

Data Structures and Algorithms

HNDIT3032

Module Code	HNDIT3032	Module Title	Data Structures and Algorithms			
Credits	2	Hours/Week	Lecturers	30		
GPA/NGPA	GPA		Lab/Tutorial	30		
Semester	S3	Core Compulsory	Compulsory			

Assessment methods /tasks	% weightage	Outcomes			
		1	2	3	4
Final Examination	60	x	x	x	x
Practical Assignments	20				x
On-line quizzes including SEQs	20	x	x	x	x

Recommended Additional Resources
 A Common-Sense Guide to Data Structures and Algorithms, Second Edition: Level Up Your Core Programming Skills by Jay Wengrow

The Need for Data Structures

Data structures organize data

⇒ more efficient programs.

More powerful computers ⇒ more complex applications.

More complex applications demand more calculations.

Organizing Data

Any organization for a collection of records can be searched, processed in any order, or modified.

The choice of data structure and algorithm can make the difference between a program running in a few seconds or many days.

Efficiency

A solution is said to be efficient if it solves the problem within its resource constraints.

- Space
- Time
- The cost of a solution is the amount of resources that the solution consumes.

Selecting a Data Structure

Select a data structure as follows:

1. Analyze the problem to determine the resource constraints a solution must meet.
2. Determine the basic operations that must be supported. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

Some Questions to Ask

- Are all data inserted into the data structure at the beginning, or are insertions interspersed with other operations?
- Can data be deleted?
- Are all data processed in some well-defined order, or is random access allowed?

Data Structure Philosophy

Each data structure has costs and benefits.
Rarely is one data structure better than another in all situations.

A data structure requires:

- space for each data item it stores,
- time to perform each basic operation,
- programming effort.

What is a Data Structure ?

- **Definition :**

An organization and representation of data

- representation
 - data can be stored variously according to their type
 - signed, unsigned, etc.
 - example : integer representation in memory
- organization
 - the way of storing data changes according to the organization
 - ordered, inordered, tree
 - example : if you have more than one integer ?

Abstract Data Types

Abstract Data Type (ADT): a definition for a data type solely in terms of a set of values and a set of operations on that data type.

Each ADT operation is defined by its inputs and outputs.

Encapsulation: Hide implementation details.

Data Structure (cont.)

- A data structure is the physical implementation of an ADT.
 - Each operation associated with the ADT is implemented by one or more subroutines in the implementation.
- Data structure usually refers to an organization for data in main memory.
- File structure is an organization for data on peripheral storage, such as a disk drive.

Properties of a Data Structure ?

- Efficient utilization of medium
- Efficient algorithms for
 - creation
 - manipulation (insertion/deletion)
 - data retrieval (Find)
- A well-designed data structure allows using little
 - resources
 - execution time
 - memory space

13

Algorithms and Programs

Algorithm: A finite, clearly specified sequence of instructions to be followed to solve a problem.

An algorithm takes the input to a problem (function) and transforms it to the output.

- A mapping of input to output.

A problem can have many algorithms.

What is An Algorithm ?

Problem : Write a program to calculate

$$\sum_{i=1}^N i^3$$

```
int Sum (int N)
PartialSum ← 0
i ← 1
foreach (i > 0) and (i <= N)
    PartialSum ← PartialSum + (i*i*i)
    increase i with 1
return value of PartialSum
```

```
int Sum (int N)
{
    int PartialSum = 0 ;
    for (int i=1; i<=N; i++)
        PartialSum += i * i * i;
    return PartialSum;
}
```

15

Algorithm Properties

An algorithm possesses the following properties:

- It must be correct.
- It must be composed of a series of concrete steps.
- There can be no ambiguity as to which step will be performed next.
- It must be composed of a finite number of steps.
- It must terminate.

A computer program is an instance, or concrete representation, for an algorithm in some programming language.

Algorithm Efficiency

There are often many approaches (algorithms) to solve a problem. How do we choose between them?

At the heart of computer program design are two (sometimes conflicting) goals.

1. To design an algorithm that is easy to understand, code, debug.
2. To design an algorithm that makes efficient use of the computer's resources.

How to Measure Efficiency?

