12.10.20

S3

Fall 2020-21:: Data Structure Theory (F)

Course Teacher: Mahfujur Rahman

consulting hour: (8 am – 8 pm) on MS teams t.

Evaluation:

Quiz — (best 2/3 or 1/2)

Assignment

MCQ Quiz

Vivan

(Mid)

Lecture 1

Data: Data means raw facts or information that can be processed to get results.

Structure: A structure may be treated as a frame where we organize some elementary items in different ways.

What is data structure?

⇒ Data structure is the systematic way to organize elementary data and their exits structural relationship, so that it can be used efficiency.

Elements of a data structure : Elementary data items.

Character, Integer, Floating point number etc.

- A data structure may be an element of another data structure.

For example : Array, Structure, Stack etc.

DS in two ways:

(i) Having a concrete implementation: **Basic**

Example : Variable, Pointer, Array.

(ii) Abstract Data Types — ADTs :

Don't have any concrete implementation.

Example : List, Stack, Queue.

Operations on Data Structure:

- Basic : (i) Insertion — addition of new element.

(ii) Deletion — removal of element.

(iii) Traversal —— accessing data elements.

■ Additional: (i) Searching —— locating a certain element.

(ii) Sorting —— arranging elements in a specified order.

(iii) Merging —— combining elements of two similar data structures.

* Etc.

Program: Sequence of instructions of any programming language, that can be followed to perform a particular task.

- A program has 3 sections: input, processing, output.
- program uses data structure.

algorithm ⟶ program

Alogrithm: Set of instructions that can be followed to perform a task.

or, Sequence of steps that can be followed to solve a problem.

- To write an algorithm, we don't need to follow strictly grammar of any particular programming language. However, it may be near to a program.
- Each and every algorithm can be divided into three sections : input, operational or processing section, output.
- The second section of algorithm is the most important one. Here, we have to do all necessary operations : (i) computation, (ii) taking decision, (iii) calling other procedures, etc.

14.10.20

DS

Lecture 2

Array [1-Dimensional]

- numeric numbers & symbolic ones.

distinguish — difference contiguous — সংলগ্ন

we have to use vaild identifiers.

Variable: A variable is a storage location and an associated symbolic name (an identifier) which contains some known or unknown quality or information, a value.

* A variable is an area of memory.

index — সূচক

An array can hold a series of elements of the same type placed in contiguous memory locations.

- One can use the name to specify where to store data.

int x ;
- &X represents the memory area of variable named x
- X represents the value stored inside the area of variable named X.

type name [total_numbers_of_elements]
(data type)

* Arrays are blocks of non-dynamic memory. Each of these elements can be individually referenced by using an index to a unique identifier. Arrays are a convenient way of grouping values of same type.

Array's natural partner → ~~for~~ Loop

[ ] has two uses
(i) size declaration.
(ii) specify indices for concrete array elements.

```
char array[5];
array[7] = "*";
```

→ memory segmentation error / fault.

~~Segmentation — Fusion~~

* The array limits run from zero to the size of the array minus one.

21.10.20

Lecture 3

Array [2-Dimensional]

2D array → arrays of array.

uniform / same data type. nested loop using

int X[a][b]; a → row b → column

2D → just an abstraction for programmers.

- Memory of each element of an array can be accessed using "&" operator.

* If the element is more than 1 byte, it gives the starting byte of the element.

mimo = &mimo[0] // location of the array.

&array[index] = start_location_array + index * size_of_data

=> &mimo[2] = &mimo[0] + 2 * sizeof(int)

=> &mimo[2] = 567 + 2*4 = 575

For 2D-array:

$0 < i \leq Row, \quad 0 < j \leq Column$

$\&array[i][j] = start\_location + (i * (C * size\_of\_data)) + (j * size\_of\_data)$

$\&mimo[1][1] = mimo[0][0] + (1 * (3 * sizeof(int))) + (1 * sizeof(int))$

$= 567 + (1*3*4) + (1*4)$

$= 583$

$\&mimo[4][3] = 567 + (4*3*4) + (3*4) = 627$

String

Strings are sequence of characters representing a piece of text.

String → array of characters.

* end marker → zero (or Null) byte '\0'.
* The null character indicates where the processing will stop for cout.

22.10.20

Lecture 4

```
char Animal[] = "Tiger";
cout << Animal;
```

* cin always stops at white spaces during taking input.

To overcome this ——

```
cin.get()
```

which takes only 2 parameters : variable name of the size & it's size.

Example: `cin.get (Name, 30);`

* C++ supports a large number of string handling functions. ⟶ defined in "cstring" header file.

(i) strlen (str) : calculates the length of string str.

(ii) strcpy (s1, s2) : copies a string to another string.

(iii) strcat (s1, s2) : concatenates / joins two strings.

(iv) strcmp (s1, s2) : compares two strings.

a) if $s_1 = s_2$, strcmp returns 0

b) if $s_1 < s_2$, strcmp returns < 0

[$s_1 > s_2$] otherwise returns > 0;

String as object

Instead of using cin / cin.get() ⟶ getline()

* getline() function takes the input stream as the 1st parameter which is cin and string object as the second parameter.

```
string s;
getline (cin,s);
cout << s;
```

```
i<s.length()
 cout<<s[i];
```

* length() function is to get the total size of the string object.

Pointer *

memory map → location : address

area of memory → "size of datatype" as area.
↳ variable

location of variable → & variable.

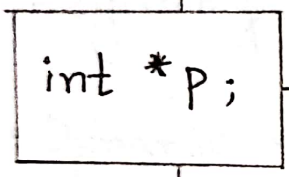
Example: &x

* A pointer is a variable that stores the location of a memory / variable.

```
int *p, q;
p = &q;
```

address → uniquely defined by a number.

