ADVANCED DATA STRUCTURES & ALGORITHMS

Prof. P.G.Joshi

SYLLABUS

ADVANCED DATA STRUCTURES & ALGORITHMS

UNIT I 15 Hrs Algorithm – Definition and Characteristics, Efficiency of an algorithm, Time and space complexity Asymptotic notations: big-oh notation (O), big-omega notation (Ω), big-theta notation (θ) , Time space trade-off , Abstract data type, Introduction to data structures: Need of data structure, Categories of data structures, operations; Arrays storage representation of 1D, 2D and Multi-dimensional arrays, Sparse matrix, operations and Representation, Applications of array

UNIT II 12 Hrs Stacks & Queues: Stack - Definition, Operation on stack, Implementation using arrays and linked lists, Applications of Stacks, Function Calling, Recursion- direct & indirect recursion,Evaluation of arithmetic Expressions, Conversion of Expressions - Prefix, Infix and Postfix expressions. Queues: Definition, Implementations using arrays and linked lists, Circular queue, Dequeue, Priority queues, Applications of queues.

UNIT III 15 Hrs Linked list: Concept of linked list, Operations on Linked list: Creation, insertion, Deletion, Insertion and deletion examples, Types of linked list: Circular linked list, Doubly linked list, Implementation of stack and queue using linked list Trees Graphs and their applications Trees terminology, Binary tree, Traversal methods, Tree traversal algorithms, Threaded binary tree, Graph terminologies, Graph Representation: Sequential and Linked list, Graph Traversal – Depth First Search, Breadth First Search, Applications, AOV network, topological sort

UNIT IV 12 Hrs Searching: Linear and Binary Sorting: Bubble sort, Selection sort, Insertion sort More algorithms: Divide and Conquer, Greedy Method, Dynamic Programming, Backtracking

Reference books 1. Data Structure using C by A.M. Tanenbaum, Yecidyan lang 2. Data Structure Through C- By Dr. Sahani. 3. Data Structures Using C Yashwant Kanitkar – BPB Publication 4. Data Structures Through C (A Practical Approach), G.S Baluja Danapat Rai & Co. 5. Fundamentals of Data Structures, Ellis Horowitz and Sartaj Sajni Galgotia Publications 6. Introduction to data structures in C , Ashok N. Kamthane, Pearson Education 7. Theory and Problems of Data Structures, Schaums Outline Series, Seymour Lipschutz

CONTENTS

UNIT –I

Introduction to Data structures

In computer terms, a data structure is a Specific way to store and organize data in a computer's memory so that these data can be used efficiently later. Data may be arranged in many different ways such as the logical or mathematical model for a particular organization of data is termed as a data structure. The variety of a particular data model depends on the two factors -

- Firstly, it must be loaded enough in structure to reflect the actual relationships of the data with the real world object.
- Secondly, the formation should be simple enough so that anyone can efficiently process the data each time it is necessary.

Categories of Data Structure:

The data structure can be sub divided into major types:

- Linear Data Structure
- Non-linear Data Structure

Linear Data Structure:

A data structure is said to be linear if its elements combine to form any specific order. There are basically two techniques of representing such linear structure within memory.

- First way is to provide the linear relationships among all the elements represented by means of linear memory location. These linear structures are termed as arrays.
- The second technique is to provide the linear relationship among all the elements represented by using the concept of pointers or links. These linear structures are termed as linked lists.

The common examples of linear data structure are:

- Arrays
- Queues
- Stacks
- Linked lists

Non linear Data Structure:

This structure is mostly used for representing data that contains a hierarchical relationship among various elements.

Examples of Non Linear Data Structures are listed below:

- Graphs
- family of trees and
- table of contents

Tree: In this case, data often contain a hierarchical relationship among various elements. The data structure that reflects this relationship is termed as rooted tree graph or a tree.

Graph: In this case, data sometimes hold a relationship between the pairs of elements which is not necessarily following the hierarchical structure. Such data structure is termed as a Graph.

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

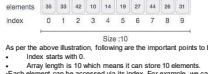
- **Element** − Each item stored in an array is called an element.

•**Index** − Each location of an element in an array has a numerical index, which is used to identify the element.

Array Representation:(Storagestructure)

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



As per the above illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.

•Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.

Basic Operations

Following are the basic operations supported by an array.

- **Traverse** − print all the array elements one by one.
- **Insertion** − Adds an element at the given index.
- **Deletion** − Deletes an element at the given index.
- **Search** − Searches an element using the given index or by the value.
- **Update** − Updates an element at the given index.

In C, when an array is initialized with size, then it assigns defaults values to its elements in followino order.

Data Type	Default Value
bool	false
char	0
int	0
float	0.0
double	0.0f
void	
wchar_t	0

Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Here, we see a practical implementation of insertion operation, where we add data at the end of the array −

Algorithm

Let **LA** be a Linear Array (unordered) with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm where ITEM is inserted into the Kth position of LA −

```
1. Start
2. Set J = N
3. Set N = N+1
4. Repeat steps 5 and 6 while J >= K
5. Set LA[J+1] = LA[J]
6. Set J = J-1
7. Set LA[K] = ITEM
8. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>

main() {
   int LA[] = {1,3,5,7,8};
   int item = 10, k = 3, n = 5;
   int i = 0, j = n;
   printf("The original array elements are :\n");
   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
```

```
   n = n + 1;
   while( j >= k) {
      LA[j+1] = LA[j];
      j = j - 1;
   }
   LA[k] = item;
   printf("The array elements after insertion :\n");
   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after insertion :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 10
LA[4] = 7
LA[5] = 8
```

Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to delete an element available at the Kth position of LA.

```
1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J] = LA[J + 1]
5. Set J = J+1
6. Set N = N-1
7. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>

void main() {
   int LA[] = {1,3,5,7,8};
   int k = 3, n = 5;
   int i, j;
   printf("The original array elements are :\n");
   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }

   j = k;
   while( j < n) {
      LA[j-1] = LA[j];
      j = j + 1;
   }
   n = n -1;
   printf("The array elements after deletion :\n");
   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after deletion :
LA[0] = 1
LA[1] = 3
LA[2] = 7
LA[3] = 8
```

Lecture-02

Search Operation

You can perform a search for an array element based on its value or its index.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to find an element with a value of ITEM using sequential search.

```
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>

void main() {
   int LA[] = {1,3,5,7,8};
   int item = 5, n = 5;
   int i = 0, j = 0;
      printf("The original array elements are :\n");
         for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
      while( j < n){
      if( LA[j] == item ) {
         break;
      }
      j = j + 1;
   }
   printf("Found element %d at position %d\n", item, j+1);
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
Found element 5 at position 3
```

Update Operation

Update operation refers to updating an existing element from the array at a given index.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to update an element available at the Kth position of LA.

```
1. Start
2. Set LA[K-1] = ITEM
3. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>

void main() {
   int LA[] = {1,3,5,7,8};
   int k = 3, n = 5, item = 10;
   int i, j;
      printf("The original array elements are :\n");
   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }

   LA[k-1] = item;
   printf("The array elements after updation :\n");
   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
```

```
The array elements after updation :
LA[0] = 1
LA[1] = 3
LA[2] = 10
LA[3] = 7
LA[4] = 8
```

Lecture-03

Sparse Matrix and its representations

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have **0 value**, then it is called a sparse matrix.

Why to use Sparse Matrix instead of simple matrix ?

-Storage: There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.

-Computing time: Computing time can be saved by logically designing a data structure traversing only non-zero elements..

Example:

```
0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0
```

Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with **triples- (Row, Column, value).**

Sparse Matrix Representations can be done in many ways following are two common representations:

1. Array representation
2. Linked list representation

Method 1: Using Arrays

```
#include<stdio.h>
 int main()
{
   // Assume 4x5 sparse matrix
   int sparseMatrix[4][5] =
   {
      {0 , 0 , 3 , 0 , 4 },
      {0 , 0 , 5 , 7 , 0 },
      {0 , 0 , 0 , 0 , 0 },
      {0 , 2 , 6 , 0 , 0 }
   };

   int size = 0;
   for (int i = 0; i < 4; i++)
      for (int j = 0; j < 5; j++)
         if (sparseMatrix[i][j] != 0)
            size++;
   int compactMatrix[3][size];
   // Making of new matrix
```

```
int k = 0;
for (int i = 0; i < 4; i++)
    for (int j = 0; j < 5; j++)
        if (sparseMatrix[i][j] != 0)
        {
            compactMatrix[0][k] = i;
            compactMatrix[1][k] = j;
            compactMatrix[2][k] = sparseMatrix[i][j];
            k++;
        }
 for (int i=0; i<3; i++)
{
    for (int j=0; j<size; j++)
        printf("%d ", compactMatrix[i][j]);
    printf("\n");
}
 return 0;
}
```

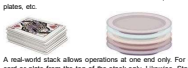
0	0	3	0	4		
0	0	5	7	0		
0	0	0	0	0		
0	2	6	0	0		

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

UNIT II Lecture-04

STACK

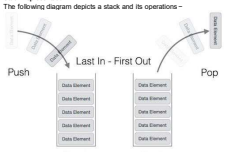
A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.



A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.
This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.

- **pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.
- **isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions −

peek()

Algorithm of peek() function −

```
begin procedure peek
   return stack[top]
end procedure
```

Implementation of peek() function in C programming language −

Example

```
int peek() {
   return stack[top];
}
```

isfull()

Algorithm of isfull() function −

```
begin procedure isfull

   if top equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

Implementation of isfull() function in C programming language −

Example

```
bool isfull() {
   if(top == MAXSIZE)
      return true;
   else
      return false;
}
```

isempty()

Algorithm of isempty() function −

```
begin procedure isempty

   if top less than 1
      return true
   else
      return false
   endif

end procedure
```

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code −

Example

```
bool isempty() {
   if(top == -1)
      return true;
   else
      return false;
}
```

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.

If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows −

```
begin procedure push: stack, data

   if stack is full
      return null
   endif

   top ← top + 1
   stack[top] ← data

end procedure
```

Implementation of this algorithm in C, is very easy. See the following code −

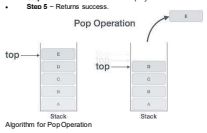
Example

```
void push(int data) {
   if(!isFull()) {
      top = top + 1;
      stack[top] = data;
   } else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space. A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.



Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows −

```
begin procedure pop: stack

   if stack is empty
      return null
   endif

   data ← stack[top]
   top ← top - 1
   return data

end procedure
```

Implementation of this algorithm in C, is as follows −

Example

```
int pop(int data) {

   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}
```

Lecture-05

Stack Applications

Three applications of stacks are presented here. These examples are central to many activities that a computer must do and deserve time spent with them.