1. Empirical comparison (run programs)
2. Asymptotic Algorithm Analysis

Critical resources:

Factors affecting running time:

For most algorithms, running time depends on “size” of the input.

Running time is expressed as $T(n)$ for some function T on input size n .

Algorithm Efficiency (cont)

Goal (1) is the concern of Software Engineering.

Goal (2) is the concern of data structures and algorithm analysis.

When goal (2) is important, how do we measure an algorithm's cost?

The Process of Algorithm Development

- Design
 - divide&conquer, greedy, dynamic programming
- Validation
 - check whether it is correct
- Analysis
 - determine the properties of algorithm
- Implementation
- Testing
 - check whether it works for all possible cases

Analysis of Algorithm

- Analysis investigates
 - What are the properties of the algorithm?
 - in terms of time and space
 - How good is the algorithm ?
 - according to the properties
 - How it compares with others?
 - not always exact
 - Is it the best that can be done?
 - difficult !

21

Mathematical Background

If the unit of running time of algorithms A and B is μsec

N	T_A	T_B
10	10^{-2} sec	10^{-4} sec
100	10^{-1} sec	10^{-2} sec
1000	1 sec	1 sec
10000	10 sec	100 sec
100000	100 sec	10000 sec

So which algorithm is faster ?

23

Mathematical Background

- Assume the functions for running times of two algorithms are found !

For input size N

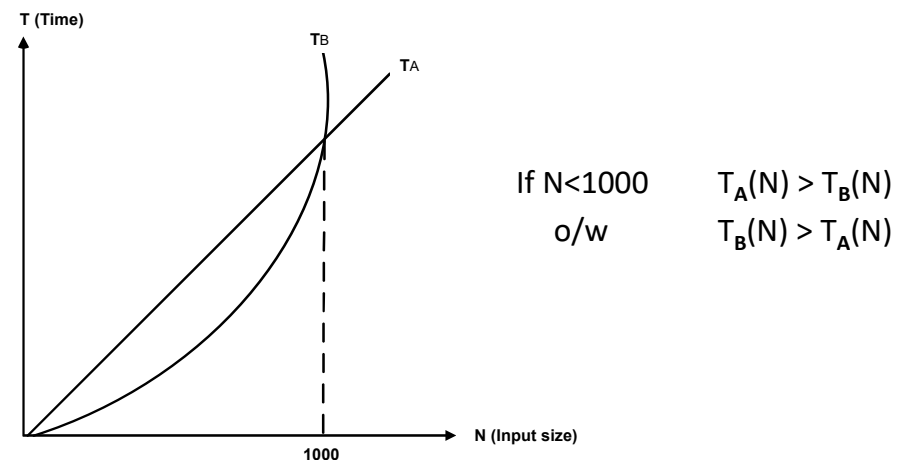
Running time of Algorithm A = $T_A(N) = 1000 N$

Running time of Algorithm B = $T_B(N) = N^2$

Which one is faster ?

22

Mathematical Background



Compare their relative growth ?

24

Mathematical Background

- Is it always possible to have definite results?

NO !

The running times of algorithms can change because of the platform, the properties of the computer, etc.

We use asymptotic notations (O , Ω , θ , o)

- compare relative growth
- compare only algorithms

25

Big Oh Notation (O)

- **Analysis of Algorithm A**

$$T_A(N) = 1000N = O(N)$$

$$1000N \leq cN \quad \forall N \geq n_0$$

if $c = 2000$ and $n_0 = 1$ for all N

$T_A(N) = 1000N = O(N)$ is right

27

Big Oh Notation (O)

Provides an “upper bound” for the function f

- **Definition :**

$T(N) = O(f(N))$ if there are positive constants c and n_0 such that

$$T(N) \leq cf(N) \text{ when } N \geq n_0$$

- $T(N)$ grows no faster than $f(N)$
- growth rate of $T(N)$ is less than or equal to growth rate of $f(N)$ for large N
- $f(N)$ is an upper bound on $T(N)$
 - not fully correct !

