0, 5, 7, 9, 0 } };

  // Print the solution
  primMST(graph);

  return 0;
}
```

Kruskal’s Algorithm

```
// Kruskal's algorithm in C++

#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

#define edge pair<int, int>

class Graph {
   private:
  vector<pair<int, edge> > G;  // graph
  vector<pair<int, edge> > T;  // mst
  int *parent;
  int V;  // number of vertices/nodes in graph
   public:
  Graph(int V);
  void AddWeightedEdge(int u, int v, int w);
  int find_set(int i);
  void union_set(int u, int v);
  void kruskal();
  void print();
};
Graph::Graph(int V) {
  parent = new int[V];

  //i 0 1 2 3 4 5
  //parent[i] 0 1 2 3 4 5
  for (int i = 0; i < V; i++)
    parent[i] = i;

  G.clear();
  T.clear();
}
void Graph::AddWeightedEdge(int u, int v, int w) {
  G.push_back(make_pair(w, edge(u, v)));
}
int Graph::find_set(int i) {
  // If i is the parent of itself
  if (i == parent[i])
    return i;
  else
```

```
    // Else if i is not the parent of itself
    // Then i is not the representative of his set,
    // so we recursively call Find on its parent
    return find_set(parent[i]);
}

void Graph::union_set(int u, int v) {
  parent[u] = parent[v];
}
void Graph::kruskal() {
  int i, uRep, vRep;
  sort(G.begin(), G.end());  // increasing weight
  for (i = 0; i < G.size(); i++) {
    uRep = find_set(G[i].second.first);
    vRep = find_set(G[i].second.second);
    if (uRep != vRep) {
      T.push_back(G[i]);  // add to tree
      union_set(uRep, vRep);
    }
  }
}
void Graph::print() {
  cout << "Edge :"
     << " Weight" << endl;
  for (int i = 0; i < T.size(); i++) {
    cout << T[i].second.first << " - " << T[i].second.second << " : "
       << T[i].first;
    cout << endl;
  }
}
int main() {
  Graph g(6);
  g.AddWeightedEdge(0, 1, 4);
  g.AddWeightedEdge(0, 2, 4);
  g.AddWeightedEdge(1, 2, 2);
  g.AddWeightedEdge(1, 0, 4);
  g.AddWeightedEdge(2, 0, 4);
  g.AddWeightedEdge(2, 1, 2);
  g.AddWeightedEdge(2, 3, 3);
  g.AddWeightedEdge(2, 5, 2);
  g.AddWeightedEdge(2, 4, 4);
  g.AddWeightedEdge(3, 2, 3);
  g.AddWeightedEdge(3, 4, 3);
  g.AddWeightedEdge(4, 2, 4);
  g.AddWeightedEdge(4, 3, 3);
  g.AddWeightedEdge(5, 2, 2);
  g.AddWeightedEdge(5, 4, 3);
  g.kruskal();
  g.print();
  return 0;
}
```

Topological Sort

soritng algorithm for graphs

```
// A C++ program to print topological sorting of a DAG
#include <iostream>
#include <list>
#include <stack>
using namespace std;

// Class to represent a graph
class Graph {
  int V; // No. of vertices'

  // Pointer to an array containing adjacency listsList
  list<int>* adj;

  // A function used by topologicalSort
  void topologicalSortUtil(int v, bool visited[], stack<int>& Stack);

public:
  Graph(int V); // Constructor

  // function to add an edge to graph
  void addEdge(int v, int w);

  // prints a Topological Sort of the complete graph
  void topologicalSort();
};
```

```
Graph::Graph(int V)
{
  this->V = V;
  adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
  adj[v].push_back(w); // Add w to v’s list.
}

// A recursive function used by topologicalSort
void Graph::topologicalSortUtil(int v, bool visited[],
                stack<int>& Stack)
{
  // Mark the current node as visited.
  visited[v] = true;

  // Recur for all the vertices adjacent to this vertex
  list<int>::iterator i;
  for (i = adj[v].begin(); i != adj[v].end(); ++i)
    if (!visited[*i])
      topologicalSortUtil(*i, visited, Stack);

