Data Structures

Complexity Analysis

Algo:
1- sum=0
2- for i=1 to i=n
3- for j=1 to j=n
4- sum++
big O notation is $O(n^2)$

$3n+2$

$n \quad n(n+1) \quad 2n^2 =$

$2n^2$

Statement	equation	operations	iterations	subtotal
1.	sum=0	1	1	$1 \times 1 = 1$
2a	i=1	1	1	$1 \times 1 = 1$
2b	i<n	1	n+1	n+1
2c	i++	2	n	2n
3a	j=1	1	n(1)	n
3b	j<n	1	n(n+1)	n^2+n
3c	j++	2	n(n)	$2n^2$
4	sum++	2	n(n)	$2n^2$
				$3+5n+5n^2$
1	b=A[0]	2	1	2

Recursion

Direct recursion

A function is called direct recursive if it calls the same function again.

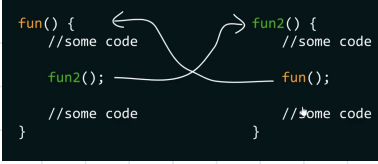
Structure of Direct recursion:

```
fun() {
    //some code

    fun();

    //some code
}
```

Indirect recursion

A function (let say fun) is called indirect recursive if it calls another function (let say fun2) and then fun2 calls fun directly or indirectly.

Structure of Indirect recursion:



TAIL RECURSIVE

A recursive function is said to be tail recursive if the recursive call is the last thing done by the function. There is no need to keep record of the previous state.

```
void fun(int n) {
    if(n == 0)
        return;
    else
        printf("%d ", n);
    return fun(n-1);
}
int main() {
    fun(3);
    return 0;
}
```

NON TAIL RECURSIVE

A recursive function is said to be non-tail recursive if the recursive call is not the last thing done by the function. After returning back, there is some something left to evaluate.

```
void fun(int n) {
    if(n == 0)
        return;
    fun(n-1);
    printf("%d ", n);
}
int main() {
    fun(3);
    return 0;
}
```

ONE MORE EXAMPLE (NON-TAIL)

```
int fun(int n) {
    if(n == 1)
        return 0;
    else
        return 1 + fun(n/2);
}
int main() {
    printf("%d", fun(8));
    return 0;
}
```

another evaluation

Backtracking

missing

Elementary Sorting

	N.O	Algorithm name	when to use	Advantage	Disadvantage
✓	1.	Bubble sort	small data set	no extra temp memory takes min space	too many steps : n^2
✓	2.	Insersion Sort	small data set		
✓	3.	selection sort		no extra temp memory takes min space	
✓	4.	Shell sort			
✓	5.	Quick sort	medium data set		
✓	6.	Merge sort	link list		
✓	7.	Radix Sort			
✓	8.	Comb sort			

dynamic array

```
int * a = new int( r ) → 1D array
int * a = new int( r x c) →2D array
int ** a = new *int (n)
```

find array size

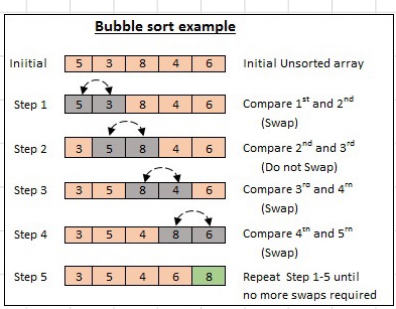
```
int n = sizeof(array) / sizeof(array[0]);
```

total array size / first element size

1. Bubble sort

↳ swaping of elements

```
void bubbleSort(int array[], int n)
{
  for (int i = 0; i < n; ++i)
  {
    for (int j = 0; j < n - i; ++j)
    {
      if (array [j] > array[j + 1])
      {
        int temp = array[j];
        array[j] = array[j + 1];
        array[j + 1] = temp;
      }
    }
  }
```



2. Insersion sort

↳ moving elements to sorted side (left)

↳ compares adjaced values

↳ moves to the left one by one

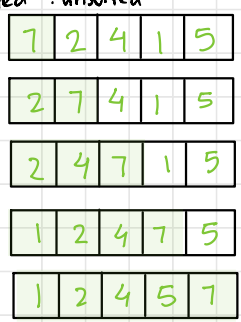
```
void insertionSort(int arr[], int n)
{
    int i, temp, j;
    for (i = 1; i < n; i++)
    {
        temp = arr[i]; → store's forward of j
        j = i - 1; → stores pre of i

        while (j >= 0 && arr[j] > temp)
        {
            arr[j + 1] = arr[j]; store prev value in for
            j = j - 1; to no check for old values
        }
        arr[j + 1] = temp; when position found store
    }
}
```

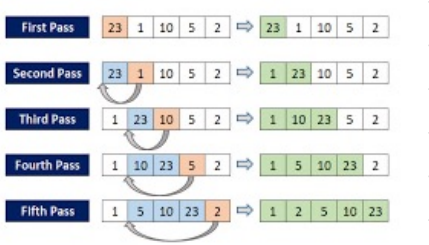
prev > for

j temp=2 | 7 | 2 |

j temp=2 | 7 | 7 |

sorted : unsorted





if last element has smallest value then insertion sort takes numerous sorts hence in this case shell sort better

3. Selection sort

↳ finds smaller element in array

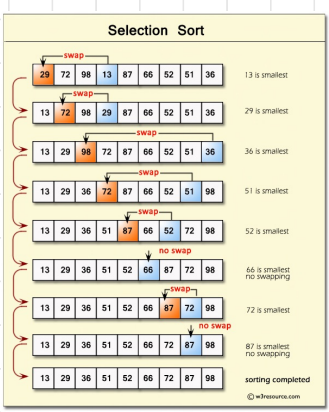
↳ exchanges with first position

↳ then second smallest element

↳ swaped with second position

```
void selectionSort(int arr[], int n)
{
   int i, j, min_idx;
   for (i = 0; i < n-1; i++)
   {
      min_idx = i;
      for (j = i+1; j < n; j++)
      {
            if (arr[j] < arr[min_idx])
            min_idx = j;
      }
       swap(arr[min_idx], arr[i]);
   }
}
```

0	1	2	3	4	i	j	min_indx
7	2	4	1	5	0	1	7 2 1
1	2	4	7	5	1	2	2
1	2	4	7	5	2	3	4
1	2	4	7	5	3	4	7 5
1	2	4	5	7			