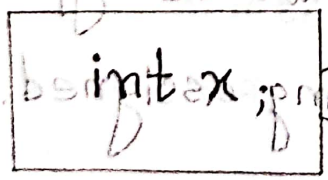
- *p is called asterisk-P.



→ &p represents the memory area of variable named *p.

→ p represents the address stored inside the area of variable named *p.

→ *p represents the value stored in the area represented to by address p.



→ &x represents the memory area of variable named x.

→ x represents the value stored inside the area of variable named x.

Example: int *p, x=10;
p = &x;

x = 10 0x7ffe0f54

location of x = [hexadecimal numbers]

→ P = 0x7ffe0f54

location of P = 0x7ffe0f58

→ pointers P = 10

Void Pointers: allows void pointers to point to any data type, which represents absense of type.

→ • We need to cast the address in it.

* As *data has no type, it must be type casted to (char*) before being assigned.

Null Pointers: indicates not pointing to any valid reference or memory address.

int *p; ⟶ p=0; // p=NULL; (null pointer value)

** C++ integrates the operators new and delete for dynamic memory allocation.

- Data type specifier is ~~followed~~ pointed to be beginning of the new block of memory allocated.
- Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no speed is required.

pointer = new ntype [number_of_elements];
↓
allocate memory

delete [ ] pointer;

28.10.20

Structure

A structure is an aggregate data type built using elements of other types.

- aggregate ——— group, total.

* Structure is a collection of variable of different types, under a single name.
* Structure ——→ user defined data type.
- struct ——→ গুদামঘর

Example: ——→ Keyword

```
struct Employee ——→ Identifier
{
    char name[20];
    int age;
    float salary;
};
```

} → list of component variables

} ; ——→ ends with semi-colon

char ⟶ 2 byte float ⟶ 8 byte

int ⟶ 4 byte double ⟶ 16 byte

{ } a ;

EmployeeRecord b, *c, d[5];

c = &a

variable a takes {20×2 + 1×4 + 1×8} bytes or 52 bytes in the memory.

So, each index contains 52 bytes.

- The dot (.) or combination of dash-line and (⟶) greater than sign is used as operator to refer to members of struct.

---

* To initialize structure variable:

EmployeeRecord x = {"Smith", 22, 1234.56};

(x.name, x.age, x.salary) ⟶ refers

* Memory is allocated when variables are created.

* Nested Structure: Structure in structure.

Example 1:

```
struct DateofBirth
{
  int day, month, year;
};
struct Student
{
  char StuName[30];
  int age;
  DateofBirth dob;
};
```

Example 2:

```
struct Appointment
{
  struct AppDate
  {
    int day, month, year;
  } dt;
  struct AppTime
  {
    int minute, hour;
  } tm;
  char venue[100];
};
```

* To initialize and show as output:

```
Student Stu1;
Stu1.dob.month = "March";
cout << stu.dob.month << endl;
```

* Self-referential Structure: One or more pointers points to the structure of the same type.

→ [pointer to instance of enclosing struct]

Example: Every person may have a child who is also a person.

```
struct Person{
    char name[30];
    person *child;
};
```

- Used for linked lists, queue, stack and trees.

31.10.20

Stack

A stack is an abstract data type, that serves as a collection of elements, in LIFO (last in, first out) method, with two principal operations.

(i) push adds elements.

(ii) pop removes the last element that was added.

Bounded capacity:

(i) If the stack is full and does not contain enough space to accept an entity to be pushed, then it is considered as overflow stack.

(ii) A pop either reveals previously concealed (hidden) items, or results in an empty stack (no items are present in stack to be removed)

Non-Bounded capacity: Dynamically allocate

memory for stack. • No overflow.

* Stack Implementation in C++ :

```
int stack[100], top = 0, maxSize = 100;
// stack[]: holds the elements
   top[]: index of stack[], holds first element
   maxSize: array size of stack[]
```

(i) bool isEmpty(); // returns True if stack has no element

```
{
  if (top == 0)
     return true;          // underflow
  else return false;
}
```

(ii) bool isFull() // returns True if stack full

```
{
  if (top == maxSize)  --> (when top==-1 ; top==maxSize-1)
     return true;          // overflow
  else return false;
}
```

(iii) bool push (int Element)

```
{                          // inserts element at the
                              top of the stack
 if (isFull())
 {
  cout << "Stack is FULL!" << endl;
  return false;
 }                    (when we consider top == -1)
stack[top] = element;
top++;
return true;
}
```

(iv) bool pop() // deletes top element from stack and puts it in

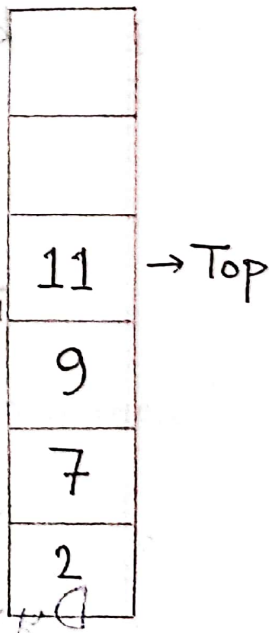
```
{
 if (isEmpty())
 {
  cout << "Stack is already empty!
           Cannot pop!" << endl;
  return false;
 }
top--  ;  return true;
}
```

(v) int topElement() // gives the top element

```
{
  return stack[top-1];
}
```

(vi) bool show() // prints the whole current stack

```
{
  if (isEmpty())
  {
    cout << "Stack is empty!
         Nothing to show!" << endl;
    return false;
  }
  for (int i = top-1 ; i>=0 ; i--)
  {
    cout << stack[i] << endl;
  }
  return true;
}
```



(Example)

*There are three ways to declare stack:

Object Oriented Approach:

```
class myStack
{
 int stack[100], top, maxSize;
 public:
  myStack (int size = 100)
   {
     maxSize = size;
          top = 0;
   }
  bool isEmpty ();
  bool isFull ();
  bool push (int Element);
  bool pop ();
  int topElement ();
  void show ();
};                                  //myStack
```

Dynamic Stack: `int *Stack, top, maxSize;`

`public: myStack (int);`

```
~myStack();
   // empty, full, push, pop, top element, show
void resize(int size);   // Resize the stack
};
```

- Constructor will create the array dynamically; Destructor will release it.

```
myStack::myStack(int size = 100)
{
  maxSize = size;   // get size
  stack = new int[maxSize];
  top = -1;   // start the stack
}

myStack::~myStack()
{
  delete[] stack;   // release the memory for stack
}
```

- resize creates a new array dynamically,

copies all element from previous stack,
releases old array and makes the point
stack point to the new array.

```
void resize (int size = 100)
{
   maxSize + size  = newSize ;
   int *tempStk = new int [newSize];
   for (int i= 0 ; i <maxSize ; i++)
    tempStr [i] = stack [i] ; // copy from
                                 old to new
    maxSize += size;
    delete [] stack; //release old stack
    stack = tempStk ;
}

 void push (int Element)
{
   if (isFull () )    resize ();
                    stack [top]= element;
                      top ++ ;
}
```

* Generic Stack: template <typename T>

```
class myStack
  {
    T *stack;
    int top, maxSize;
  public:
    myStack(int ---); ~myStack();
    bool push(T);
  void resize(int ------); // release and
                              resize the stack
};
```

* Stack Library Function: #include <stack>

↳ using : not need for implementation

** Practice in partical.

2.11.20

Queue

A queue is like a container with both end open, in FIFO (first in, first out) method.

* A queue is a waiting line — seen in daily life.
  - rear & front. (called ends)

Example: Ticket system, Bank system,

- There is many kinds of queue

In queue —
- Items are added only at rear.
- Items are removed only at front.

* Queue Implementation in C++:

```
int queue[100], rear, front, maxSize = 4;
// where front = rear = -1
// queue[] : holds the all elements.

bool isEmpty()   // returns True if queue has no elements
{
  if (front == -1) && (rear == -1)
```

```
return true;                    //underflow
else return false;
}
```

(ii) `bool isFull()` // returns True if queue full

```
{
  if (rear == maxSize - 1) && (front == rear)
    return true;
  else return false;            // overflow
}
```

(iii) `bool enqueue(int Element)`

```
{
  if (isEmpty())                // add element to back at the rear
  { front = rear = 0; }
else if (isFull())
  {
    cout << "Already FULL!" << endl;
    return true;
  }
  else { rear++; }
```

```
    queue[rear] = Element;
    return true;
}
```

(iv) bool dequeue() // removes element from the front

```
{
  if (isEmpty())
  {
    cout << "Can't Dequeue! Empty queue!"
         << endl;
    return false;
  }
else if (front != rear)
  {
    front = rear = -1; }
else      { front ++ ; }
  return true;
};
```

(v) int frontValue() // retrieve value of element from front

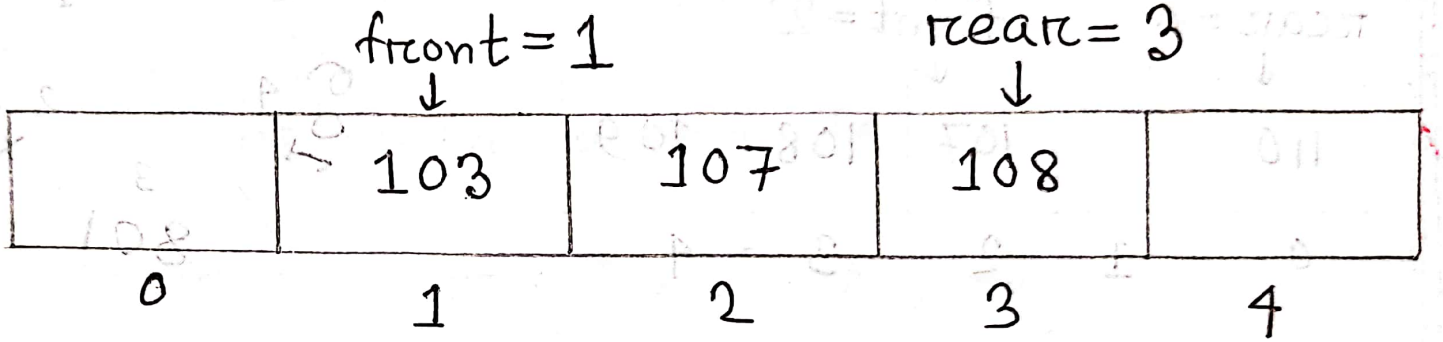
```
{ return queue[front];}
```

(vi) bool show() // print all the values of queue [front to rear]

```
{
 if (isEmpty())
 {
  cout << " Nouthing to show! Empty queue !" << endl;
  return false;
 }
 else
 for (int i = front ; i <= rear ; i++)
 {
   cout << queue[i] << "  " ;
 }
 cout << endl;  return true;
}
```

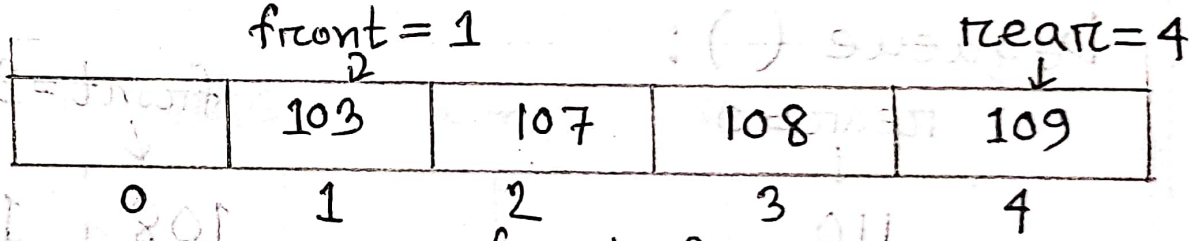
}

Example:



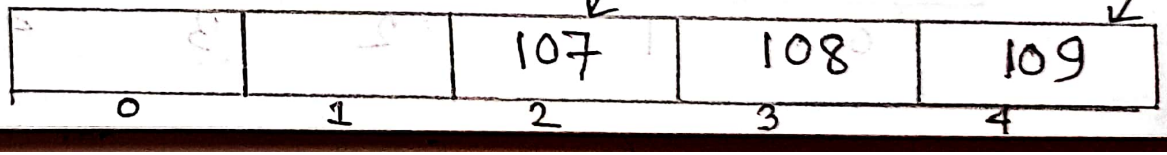
Enqueue (109):

rear++



Dequeue ():

front++



4. 11. 20

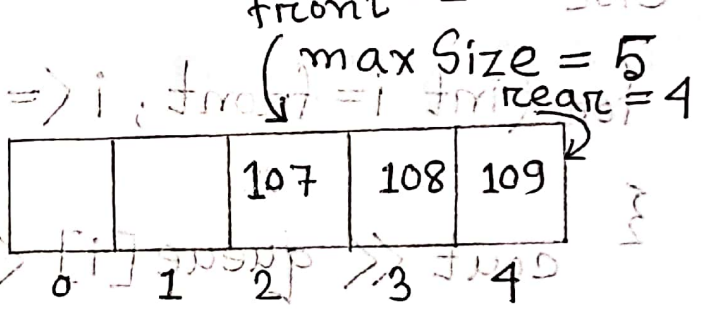
Circular Queue

Unlike a linear queue, a circular one can reuse an unused memory in the queue by circling back to it.

* Circular queue is more memory efficient.

Enqueue (110):

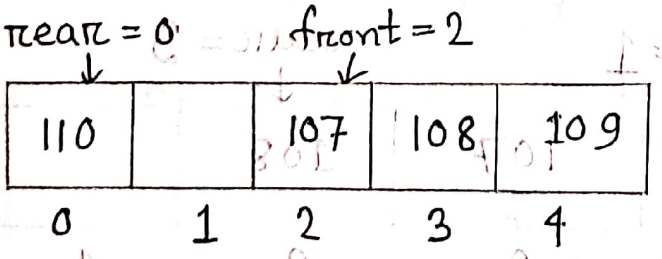
- When a linear queue:

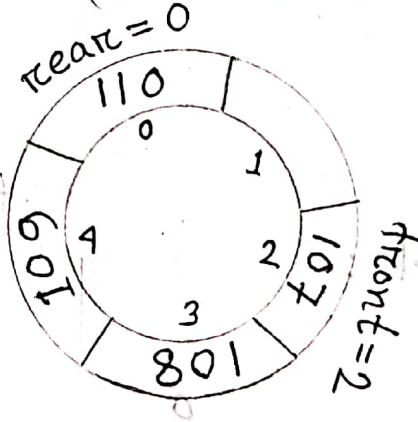


Here, rear == (max Size - 1) rear ≠ 5

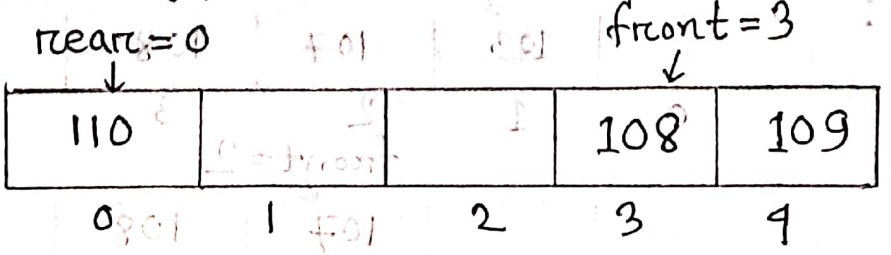
So, it is not possible to enqueue 110.

When a circular queue:





Dequeue ():



[max Size = 5]

* Simulation of Operations: front = rear = -1 (initialize)

enqueue: rear = (rear + 1) % maxSize ;

dequeue: front = (front + 1) % maxSize ;

9. 11. 20

Algebraic Expression

AE is a legal combination of operands and the operators.

*Operand ⟶ variable, constant (x, y, z, 0, 1, 2, 8)

⟶ quantity (unit of data): performed for mathematical operation.

* Operator ⟶ symbol: signifies logical operation between operands.

⟶ +, −, *, /, ^, %.

So, a expression is like: x + y * 5 − z.

- expression is of 3 types: infix, prefix, postfix.

Infix : operand operator operand

x + y

6 * 3

between

Postfix : operand operand operator

x y +

comes after

x y z + *

Prefix : operator operand operand

comes before

* + x y z

Simple Notation : (x+y) * z

Infix

- Infix notations are not as simple as they seem, while evaluating them

- To evaluate → consider: (i) Precedence & (ii) Associative Property.
  - ↳ governs the evaluation order of an expression. (পরিচালনা)

two rules:

(i) operator with higher precedence before an operator with lower precedence.

(ii) same precedence order → according to associativity order.

Example: $3+5*4+2-6$

$= 3+20+2-6 = 23+2-6 = 25-6 = 19$

Postfix → Left to right

Prefix → Right to left

* Infix expression is hard to parse.