1. Expression evaluation
2. Backtracking (game playing, finding paths, exhaustive searching)
3. Memory management, run-time environment for nested language features.

Expression evaluation
In particular we will consider arithmetic expressions. Understand that there are boolean and logical expressions that can be evaluated in the same way. Control structures can also be treated similarly in a compiler.
This study of arithmetic expression evaluation is an example of problem solving where you solve a simpler problem and then transform the actual problem to the simpler one.
Aside: *The NP-Complete problem*. There are a set of apparently intractable problems: finding the shortest route in a graph (Traveling Salesman Problem), bin packing, linear programming, etc. that are similar enough that if a polynomial solution is ever found (exponential solutions abound) for one of these problems, then the solution can be applied to all problems.

Infix, Prefix and Postfix Notation
We are accustomed to write arithmetic expressions with the operation between the two operands: a+b or c/d. If we write a+b*c, however, we have to apply precedence rules to avoid the ambiguous evaluation (add first or multiply first?).
There's no real reason to put the operation between the variables or values. They can just as well precede or follow the operands. You should note the advantage of prefix and postfix: the need for precedence rules and parentheses are eliminated.

Infix	Prefix	Postfix
a + b	+ a b	a b +
a + b * c	+ a * b c	a b c * +
(a + b) * (c - d)	* + a b - c d	a b + c d - *
b * b - 4 * a * c		
40 - 3 * 5 + 1		

Postfix expressions are easily evaluated with the aid of a stack.

Infix, Prefix and Postfix Notation KEY

Infix	Prefix	Postfix
a + b	+ a b	a b +

a + b * c	+ a * b c	a b c * +
(a + b) * (c - d)	* + a b - c d	a b + c d - *
b * b - 4 * a * c	- * b b * * 4 a c	b b * 4 a * c * -
40 - 3 * 5 + 1 = 26	+ - 40 * 3 5 1	40 3 5 * - 1 +

Postfix Evaluation Algorithm
Assume we have a string of operands and operators, an informal, by hand process is

1. Scan the expression left to right
2. Skip values or variables (operands)
3. When an operator is found, apply the operation to the preceding two operands
4. Replace the two operands and operator with the calculated value (three symbols are replaced with one operand)
5. Continue scanning until only a value remains–the result of the expression

The time complexity is O(n) because each operand is scanned once, and each operation is performed once.
A more formal algorithm:

```
create a new stack
while(input stream is not empty){
  token = getNextToken();
  if(token instanceof operand){
    push(token);
  } else if (token instance of operator)
    op2 = pop();
    op1 = pop();
    result = calc(token, op1, op2);
    push(result);
  }
}
return pop();
```

Demonstration with 2 3 4 + * 5 -

Infix transformation to Postfix
This process uses a stack as well. We have to hold information that's expressed inside parentheses while scanning to find the closing ")". We also have to hold information on operations that are of lower precedence on the stack. The algorithm is:

1. Create an empty stack and an empty postfix output string/stream
2. Scan the infix input string/stream left to right
3. If the current input token is an operand, simply append it to the output string (note the examples above that the operands remain in the same order)
4. If the current input token is an operator, pop off all operators that have equal or higher precedence and append them to the output string; push the operator onto the stack. The order of popping is the order in the output.
5. If the current input token is '(', push it onto the stack
6. If the current input token is ')', pop off all operators and append them to the output string until a '(' is popped; discard the '(';
7. If the end of the input string is found, pop all operators and append them to the output string.

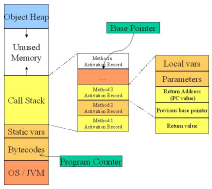
This algorithm doesn't handle errors in the input, although careful analysis of parenthesis or lack of parenthesis could point to such error determination.
Apply the algorithm to the above expressions.

Backtracking
Backtracking is used in algorithms in which there are steps along some path (state) from some starting point to some goal.

- Find your way through a maze.
- Find a path from one point in a graph (roadmap) to another point.
- Play a game in which there are moves to be made (checkers, chess).

In all of these cases, there are choices to be made among a number of options. We need some way to remember these decision points in case we want/need to come back and try the alternative
Consider the maze. At a point where a choice is made, we may discover that the choice leads to a dead-end. We want to retrace back to that decision point and then try the other (next) alternative.
Again, stacks can be used as part of the solution. Recursion is another, typically more favored, solution, which is actually implemented by a stack.

Memory Management
Any modern computer environment uses a stack as the primary memory management model for a running program. Whether it's native code (x86, Sun, VAX) or JVM, a stack is at the center of the run-time environment for Java, C++, Ada, FORTRAN, etc.
The discussion of JVM in the text is consistent with NT, Solaris, VMS, Unix runtime environments.
Each program that is running in a computer system has its own memory allocation containing the typical layout as shown below.



Call and return process
When a method/function is called

1. An activation record is created; its size depends on the number and size of the local variables and parameters.
2. The Base Pointer value is saved in the special location reserved for it
3. The Program Counter value is saved in the Return Address location
4. The Base Pointer is now reset to the new base (top of the call stack prior to the creation of the AR)
5. The Program Counter is set to the location of the first bytecode of the method being called
6. Copies the calling parameters into the Parameter region
7. Initializes local variables in the local variable region

While the method executes, the local variables and parameters are simply found by adding a constant associated with each variable/parameter to the Base Pointer.
When a method returns

1. Get the program counter from the activation record and replace what's in the PC
2. Get the base pointer value from the AR and replace what's in the BP
3. Pop the AR entirely from the stack.

Lecture-06

QUEUE
Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation
As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure –

Queue

As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations
Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues –

- **enqueue()** – add (store) an item to the queue.
- **dequeue()** – remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are –

- **peek()** – Gets the element at the front of the queue without removing it.
- **isfull()** – Checks if the queue is full.
- **isempty()** – Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.
Let's first learn about supportive functions of a queue –
peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows –

Algorithm

```
begin procedure peek
   return queue[front]
end procedure
```

Implementation of peek() function in C programming language –

Example

```
int peek() {
   return queue[front];
}
```

isfull()
As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ.Algorithm of isfull() function –

Algorithm

```
begin procedure isfull

   if rear equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

Implementation of isfull() function in C programming language –

Example

```
bool isfull() {
   if(rear == MAXSIZE - 1)
      return true;
   else
      return false;
}
```

isempty()
Algorithm of isempty() function –

Algorithm

```
begin procedure isempty

   if front is less than MIN OR front is greater than rear
      return true
```

```
   else
      return false
   endif

end procedure
```

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code −

Example

```
bool isempty() {
   if(front < 0 || front > rear)
      return true;
   else
      return false;
}
```

Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − return success.

Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue operation

```
procedure enqueue(data)

   if queue is full
      return overflow
   endif

   rear ← rear + 1
   queue[rear] ← data
   return true

end procedure
```

Implementation of enqueue() in C programming language −

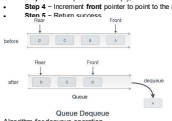
Example

```
int enqueue(int data)
   if(isfull())
      return 0;

   rear = rear + 1;
   queue[rear] = data;

   return 1;
end procedure
```

Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation −

- **Step 1** − Check if the queue is empty.
- **Step 2** − If the queue is empty, produce underflow error and exit.
- **Step 3** − If the queue is not empty, access the data where **front** is pointing.
- **Step 4** − Increment **front** pointer to point to the next available data element.
- **Step 5** − Return success.



Queue Dequeue

Algorithm for dequeue operation

```
procedure dequeue

   if queue is empty
      return underflow
   end if

   data = queue[front]
   front ← front + 1
   return true

end procedure
```

Implementation of dequeue() in C programming language −

Example

```
int dequeue() {
   if(isempty())
      return 0;

   int data = queue[front];
   front = front + 1;

   return data;
}
```

UNIT III Lecture-07

LINKED LIST

A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** − Each link of a linked list can store a data called an element.
- **Next** − Each link of a linked list contains a link to the next link called Next.

•**LinkedList** − A Linked List contains the connection link to the first link called First.

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.

As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

Types of Linked List

Following are the various types of linked list.

- **Simple Linked List** − Item navigation is forward only.
- **Doubly Linked List** − Items can be navigated forward and backward.

•**Circular Linked List** − Last item contains link of the first element as next and the first element has a link to the last element as previous.

Basic Operations

Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Display** − Displays the complete list.
- **Search** − Searches an element using the given key.
- **Delete** − Deletes an element using the given key.

Insertion Operation

Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.

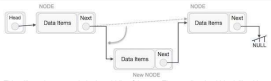
Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode) and **C** (RightNode). Then point B.next to C −

```
NewNode.next −> RightNode;
```

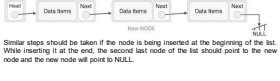
It should look like this −

Now, the next node at the left should point to the new node.

```
LeftNode.next −> NewNode;
```



This will put the new node in the middle of the two. The new list should look like this −



Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.

The left (previous) node of the target node now should point to the next node of the target node −

```
LeftNode.next −> TargetNode.next;
```

This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

```
TargetNode.next −> NULL;
```

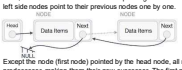
We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

Reverse Operation

This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.

First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node –

We have to make sure that the last node is not the lost node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their *new* successor. The first node will point to NULL.

We'll make the head node point to the new first node by using the temp node.

The linked list is now reversed.

Program:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

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

```
struct node *head = NULL;
struct node *current = NULL;

//display the list
void printList() {
   struct node *ptr = head;
   printf("\n[ ");

   //start from the beginning
   while(ptr != NULL) {
      printf("(%d,%d) ",ptr->key,ptr->data);
      ptr = ptr->next;
   }

   printf(" ]");
}

//insert link at the first location
void insertFirst(int key, int data) {
   //create a link
   struct node *link = (struct node*) malloc(sizeof(struct node));

   link->key = key;
   link->data = data;

   //point it to old first node
   link->next = head;

   //point first to new first node
   head = link;
}

//delete first item
struct node* deleteFirst() {

   //save reference to first link
   struct node *tempLink = head;

   //mark next to first link as first
   head = head->next;

   //return the deleted link
```

```
   return tempLink;
}

//is list empty
bool isEmpty() {
   return head == NULL;
}

int length() {
   int length = 0;
   struct node *current;

   for(current = head; current != NULL; current = current->next) {
      length++;
   }

   return length;
}

//find a link with given key
struct node* find(int key) {

   //start from the first link
   struct node* current = head;