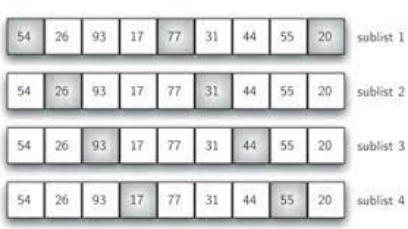


4. Shell sort

↳ compares distanced values

↳ then swaps

when gap=1 it will work as insersion sort

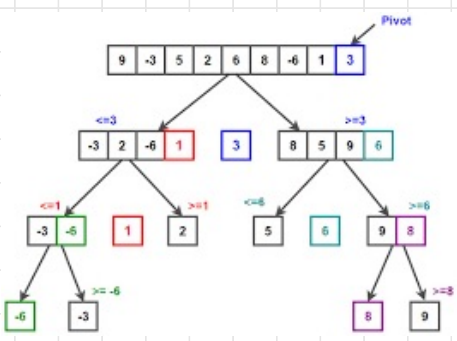



```
int shellSort(int arr[], int n)
{
     for (int gap = n/2; gap > 0; gap= n/2)        // reduce till gap reaches 1
     {
          for (int j = gap; j < n; j++)
          {
               for (int i=j-gap; i>=0; i=i-gap)   // i=j-gap → no backward gap ; i=i-gap → to check backward gap
               {
                    if(arr[i+gap]>arr[i])
                         { break; }               // no swapping then exit loop
                    else
                         { swap(arr[i+gap],arr[i] )
                    }                             // if swapping then compare backward gap
               }
          }
     }
```

5. quick sort

↳ partitions

```
void quickSort(int arr[], int lb, int ub)
{
     int start = lb;
     int end = ub;
     int pivot = arr[ lb ];                       // → can start partition anywhere

    while (start < end)
    {
          while (arr[start] <pivot)
               { start++; }
          while (arr[end] > pivot)
               { end--; }
          if (start <= end)
          {
               swap(arr[start],arr[end]) ;
               start++;
               end--;
          }
    }
     Recursion
     if (lb < end)
          { quickSort(arr, lb, end); }
     if (ub > start)
          { quickSort(arr, start, ub); }
}
```



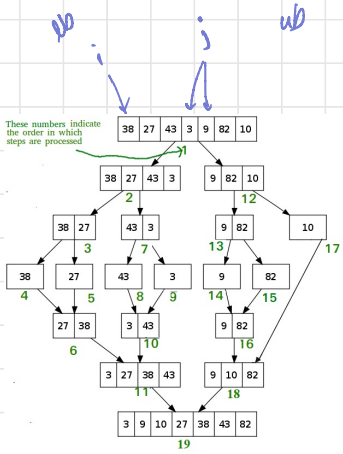
6. merge sort

↳ divide into subarrays till 1 element is left

↳ merge arrays sortedly

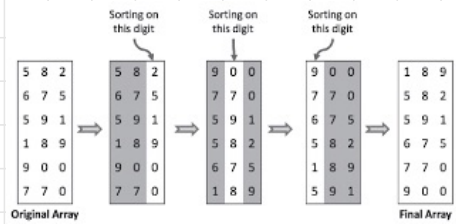
```
void Merge(int a[], int lb, int mid, int ub)
{
    // We have lb to mid and mid+1 to ub already sorted.
    int temp[ub];
    int i = lb;
    int k = 0;  → output array index
    int j = mid + 1;

    // Merge the two parts into temp[].
    while (i <= mid && j <= ub)
    {
        if (a[i] < a[j])
        {
            temp[k] = a[i];
            k++;
            i++;
        }
        else
        {
            temp[k] = a[j];
            k++;
            j++;
        }
    }

    // Insert all the remaining values from i to mid into temp[].
    while (i <= mid)
    {
        temp[k] = a[i];
        k++;
        i++;
    }

    // Insert all the remaining values from j to ub into temp[].
    while (j <= ub)
    {
        temp[k] = a[j];
        k++;
        j++;
    }

    // Assign sorted data stored in temp[] to a[].
    for (i = lb; i <= ub; i++)
    {
        a[i] = temp[i-lb];
    }
}

// A function to split array into two parts.
void MergeSort(int *a, int lb, int ub)
{
    int mid;
    if (lb < ub)
    {
        mid=(lb+ub)/2;
        MergeSort(a, lb, mid);
        MergeSort(a, mid+1, ub);
        Merge(a, lb, mid, ub);
    }
}
```

↳ inserts smaller value



7. radix sort

```
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;
}

void countSort(int arr[], int n, int pos)
{
    int output[n]; // output array
    int i, count[10] = { 0 };

    for (i = 0; i < n; i++) count occurences of digit
        count[(arr[i] / pos) % 10]++;

    for (i = 1; i < 10; i++) store positions of digits
        count[i] = count[i] + count[i - 1];
                   current     + prev
    for (i = n - 1; i >= 0; i--) { output array
        output[count[(arr[i] / pos) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }
        copy to array
    for (i = 0; i < n; i++)
        arr[i] = output[i];
}

void radixsort(int arr[], int n)
{
    get no. of digits
    int max= getMax(arr, n);
    for (int pos = 1; max/pos> 0; pos= *10)
        countSort(arr, n, pos);
}
```



8. Comb Sort

↳ like bubble sort but using gap

```
int getNextGap(int gap)
{
    gap = (gap*10)/13; gap/1.3
    if (gap < 1)
        return 1;
    return gap;
}

void combSort(int a[], int n)
{
    int gap = n;          so that loop runs
    bool swapped = true;

    while (gap != 1 || swapped == true)
    {
        gap = getNextGap(gap);
        swapped = false; → to check wether swapping occurs

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

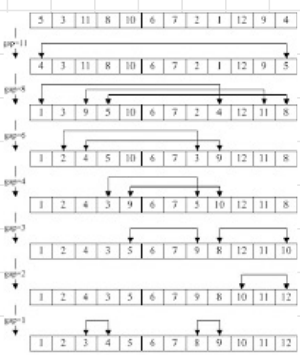


Figure 1: Comb sort

Link list

3 types of Link lists

- Single linked list: navigation is forward only
- Doubly linked list: navigation forward and backward
- Circular linked list: last element is linked to first
- Doubly Circular linked list:

A link list is made up of nodes

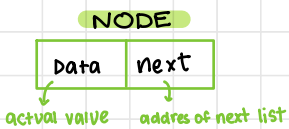
To check if pointer isnt pointing at any node

- if (head == NULL)
- if (head == 0)
- if (!head)

To check last node in list

if (head → next == NULL)

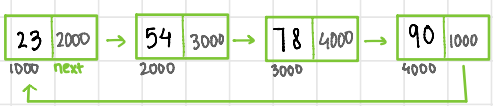
Single linked list



Doubly linked list

LAB 6

Circular linked list



Doubly Circular linked list