26

Examples

- $7n+5 = O(n)$

for $c=8$ and $n_0=5$

$$7n+5 \leq 8n \quad n \geq 5 = n_0$$

- $7n+5 = O(n^2)$

for $c=7$ and $n_0=2$

$$7n+5 \leq 7n^2 \quad n \geq n_0$$

28

Advantages of O Notation

- It is possible to compare of two algorithms with running times
- Constants can be ignored.
 - Units are not important
 $O(7n^2) = O(n^2)$
- Lower order terms are ignored
 - $O(n^3+7n^2+3) = O(n^3)$

Running Times of Algorithm A and B

$$T_A(N) = 1000 N = O(N)$$

$$T_B(N) = N^2 = O(N^2)$$

A is asymptotically faster than B !

29

30

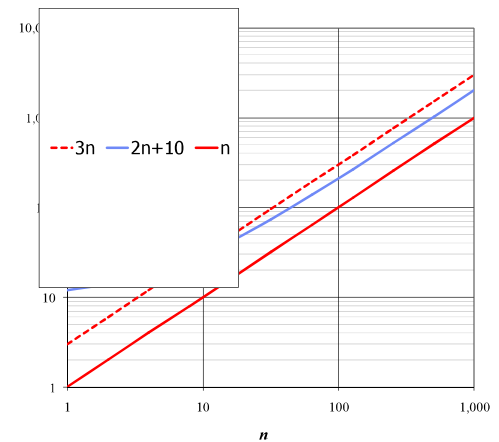
Big-Oh Notation

- To simplify the running time estimation, for a function $f(n)$, we ignore the constants and lower order terms.

Example: $10n^3+4n^2-4n+5$ is $O(n^3)$.

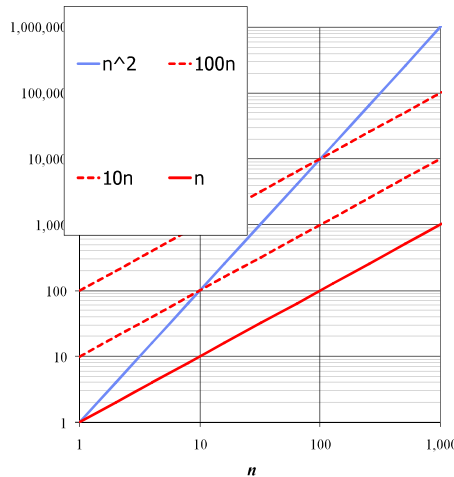
Big-Oh Notation (Formal Definition)

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that $f(n) \leq cg(n)$ for $n \geq n_0$
- Example: $2n + 10$ is $O(n)$
 - $2n + 10 \leq cn$
 - $(c - 2)n \geq 10$
 - $n \geq 10/(c - 2)$
 - Pick $c = 3$ and $n_0 = 10$



Big-Oh Example

- Example: the function n^2 is not $O(n)$
 - $n^2 \leq cn$
 - $n \leq c$
 - The above inequality cannot be satisfied since c must be a constant
 - n^2 is $O(n^2)$.



More Big-Oh Examples



◆ $7n-2$

$7n-2$ is $O(n)$

need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \cdot n$ for $n \geq n_0$

this is true for $c = 7$ and $n_0 = 1$

■ $3n^3 + 20n^2 + 5$

$3n^3 + 20n^2 + 5$ is $O(n^3)$

need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ for $n \geq n_0$

this is true for $c = 4$ and $n_0 = 21$

■ $3 \log n + 5$

$3 \log n + 5$ is $O(\log n)$

need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \cdot \log n$ for $n \geq n_0$

this is true for $c = 8$ and $n_0 = 2$

Big-Oh Rules



- If $f(n)$ is a polynomial of degree d , then $f(n)$ is $O(n^d)$, i.e.,
 1. Drop lower-order terms
 2. Drop constant factors
- Use the smallest possible class of functions
 - Say “ $2n$ is $O(n)$ ” instead of “ $2n$ is $O(n^2)$ ”
- Use the simplest expression of the class
 - Say “ $3n + 5$ is $O(n)$ ” instead of “ $3n + 5$ is $O(3n)$ ”

Growth Rate of Running Time

- Consider a program with time complexity $O(n^2)$.
- For the input of size n , it takes 5 seconds.
- If the input size is doubled ($2n$), then it takes 20 seconds.
- Consider a program with time complexity $O(n)$.
- For the input of size n , it takes 5 seconds.
- If the input size is doubled ($2n$), then it takes 10 seconds.
- Consider a program with time complexity $O(n^3)$.
- For the input of size n , it takes 5 seconds.
- If the input size is doubled ($2n$), then it takes 40 seconds.