Infix	Postfix	Prefix
(A+B)*(C+D)	AB+CD+*	*+AB+CD
A-B/(C*D^E)	ABCDE^*/-	-A/B*C^DE

Slide—12, 13, 14, 15 [reading & coding]

Parenthesis — বন্ধনী / Bracket

get:

(i) opening parenthesis → push into stack.

(ii) closing → check the top of stack.

→ if ToS has same p but
(i) opening → then pop it.

(iii) no parenthesis → skip the character.

For evaluating:

```
string::iterator i;
for (i = postfix.begin(); i != postfix.end(); i++)
{   if (operator(*i) != 1)
    {  a = stk.topElement();
       stk.pop();  //same as b
       stk.push(operation(a, b, *i)); }
    else if (operand(*i) > 0)
       { stk.push(number(*i)); }
    return stk.topElement();
```

11.11.20

Sorting [আজানো]

Arranges values in an specific order, in an array is called sorting.

- → alphabetical
- → increasing numerical
- → decreasing numerical

* Easy to find data → keeping datas in sorted order.

Huffman compression : classical data compression.

Combinatorial search : a classic paradigm in artifical intelligence.

* String processing algorithms are often based on
  → sorting.

- Sorting are of 3 types.

Bubble Sort

- compares adjacent numbers in pairs ;

  a > b → False → swap [a b → b a]

  a < b → True [a b → a b]

- pass one by one element and check (a>b or not).... and last element will be bubbled.

* Bubble sorting is really slow.

  But easy to understand and fast to implement.

Increasing order → Simulation:

(i) Given: 5 7 3 9 2 10

5 7 3 9 2 10

5 3 7 2 9 10

3 5 2 7 9 10

3 2 5 7 9 10

sorting order: 2 3 5 7 9 10

Total Pass: 5

[N-1]

(ii) Given: 62 43 76 45 34 90 66

43 62 45 76 34 76 66 90

43 45 34 62 66 76 90

43 34 45 62 66 76 90

34 43 45 62 66 76 90 Total Pass: 4

sorting order

Algorithm:

```
bool sorted = False;
do {
    sorted = false;
    for (int i=0 ; i<size - 1; i++)
    {
    if (arr [i] > arr [i+1]
        swap (arr, i, (i+1));
        sorted = true;
    }
} while (sorted);
```

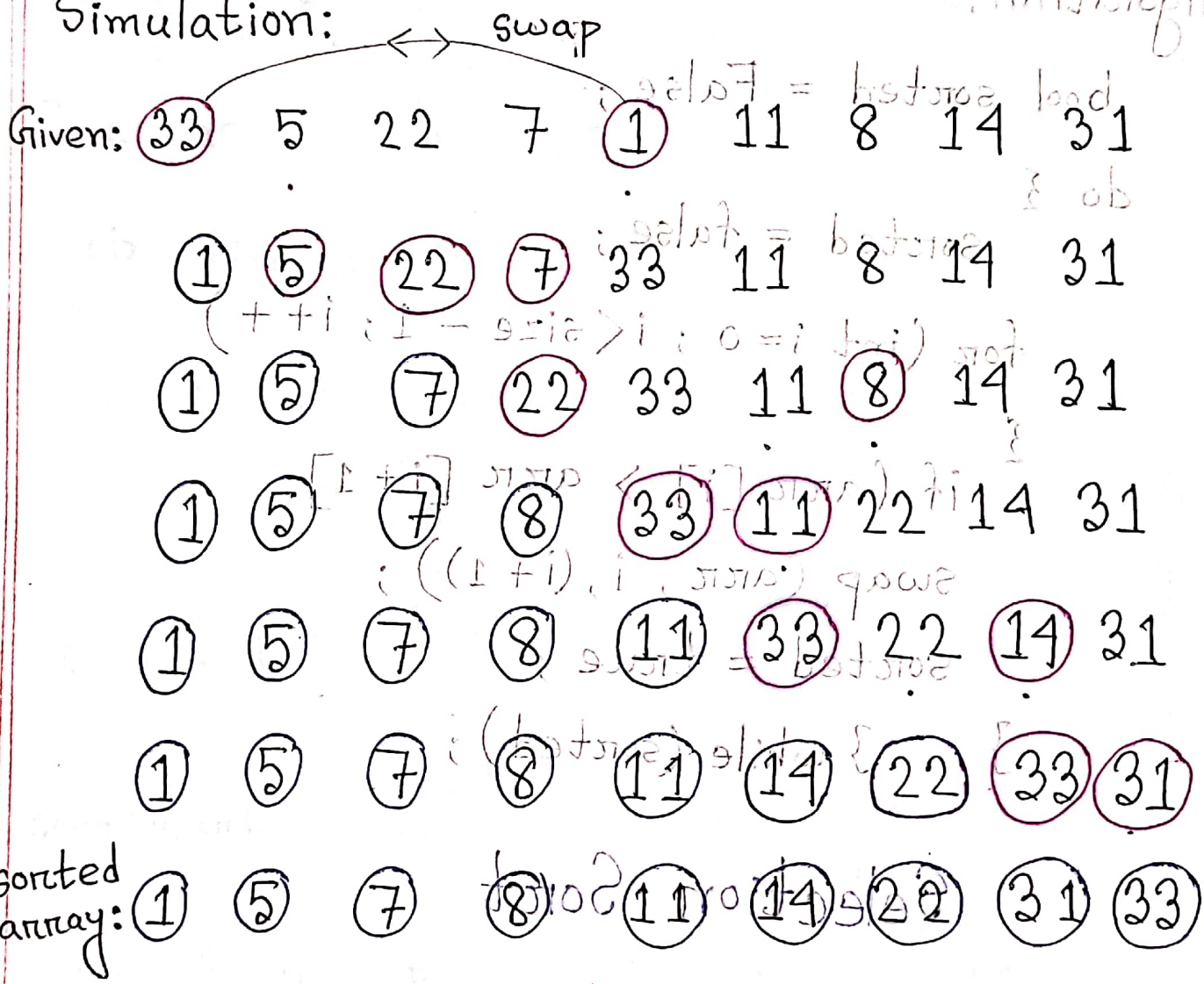
Selection Sort

- Locate smallest element in array.
- Interchange it with elements one by one in position 0, 1, 2, 3 .... accordingly.