```
int main()
{
    Linkedlist list;
    list.add(15);
    list.add(30);
    list.print();


}
```

creating node

```
class node
{
   public:
   int data;
   node* next;

   node()
   {
      data = 0;
      next = NULL;
   }

   node(int a)
   {
      data = a;
      next = NULL;
   }
};
```

empty node

creating link list

```
class Linkedlist
{
   public:
   node* head;

   Linkedlist()
   {
      head = NULL;
   }
   Linkedlist(node *n)
   {
   head=n;
   }

   Linkedlist(int a)
   {
      head=new node(a);
   }
```

no node

gives head address of first node → head stores value of first node

creates new node and head stores address → creates first node, stores value in head

adding a node in end

```
void add-fist(int n)
   {
      if (head==NULL)
      {
         head=new node(n);
      }
      else
      {
node *temp=head;
while (temp->next!= NULL)
      {
temp=temp->next;
      }
temp->next=new node(n);
      }
   }
```

if no node exists

head stores value of first node created

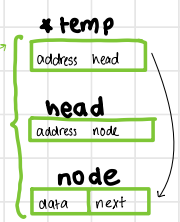
always dynamically

stores first node

loop till last node

stores next node

adds new node in end



displaying all node

```
void print()
   {
      node *temp=head;
      while (temp!=NULL)
      {
         cout<<temp->data<< " ";
         temp=temp->next;
      }
   }
```

stores first node value

loop till last node

print

moves to next node

reverse

```
void reverse()
  {
     Node* current = head;
     Node *prev = NULL;
     Node *next = NULL;

     while (current != NULL)
     {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
     }
     head = prev;
  }
```

loop till last node

adding node in begining

```
void add-last(int n)
   {
      if (head==NULL)
      {
         head=new node(n);
      }
      else
      {
         node *temp=head;
         head=new node(n);
         head->next=temp;

      }
   }
```

if no node exists

add new node, store in head

store first node in temp

create new node, store in head

Point head to temp

stores old first node

Searching

Binary Search

searches from mid value

```
void binarySearch(int arr[], int lo, int hi, int x)
{
    int found=0;
    while (hi>=lo)
    {
        int mid = (lo + hi)/2;
        if ( arr[mid] == x)
        {
                found =1;
                cout<< "found at index " <<mid;
                break;
        }
        else if ( arr[mid] < x )
                {lo = mid + 1; }
        else
                { hi = mid - 1; }
    }
    if (!found)
        { cout << "Element is not present in array"; }
}
```

(lo+hi)/2

if at mid

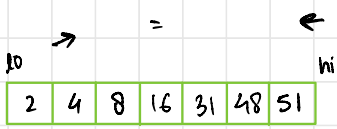
Interpolation Search

searches from formula value

```
void interpolationSearch(int arr[], int lo, int hi, int x)
{
    int found=0;

    while (x >= arr[lo] && x <= arr[hi] && arr[hi]!=arr[lo])
    {
        int pos=  lo + ((x - arr[lo]) * (hi - lo) / (arr[hi] - arr[lo]));
        if ( arr[pos] == x)
        {
                found =1;
                cout<< "found at index " <<pos;
                break;
        }
        else if ( arr[pos] < x )
                {lo = pos + 1; }
        else
                { hi = pos - 1; }
    }

    if (arr[lo] == x)
    {
        found = 1;
        cout<<"Found key at index ",lo;
    }

if (!found)
        { cout << "Element is not present in array"; }
}
```

$lo + \frac{(x - arr[lo]) \times (hi - lo)}{arr[hi] - arr[lo]}$

if at first index



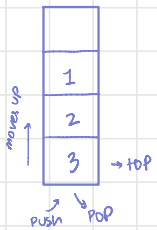
Linear Search

searches 1 by 1

```
int search(int arr[], int n, int x)
{
        for (int i = 0; i < n; i++)
        {
                if (arr[i] == x)
                        {cout<< "found at index " << i; }
                else
                        {cout<< "Element is not present in array"; }
}
```

STACK (LIFO)

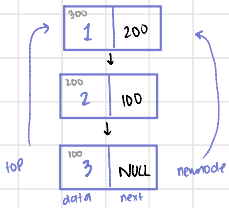
array




```
class stack
{
      int n; //size of array
      int top;
      int *arr;

      public:
      stack(int a)
      {
            top=-1;
            n=a;
            arr= new int[n];
      }

      void push(int val)
      {
            if (top>=n-1)
               { cout <<"Stack overflow "; }
            else
            {
                  top++;
                  arr[top]=val;
            }
      }

      void pop()
      {
            if(top<=-1)
               { cout<<"Stack Underflow"<<endl; }
            else
               { top--; }
      }

      void display()
      {
            if(top>=0)
            {
               for(int i=top; i>=0; i--)
               {
                  cout<<arr[i]<<" ";
                  }
            }
            else
               { cout<<"Stack is empty"; }
      }
};
```

```
int Peek( )
{ return top; }

bool isEmpty()
{ return (top == -1); }
```

Linklist




```
class node
{
   public:
   int data;
   node *next;

   node(int d)
   {
      data = d;
      next = NULL;
   }
};


class Linklist
{
   public:
   node *top;
   Linklist()
   { top = NULL; }

   void push(int d)
   {
      node *newnode= new node(d);
      newnode->next=top;
      top=newnode;
   }

   void pop()
   {
      if(top==NULL)
         {cout<<"Stack is empty"; }
      else
         { top=top->next; }
   }

   void display()
   {
      node *temp=top;
      while (temp!=NULL)
      {
         cout<< temp->data <<" ";
         temp=temp->next;
      }
   }
};
```

INFIX PREFIX POSTFIX

infix

1. ( ) , [ ] , { }
2. ^
3. * , /
4. + , -

<num><sign><num>

(5+1) * 6

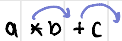
Prefix

<sign><num><num>

+ * a b c

Posthix

<num><num><sign>

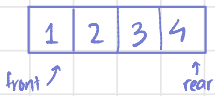


ab* c+

QUEUE FIFO

array