   //if list is empty
   if(head == NULL) {
      return NULL;
   }

   //navigate through list
   while(current->key != key) {

      //if it is last node
      if(current->next == NULL) {
         return NULL;
      } else {
         //go to next link
         current = current->next;
      }
   }
```

```
   //if data found, return the current Link
   return current;
}

//delete a link with given key
struct node* delete(int key) {

   //start from the first link
   struct node* current = head;
   struct node* previous = NULL;

   //if list is empty
   if(head == NULL) {
      return NULL;
   }

   //navigate through list
   while(current->key != key) {

      //if it is last node
      if(current->next == NULL) {
         return NULL;
      } else {
         //store reference to current link
         previous = current;
         //move to next link
         current = current->next;
      }
   }

   //found a match, update the link
   if(current == head) {
      //change first to point to next link
      head = head->next;
   } else {
      //bypass the current link
      previous->next = current->next;
   }

   return current;
}
```

```
void sort() {

   int i, j, k, tempKey, tempData;
   struct node *current;
   struct node *next;

   int size = length();
   k = size ;

   for ( i = 0 ; i < size - 1 ; i++, k-- ) {
      current = head;
      next = head->next;

      for ( j = 1 ; j < k ; j++ ) {

         if ( current->data > next->data ) {
            tempData = current->data;
            current->data = next->data;
            next->data = tempData;

            tempKey = current->key;
            current->key = next->key;
            next->key = tempKey;
         }

         current = current->next;
         next = next->next;
      }
   }
}

void reverse(struct node** head_ref) {
   struct node* prev = NULL;
   struct node* current = *head_ref;
   struct node* next;

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

```
   *head_ref = prev;
}

void main() {
   insertFirst(1,10);
   insertFirst(2,20);
   insertFirst(3,30);
   insertFirst(4,1);
   insertFirst(5,40);
   insertFirst(6,56);

   printf("Original List: ");

   //print list
   printList();

   while(!isEmpty()) {
      struct node *temp = deleteFirst();
      printf("\nDeleted value:");
      printf("(%d,%d) ",temp->key,temp->data);
   }

   printf("\nList after deleting all items: ");
   printList();
   insertFirst(1,10);
   insertFirst(2,20);
   insertFirst(3,30);
   insertFirst(4,1);
   insertFirst(5,40);
   insertFirst(6,56);

   printf("\nRestored List: ");
   printList();
   printf("\n");

   struct node *foundLink = find(4);

   if(foundLink != NULL) {
      printf("Element found: ");
      printf("(%d,%d) ",foundLink->key,foundLink->data);
      printf("\n");
```

```
} else {
   printf("Element not found.");
}

delete(4);
printf("List after deleting an item: ");
printList();
printf("\n");
foundLink = find(4);

if(foundLink != NULL) {
   printf("Element found: ");
   printf("(%d,%d) ",foundLink->key,foundLink->data);
   printf("\n");
} else {
   printf("Element not found.");
}

printf("\n");
sort();

printf("List after sorting the data: ");
printList();

reverse(&head);
printf("\nList after reversing the data: ");
printList();
}
```

If we compile and run the above program, it will produce the following result −

Output

```
Original List:
[ (6,56) (5,40) (4,1) (3,30) (2,20) (1,10) ]
Deleted value:(6,56)
Deleted value:(5,40)
Deleted value:(4,1)
Deleted value:(3,30)
Deleted value:(2,20)
Deleted value:(1,10)
List after deleting all items:
[ ]
Restored List:
[ (6,56) (5,40) (4,1) (3,30) (2,20) (1,10) ]
```

```
Element found: (4,1)
List after deleting an item:
[ (6,56) (5,40) (3,30) (2,20) (1,10) ]
Element not found.
List after sorting the data:
[ (1,10) (2,20) (3,30) (5,40) (6,56) ]
List after reversing the data:
[ (6,56) (5,40) (3,30) (2,20) (1,10) ]
```

Lecture-08

Polynomial List

A polynomial p(x) is the expression in variable x which is in the form ($ax^n + bx^{n-1} + ... + jx + k$), where a, b, c ...., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.

An important characteristics of polynomial is that each term in the polynomial expression consists of two parts:

- one is the coefficient

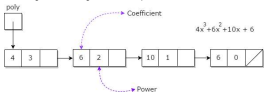
• other is the exponent

Example:

$10x^2 + 26x$, here 10 and 26 are coefficients and 2, 1 are its exponential value.

Points to keep in Mind while working with Polynomials:

- •The sign of each coefficient and exponent is stored within the coefficient and the exponent itself
- Additional terms having equal exponent is possible one
- •The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent



Representation of Polynomial

Polynomial can be represented in the various ways. These are:

- By the use of arrays
- By the use of Linked List

Representation of Polynomials using Arrays

There may arise some situation where you need to evaluate many polynomial expressions and perform basic arithmetic operations like: addition and subtraction with those numbers. For this you will have to get a way to represent those polynomials. The simple way is to represent a polynomial with degree 'n' and store the coefficient of n+1 terms of the polynomial in array. So every array element will consists of two values:

- Coefficient and
- Exponent

Representation of Polynomial Using Linked Lists

A polynomial can be thought of as an ordered list of non zero terms. Each non zero term is a two tuple which holds two pieces of information:

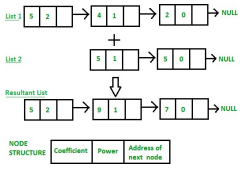
- The exponent part
- The coefficient part

Adding two polynomials using Linked List

Given two polynomial numbers represented by a linked list. Write a function that add these lists means add the coefficients who have same variable powers.

Example:

```
Input:
   1st number = 5x^2 + 4x^1 + 2x^0
   2nd number = 5x^1 + 5x^0
Output:
   5x^2 + 9x^1 + 7x^0
Input:
   1st number = 5x^3 + 4x^2 + 2x^0
   2nd number = 5x^1 + 5x^0
Output:
   5x^3 + 4x^2 + 5x^1 + 7x^0
```




```
struct Node
{
   int coeff;
   int pow;
   struct Node *next;
};
void create_node(int x, int y, struct Node **temp)
{
   struct Node *r, *z;
   z = *temp;
   if(z == NULL)
   {
      r =(struct Node*)malloc(sizeof(struct Node));
      r->coeff = x;
      r->pow = y;
      *temp = r;
      r->next = (struct Node*)malloc(sizeof(struct Node));
      r = r->next;
      r->next = NULL;
   }
   else
   {
      r->coeff = x;
      r->pow = y;
      r->next = (struct Node*)malloc(sizeof(struct Node));
      r = r->next;
      r->next = NULL;
   }
}
void polyadd(struct Node *poly1, struct Node *poly2, struct Node *poly)
{
while(poly1->next && poly2->next)
   {
      if(poly1->pow > poly2->pow)
      {
         poly->pow = poly1->pow;
         poly->coeff = poly1->coeff;
         poly1 = poly1->next;
      }
      else if(poly1->pow < poly2->pow)
      {
         poly->pow = poly2->pow;
         poly->coeff = poly2->coeff;
         poly2 = poly2->next;
      }
      else
      {
         poly->pow = poly1->pow;
         poly->coeff = poly1->coeff+poly2->coeff;
```

```
        poly1 = poly1->next;
        poly2 = poly2->next;
      }
      poly->next = (struct Node *)malloc(sizeof(struct Node));
      poly = poly->next;
      poly->next = NULL;
    }
  while(poly1->next || poly2->next)
    {
      if(poly1->next)
      {
        poly->pow = poly1->pow;
        poly->coeff = poly1->coeff;
        poly1 = poly1->next;
      }
      if(poly2->next)
      {
        poly->pow = poly2->pow;
        poly->coeff = poly2->coeff;
        poly2 = poly2->next;
      }
      poly->next = (struct Node *)malloc(sizeof(struct Node));
      poly = poly->next;
      poly->next = NULL;
    }
```

```
}
void show(struct Node *node)
{
while(node->next != NULL)
  {
  printf("%dx^%d", node->coeff, node->pow);
  node = node->next;
  if(node->next != NULL)
    printf(" + ");
  }
}
int main()
{
  struct Node *poly1 = NULL, *poly2 = NULL, *poly = NULL;
  // Create first list of 5x^2 + 4x^1 + 2x^0
  create_node(5,2,&poly1);
  create_node(4,1,&poly1);
  create_node(2,0,&poly1);
  // Create second list of 5x^1 + 5x^0
  create_node(5,1,&poly2);
  create_node(5,0,&poly2);
  printf("1st Number: ");
  show(poly1);
  printf("\n2nd Number: ");
  show(poly2);
```

```
  poly = (struct Node *)malloc(sizeof(struct Node));
  // Function add two polynomial numbers
  polyadd(poly1, poly2, poly);
  // Display resultant List
  printf("\nAdded polynomial: ");
  show(poly);
return 0;
}
```

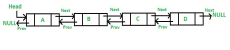
Output:

```
1st Number: 5x^2 + 4x^1 + 2x^0
2nd Number: 5x^1 + 5x^0
Added polynomial: 5x^2 + 9x^1 + 7x^0
```

Lecture-09

Doubly Linked List

A Doubly Linked List (DLL) contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list.



Following is representation of a DLL node in C language.

```
/* Node of a doubly linked list */
struct Node {
    int data;
    struct Node* next; // Pointer to next node in DLL
    struct Node* prev; // Pointer to previous node in DLL
};
```

Following are advantages/disadvantages of doubly linked list over singly linked list.

Advantages over singly linked list

1) A DLL can be traversed in both forward and backward direction.
2)The delete operation in DLL is more efficient if pointer to the node to be deleted is given.
3) We can quickly insert a new node before a given node.
In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.

Disadvantages over singly linked list

1)Every node of DLL Require extra space for an previous pointer. It is possible to implement DLL with single pointer though
2) All operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with next pointers. For example in following functions for insertions at different positions, we need 1 or 2 extra steps to set previous pointer.

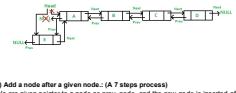
Insertion

A node can be added in four ways
1) At the front of the DLL
2) After a given node.
3) At the end of the DLL
4) Before a given node.

1) Add a node at the front: (A 5 steps process)

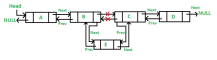
The new node is always added before the head of the given Linked List. And newly added node becomes the new head of DLL. For example if the given Linked List is

10152025 and we add an item 5 at the front, then the Linked List becomes 510152025. Let us call the function that adds at the front of the list is push(). The push() must receive a pointer to the head pointer, because push must change the head pointer to point to the new node



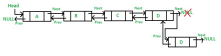
2) Add a node after a given node.: (A 7 steps process)

We are given pointer to a node as prev_node, and the new node is inserted after the given node.