Best, Worst, Average Cases

Not all inputs of a given size take the same time to run.

Sequential search for K in an array of n integers:

- Begin at first element in array and look at each element in turn until K is found

Best case: Find at first position.

Cost is 1 compare.

Worst case: Find at last position.

Cost is n compares.

Average case: $(n+1)/2$ compares

IF we assume the element with value K is equally likely to be in any position in the array.

Exercise

1. Find BigOh and Ω of the following functions:

1.1 2^{n+1}

1.2 $n^3 + 3n^2 + 4$

1.3 $(x^2 + y^3)$

1.4 $n^6 + n^5 + n^{\frac{1}{2}} + 1$

HNDIT3032 Data Structures and Algorithms



Week 2



Data Structures and Algorithms HNDIT3032

Lecture 02- Array

Data Abstraction

Data abstraction is the process of defining a collection of data and relative operations, by specifying **what the operations do on the data, not how the data and the operations are implemented.**

Example: use of “dates” in a program

- In **abstract** form we think of dates as “**Day Month Year**”
- We identify a number of operations that make sense when applied to a date
 - the day after a date, the day before a date, equality between two dates,..

How might such dates be implemented?

1. **Julian form** – as the number of days since 1 January 1995
2. **With three data fields** – year, month, day

2 January 1996 → 0366 (Julian form)
 → 96 01 02 (Three data field)

Example of Levels of Data Abstraction

Level of Abstraction	Example
Abstract Data Type	List
User-defined Data Types	Classes
Predefined structured Data Types	Array of Double
Predefined simple Data Types	Double
Machine Language Type	0110111011

Abstract Data Types defines **data abstraction**, which is used to control the interaction between a program and its data structures.

It guards against:

- Inadvertently erroneous use of the data
- Deliberate misuse of the data
- Modification of purpose or implementation of shared data

Definitions

- An **Abstract Data Type** is a collection of data together with a set of data management operations, called **Access Procedures**, defined on these data.

Definition and use of an ADT are **independent** of the **implementation** of the data and its access procedures.

- A **Data Structure**, or structured data type, is an organised collection of data elements, created using
 - Predefined structured data types (e.g., array, vectors)
 - Predefined simple data types (e.g. Boolean, real, integer)
 - User-defined data types (e.g., a “date” class)

Abstract Data Types (ADTs)

- ✦ An abstract data type (ADT) is an abstraction of a data structure
- ✦ An ADT specifies:
 - Data stored
 - Operations on the data
 - Error conditions associated with operations
- ✦ **Example:** ADT modeling a simple stock trading system
 - The data stored are buy/sell orders
 - The operations supported are
 - ✦ order **buy**(stock, shares, price)
 - ✦ order **sell**(stock, shares, price)
 - ✦ void **cancel**(order)
 - Error conditions:
 - ✦ Buy/sell a nonexistent stock
 - ✦ Cancel a nonexistent order

ADTs are **not** implementations

- We can use different implementations for ADTs
- For instance: Stack
 - Last in, first out
 - Basic mechanism for function calls, delimiter checks in compilers, etc.
 - Operations: new, push, pop, peek, empty?

Abstract Data Type (ADT)

- Vector, Matrix
 - Random access to any element by coordinates
- Queue (Buffer)
 - First in, First out
- Set (unordered), Ordered Lists (Dictionary)
- Graphs (of nodes and vertices)
-

Types of Array

Two types:

- **Static Array:** Memory allocated at compile time .It is define as fixed size of array. It cannot be edit or update the size of this array by user.
- **Dynamic Array:** memories are allocated at run time. So that not having a fixed size. Suppose, user wants to declare any random size of an array.

myArray =

3	6	3	1	6	3	4	1
0	1	2	3	4	5	6	7

myArray has room for 8 elements

- The elements are accessed by their index
- In Java, array indices start at 0

Static Arrays

- An array is a list of similar things
- An array has a fixed:
 - name
 - type
 - length
- These must be declared when the array is created.
- Arrays sizes cannot be changed during the execution of the code

Example -Static Array

- `int Number[5] = {10,12,14,18,100}`
- `char ch[10]={‘a’,‘e’,‘i’,‘o’,‘u’,‘A’,‘E’,‘I’,‘O’,‘U’}`
- Size of the array define as numbers values in the Array
- Array size cannot be changed once defined.