```
class queue
{
    int n;
    int rear; //inserts , points to last element
    int front;  //to delete , points to first element inserted
    int *arr;

    public:
    queue(int a)
    {
        rear=-1;
        front=-1;
        n=a;
        arr= new int[n];
    }

    void insert(int val)
    {
        if (rear==n-1)
        {cout <<"Queue is full "<<endl;}

        else if(front==-1) //first input
        {
            rear++;
            front++;
        }
        else  //rest inputs
        {rear++;}
    arr[rear]=val;
    }

    void deletes()
    {
        if(front==-1)
        {cout<<"Queue is empty"<<endl; }

        else if(front==rear) // one element in queue
        {
            rear=-1;
            front=-1;
        }
        else
        {front++; }
    }

    void display()
    {
        if(rear>=0)
        {
        cout<<"Queue elements are: ";
           for(int i=rear; i>front-1; i--)
           {
              cout<<arr[i]<<" ";
           }
        }
        else
        { cout<<"Queue is empty"; }
    }

};
```

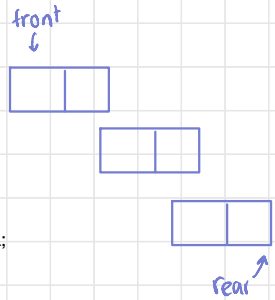


1. full
2. first input
3. rest

1. empty
2. 1 element left
3. rest

LAB 9

linklist




```
class node
{
   public:
   int data;
   node *next;

   node(int d)
   {
      data = d;
      next = NULL;
   }
};

class Queue
{
   node *front; -> delete node
   node *rear; -> add node
   public:
   Queue()
   {
      front=NULL;
      rear=NULL;
   }

   void insert(int n)
   {
      if(front==NULL) //for first node
      {
         front= new node(n);
         rear= front;
      }
      else
      {
         rear->next= new node(n);
         rear=rear->next;
      }
   }

   void deletes()
   {
      if(front==NULL)
      { cout<<"Queue is empty"; }
      else
      {
         front=front->next;

         if(front==NULL)
            { rear=NULL; }
      }
   }

   void display()
   {
      node *temp=front;
      while (temp!=NULL)
      {
         cout<<temp->data<<" ";
         temp=temp->next;
      }
   }
};
```

1. first node
2. rest

1. Empty
2. rest
3. last elemen

Circular queue array

```
class queue
{
        int n;
        int rear; //inserts , points to last element
        int front;  //to delete , points to first element inserted
        int *arr;

        public:
        queue(int a)
        {
                rear=-1;
                front=-1;
                n=a;
                arr= new int[n];
        }

        void insert(int val)
        {
                if (front==0 && rear==n-1 || front==rear+1)
                {cout <<"Queue is full "<<endl;}
                else if(front==-1) //first input
                {
                        rear++;
                        front++;
                }
                else //rest inputs
                {
                        if(rear== n-1)
                                { rear=0; }
                        else
                                { rear++; }

                        arr[rear]=val;
                }
        }

        void deletes()
        {
                if(front==-1)
                {cout<<"Queue is empty"<<endl; }
                else if(front==rear) // one element in queue
                {
                        rear=-1;
                        front=-1;
                }
                else
                {
                        if(front==n-1)
                                { front=0; }
                        else
                                { front++;}
                }
        }

        void display()
        {
                if(front ==-1 )
                        { cout<<"Queue is empty"; }
                else if(front<=rear)
                {
                    for(int i=rear; i>front-1; i--)
                    { cout<<arr[i]<<" "; }
                }
                else
                {
                        for(int i=front; i<=n-1;i++)
                                { cout<<arr[i]<<" ";}

                        for(int i=0; i<front; i++)
                                { cout<<arr[i]<<" ";}
                }
        }
};
```

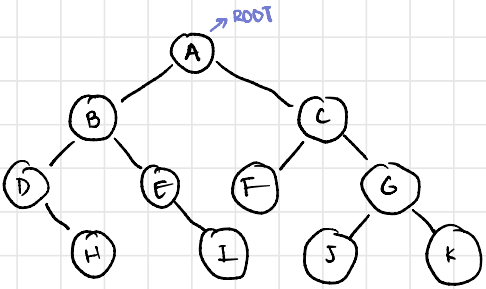
Circular queue linklist

```
class node
{
   public:
   int data;
   node *next;

   node(int d)
   {
      data = d;
      next = NULL;
   } '
};

class Queue
{
   node *front;
   node *rear;
   public:
   Queue()
   {
      front=NULL;
      rear=NULL;
   }

   void insert(int n)
   {

      if(front==NULL) //for first node
      {
         front= new node(n);
         rear= front;
      }
      else
      {
         rear->next= new node(n);
         rear=rear->next;
         rear->next=front;
      }
   }

   void deletes()
   {
      if(front==NULL)
      { cout<<"Queue is empty"; }
      else if(front==rear)
      {
         front=NULL;
         rear=NULL;
      }
      else
      {
         front=front->next;
         rear->next=front;//for circular
      }
   }