* More efficient than Bubble sort; fewer exchanges.

- An array with N elements needs exactly N-1 selections for fixing index 0 to N-2.

Simulation:



Algorithm:

```
int arr[], size)
    int min = 0;
for(int i=0; i<size-1; i++)
```

```
{   min = i ;
    for (int j= i+1 ; j<size ; j++)
      if (arr [j] <arr [min])
       {  min = j ;  }
    if (min ! = i)
      swap (arr, min, i)
}
```

```
// swap :
swap (arr [i], a, b)
{ int temp = arr [a];
  arr [a] = arr [b];
  arr [b] = temp ; }
```

Insertion Sort

- value moved to it's right place ; between the range.

Time complexity : n^2 .

Simulation :

Given : 5 21 70 24 77 11 3 42

5 21 (24) 70 77 11 3 42

5 (11) 21 24 70 77 3 42

(3) 5 11 21 24 70 77 42

Sorted array: 3 5 11 21 24 (42) 70 77

Here, element moving numbers: 4.

Algorithm:

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

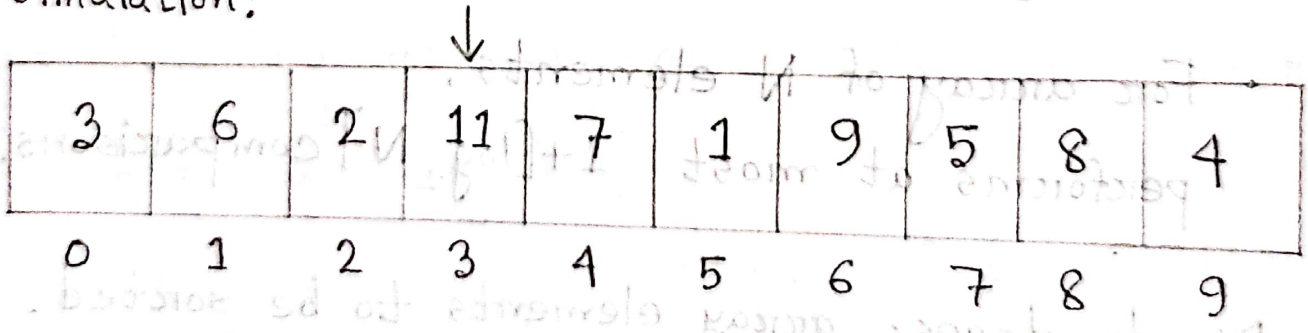
* The insertion may need to shift one or more elements to place the element at right position.

Searching

Search: locate an item in a list of data/information.

Linear Search	Binary Search
• searches sequentially (serially) for an element.	• searches an element by dividing the sorted elements in a list into two sub-lists.
• starts from first element.	• starts with middle element.

Linear Search Simulation:



value to search: 11

found: true

position: 3

[starting position = -1]

[false; position = -1]

Algorithm:

```
int LinearSearch (int arr[], int key, int size)
{
  for (int i = 0 ; i < size ; i++)
  {  if (arr[i] = key)
     return i ;
  }  return -1;
}
```

Binary Search

* Binary search is more efficient than linear search.

- For array of N elements,
  performs at most $1+[\log_2 N]$ comparisons.

Disadvantages: array elements to be sorted.

Simulation: [$\log_2$ base$_2$ → better]

Sorted array:

3	4	5	6	7	8	9	11
0	1	2	3	4	5	6	7

value to search: 8

index

first : 0 → 3

Here, 8 > mid value (6)

last : 7 → 11

So, first = mid + 1

mid : ⌊(0+7)/2⌋ = 3 ↓ 6

= 4 → 7

(new) mid = ⌊(4+7)/2⌋ = 5 → 8

∴ value found at index 5.

value to search: 5.

So, last = mid − 1

= 2 → 5

Here, 5 < mid value (6)

∴ value found at index 2.

Algorithm:

```
int BinarySearch (int arr[ ], int low, int high, int key)
{  while (low <= high)
   { int mid = (high + low) / 2;
     if (arr [mid] == key)
     return mid ;
     if (arr [mid] < key)
       low = mid + 1 ;
     BinarySearch (arr, low, (mid -1), key); }
     if (arr [mid] > key )
     high = mid - 1;
     BinarySearch (arr, (mid+1),
                   high , key);
     }
     return -1 ;
       " Not found" ;
```

(Final)