3) Add a node at the end: (7 steps process)

The new node is always added after the last node of the given Linked List. For example if the given DLL is 510152025 and we add an item 30 at the end, then the DLL becomes 5101520253O. Since a Linked List is typically represented by the head of it, we have to traverse the list till end and then change the next of last node to new node.

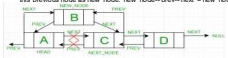


4) Add a node before a given node:

Steps

Let the pointer to this given node be next_node and the data of the new node to be added as new_data.

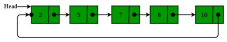
1. Check if the next_node is NULL or not. If it's NULL, return from the function because any new node can not be added before a NULL
2. Allocate memory for the new node, let it be called new_node
3. Set new_node->data = new_data
4. Set the previous pointer of this new_node as the previous node of the next_node, new_node->prev = next_node->prev
5. Set the previous pointer of the next_node as the new_node, next_node->prev = new_node
6. Set the next pointer of this new_node as the next_node, new_node->next = next_node;
7. If the previous node of the new_node is not NULL, then set the next pointer of this previous node as new_node. new_node->prev->next = new_node



Lecture-10

Circular Linked List

Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.



Advantages of Circular Linked Lists:

1)Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.

2)Useful for implementation of queue. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.

3)Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.

4)Circular Doubly Linked Lists are used for implementation of advanced data structures like Fibonacci Heap.

Insertion in an empty List

Initially when the list is empty, *last* pointer will be NULL.

After inserting a node T,

After insertion, T is the last node so pointer *last* points to node T. And Node T is first and last node, so T is pointing to itself.

Function to insert node in an empty List,

```
struct Node *addToEmpty(struct Node *last, int data)
{
   // This function is only for empty list
   if (last != NULL)
      return last;
```

```
   // Creating a node dynamically.
   struct Node *last =
         (struct Node*)malloc(sizeof(struct Node));

   // Assigning the data.
   last -> data = data;

   // Note ; list was empty. We link single node
   // to itself.
   last -> next = last;

   return last;
}
```

Run on IDE

Insertion at the beginning of the list

To insert a node at the beginning of the list, follow these step:
1. Create a node, say T.
2. Make T -> next = last -> next.
3. last -> next = T.

After insertion,

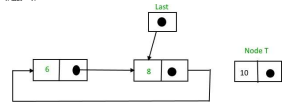
Function to insert node in the beginning of the List,

```
struct Node *addBegin(struct Node *last, int data)
{
 if (last == NULL)
    return addToEmpty(last, data);

 // Creating a node dynamically.
 struct Node *temp
      = (struct Node *)malloc(sizeof(struct Node));

 // Assigning the data.
 temp -> data = data;

 // Adjusting the links.
 temp -> next = last -> next;
 last -> next = temp;

 return last;
```

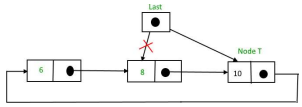
```
}
```

Insertion at the end of the list

To Insert a node at the end of the list, follow these step:
1. Create a node, say T.
2. Make T -> next = last -> next;
3. last -> next = T.
4. last = T.



After insertion,



Function to insert node in the end of the List,

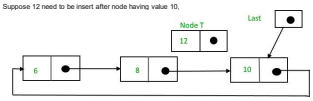
```
struct Node *addEnd(struct Node *last, int data)
{
 if (last == NULL)
    return addToEmpty(last, data);

 // Creating a node dynamically.
```

```
   struct Node *temp =
         (struct Node *)malloc(sizeof(struct Node));

   // Assigning the data.
   temp -> data = data;

   // Adjusting the links.
   temp -> next = last -> next;
   last -> next = temp;
   last = temp;

   return last;
}
```

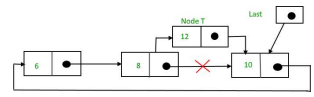
Insertion in between the nodes

To insert a node at the end of the list, follow these step:
1. Create a node, say T.
2. Search the node after which T need to be insert, say that node be P.
3. Make T -> next = P -> next;
4. P -> next = T.

Suppose 12 need to be insert after node having value 10,



After searching and insertion,



Function to insert node in the end of the List,

```
struct Node *addAfter(struct Node *last, int data, int item)
{
  if (last == NULL)
     return NULL;
  struct Node *temp, *p;
  p = last -> next;
  // Searching the item.
  do
  {
     if (p ->data == item)
     {
        temp = (struct Node *)malloc(sizeof(struct Node));
        // Assigning the data.
        temp -> data = data;
        // Adjusting the links.
        temp -> next = p -> next;
        // Adding newly allocated node after p.
        p -> next = temp;
        // Checking for the last node.
        if (p == last)
           last = temp;
        return last;
     }
     p = p -> next;
  } while (p != last -> next);

  cout << item << " not present in the list." <<endl;
  return last;
}
```

Module-2:
Lecture-11

Memory Allocation-

Whenever a new node is created, memory is allocated by the system. This memory is taken from list of those memory locations which are free i.e. not allocated. This list is called AVAIL List. Similarly, whenever a node is deleted, the deleted space becomes reusable and is added to the list of unused space i.e. to AVAIL List. This unused space can be used in future for memory allocation.

Memory allocation is of two types-

1. Static Memory Allocation
2. Dynamic Memory Allocation

1. **Static Memory Allocation:**
   When memory is allocated during compilation time, it is called 'Static Memory Allocation'. This memory is fixed and cannot be increased or decreased after allocation. If more memory is allocated than requirement, then memory is wasted. If less memory is allocated than requirement, then program will not run successfully. So exact memory requirements must be known in advance.
2. **Dynamic Memory Allocation:**
   When memory is allocated during run/execution time, it is called 'Dynamic Memory Allocation'. This memory is not fixed and is allocated according to our requirements. Thus in it there is no wastage of memory. So there is no need to know exact memory requirements in advance.

Garbage Collection-

Whenever a node is deleted, some memory space becomes reusable. This memory space should be available for future use. One way to do this is to immediately insert the free space into availability list. But this method may be time consuming for the operating system. So another method is used which is called 'Garbage Collection'. This method is described below: In this method the OS collects the deleted space time to time onto the availability list. This process happens in two steps. In first step, the OS goes through all the lists and tags all those cells which are currently being used. In the second step, the

OS goes through all the lists again and collects untagged space and adds this collected space to availability list. The garbage collection may occur when small amount of free space is left in the system or no free space is left in the system or when CPU is idle and has time to do the garbage collection.

Compaction

One preferable solution to garbage collection is compaction. The process of moving all marked nodes to one end of memory and all available memory to other end is called compaction. Algorithm which performs compaction is called compacting algorithm.

Lecture-12

Infix to Postfix Conversion

```
1   #include<stdio.h>
2   char stack[20];
3   int top = -1;
4   void push(char x)
5   {
6     stack[++top] = x;
7   }
8
9   char pop()
10  {
11    if(top == -1)
12      return -1;
13    else
14      return stack[top--];
15  }
16
17  int priority(char x)
18  {
19    if(x == '(')
20      return 0;
21    if(x == '+' || x == '-')
22      return 1;
23    if(x == '*' || x == '/')
24      return 2;
25  }
26
27  main()
28  {
29    char exp[20];
30    char *e, x;
31    printf("Enter the expression :: ");
32    scanf("%s",exp);
33    e = exp;
34    while(*e != '\0')
35    {
36      if(isalnum(*e))
37        printf("%c",*e);
38      else if(*e == '(')
39        push(*e);
40      else if(*e == ')')
41      {
```

```
42        while((x = pop()) != '(')
43          printf("%c", x);
44      }
45      else
46      {
47        while(priority(stack[top]) >= priority(*e))
48          printf("%c",pop());
49        push(*e);
50      }
51      e++;
52    }
53    while(top != -1)
54    {
55      printf("%c",pop());
56    }
57  }
```

OUTPUT:

```
Enter the expression :: a+b*c
abc*+

Enter the expression :: (a+b)*c+(d-a)
ab+c*da-+
```

Evaluate POSTFIX Expression Using Stack

```
1   #include<stdio.h>
2   int stack[20];
3   int top = -1;
4   void push(int x)
5   {
6     stack[++top] = x;
7   }
8
9   int pop()
10  {
11    return stack[top--];
12  }
13
14  int main()
15  {
16    char exp[20];
17    char *e;
18    int n1,n2,n3,num;
19    printf("Enter the expression :: ");
20    scanf("%s",exp);
21    e = exp;
22    while(*e != '\0')
23    {
24      if(isdigit(*e))
```

```
25      {
26        num = *e - 48;
27        push(num);
28      }
29      else
30      {
31        n1 = pop();
32        n2 = pop();
33        switch(*e)
34        {
35          case '+':
36          {
37            n3 = n1 + n2;
38            break;
39          }
40          case '-':
41          {
42            n3 = n2 - n1;
43            break;
44          }
45          case '*':
46          {
47            n3 = n1 * n2;
48            break;
49          }
```

```
50          case '/':
51          {
52            n3 = n2 / n1;
53            break;
54          }
55        }
56        push(n3);
57      }
58      e++;
59    }
60    printf("\nThe result of expression %s = %d\n\n",exp,pop());
61    return 0;
62
63  }
64
```

Output:

```
Enter the expression :: 245+*

The result of expression 245+* =  18
```

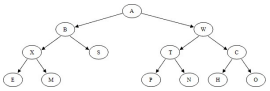
Lecture-13

Binary Tree

A *binary tree* consists of a finite set of nodes that is either empty, or consists of one specially designated node called the *root* of the binary tree, and the elements of two disjoint binary trees called the *left subtree* and *right subtree* of the root.

Note that the definition above is recursive: we have defined a binary tree in terms of binary trees. This is appropriate since recursion is an innate characteristic of tree structures.

Diagram 1: A binary tree



Binary Tree Terminology

Tree terminology is generally derived from the terminology of family trees (specifically, the type of family tree called a *lineal chart*).

- Each root is said to be the *parent* of the roots of its subtrees.
- Two nodes with the same parent are said to be *siblings*; they are the *children* of their parent.
- The root node has no parent.
- A great deal of tree processing takes advantage of the relationship between a parent and its children, and we commonly say a *directed edge* (or simply an *edge*) extends from a parent to its children. Thus edges connect a root with the roots of each subtree. An *undirected edge* extends in both directions between a parent and a child.

- *Grandparent* and *grandchild* relations can be defined in a similar manner; we could also extend this terminology further if we wished (designating nodes as cousins, as an uncle or aunt, etc.).