   void display()
   {
      node *temp=front;
      while (temp->next!=front)
      {
         cout<<temp->data<<" ";
         temp=temp->next;
      }
      cout<<temp->data<<" ";
   }
};
```

TREES

non linear data structure

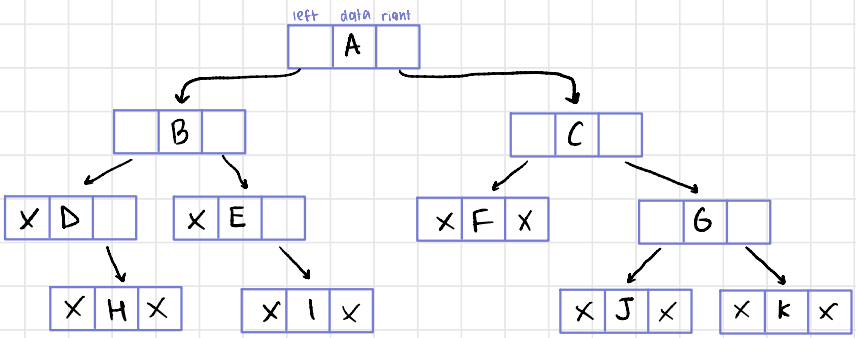
collection of nodes



can move top to bottom

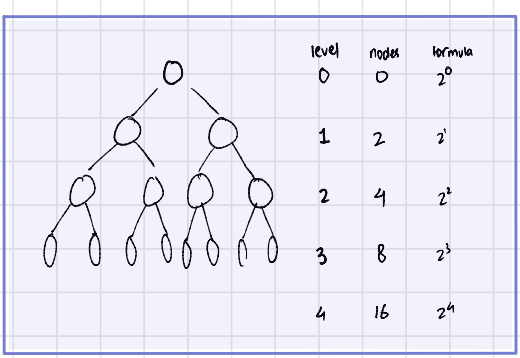
cant move bottom to top

- root
- nodes
- parent node
- child node
- leaf node
- non leaf node / internal node
- path
- edges
- ancestor
- decendant
- Sibling
- degree : no of children



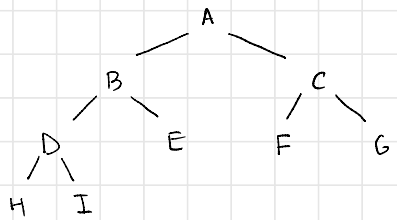
Binary Tree

- at most 2 children
- max node at any level $i = 2^{i}$ → level
- max nodes at heigh $h = 2^{h+1} - 1$



BINARY TREE

at most 2 children



ARRAYS

A	B	C	D	E	F	G	H	I
0	1	2	3	4	5	6	7	8

Complete binary tree:

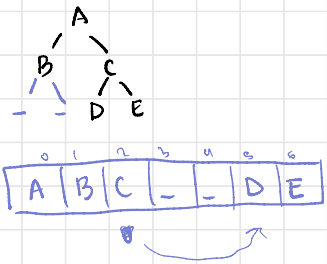
↳ all levels filled except last

↳ all as left as possible

left child : $2i+1$

right child: $2i+2$

parent : $\frac{i}{2}$ -1 floored



array

```
char tree[10];
void root(char val)
{
   if(tree[0] != NULL)
      cout<<"Tree already had root"<<endl;
   else
      tree[0] = val;
}

void left(char val, int parent)
{
   if(tree[parent] == NULL)
      cout <<"no parent found"<<endl;
   else
      tree[(parent * 2) + 1] = val;
}

void right(char val, int parent)
{
   if(tree[parent] == NULL)
      cout<<"no parent found"<<endl;
   else
      tree[(parent * 2) + 2] = val;
}

void print()
{
   cout << " ";
   for(int i = 0; i < 10; i++)
   {
      if(tree[i] != '\0')
         cout<<tree[i];
      else
         cout<<"-";
   }
}

int main()
{
   root('A');
   right('D', 0);
   left('B', 0);
   right('E', 1);
   right('F', 2);
   print();
}
```

BINARY TREE TRAVERSALS

link list

```
class node
{
public:
    char data;
    node *left,*right;
};

node* create()
{
    char x;
    node *temp=new node();
    cout<<"data : ";
    cin>>x;
    if(x=='1') → empty condition
        return NULL;
    temp->data=x;

    cout<<x<<" Left child ";
    temp->left=create();

    cout<<x<<" Right child ";
    temp->right=create();
    return temp;

}
```

LDR

```
void inorder(node *root)
{
      if (root)
      {
            inorder( root->left );
            cout << root->data << " ";
            inorder( root->right );
      }
}
```

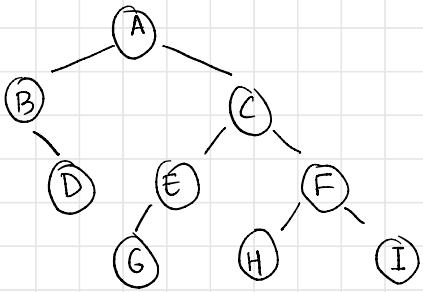
DLR

```
void preorder(node *root)
{
      if (root)
      {
      cout << root->data << " ";
      preorder( root->left );
      preorder( root->right );
      }
}
```

LRD

```
void postorder(node *root)
{
      if (root)
      {
      postorder( root->left );
            postorder( root->right );
      cout << root->data << " ";
      }
}
```

```
int main()
{
    node *root;
    root=NULL;
    root=create();
    inorder(root);
    cout<<endl;
    preorder(root);
    cout<<endl;
    postorder(root);
}
```



Preorder: (Root, Left, Right)

A, B, D, C, E, G, F, H, I

inorder: (Left, Root, right)

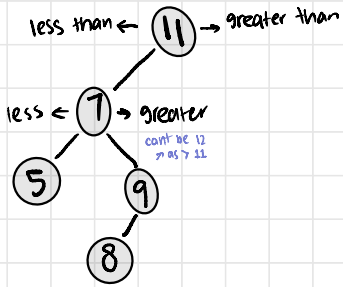
B, D, A, G, E, C, H, F, I

Postorder: (Left, right, Root)

D, B, G, E, H, I, F, C, A

BINARY SEARCH TREE

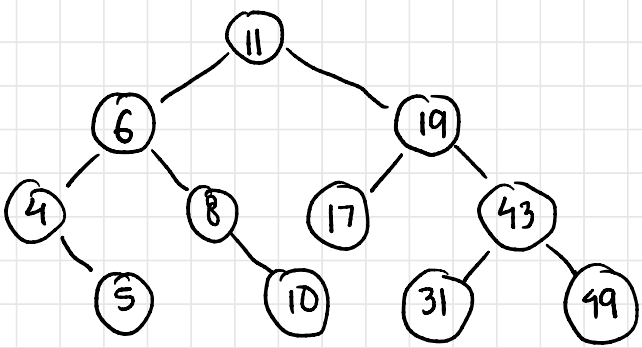
↳ no duplicate elements



at most 2 children

start always root

11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31



INORDER (Left, Root, Right)

delete a 2 child node

↳ inorder predecesser — in ascending order only for BST

- largest node from left subtree

↳ inorder sucessor

- smallest node from right subtree

insert/search

```
class node
{
    public:
        int data;
        node *left, *right;

        node()
    {
        data=0;
        left=NULL;
        right=NULL;
    }

        node(int n)
    {
        data=n;
        left=NULL;
        right=NULL;
    }
};