Declaring Arrays

One-D Arrays

Declaration

type var-name[];
var_name=new type[size];
or
type var-name[]=new type[size];

Example

int month_date[];
month_date=new int[12];
Or
int month_date[]=new int[12];

Declaring Arrays

```
int myArray[];
```

declares *myArray* to be an array of integers

```
myArray = new int[8];
```

sets up 8 integer-sized spaces in memory, labelled
myArray[0] to *myArray[7]*

```
int myArray[] = new int[8];
```

combines the two statements in one line

Assigning Values

- refer to the array elements by index to store values in them.

```
myArray[0] = 3;  
myArray[1] = 6;  
myArray[2] = 3; ...
```

- can create and initialise in one step:

```
int myArray[] = {3, 6, 3, 1, 6, 3, 4, 1};
```

Iterating Through Arrays

- for* loops are useful when dealing with arrays:

```
for (int i = 0; i < myArray.length;  
    i++)  
{  
    myArray[i] = getsomevalue();  
}
```

Notes on Arrays

- index starts at 0.
- arrays can't shrink or grow.
 - e.g., use Vector instead.(Vector , It will discuss next session more details)
- each element is initialized.
- array bounds checking (no overflow!)
 - ArrayIndexOutOfBoundsException
- Arrays have *array.length*

Array Literals

```
int[] foo = {1,2,3,4,5};
```

```
String[] names = {"Joe", "Sam"};
```

Arrays

Example program

```
public class OneDArray
{
    public static void main (String args[])
    {
        int month_date[];
        month_date=new int[5];
        // or int month_date[]=new int[5];
        month_date[0]=31;
        month_date[1]=28;
        month_date[2]=31;
        month_date[3]=30;
        month_date[4]=31;

        System.out.println("April has "+ month_date[3]+"
Days.");
    }
}
```

Arrays

Example program

```
public class OneDArray
{
    public static void main (String args[])
    {
        int month_date[]={31,28,31,30,31}

        System.out.println("April has "+
            month_date[3]+" Days.");
    }
}
```

Arrays

Two-D Arrays

Declaration

```
type var_name[][];  
var_name=new type[size][size];
```

or

```
type var_name[][]=new type[size][size];
```

Example

```
int month_date [][];  
month_date=new month_date [2][3];
```

E.g.

```
class MultiDimArrayDemo
```

```
{
```

```
public static void main(String[] args)
```

```
{
```

```
String[][] names = {"Mr. ", "Mrs. ", "Ms. "}, {"Smith", "Jones"};
```

```
System.out.println(names[0][0] + names[1][0]); //Mr. Smith
```

```
System.out.println(names[0][2] + names[1][1]); //Ms. Jones
```

```
}
```

```
}
```

Arrays

Example program

```
public class TwoDArray  
{  
    public static void main (String args[])  
    {  
        int TwoD[][]=new int[3][4];  
        int i,j,k=0;  
  
        for(i=0; i<3; i++)  
            for(j=0; j<4; j++)  
            {  
                TwoD[i][j]=k;  
                k=k+1  
            }  
        for(i=0; i<3; i++){  
            for(j=0; j<4; j++){  
                System.out.print(TwoD[i][j]+" ");  
                System.out.println();  
            }  
        }  
    }  
}
```

Arrays

Example program

```
public class TwoDArray  
{  
    public static void main (String args[])  
    {  
        double m[][]={ {0*0,1*0,2*0,3*0},  
                        {0*1,1*1,2*1,3*1}  
                        {0*2,1*2,2*2,3*2}  
                        {0*3,1*3,2*3,3*3}};  
  
        for(int i=0; i<4; i++){  
            for(int j=0; j<4; j++){  
                System.out.print(m[i][j]+ " ");  
                System.out.println();  
            }  
        }  
    }  
}
```

Copying Arrays

The System class has an **arraycopy** method that you can use to efficiently copy data from one array into another:

E.g.

```
class ArrayCopyDemo
{
    public static void main(String[] args)
    {
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e', 'i', 'n', 'a',
                             't', 'e', 'd' };
        char[] copyTo = new char[7];
        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
        System.out.println(new String(copyTo));
    }
}
```

The output from this program is:

caffeine

Array Limitations

- Arrays
 - Simple,
 - Fast
 - but*
 - Must specify size at construction time
 - Murphy's law
 - Construct an array with space for n
 - n = twice your estimate of largest collection
 - Tomorrow you'll need $n+1$
 - More flexible system?