23.11.20

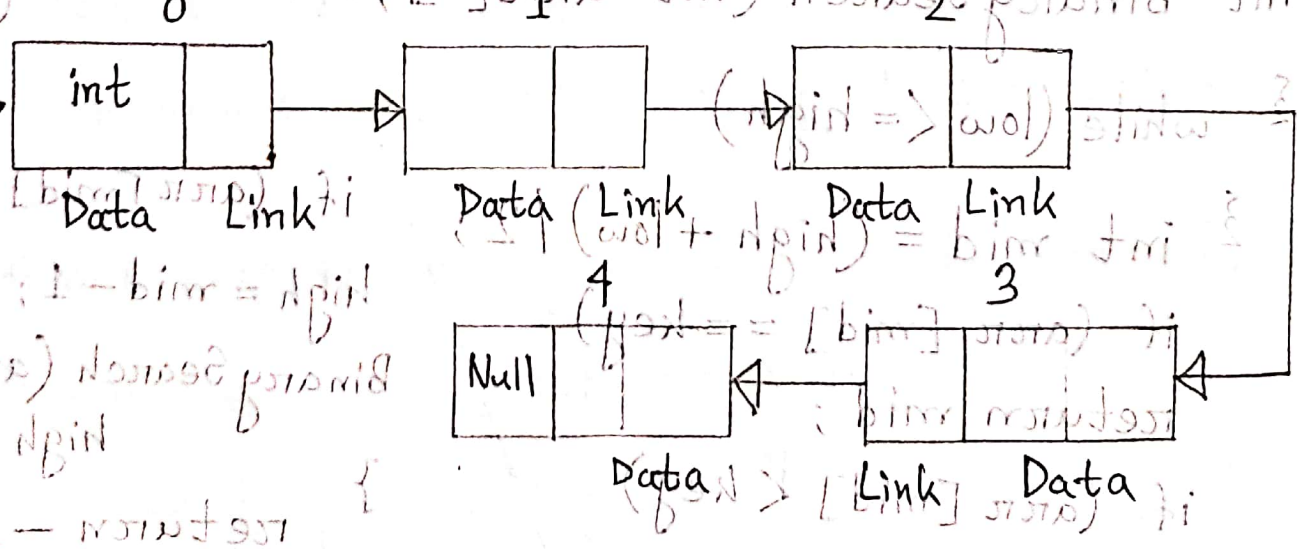
Lecture 12

Linked List

→ Linear data structure

Linked list is a data structure, consisting of a group of memory space, which together represent a list → a sequence of data.

- Each data is stored in a separate memory space / block → called node.
- Each memory block contains the data along with link / location / address to the memory location for the next data in the list.



consecutive → পরপর

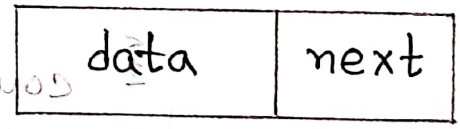
- In a linked list, the data aren't stored consecutively / serially in the memory.

sparse → বিরল

Example: Image viewers, Previous and next page in web browsers, Music Players.

* Representation of a Node:

```
struct ListNode
{
    int data;
    ListNode *next;
}; ListNode node;
```



Traversal → ~~sort~~ way of sorting

Linked List : (i) Searching

(ii) Insertion

(iii) Deletion

30.11.20

TREE

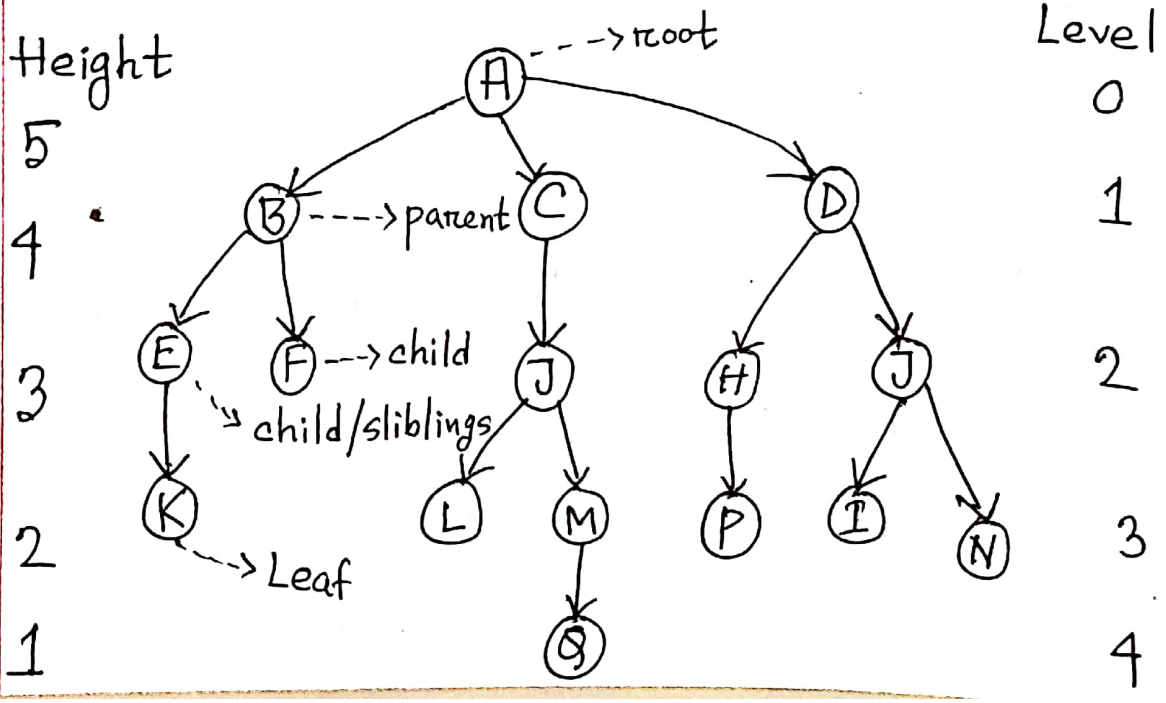
- Non-linear data structure. || • represents hierarchical data.
- Storing data in a way that makes it easily searchable.
- represents → sorted lists of data.
- Routing algorithms.

Rout-ing → deciding that incoming data will go which route (way) : left or right.

Each node → references a value and a list of references. (to it's children)

Every root has a root; never referenced by any other node.

- One node → root node in tree.



Leaf → which node has no ~~more~~ children.

Internal vertex / Parent → which node has children.

Subtree, Ancestor – পূর্বপুরুষ Descendant – বংশধর

m-ary tree : has no more than m children.

2-ary tree / Binary tree → at most 2 children.

[1/2] "

Complete Binary Tree → last 2 levels has leaves only.
→ might be absent.