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

    if (value > root->data) //if val > root
        root->right = Insert(root->right, value);

    else if (value < root->data) //if val < root
        root->left = Insert(root->left, value);

    return root;
}

void Inorder(node* root)
{
    if (root) //if root exists
    {
        Inorder(root->left);
        cout << root->data << " ";
        Inorder(root->right);
    }
}

node* search(node* root, int value)
{
        if (root == NULL || root->data == value)
        return root;

        if (root->data < value)
            return search(root->right, value);
        else
        return search(root->left, value);
}
```

delete

```
node *minValueNode( node *n)
{
    node *current = n;
    while (current && current->left != NULL)
        { current = current->left; }
    return current;
}

node *deleteNode( node *root, int value)
{
    if (root == NULL)
        return root;

    if (value < root->data)
        root->left = deleteNode(root->left, value);

    else if (value > root->data)
         root->right = deleteNode(root->right, value);

    else //delete
    {
        if (root->left == NULL)
        {
            node *temp = root->right;
            return temp;
        }
        else if (root->right == NULL)
        {
            node *temp = root->left;
            return temp;
        }
        node* temp = minValueNode(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

int main()
{
    node *root = NULL;
        root= Insert(root, 50);
        Insert(root, 30);
        Insert(root, 20);
        Insert(root, 40);
        Insert(root, 70);
    deleteNode(root, 20);
        Inorder(root);

}
```

lowest left value

searching

delete root

if left empty

if right empty

if both filled

ishma hafeez notes

AVL Tree

LAB 10

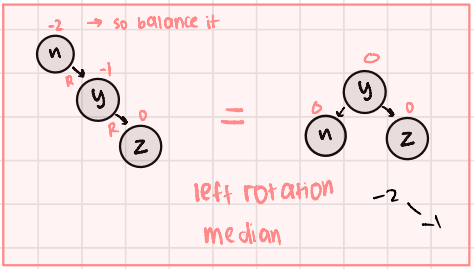
↳ Binary Search Tree

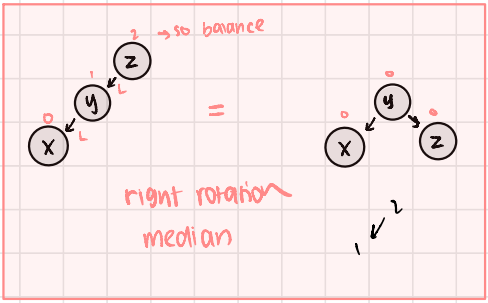
↳ balance factor

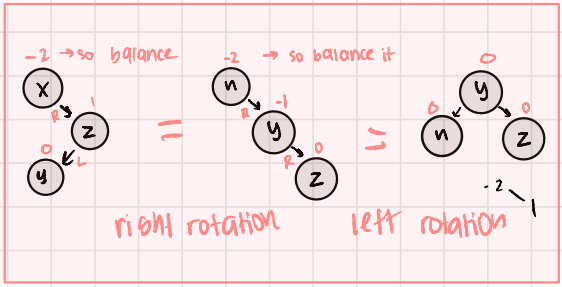
↳ h leftsubtree - h rightsubtree : {-1/0/1} for each node

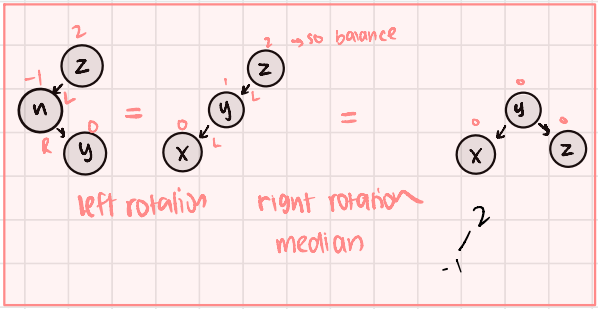
↳ after every insertion check balance factor

↳ if ≠ {-1/0/1} then balance it









ROTATIONS

```
class node
{
    public:
    int data;
    int height;
    node *right;
    node*left;

    node()
    {
        right=NULL;
        left=NULL;
    }

    node(int d)
    {
        data=d;
        right=NULL;
        left=NULL;
        height=1;
    }
};
```

```
node* LLRotation(node *&root)
{
    node* rootl=root->left;
    root->left=rootl->right;
    rootl->right=root;

    root->height=NodeHeight(root);
    rootl->height=NodeHeight(rootl);

    root=rootl;
    return root;
}

node* LRRotation(node *&root)
{
    root->left=RRRotation(root->left);
    root=LLRotation(root);
}
```

```
node* RRRotation(node* &root)
{
    node* rootr=root->right;
    root->right=rootr->left;
    rootr->left=root;

    root->height=NodeHeight(root);
    rootr->height=NodeHeight(rootr);

    root=rootr;
    return root;
}

node* RLRotation(node* &root)
{
    root->right=LLRotation(root->right);
    root=RRRotation(root);
}
```

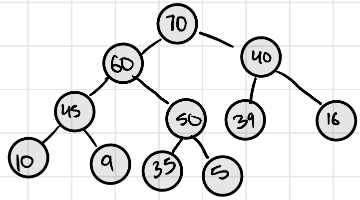
```
int NodeHeight(node *&root)
{
    int l,r;   root exists   root left exists
    if(root!=NULL && root->left!=NULL)
        { l=root->left->height;}
    else
        { l=0;}

    if(root!= NULL && root->right!=NULL)
        { r=root->right->height; }
    else
        { r=0; }

     if(l >r)
        { return l+1; }
    else
        { return r+1; }
}

int BalanceFactor(node *&root)
{
    int l,r;
    if(root!=NULL && root->left!=NULL)
        { l=root->left->height; }
    else
        { l=0;}

    if(root!=NULL && root->right!=NULL)
        { r=root->right->height; }
    else
        { r=0;}

return l-r;
}
```

```
node* insertinBST(node *&root,int d)
{
    if(root==NULL)
    {
        root=new node(d);
        return root;
    }

    if(d>root->data)
        { root->right=insertinBST(root->right,d); }
    else
        { root->left=insertinBST(root->left,d); }

    root->height=NodeHeight(root);
    if(BalanceFactor(root)==2 && BalanceFactor(root->left)==1)
        { LLRotation(root); }
    else if(BalanceFactor(root)==2 && (BalanceFactor(root->left)==-1))
        { LRRotation(root); }
    else if(BalanceFactor(root)==-2 && (BalanceFactor(root->right)==-1))
        { RRRotation(root); }
    else if(BalanceFactor(root)==-2 && BalanceFactor(root->right)==1)
        { RLRotation(root); }

    return root;
}

void takeInput(node* &root)
{
    int d; cin>>d;
    while(d!=-1)
    {
        insertinBST(root,d);
        cin>>d;
    }
}
```