References

- Next Session Dynamic Array (Java Collection Framework ArrayList ,Vector , LinkedList)

- <https://www.scaler.com/topics/dynamic-array-in-java/>
- <https://www.geeksforgeeks.org/arraylist-in-java/>
- https://www.tutorjoes.in/java_programming_tutorial/array_exercise_programs_in_java

Why need Dynamic Array

We often come across to run our program when we don't know the exact input size to be entered. example that-how many student joint my lecture online today, In that case, an error occurs when the input size is recorded by coding and the program will be error.(Array outbound exception)

Therefore, we need a method that sets the size of the computer memory when it is run.

There is a need for an array that can be set according to the size of our input. Therefore, dynamic array can be introduced.

HNDIT3032 Data Structures and Algorithms



Week 3- Dynamic Array



What is Dynamic Array

- A dynamic array size assign at running time
- It can be automatic resizing.
- It array expands as add more elements. So not need to determine the size ahead of time.
- It is a growable array
- It is a resizable array
- If the array is filled to the size we have given, its size will automatically be resizable. Maybe double. If it is an ArrayList, the capacity will increase by $3/2 + 1$ in the array.
- Heterogeneous elements are allowed

Dynamic Array

- resizable array

Array size is 10 –ArrayList default size is 10

$$10 * 3/2 + 1 = 15 + 1 = 16$$

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	11	12	13	14	15	16	17	18	19						

Resize by capacity =16

Advantage of Dynamic Array

- Random Access of Elements
- Good locality of reference and data cache utilization
- Easy to Insert /delete at the time

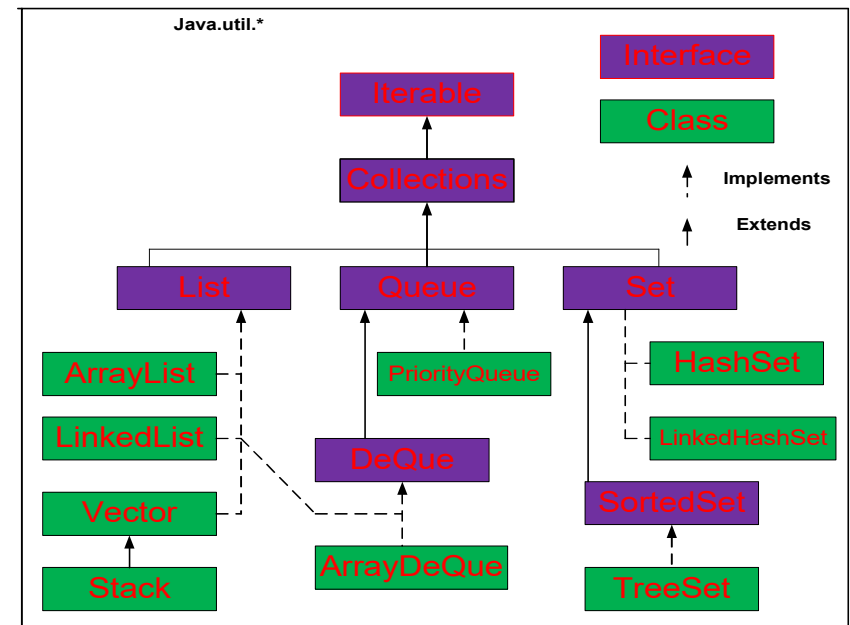
Disadvantage of Dynamic Array

- Waster more memory
- Shifting elements is time consuming
- Expanding /Shrinking the array is time consuming

Collections in Java

- 1.It is a Framework
- 2.it is provide as Architecture to store and Manipulate the group of Object
- 3.it means a single unit of object.
- 4.it framework provide many interface (Set, List, Queue, Deque) and Classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet)

Hierarchy of Collection Framework






What is Java Framework?