  // Push current vertex to stack which stores result
  Stack.push(v);
}

// The function to do Topological Sort. It uses recursive
// topologicalSortUtil()
void Graph::topologicalSort()
{
  stack<int> Stack;

  // Mark all the vertices as not visited
  bool* visited = new bool[V];
  for (int i = 0; i < V; i++)
    visited[i] = false;

  // Call the recursive helper function to store Topological
  // Sort starting from all vertices one by one
  for (int i = 0; i < V; i++)
    if (visited[i] == false)
      topologicalSortUtil(i, visited, Stack);

  // Print contents of stack
  while (Stack.empty() == false) {
    cout << Stack.top() << " ";
    Stack.pop();
  }
}

// Driver program to test above functions
int main()
{
  // Create a graph given in the above diagram
  Graph g(6);
  g.addEdge(5, 2);
  g.addEdge(5, 0);
  g.addEdge(4, 0);
  g.addEdge(4, 1);
  g.addEdge(2, 3);
  g.addEdge(3, 1);

  cout << "Following is a Topological Sort of the given graph  ";
  g.topologicalSort();

  return 0;
}
```

▼ important

1. MST and Difference between prims and Kruskal’s Algorithm
    a. minimum spanning tree is a sub tree of a graph in which all vertex are visited whereas the total weight of edges must be as minimum as possible
    b. difference between prims and Kruskal

PRIMS	KRUSKAL
start with root vertex	starts with the vertex having minimum cost
select shortest connected vertex	select next shortest vertex
to obtain minimum tree it traverse one node more than once	it traverse a node only once
o(v^2)	o(e log v)
best for dense graph	best for sparse graph
best for list	best for heaps

2. benefits of chaining over linear probing in hash table
3. differences between Kruskal, Prim and Dijkstra algorithms
4. a scenario where you would avoid the use of linked lists
5. Graph or Tree. Choose and explain which you use to plot a water distribution scheme in cities.
6. Explain the algorithm you would use to describe the 2nd best path in a 5 node graph
7. How sorting can be performed without making comparisons?
8. e benefits of using a built-in Tree Abstract Data Type (ADT) in your code.
9. If the time comes and you have to prefer one over the other, how you justify the time complexity over space complexity and vice versa. Write arguments on both preferences.
10. why do you think allocating multiple linked list elements at a time is a good idea? Assume that the linked list is actively used for incoming data
11. Why can a null pointer not be accesd?
    a. beacause null pointer will invoke undefined behaviour , since it does not points to meaningful object. so an attemot to dereference it will cause runtime error

Merge Sort

- divide and conquer algorithm
- array is divided into halve and then combined in sorted manner
- recursive algorithm: continuously divides into half until it can not be divided further

Algorithm:

- `base case: array itni divide hochuki hai kay ab sirf aik element of array hi bacha hai. stop recursion`
- `if(low<high)`
    - `calculate mid point`
    - `after dividing arrays into smallest units start merging the elements`
    - `firstr compare element for each list and combine into sorted manner`

```
step1: start
step2: declare arr, left, right, mid
step3: perform merge function
    if left>right return
    mid=(left+right)/2
    mergesort(arr,left,mid)
    mergesort(arr,mid+1,right
    merge(array,left,mid,right)
step4: stop
```

Code

```
void merge(int arr[], int left, int mid, int right)
{
int i,j,k;
//create temp sub arrays
//first sub aray is from low - mid
int n1=mid - low +1;
int L[n1];
//copy data
for(i=0;i<n1;i++)
{
  L[i]=arr[low+1];
}
//second sub array is from mid+1 - high
int n2=right -mid;
int R[n2];
//copy data
for(j=0;j<n2;j++)
{
R[j]=arr[mid+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++;
    }
}//function merge ends

void mergesort(int arr[], int left, int right)
{
if(l<r)
{
// Same as (l+r)/2, but avoids overflow for
        // large l and h
int mid=left+(right-1)/2;
//sort first and second halves
mergesort(arr,left, mid);
mergesort(arr,mid+1,right);

merge(arr,left,mid,right);
```