```
int main()
{
    Node * root = NULL
    take input (root)
}
```

LAB 11

max heap
↳ A [parent [i]] ≥ A[i]
↳ output left to right of each level



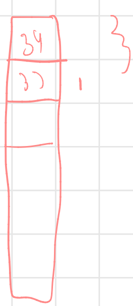
index i
left child 2i+1
right child 2i+2
parent i/2 -1

70	60	40	45	50	39	16	10	9	35	5
0	1	2	3	4	5	6	7	8	9	10

Max heap

```
n=0;  → updated when inserting or deleting
void heap(int a[], int n, int i)
{                              2   1
   int max = i;  1
   int l= 2*i+1;  3
   int r= 2*i+2;  4
                                   change for min heap
   if (r < n && a[r] > a[max])  check for right children
   {   max = r;    }

   if (l < n && a[l] > a[max])  check for left children
   {   max = l;    }

   if (max!= i)   not balanced
   {
      swap(a[max], a[i]);
      heap(a, n, max);
   }                       2   1
}


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

void del(int a[], int val)
{
   int i;
   for (i=0;i<n; i++)
   {
      if (val == a[i])
      { break; }
   }   find value  index=i

   swap(a[i],a[n-1]);  → swap with last value
   n--;  decrement array
   for (int i=n/2 - 1; i>= 0; i--)
   {   heap(a, n, i);    }
}


void insert(int a[], int val)
{                          loop in reverse
   a[n] = val;
   n++;                 parent I think
   for (int i = n/2 -1 ; i >= 0; i--)
   {   heap(a, n, i);  }
}
```

n=X 2



1

heap sort

```
void sort(int a[],int n)
{
//building
   for (int i = n/2 -1; i >= 0; i--)
   {   heap(a, n, i);  }

   for (int i = n-1; i >= 0; i--)
   {
      swap(a[0], a[i]);
      heap(a, i, 0);
   } //so root is at highest element again

   //reverse array
   for (int i = 0; i < n/2; i++)
   {   swap(a[i], a[n-i-1]);   }
}
```

priority queue using heap

```
void del(int a[]) //deleting from root
{
   int i;
   swap(a[0],a[n-1]);
   n--;
   for (int i=n/2 - 1; i>= 0; i--)
   {   heap(a, n, i);    }
```

TYPES OF HASHING

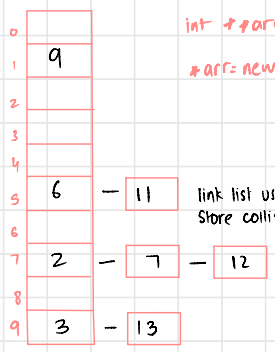
LAB 12

1) Open hashing (closed addressing) → Chaining
2. Closed hashing (Open addressing)

- linear Probing
- Quadratic Probing
- Double Hashing

3,2,9,6,11,13,7,12 → key
h(k) = 2k+3 m = 10 → location

1. Chaining



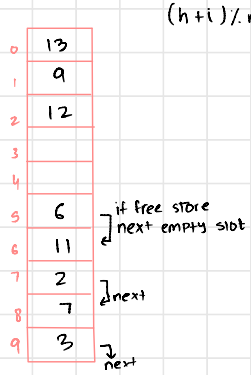
int **arr

*arr = new node [n]

link list used to store collisions

2. linear Probing

(h+i)%m



h_1: 2k+3

3. Quadratic Probing

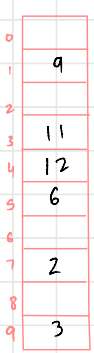
$(h+i^2)\%m$ i=0 to i<m

resolves clustering

4. Double Hashing

$h_2(k)$: 3k+1 → given in question

$(h_1 + h_2 * i)\%m$ i=0 to i<m



- use h_2 when collision, hence formula when collision

LINEAR PROBING

```
class HashTable
{
    public:
    int *arr;
    int n;
    HashTable(int size)
    {
        n=size;
        arr=new int [n];
        for(int i=0;i<n;i++)
        {
            arr[i]=NULL;
        }
    }
    int hashfunction(int key)
    {
        return key%n;
    }


    void insert(int key)
    {
        int index =hashfunction(key);
        int repeat=index;

        if(arr[index]==NULL)
        {   arr[index]=key;    }

        else
        {
            int i=1;
            while(arr[index+i]!=NULL)
            {
                if(index+i==repeat)
                {
                    cout<<"Hash Table is full"<<endl;
                        break; add a bool variable to
                }                stop inserting
                if (i==n)
                {
                        index=0;
                }
                i++;
            }
        arr[(index+i)%n]=key;
    }
}
```

```
int search(int key)
{
    int index=hashfunction(key);
    int repeat=index;
    if(arr[index]==key)
    {
        return index;
    }
else
    {
        while(arr[index+1]!=key)
        {
            index++;
            if(index==n)
            {
                index=0; }
            if(index==repeat)
            {
                return -1;
            }
        }
            return index;
     }
   }


void deletekey(int key)
{
    int index=search(key);
    if(index==-1)
    {
        cout<<"Key not found"<<endl;
    }
        else
    {
        arr[index]=NULL;
    }
}

};
```

CHAINING

```
class node {
    public:
    int data;
    node* next;
node() {
data=0;
next = NULL; }
    node(int val)
    {
data=val;
next = NULL; }
};

class Hashtable
{
public:
    node **arr;
    int n;
    Hashtable(int size)
    {
        n=size;
        *arr=new node [n];
        for(int i=0;i<10;i++)
        {
            arr[i]=NULL;
        }
    }

    int hashfunction(int key)
    {
        return key%n;
    }


void insert(int key)
{
    node *newnode=new node(key);
    int index =hashfunction(key);
    if(arr[index]==NULL)
    {   arr[index]=newnode;    }
else
    {
        node *temp=arr[index];
        while(temp->next!=NULL)
        {
            temp=temp->next;
        }
        temp->next=newnode;
    }
}
```