- Java Framework is readymade Architecture
- Java Framework represents a set of class and interface .
- Java Framework is optional



What is Collection Framework?

- It is a unified architecture for storing and manipulating a group of objects.
- Included –Interfaces with implementation – class
- Algorithms

- 
- **Interfaces:** Interface in Java refers to the abstract data types. They allow Java collections to be manipulated independently from the details of their representation. Also, they form a hierarchy in object-oriented programming languages.
 - **Classes:** Classes in Java are the implementation of the collection interface. It basically refers to the data structures that are used again and again.
 - **Algorithm:** Algorithm refers to the methods which are used to perform operations such as searching and sorting, on objects that implement collection interfaces. Algorithms are polymorphic in nature as the same method can be used to take many forms or you can say perform different implementations of the Java collection interface.
 - So why do you think we need Java collections? The Java collection framework provides the developers to access prepackaged data structures as well as algorithms to manipulate data.



Why use Java collection?

- Reducing the effort required to write the code by providing useful data structures and algorithms
- Java collections provide high-performance and high-quality data structures and algorithms thereby increasing the speed and quality
- Unrelated APIs can pass collection interfaces back and forth
- Decreases extra effort required to learn, use, and design new API's
- Supports reusability of standard data structures and algorithms

Iterable Interface

- root interface for all the collection classes.
- Collection interface **extends** Iterable interface
- subclasses of Collection interface also implement the Iterable interface.
- It contains only one abstract method.
- `Iterator<T> iterator()` returns the iterator over the elements of type T

List Interface

- The child interface of Collection interface.
- list type data structure we can store the ordered collection of objects.
- List Interface can have duplicate values.
- It is implemented by the classes ArrayList, LinkedList, Vector, and Stack.
- `List <data-type> l1= new ArrayList();`
- `List <data-type> l2 = new LinkedList();`
- `List <data-type> l3 = new Vector();`
- `List <data-type> l4 = new Stack();`

ArrayList

- ArrayList is dynamic array .
- it can be uses store the duplicate element of different data types.
- it is maintains insert order
- it is non-synchronized
- it elements can be randomly accessed
- Will discussed more ArrayList Lecture 4

ArrayList Example program

```
import java.util.*;
class TestCollection1{
    public static void main(String args[]) {
        //Creating arraylist
        ArrayList<String> l=new ArrayList<String>();
        l.add("Saman");//Adding object in arraylist
        l.add("Kamal");
        l.add("Saman");
        l.add("Maharuf");
        //Traversing list through Iterator
        Iterator IT=l.iterator();
        while(IT.hasNext()){ //travers end of the list
            System.out.println(IT.next());
        }
    }
}
```

//hash Next() and next method use to iterator

LinkedList

- LinkedList uses a doubly linkedlist to store element internally.
- LinkedList can be store duplicate elements.
- It also maintance insert order
- it is not synchronized
- LinkedList manipulation is fast becouse that no shifting is required
- (will discusedd next topic about LinkedList more details)

LinkedList Example program

```
import java.util.*;
public class TestCollection2 {
    public static void main(String args[]){
        LinkedList<String> linkl=new LinkedList<String>();
        linkl.add("Saman");//Adding object in linklist
        linkl.add("Kamal");
        linkl.add("Saman");
        linkl.add("Maharuf");
        //Traversing list through Iterator
        Iterator IT=linkl.iterator();
        while(IT.hasNext()){ //travers end of the linklist
            System.out.println(IT.next());
        }
    }
}
```

Vector

- It is dynamic array to used way to data store elements.
- It is same as ArrayList
- Vector is synchronized

Vector Example Program

```
import java.util.*;
public class TestCollection3 {
    public static void main(String args[]){
        Vector<String> ve=new Vector<String>();
        ve.add("Saman");//Adding object in Vector
        ve.add("Kamal");
        ve.add("Saman");
        ve.add("Maharuf");
        //Traversing list through Iterator
        Iterator IT=ve.iterator();
        while(IT.hasNext()){ //travers end of the Vector
            System.out.println(IT.next());
        }
    }
}
```

Exercise

- Discussed Static array vs Dynamic Array

Significant Differences between ArrayList and Vector:

- **Synchronization:** Vector is **synchronized**, which means only one thread at a time can access the code, while ArrayList is **not synchronized**, which means multiple threads can work on ArrayList at the same time. For example, if one thread is performing an add operation, then there can be another thread performing a remove operation in a multithreading environment. If multiple threads access ArrayList concurrently, then we must synchronize the block of the code which modifies the list structurally or allow simple element modifications. Structural modification means the addition or deletion of element(s) from the list. Setting the value of an existing element is not a structural modification.