Full Binary Tree → last / bottom level must be filled up.

• Full Binary Tree is complete binary tree, but ~~ve~~ cbt isn't fbt.

Nodes at each level : 2^{level} ; height: $\lceil \log_2(node) \rceil$

maximum node : 2^{level} , minimum node : $2^{level-1}$

Total ~~to~~ nodes : $2^0 + 2^1 + 2^2 + \dots + 2^{level-1}$

$= 2^{level} - 1$

→ Tree : Node Height = Tree Height − Node Level

2.12.20

Tree Traversal

Traversal: Systematic way of visiting all nodes.

Method: Inorder, Pre order, Post order

Traverse left subtree > right subtree

Inorder → left, root, right.

Pre order → root, left, right.

Post order → ~~right~~ left, right, root.

Binary Search Tree

1st node → root

value < root ----> left

value > root ----> right

Delete: (i) leaf node → just delete

(ii) node has only one child →

connect node's child to it's parent

and delete ... (common)

(iii) node has both (two) child ⟶ replace with the child with the highest value from left sub-tree ~~or~~ or with the lowest value from right sub-tree

BST doesn't insert duplicate values.

5.12.20

Heap

Heap is a index wise binary tree.

It must be a complete binary tree ; right leaf can only be missing.

- parent ≥ children.

Heap has two properties : Structure property and Order (heap) property.

~~If~~ ~~he~~ ⊛ left ⟶ right ; top ⟶ bottom.

⊛ Heap's elements are ordered top to down , but not ordered from left to right.

A Heap Not a heap Not a heap

- Heap → two types.

Max Heap	Min Heap
root → largest	root → smallest
nodes: parent ≥ child	parent ≤ chid

Draw Heap

array [heap as an array] : $n = 11$

i=	0	1	2	3	4	5	6	7	8	9	10

parent index : i

left child index : $2i+1$

right child index : $2i+2$

Draw Heap

parent : $\lfloor (i-1)/2 \rfloor$

Heapsize [arr] $\leq$ length [arr]

Max Heap properties:

(i) Max heapify (ii) Build max-heap (iii) Heapsort

How to return --> i) compare i^{th} value with it's left and right child key value, to find the largest one.

ii) Sort : i = 0 exchange with i = (size − 1) and repeat it recursively and heapify it.

Insertion : insert at last index + and heapify it.

Removal : heapify down -- called.

(i) replace / exchange → with last element.

(ii) reduce heap size by one and then heapify it.

[· delete root → exchange with last element.

Heapsort : {small to large}.

· Heap implements priority queue.

(ii) Sorting. → heap sort

Height, $h = \lfloor \log_2 n \rfloor$; n = node

· Heaps are based upon trees.

9.12.20

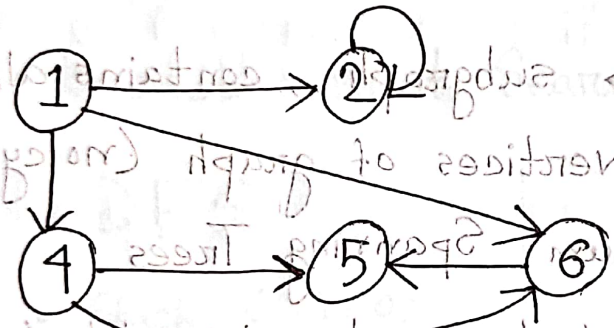
Graphs

Directed ↳ ordered $(u,v) \neq (v,u)$

Undirected $(u,v) = (v,u)$

Acyclic Graph → circuit
- starts and end at same vertex.

Acyclic Graph → path (not circuit)



← self loop : (2,2)

degree : in degree + out degree

self loop deg : 2

adjacent : connected nodes.

degree of 6 : 3 in - : 2 out - 1

Complete graph : maximum edges.
connected to every nodes.

Dense graph : number of edges is closed to maximum.

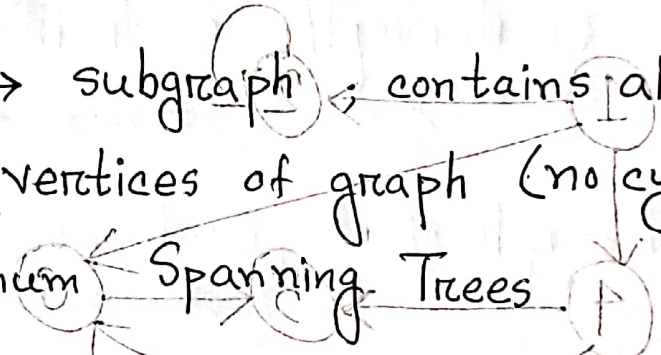
Sparse graph : number of edges is very few.
(using → adjacency matrix to store)

Weighted graph → associates weights with either vertices or edges.

Connected component → subgraph of supergraph.

Spanning Tree → subgraph ; contains all vertices of graph (no cycle)

MST → Minimum Spanning Trees



- Undirected connected graph - weight (w) on (V, E)
- number of edges = vertex − 1

MST : total weight of tree is minimum.

Prim's algorithm:

(i) vertex based algorithm

(ii) edge can be found with priority queue.

- Which one is better?

  => Prim's algorithm.

Kruskal's algorithm: (i) edge based algorithm.

(ii) maintains → forest of trees.

(iii) edge can be done just by sorting.

Way of graph traversing and searching : 2

- Depth-first Search (DFS)
- Breadth-first Search (BFS)

DFS	BFS
• Stack (LIFO)	• Queue (FIFO)
• Path wise search.	• Level wise search
• Tree & edges & cross edges.	• Back edges & forward edges.

Back: child to parent

Forward: parent to child

cross: all other edges (undiscovered)

Motive : Find shortest path.

- Which one is better?

=> BFS.