Other Tree Terms

- The number of subtrees of a node is called the *degree* of the node. In a binary tree, all nodes have degree 0, 1, or 2.
- A node of degree zero is called a *terminal node* or *leaf node*.
- A non-leaf node is often called a *branch node*.
- The *degree of a tree* is the maximum degree of a node in the tree. A binary tree is degree 2.
- A *directed path* from node n_1 to n_k is defined as a sequence of nodes n_1, n_2, ..., n_k such that n is the parent of $n+1$ for $1 <= i < k$. An *undirected path* is a similar sequence of undirected edges. The length of this path is the number of edges on the path, namely $k - 1$ (i.e., the number of nodes – 1). There is a path of length zero from every node to itself. Notice that in a binary tree there is exactly one path from the root to each node.
- The *level* or *depth* of a node with respect to a tree is defined recursively: the level of the root is zero; and the level of any other node is one higher than that of its parent. Or to put it another way, the level or depth of a node n_i is the length of the unique path from the root to n_i.
- The *height* of n is the length of the longest path from n to a leaf. Thus all leaves in the tree are at height 0.
- The *height of a tree* is equal to the height of the root. The *depth of a tree* is equal to the level or depth of the deepest leaf; this is always equal to the height of the tree.
- If there is a directed path from n_1 to n_2, then n_1 is an ancestor of n_2 and n_2 is a descendant of n_1.

Lecture-14

Special Forms of Binary Trees

There are a few special forms of binary tree worth mentioning.

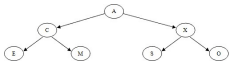
If every non-leaf node in a binary tree has nonempty left and right subtrees, the tree is termed a *strictly binary tree*. Or, to put it another way, all of the nodes in a strictly binary tree are of degree zero or two, never degree one. A strictly binary tree with N leaves always contains $2N - 1$ nodes.

Some texts call this a "full" binary tree.

A *complete binary tree* of depth d is the strictly binary tree all of whose leaves are at level d.

The total number of nodes in a complete binary tree of depth d equals $2^{d+1} - 1$. Since all leaves in such a tree are at level d, the tree contains 2^d leaves and, therefore, $2^d - 1$ internal nodes.

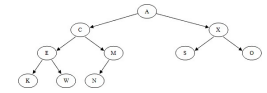
Diagram 2: A complete binary tree



A binary tree of depth d is an *almost complete binary tree* if:

- Each leaf in the tree is either at level d or at level $d - 1$.
- For any node n_R in the tree with a right descendant at level d, all the left descendants of n_R that are leaves are also at level d.

Diagram 3: An almost complete binary tree



An almost complete strictly binary tree with N leaves has $2N - 1$ nodes (as does any other strictly binary tree). An almost complete binary tree with N leaves that is not strictly binary has $2N$ nodes. There are two distinct almost complete binary trees with N leaves, one of which is strictly binary and one of which is not.

There is only a single almost complete binary tree with N nodes. This tree is strictly binary if and only if N is odd.

Representing Binary Trees in Memory

Array Representation

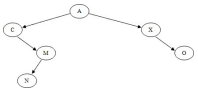
For a complete or almost complete binary tree, storing the binary tree as an array may be a good choice.

One way to do this is to store the root of the tree in the first element of the array. Then, for each node in the tree that is stored at subscript k, the node's left child can be stored at subscript $2k+1$ and the right child can be stored at subscript $2k+2$. For example, the almost complete binary tree shown in **Diagram 2** can be stored in an array like so:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	C	X	E	M	S	O	K	W	N

However, if this scheme is used to store a binary tree that is not complete or almost complete, we can end up with a great deal of wasted space in the array.

For example, the following binary tree

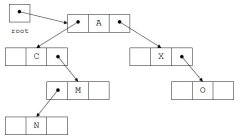


would be stored using this technique like so:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	C	X		M		O			N

Linked Representation

If a binary tree is not complete or almost complete, a better choice for storing it is to use a linked representation similar to the linked list structures covered earlier in the semester:



Each tree node has two pointers (usually named left and right). The tree class has a pointer to the root node of the tree (labeled root in the diagram above).

Any pointer in the tree structure that does not point to a node will normally contain the value NULL. A linked tree with N nodes will always contain $N + 1$ null links.

Lecture-15

Tree Traversal:

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

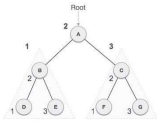
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −

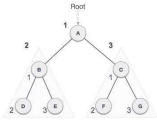
D → B → E → A → F → C → G

Algorithm

Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
Step 3 − Recursively traverse right subtree.

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

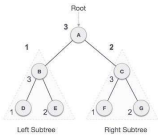
A → B → D → E → C → F → G

Algorithm

Until all nodes are traversed −
Step 1 − Visit root node.
Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree.

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

D → E → B → F → G → C → A

Algorithm

Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.

Lecture-16

AVL Trees

An ***AVL tree*** is another balanced binary search tree. Named after their inventors, **A**delson-**V**elskii and **L**andis, they were the first dynamically balanced trees to be proposed. Like red-black trees, they are not perfectly balanced, but pairs of sub-trees differ in height by at most 1, maintaining an ***O(logn)*** search time. Addition and deletion operations also take ***O(logn)*** time.

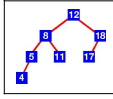
Definition of an AVL tree

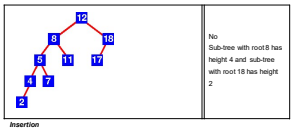
An AVL tree is a binary search tree which has the following properties:

1. The sub-trees of every node differ in height by at most one.
2. Every sub-tree is an AVL tree.

Balance requirement for an AVL tree: the left and right sub-trees differ by at most 1 in height.

You need to be careful with this definition: it permits some apparently unbalanced trees! For example, here are some trees:

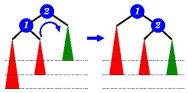
Tree	AVL tree?
	Yes Examination shows that each left sub-tree has a height 1 greater than each right sub-tree.



No
Sub-tree with root 8 has height 4 and sub-tree with root 18 has height 2

Insertion

As with the red-black tree, insertion is somewhat complex and involves a number of cases. Implementations of AVL tree insertion may be found in many textbooks: they rely on adding an extra attribute, the **balance factor** to each node. This factor indicates whether the tree is *left-heavy* (the height of the left sub-tree is 1 greater than the right sub-tree), *balanced* (both sub-trees are the same height) or *right-heavy*(the height of the right sub-tree is 1 greater than the left sub-tree). If the balance would be destroyed by an insertion, a rotation is performed to correct the balance.



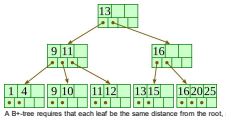
A new item has been added to the left subtree of node 1, causing its height to become 2 greater than 2's right sub-tree (shown in green). A right-rotation is performed to correct the imbalance.

Lecture-17

B+-tree

In B+-tree, each node stores up to *d* references to children and up to *d* − 1 keys. Each reference is considered "between" two of the node's keys; it references the root of a subtree for which all values are between these two keys.

Here is a fairly small tree using 4 as our value for *d*.



A B+-tree requires that each leaf be the same distance from the root, as in this picture, where searching for any of the 11 values (all listed on the bottom level) will involve loading three nodes from the disk (the root block, a second-level block, and a leaf).

In practice, *d* will be larger — as large, in fact, as it takes to fill a disk block. Suppose a block is 4KB, our keys are 4-byte integers, and each reference is a 6-byte file offset. Then we'd choose *d* to be the largest value so that 4 (*d* − 1) + 6 *d* ≤ 4096; solving this inequality for *d*, we end up with *d* ≤ 410, so we'd use 410 for *d*. As you can see, *d* can be large.

A B+-tree maintains the following invariants:

- Every node has one more references than it has keys.
- All leaves are at the same distance from the root.
- For every non-leaf node *N* with *k* being the number of keys in *N*: all keys in the first child's subtree are less than *N*'s first key; and all keys in the *i*th child's subtree (2 ≤ *i* ≤ *k*) are between the (*i* − 1)th key of *n* and the *i*th key of *n*.
- The root has at least two children.
- Every non-leaf, non-root node has at least floor(*d* / 2) children.

- Each leaf contains at least *floor(d / 2)* keys.
- Every key from the table appears in a leaf, in left-to-right sorted order.

In our examples, we'll continue to use 4 for d. Looking at our invariants, this requires that each leaf have at least two keys, and each internal node to have at least two children (and thus at least one key).

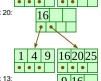
2. Insertion algorithm

Descend to the leaf where the key fits.

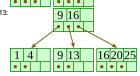
1. If the node has an empty space, insert the key/reference pair into the node.
2. If the node is already full, split it into two nodes, distributing the keys evenly between the two nodes. If the node is a leaf, take a copy of the minimum value in the second of these two nodes and repeat this insertion algorithm to insert it into the parent node. If the node is a non-leaf, exclude the middle value during the split and repeat this insertion algorithm to insert this excluded value into the parent node.

Initial:

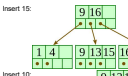
Insert 20:



Insert 13:



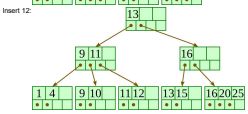
Insert 15:



Insert 10:

Insert 11:

Insert 12:



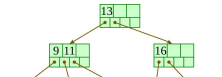
3. Deletion algorithm

Descend to the leaf where the key exists.

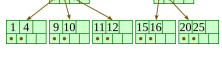
1. Remove the required key and associated reference from the node.
2. If the node still has enough keys and references to satisfy the invariants, stop.

3. If the node has too few keys to satisfy the invariants, but its next oldest or next youngest sibling at the same level has more than necessary, distribute the keys between this node and the neighbor. Repair the keys in the level above to represent that these nodes now have a different "split point" between them; this involves simply changing a key in the levels above, without deletion or insertion.
4. If the node has too few keys to satisfy the invariant, and the next oldest or next youngest sibling is at the minimum for the invariant, then merge the node with its sibling; if the node is a non-leaf, we will need to incorporate the "split key" from the parent into our merging. In either case, we will need to repeat the removal algorithm on the parent node to remove the "split key" that previously separated these merged nodes — unless the parent is the root and we are removing the final key from the root, in which case the merged node becomes the new root (and the tree has become one level shorter than before).