```
int search(int key)
{
    int index=hashfunction(key);
    if(arr[index]==NULL)
    {
        cout<<"Key not found"<<endl;
return -1; }
else
    {
        node *temp=arr[index];
        while(temp)
        {
            if(temp->data==key)
            {
                cout<<"Key found at index "<<index<<endl;
                return index;
            }
            temp=temp->next;
        }
        cout<<"Key not found"<<endl;

return -1; }
}


void deletes(int key)
{
    int index=search(key);
    if(index==-1)
    {
        cout<<"Key not found"<<endl;
    }
else
    {
        node *temp=arr[index];
        node *prev=NULL;
        while(temp)
        {

    if(temp->data==key)
    {
        if(prev==NULL)
        {
            arr[index]=temp->next;
            delete temp;
            break;
}
else
        {
            prev->next=temp->next;
            delete temp;
            break;
} }
prev=temp;
    temp=temp->next;
}

} }
```

```
void print()
{

for(int i=0;i<n;i++)
{
    node *temp=arr[i];
    if(temp==NULL)
    {

cout<<"0"; }
    while(temp)
    {
        cout<<temp->data<<" -> ";
        temp=temp->next;
    }
    cout<<endl;
}
```

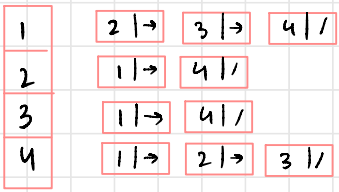
GRAPHS

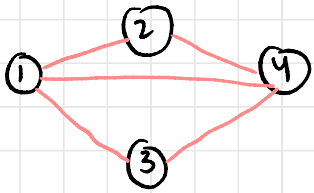
LAB 13

Adjency matrix

	1	2	3	4
1	0	1	1	1
2	1	0	0	1
3	1	0	0	1
4	1	1	1	0

Adjency List





Breadth First Search (BFS)

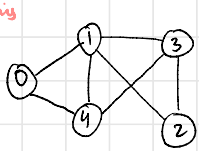
- can traverse from any node
- Queue used (FIFO)
- goes node → its neighbour then next node

Depth First Traversal (DFS)

- can traverse from any node
- uses Stack (LIFO)
- goes in depth of each node then backtracks

```
void BFS(int s)
{
   bool visited[v]; → declare visited
    for (int i = 0; i < v; i++)
      {visited[i] = false;}

queue<int> q; → declare queue
visited[s] = true; → mark starting as visited
q.push(s); → insert starting node
while (!q.empty()) → till ave empty
{
   int x = q.front();
   cout << x << " ";
   q.pop();        → remove first vertex of queue
   for (int i=0;i<v;i++)
   {
      if (matrix[x][i] == 1 && !visited[i])   =F
      {
         visited[i] = true;
         q.push(i); → Push all neighbouring vertex in queue
      }
   }
}
}
```

0, 1, 4, 2, 3




```
void DFS(int s)
  {
     bool visited[v];
     for (int i = 0; i < v; i++)
     {  visited[i] = false; }

     stack<int> stack;
     stack.push(s);

     while (!stack.empty())
     {
        int x = stack.top();
        stack.pop();
        if (!visited[x])  =F
        {
           cout << x << " ";
           visited[x] = true;
        }

        for (int i=0;i<v;i++)
        {
           if (matrix[x][i] && !visited[i])
           {  stack.push(i); }
        }
     }
  }
```

0, 4, 3, 2, 1

DIJKSTAS

```
int minDistance(int dis[], bool visited[])
{
    int min = 999;
    int index;
for (int i = 0; i < V; i++)
if (visited[i] == false && dis[i] <= min)
min = dis[i], index = i;
    return index;
}

void dijkstra(int matrix[V][V], int source)
{
    int dis[V];
    bool visited[V];
    for (int i = 0; i < V; i++)
        dis[i] = 999, visited[i] = false;

        dis[source] = 0;

for (int i = 0; i < V - 1; i++)
{
        int u = minDistance(dis, visited);
        visited[u] = true;
        for (int j = 0; j < V; j++)
        if (!visited[j] && dis[u] != 999 && dis[u] + matrix[u][j] < dis[j])
                dis[j] = dis[u] + matrix[u][j];
}

    display(dis, V);
}
```

PRIMS

```
int main()
{
  int edge_count;
  int visited[v];
  int mst=0;
memset(visited, false, sizeof(visited)); visited[0] = true;
edge_count = 0;
cout << "Edge"<< " : "<< "Weight"<<endl; int x,y;

while (edge_count < v - 1)
{
    int min = 999;
    x = 0;
    y = 0;
    for (int i = 0; i < v; i++)
    {
        if (visited[i])
        {
                for (int j = 0; j < v; j++)
                {
                        if (!visited[j] && k[i][j])
                        {
                                if (k[i][j]<min)
                                {
                                        min = k[i][j];
                                        x = i;
                                        y = j;
                                }
                        }
                }
        }
}
edge_count++;
visited[y] = true;
mst= mst + min;
cout <<x<<"-" << y << " : " <<min; cout << endl;
}
```

KUSHKALS

```
int parent[5];
int find(int i)
{
    while (parent[i] != i)
        {i = parent[i];}
        return i;
}

void pairs(int i, int j)
{
    int a = find(i);
    int b = find(j);
    parent[a] = b;
}

void kruskal(int matrix[][5])
{
        int mst = 0;
    for (int i = 0; i < v; i++)
        parent[i] = i;

    int edge_count = 0;
    while (edge_count < v - 1)
    {
        int min = 999, a = -1, b = -1; for (int i = 0; i < v; i++)
        {
                for (int j = 0; j < v; j++)
                {
                        if (find(i) != find(j) && matrix[i][j] < min)
                        {
                                min = matrix[i][j];
                                a = i;
                                b = j;
                        }
                }
        }

        pairs(a, b);
        cout<<a<<"-"<<b<<" :"<<min<<endl; edge_count++;
        mst=mst+min;
}
cout<<"Minimum cost = "<<mst;
}
```

SPANNING ALGORITHMS

KUSHKALS → 999 for no edges
PRIMS → 0's
DIJIKSTAS → GREEDY → 999
FLLOYD → Path NASH

- ✓ LINK LISTS
- ✓ RECURSION
- X BACKTRACKING YUCK
- SEARCHING
- X SORTING YUCK
- ✓ STACKS
- ✓ QUEUES
- ✓ BST
- ✓ AVL TREES
- ✓ HEAP & PRIORITY QUEUE
- ✓ HASHING
- GRAPHS