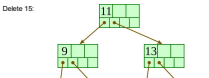
Initial:



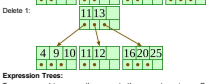
Delete 13:



Delete 15:

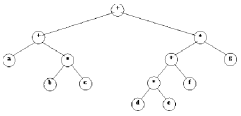


Delete 1:



Expression Trees:

Trees are used in many other ways in the computer science. Compilers and database are two major examples in this regard. In case of compilers, when the languages are translated into machine language, tree-like structures are used. We have also seen an example of expression tree comprising the mathematical expression. Let's have more discussion on the expression trees. We will see what are the benefits of expression trees and how can we build an expression tree. Following is the figure of an expression tree.



In the above tree, the expression on the left side is a + b * c while on the right side, we have d * e + f * g. If you look at the figure, it becomes evident that the inner nodes contain operators while leaf nodes have operands. We know that there are two types of nodes in the tree i.e. inner nodes and leaf nodes. The leaf nodes are such nodes which have left and right subtrees as null. You will find these at the bottom level of the tree. The leaf nodes are connected with the inner nodes. So in trees, we have some inner nodes and some leaf nodes.

In the above diagram, all the inner nodes (the nodes which have either left or right child or both) have operators. In this case, we have + or * as operators. Whereas leaf nodes contain operands only i.e. a, b, c, d, e, f, g. This tree is binary as the operators are binary. We have discussed the evaluation of postfix and infix expressions and have seen that the binary operators need two operands. In the infix expressions, one operand is on the left side of the operator and the other is on the right side. Suppose, if we have

+ operator, it will be written as 2 + 4. However, in case of multiplication, we will write as 5*6. We may have unary operators like negation (-) or in Boolean expression we have NOT. In this example, there are all the binary operators. Therefore, this tree is a binary tree. This is not the Binary Search Tree. In BST, the values on the left side of the nodes are smaller and the values on the right side are greater than the node. Therefore, this is not a BST. Here we have an expression tree with no sorting process involved.

This is not necessary that expression tree is always binary tree. Suppose we have a unary operator like negation. In this case, we have a node which has (-) in it and there is only one leaf node under it. It means just negate that operand.

Let's talk about the traversal of the expression tree. The inorder traversal may be executed here.

Lecture-18

Binary Search Tree (BST)

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −

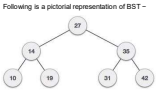
- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than to its parent node's key. Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −

left_subtree (keys) ≤ node (key) ≤ right_subtree (keys)

Representation

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST −



We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

BasicOperations

Following are the basic operations of a tree −

- **Search** − Searches an element in a tree.
- **Insert** − Inserts an element in a tree.
- **Pre-order Traversal** − Traverses a tree in a pre-order manner.
- **In-order Traversal** − Traverses a tree in an in-order manner.
- **Post-order Traversal** − Traverses a tree in a post-order manner.

Node

Define a node having some data, references to its left and right child nodes.

```
struct node {
  int data;
  struct node *leftChild;
  struct node *rightChild;
};
```

SearchOperation

Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm

```
struct node* search(int data){
  struct node *current = root;
  printf("Visiting elements: ");

  while(current->data != data){

    if(current != NULL) {
      printf("%d ",current->data);

      //go to left tree
      if(current->data > data){
        current = current->leftChild;
      } //else go to right tree
      else {
        current = current->rightChild;
      }

      //not found
      if(current == NULL){
        return NULL;
      }
    }
  }

  return current;
}
```

Insert Operation

Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm

```
void insert(int data) {
  struct node *tempNode = (struct node*) malloc(sizeof(struct node));
  struct node *current;
  struct node *parent;

  tempNode->data = data;
  tempNode->leftChild = NULL;
  tempNode->rightChild = NULL;

  //if tree is empty
  if(root == NULL) {
    root = tempNode;
  } else {
    current = root;
    parent = NULL;

    while(1) {
      parent = current;

      //go to left of the tree
      if(data < parent->data) {
        current = current->leftChild;
        //insert to the left

        if(current == NULL) {
          parent->leftChild = tempNode;
          return;
        }
      } //go to right of the tree
      else {
        current = current->rightChild;

        //insert to the right
        if(current == NULL) {
          parent->rightChild = tempNode;
          return;
        }
      }
    }
  }
}
```

Module-3:
Lecture-19

Graphs Terminology

A **graph** consists of:

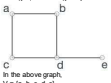
- A set, V, of **vertices** (nodes)
- A collection, E, of pairs of vertices from V called **edges** (arcs)

Edges, also called arcs, are represented by **(*u*, *v*)** and are either:

- **Directed** if the pairs are ordered **(*u*, *v*)**
  - *u* the **origin**
  - *v* the **destination**
- **Undirected** if the pairs are unordered

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E**is the set of edges, connecting the pairs of vertices. Take a look at the following graph –



In the above graph,

V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

Then a **graph** can be:

Directed graph (di-graph) if all the edges are directed

Undirected graph (graph) if all the edges are undirected

Mixed graph if edges are both directed or undirected

Illustrate terms on graphs

End-vertices of an edge are the **endpoints** of the edge.

Two vertices are **adjacent** if they are endpoints of the same edge.

An edge is **incident** on a vertex if the vertex is an endpoint of the edge.

Outgoing edges of a vertex are directed edges that the vertex is the **Incoming** origin. **edges** of a vertex are directed edges that the vertex is the destination. **Degree** of a vertex, *v*, denoted *deg(v)* is the number of incident edges.

Out-degree, *outdeg(v)*, is the number of outgoing edges.

In-degree, *indeg(v)*, is the number of incoming edges.

Parallel edges or multiple edges are edges of the same type and end-vertices

Self-loop is an edge with the end vertices the same vertex

Simple graphs have **no** parallel edges or self-loops

Properties

If graph, G, has *m* edges **then** $\Sigma_{v \in G} deg(v) = 2m$

If a di-graph, G, has *m* edges **then**

$\Sigma_{v \in G} indeg(v) = m = \Sigma_{v \in G} outdeg(v)$

If a **simple** graph, G, has *m* edges and *n* vertices:

If G is also directed **then** $m \leq n(n-1)$

If G is also undirected **then** $m \leq n(n-1)/2$

So a simple graph with *n* vertices has $O(n^2)$ edges at most

More Terminology

Path is a sequence of alternating vetches and edges such that each successive vertex is connected by the edge. Frequently only the vertices are listed especially if there are no parallel edges.

Cycle is a path that starts and end at the same vertex.

Simple path is a path with distinct vertices.

Directed path is a path of only directed edges

Directed cycle is a cycle of only directed edges.

Sub-graph is a subset of vertices and edges.

Spanning sub-graph contains all the vertices.

Connected graph has all pairs of vertices connected by at least one path.

Connected component is the maximal connected sub-graph of a unconnected graph.

Forest is a graph without cycles.

Tree is a connected forest (previous type of trees are called rooted trees, these are free trees)

Spanning tree is a spanning subgraph that is also a tree.

More Properties

If G is an **undirected** graph with *n* vertices and *m* edges:

- **If** G is connected **then** $m \geq n - 1$
- **If** G is a tree **then** $m = n - 1$
- **If** G is a forest **then** $m \leq n - 1$

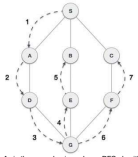
Graph Traversal:

1. Depth First Search
2. Breadth First Search

Lecture-20

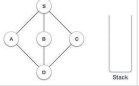
Depth First Search:

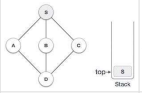
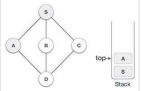
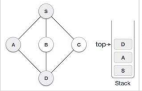
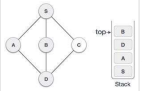
Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

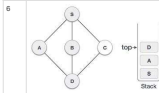
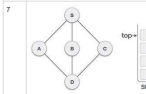


As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1		Initialize the stack.

2		Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3		Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from **A**. Both **S**and **D** are adjacent to **A** but we are concerned for unvisited nodes only.
4		Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.
5		We choose **B**, mark it as visited and put onto the stack. Here **B**does not have any unvisited adjacent node. So, we pop **B**from the stack.

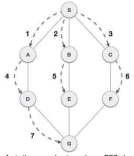
6		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.
7		Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

Lecture-21

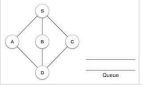
Breadth First Search

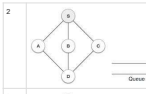
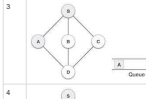
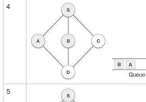
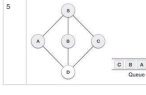
Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

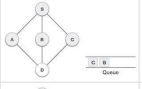
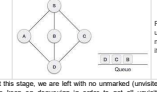


As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1		Initialize the queue.

2		We start from visiting **S**(starting node), and mark it as visited.
3		We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.
4		Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.
5		Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.

6		Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.
7		From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

Lecture-22

Graph representation

You can represent a graph in many ways. The two most common ways of representing a graph is as follows:

Adjacency matrix

An adjacency matrix is a **VxV** binary matrix **A**. Element A[i, j] is 1 if there is an edge from vertex i to vertex j else A[i, j] is 0.

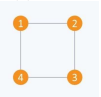
Note: A binary matrix is a matrix in which the cells can have only one of two possible values - either a 0 or 1.

The adjacency matrix can also be modified for the weighted graph in which instead of storing 0 or 1 in A[i, j], the weight or cost of the edge will be stored.

In an undirected graph, if A[i, j] = 1, then A[j, i] = 1. In a directed graph, if A[i, j] = 1, then A[j, i] may or may not be 1.

Adjacency matrix provides **constant time access (O(1))** to determine if there is an edge between two nodes. Space complexity of the adjacency matrix is **O(V2)**.

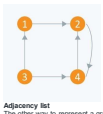
The adjacency matrix of the following graph is:



i/j: 1 2 3 4
1 : 0 1 0 1
2 : 1 0 1 0
3 : 0 1 0 1
4 : 1 0 1 0

The adjacency matrix of the following graph is:

i/j: 1 2 3 4
1 : 0 1 0 0
2 : 0 0 0 1
3 : 1 0 0 1
4 : 0 1 0 0



Adjacency list
The other way to represent a graph is by using an adjacency list. An adjacency list is an array A of separate lists. Each element of the array **Ai** is a list, which contains all the vertices that are adjacent to vertex i.
For a weighted graph, the weight or cost of the edge is stored along with the vertex in the list using pairs. In an undirected graph, if vertex j is in list Ai then vertex i will be in list Aj.
The space complexity of adjacency list is **O(V + E)** because in an adjacency list information is stored only for those edges that actually exist in the graph. In a lot of cases, where a matrix is sparse using an adjacency matrix may not be very useful. This is because using an adjacency matrix will take up a lot of space where most of the elements will be 0, anyway. In such cases, using an adjacency list is better.
Note: A sparse matrix is a matrix in which most of the elements are zero, whereas a dense matrix is a matrix in which most of the elements are non-zero.

Consider the same undirected graph from an adjacency matrix. The adjacency list of the graph is as follows:
A1 → 2 → 4

A2 → 1 → 3

A3 → 2 → 4

A4 → 1 → 3

Consider the same directed graph from an adjacency matrix. The adjacency list of the graph is as follows:
A1 → 2

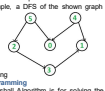
A2 → 4

A3 → 1 → 4

A4 → 2

Lecture-23

Topological Sorting:
Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.
For example, a topological sorting of the following graph is "5 4 2 3 1 0". There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is "4 5 2 3 1 0". The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no in-coming edges).
Algorithm to find Topological Sorting:
In DFS, we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don't print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack.
Topological Sorting vs Depth First Traversal (DFS):
In DFS, we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex '5' should be printed before vertex '0', but unlike DFS, the vertex '4' should also be printed before vertex '0'. So Topological sorting is different from DFS. For example, a DFS of the shown graph is "5 2 3 1 0 4", but it is not a



topological sorting
Dynamic Programming
The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.
Example:

```
Input:
    graph[][] = { {0, 5, INF, 10},
                  {INF, 0, 3, INF},
                  {INF, INF, 0,  1},
                  {INF, INF, INF, 0} }
which represents the following graph
          10
```

```
    (0)------->(3)
     |         /|\
   5 |          |
     |          | 1
     \|/        |
    (1)------->(2)
          3
```

Note that the value of graph[i][j] is 0 if i is equal to j
And graph[i][j] is INF (infinite) if there is no edge from vertex i to j.

Output:
Shortest distance matrix

```
      0      5      8      9
    INF      0      3      4
    INF    INF      0      1
    INF    INF    INF      0
```

Floyd Warshall Algorithm

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of source and destination vertices respectively, there are two possible cases.
1)k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.
2) k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j].
The following figure shows the above optimal substructure property in the all-pairs shortest path problem.

Lecture-24

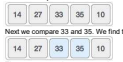
Bubble Sort
We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.

Bubble sort starts with very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.

The new array should look like this –



Next we compare 33 and 35. We find that both are in already sorted positions.

Then we move to the next two values, 35 and 10.

We know then that 10 is smaller 35. Hence they are not sorted.

We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this –

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this –

Notice that after each iteration, at least one value moves at the end.

And when there's no swap required, bubble sorts learns that an array is completely sorted.

Now we should look into some practical aspects of bubble sort.
Algorithm
We assume **list** is an array of n elements. We further assume that **swap** function swaps the values of the given array elements.

```
begin BubbleSort(list)

  for all elements of list
    if list[i] > list[i+1]
      swap(list[i], list[i+1])
    end if
  end for

  return list

end BubbleSort
```

Pseudocode
We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.
To ease-out the issue, we use one flag variable **swapped** which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.
Pseudocode of BubbleSort algorithm can be written as follows –

```
procedure bubbleSort( list : array of items )

  loop = list.count;

  for i = 0 to loop-1 do:
    swapped = false
```

```
   for j = 0 to loop-1 do:

      /* compare the adjacent elements */
      if list[j] > list[j+1] then
         /* swap them */
         swap( list[j], list[j+1] )
         swapped = true
      end if

   end for

   /*if no number was swapped that means
   array is sorted now, break the loop.*/

   if(not swapped) then
      break
   end if

  end for

end procedure return list
```

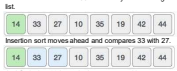
Lecture-25

Insertion Sort

We take an unsorted array for our example.

Insertion sort compares the first two elements.

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



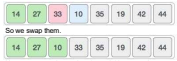
Insertion sort moves ahead and compares 33 with 27.

And finds that 33 is not in the correct position.

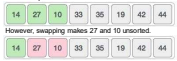
It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

These values are not in a sorted order.



So we swap them.

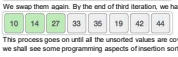


However, swapping makes 27 and 10 unsorted.

Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.



This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

Step 1 − If it is the first element, it is already sorted. return 1;
Step 2 − Pick next element
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted
Step 5 − Insert the value
Step 6 − Repeat until list is sorted

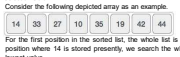
Pseudocode

```
procedure insertionSort( A : array of items )
   int holePosition
   int valueToInsert
   for i = 1 to length(A) inclusive do:
      valueToInsert = A[i]
      holePosition = i

      while holePosition > 0 and A[holePosition-1] > valueToInsert do:
         A[holePosition] = A[holePosition-1]
         holePosition = holePosition -1
      end while
      A[holePosition] = valueToInsert

   end for
         end procedure
```

Lecture-26

Selection Sort

Consider the following depicted array as an example.



For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

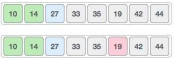
For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

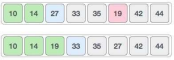
We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

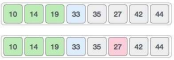
After two iterations, two least values are positioned at the beginning in a sorted manner.

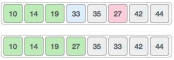
The same process is applied to the rest of the items in the array.

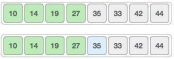
Following is a pictorial depiction of the entire sorting process −

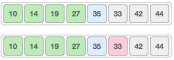


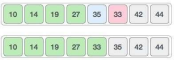


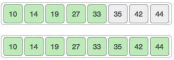












Now, let us learn some programming aspects of selection sort.

Algorithm

Step 1 − Set MIN to location 0
Step 2 − Search the minimum element in the list
Step 3 − Swap with value at location MIN
Step 4 − Increment MIN to point to next element
Step 5 − Repeat until list is sorted

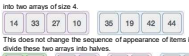
Pseudocode

```
procedure selection sort
   list : array of items
   n    : size of list

   for i = 1 to n - 1
   /* set current element as minimum*/
      min = i

      /* check the element to be minimum */

      for j = i+1 to n
         if list[j] < list[min] then
            min = j
         end if
      end for

      /* swap the minimum element with the current element*/
      if indexMin != i then
         swap list[min] and list[i]
      end if
   end for

end procedure
```

Lecture-27

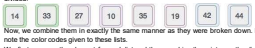
Merge Sort
To understand merge sort, we take an unsorted array as the following −

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.



This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

We further divide these arrays and we achieve atomic value which can no more be divided.



Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.
We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.

In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

After the final merging, the list should look like this −

Now we should learn some programming aspects of merge sorting.

Algorithm

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

Step 1 − if it is only one element in the list it is already sorted, return.
Step 2 − divide the list recursively into two halves until it can no more be divided.
Step 3 − merge the smaller lists into new list in sorted order.

Merge sort works with recursion and we shall see our implementation in the same way.

```
procedure mergesort( var a as array )
   if ( n == 1 ) return a
   var l1 as array = a[0] ... a[n/2]
   var l2 as array = a[n/2+1] ... a[n]
   l1 = mergesort( l1 )
   l2 = mergesort( l2 )
   return merge( l1, l2 )
end procedure
procedure merge( var a as array, var b as array )
   var c as array
   while ( a and b have elements )
      if ( a[0] > b[0] )
         add b[0] to the end of c
         remove b[0] from b
      else
         add a[0] to the end of c
         remove a[0] from a
      end if
   end while

   while ( a has elements )
      add a[0] to the end of c
      remove a[0] from a
   end while

   while ( b has elements )
      add b[0] to the end of c
      remove b[0] from b
   end while
   return c
end procedure
```

Lecture-28

Quick sort
Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.
Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst case complexity are of $O(n^2)$, where n is the number of items.
PartitionInQuick Sort
Following animated representation explains how to find the pivot value in an array.

Unsorted Array

The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.
Quick SortPivotAlgorithm
Based on our understanding of partitioning in quick sort, we will now try to write an algorithm for it, which is as follows.

Step 1 − Choose the highest index value has pivot
Step 2 − Take two variables to point left and right of the list excluding pivot
Step 3 − left points to the low index
Step 4 − right points to the high
Step 5 − while value at left is less than pivot move right
Step 6 − while value at right is greater than pivot move left
Step 7 − if both step 5 and step 6 does not match swap left and right
Step 8 − if left ≥ right, the point where they met is new pivot

Quick SortPivotPseudocode
The pseudocode for the above algorithm can be derived as −

```
function partitionFunc(left, right, pivot)
   leftPointer = left
   rightPointer = right - 1

   while True do
      while A[++leftPointer] < pivot do
         //do-nothing
      end while

      while rightPointer > 0 && A[--rightPointer] > pivot do
         //do-nothing
      end while

      if leftPointer >= rightPointer
         break
      else
         swap leftPointer,rightPointer
      end if

   end while

   swap leftPointer,right
   return leftPointer

end function
```

Quick SortAlgorithm
Using pivot algorithm recursively, we end up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quicksort as follows −

Step 1 − Make the right-most index value pivot
Step 2 − partition the array using pivot value
Step 3 − quicksort left partition recursively
Step 4 − quicksort right partition recursively

Quick SortPseudocode
To get more into it, let see the pseudocode for quick sort algorithm −

```
procedure quickSort(left, right)

   if right-left <= 0
      return
   else
      pivot = A[right]
      partition = partitionFunc(left, right, pivot)
      quickSort(left,partition-1)
      quickSort(partition+1,right)
   end if
   end procedure
```

Lecture-29

Heap Sort
Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.
What is Binary Heap?

Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible
A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.
Why array based representation for Binary Heap?

Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and right child by 2 * I + 2 (assuming the indexing starts at 0).
Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than 1.
How to build the heap?

Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.
Lets understand with the help of an example:

```
Input data: 4, 10, 3, 5, 1
         4(0)
        /   \
     10(1)  3(2)
     /   \
   5(3)   1(4)

The numbers in bracket represent the indices in the array
representation of data.
```

```
Applying heapify procedure to index 1:
         4(0)
        /   \
     10(1)   3(2)
     /   \
   5(3)   1(4)

Applying heapify procedure to index 0:
         10(0)
        /   \
     5(1)  3(2)
     /   \
   4(3)   1(4)
The heapify procedure calls itself recursively to build heap
in top down manner.
```

Radix Sort
The lower bound for Comparison based sorting algorithm (Merge Sort, Heap Sort, Quick-Sort .. etc) is Ω(nLogn), i.e., they cannot do better than nLogn.
Counting sort is a linear time sorting algorithm that sort in O(n+k) time when elements are in range from 1 to k.
What if the elements are in range from 1 to n^2?

We can't use counting sort because counting sort will take $O(n^2)$ which is worse than comparison based sorting algorithms. Can we sort such an array in linear time? Radix Sort is the answer. The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. Radix sort uses counting sort as a subroutine to sort.

Lecture-30

Radix Sort

1) Do following for each digit i where i varies from least significant digit to the most significant digit.
.....**a)** Sort input array using counting sort (or any stable sort) according to the i'th digit.
Example:
Original, unsorted list:
170, 45, 75, 90, 802, 24, 2, 66
Sorting by least significant digit (1's place) gives: ["Notice that we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs 170 & 90 and 45 & 75.]
170, 90, 802, 2, 24, 45, 75, 66
Sorting by next digit (10s place) gives: ["Notice that 802 again comes before 2 as 802 comes before 2 in the previous list.]
802, 2, 24, 45, 66, 170, 75, 90
Sorting by most significant digit (100s place) gives:
2, 24, 45, 66, 75, 90, 170, 802
What is the running time of Radix Sort?
Let there be d digits in input integers. Radix Sort takes O(d*(n+b)) time where b is the base for representing numbers, for example, for decimal system, b is 10. What is the value of d? If k is the maximum possible value, then d would be $O(\log_b(k))$. So overall time complexity is $O((n+b) * \log_b(k))$. Which looks more than the time complexity of comparison based sorting algorithms for a large k. Let us first limit k. Let k <= n^c where c is a constant. In that case, the complexity becomes $O(n\log_b(n))$. But it still doesn't beat comparison based sorting algorithms.

Linear Search

Linear search is to check each element one by one in sequence. The following method linearSearch() searches a target in an array and returns the index of the target; if not found, it returns -1, which indicates an invalid index.

```
1   int linearSearch(int arr[], int target)
2   {
3       for (int i = 0; i < arr.length; i++)
4       {
5           if (arr[i] == target)
6               return i;
7       }
8       return -1;
9   }
```

Linear search loops through each element in the array; each loop body takes constant time. Therefore, it runs in linear time O(n).

Lecture-31

Binary Search

For sorted arrays, *binary search* is more efficient than linear search. The process starts from the middle of the input array:

- If the target equals the element in the middle, return its index.
- If the target is larger than the element in the middle, search the right half.
- If the target is *smaller*, search the left half.

In the following binarySearch() method, the two index variables first and last indicates the searching boundary at each round.

```
1   int binarySearch(int  arr[], int target)
2   {
3     int first = 0, last = arr.length - 1;
4
5       while (first <= last)
6       {
7         int mid = (first + last) / 2;
8           if (target == arr[mid])
9               return mid;
10          if (target > arr[mid])
11              first = mid + 1;
12          else
13              last = mid - 1;
14      }
15      return -1;
16  }
```

```
1   arr: {3, 9, 10, 27, 38, 43, 82}
2
3   target: 10
4   first: 0, last: 6, mid: 3, arr[mid]: 27   -- go left
5   first: 0, last: 2, mid: 1, arr[mid]: 9    -- go right
6   first: 2, last: 2, mid: 2, arr[mid]: 10 -- found
7
8   target: 40
9   first: 0, last: 6, mid: 3, arr[mid]: 27   -- go right
10  first: 4, last: 6, mid: 5, arr[mid]: 43   -- go left
11  first: 4, last: 4, mid: 4, arr[mid]: 38   -- go right
12  first: 5, last: 4                         -- not found
```

Binary search divides the array in the middle at each round of the loop. Suppose the array has length n and the loop runs in t rounds, then we have $n * (1/2)^t = 1$ since at each round the array length is divided by 2. Thus $t = \log(n)$. At each round, the loop body takes constant time. Therefore, binary search runs in logarithmic time $O(\log n)$.
The following code implements binary search using recursion. To call the method, we need provide with the boundary indexes, for example,

```
binarySearch(arr, 0, arr.length - 1, target);
```

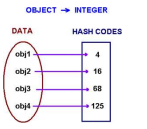
```
1
2     binarySearch(int arr[], int first, int last, int target)
3     {
4       if (first > last)
5           return -1;
6
7       int mid = (first + last) / 2;
8
9       if (target == arr[mid])
10          return mid;
11      if (target > arr[mid])
12          return binarySearch(arr,  mid + 1, last, target);
13      // target < arr[mid]
14      return binarySearch(arr, first, mid - 1, target);
15    }
```

Lecture-32

Hashing

Introduction

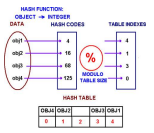
The problem at hands is to speed up searching. Consider the problem of searching an array for a given value. If the array is not sorted, the search might require examining each and all elements of the array. If the array is sorted, we can use the binary search, and therefore reduce the worse-case runtime complexity to O(log n). We could search even faster if we know in advance the index at which that value is located in the array. Suppose we do have that magic function that would tell us the index for a given value. With this magic function our search is reduced to just one probe, giving us a constant runtime O(1). Such a function is called a **hash function** . A hash function is a function which when given a key, generates an address in the table.



The example of a hash function is a *book call number.* Each book in the library has a unique call number. A call number is like an address: it tells us where the book is located in the library. Many academic libraries in the United States, uses Library of Congress Classification for call numbers. This system uses a combination of letters and numbers to arrange materials by subjects.
A hash function that returns a unique hash number is called a **universal hash function**. In practice it is extremely hard to assign unique numbers to objects. The later is always possible only if you know (or approximate) the number of objects to be processed.
Thus, we say that our hash function has the following properties

- it always returns a number for an object.
- two equal objects will always have the same number
- two unequal objects not always have different numbers

The precedure of storing objets using a hash function is the following.
Create an array of size *M*. Choose a *hash* function *h*, that is a mapping from objects into integers *0, 1, ... , M-1*. Put these objects into an array at indexes computed via the hash function *index = h(object)*. Such array is called a **hash table**.



Collisions

When we put objects into a hashtable, it is possible that different objects (by the equals() method) might have the same hashcode. This is called a **collision**. Here is the example of collision. Two different strings **"Aa"** and **"BB"** have the same key: .

```
"Aa"    =    'A'    *    31    +    'a'    =    2112
"BB" = 'B' * 31 + 'B' = 2112
```

How to resolve collisions? Where do we put the second and subsequent values that hash to this same location? There are several approaches in dealing with collisions. One of them is based on idea of putting the keys that collide in a linked list! A hash table then is an array of lists!! This technique is called a *separate chaining* collision resolution.

The big attraction of using a hash table is a constant-time performance for the basic operations add, remove, contains, size. Though, because of collisions, we cannot guarantee the constant runtime in the worst-case. Why? Imagine that all our objects collide into the same index. Then searching for one of them will be equivalent to searching in a list, that takes a liner runtime. However, we can guarantee an expected constant runtime, if we make sure that our lists won't become too long. This is usually implemnted by maintaining a *load factor* that keeps a track of the average length of lists. If a load factor approaches a set in advanced threshold, we create a bigger array and *rehash* all elements from the old table into the new one.
Another technique of collision resolution is a *linear probing*. If we cannoit insert at index k, we try the next slot k+1. If that one is occupied, we go to k+2, and so on.

Lecture-33

Hashing Functions

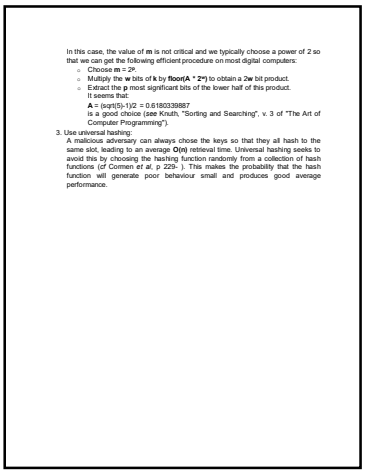
Choosing a good hashing function, **h(k)**, is essential for hash-table based searching. **h** should distribute the elements of our collection as uniformly as possible to the "slots" of the hash table. The key criterion is that there should be a minimum number of collisions.
If the probability that a key, **k**, occurs in our collection is **P(k)**, then if there are **m** slots in our hash table, a *uniform hashing* function, **h(k)**, would ensure:

$$\sum_{k:h(k)=0} P(k) = \sum_{k:h(k)=1} P(k) = \ldots = \sum_{k:h(k)=m-1} P(k) = \frac{1}{m}$$

Sometimes, this is exactly so. For example, if keys are randomly distributed in (0,r], then,
h(k) = floor(m/r)
will provide uniform hashing.
Mapping keys to natural numbers
Most hashing functions will first map the keys to some set of natural numbers, say (0,r]. There are many ways to do this, for example if the key is a string of ASCII characters, we can simply add the ASCII representations of the characters mod 255 to produce a number in (0,255) - or we could **xor** them, or we could add them in pairs mod 2^{16}-1, or ...
Having mapped the keys to a set of natural numbers, we then have a number of possibilities.

1. Use a **mod** function:
   h(k) = k mod m.
   When using this method, we usually avoid certain values of **m**. Powers of 2 are usually avoided, for **k mod 2^p** simply selects the **b** low order bits of **k**. Unless we know that all the 2^p possible values of the lower order bits are equally likely, this will not be a good choice, because some bits of the key are not used in the hash function.
   Prime numbers which are close to powers of 2 seem to be generally good choices for **m**.
   For example, if we have 4000 elements, and we have chosen an overflow table organization, but wish to have the probability of collisions quite low, then we might choose **m** = 4093. (4093 is the largest prime less than $4096 = 2^{12}$.)
2. Use the multiplication method:
   - Multiply the key by a constant **A**, $0 < A < 1$,
   - Extract the fractional part of the product,
   - Multiply this value by **m**.

   Thus the hash function is:
   h(k) = floor(m * (kA - floor(kA)))



In this case, the value of **m** is not critical and we typically choose a power of 2 so that we can get the following efficient procedure on most digital computers:

- Choose **m** = 2^**p**.
- Multiply the **w** bits of **k** by **floor(A * 2^w)** to obtain a 2**w** bit product.
- Extract the **p** most significant bits of the lower half of this product.

It seems that:

A = (sqrt(5)-1)/2 = 0.6180339887

Is a good choice (see Knuth, "Sorting and Searching", v. 3 of "The Art of Computer Programming").

3. Use universal hashing:

A malicious adversary can always chose the keys so that they all hash to the same slot, leading to an average **O(n)** retrieval time. Universal hashing seeks to avoid this by choosing the hashing function randomly from a collection of hash functions (*cf* Cormen *et al*, p 229-). This makes the probability that the hash function will generate poor behaviour small and produces good average performance.