- **Performance:** ArrayList is faster. Since it is non-synchronized, while vector operations give slower performance since they are synchronized (thread-safe), if one thread works on a vector, it has acquired a lock on it, which forces any other thread wanting to work on it to have to wait until the lock is released.
- **Data Growth:** ArrayList and Vector **both grow and shrink dynamically** to maintain optimal use of storage – but the way they resize is different. ArrayList increments 50% of the current array size if the number of elements exceeds its capacity, while vector increments 100% – essentially doubling the current array size.
- **Traversal:** Vector can use both [Enumeration and Iterator](#) for traversing over vector elements, while ArrayList can only use **Iterator** for traversing.
- **Applications:** Most of the time, programmers prefer ArrayList over Vector because ArrayList can be synchronized explicitly using [Collections.synchronizedList](#).

// runnable interface by extending Thread class

```
public class Thread1 extends Thread {
    public void run() //this method perform actions of the thread
    {
        int x= 101;
        int y=18;
        int result = x+y;
        System.out.println("Thread started running..");
        System.out.println("Sum of two numbers is: "+ result);
    }
    public static void main( String args[] )
    {
        // Creating Object of the class extend Thread class
        Thread1 t1 = new Thread1();
        //calling start method to execute the run() method of the Thread class
        t1.start();
    }
}
```

Exercise

- Describe differences ArrayList and LinkedList
- Describe differences Vector and LinkedList

References

- <https://www.edureka.co/blog/java-collections/>

Basic Operations in the Arrays

- **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.
- **Display** – Displays the contents of the array.

HNDIT3032 Data Structures and Algorithms

Week 4- Array and List



Memory Address	0x0019ff2c	0x0019ff30	0x0019ff34	0x0019ff38	0x0019ff3c
Array Values	12	34	68	77	74
Array Index	0	1	2	3	4

Print Array Address (C++)

```
#include <iostream.h>
#include <conio.h>
void main()
{
    int x[5];
    for(int i=0;i<=4;i++){
        cout<<&x[i]<<endl;
    }
    getch();
}
```

```
0x0019ff2c
0x0019ff30
0x0019ff34
0x0019ff38
0x0019ff3c
```

Traverse

- **Algorithm**
 - 1 Start
 2. Initialize an Array of certain size and datatype.
 3. Initialize another variable 'i' with 0.
 4. Print the ith value in the array and increment i.
 5. Repeat Step 4 until the end of the array is reached.
 6. End

Traverse java implementation

- **public class** Main2 {
- **public static void** main(String []args) {
- **int** X[] = **new int**[5];
- System.**out**.println("The array elements are: ");
- **for**(**int** i = 0; i < 5; i++) {
- X[i] = i*i;
- System.**out**.println("X[" + i + "] = " + X[i]);
- }}

Insertion Operation

Algorithm

- 1. Start
- 2. Create an Array of a desired data type and size.
- 3. Initialize a variable 'i' as 0.
- 4. Enter the element at it n index of the array.
- 5. Increment i by 1.
- 6. Repeat Steps 4 & 5 until the end of the array.
- 7. Stop

Insertion Operation

- **public class** Main2 {
- **public static void** main(String []args) {
- **int** X[] = **new int**[3];
- System.**out**.println("Array Before Insertion:");
- **for**(**int** i = 0; i < 3; i++)
- System.**out**.println("X[" + i + "] = " + X[i]); //prints empty array
- System.**out**.println("Inserting Elements..");
- // Printing Array after Insertion
- System.**out**.println("Array After Insertion:");
- **for**(**int** i = 0; i < 3; i++) {
- X[i] = i+1+i*i;
- System.**out**.println("X[" + i + "] = " + X[i]);
- }
- }

Deletion

Algorithm

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

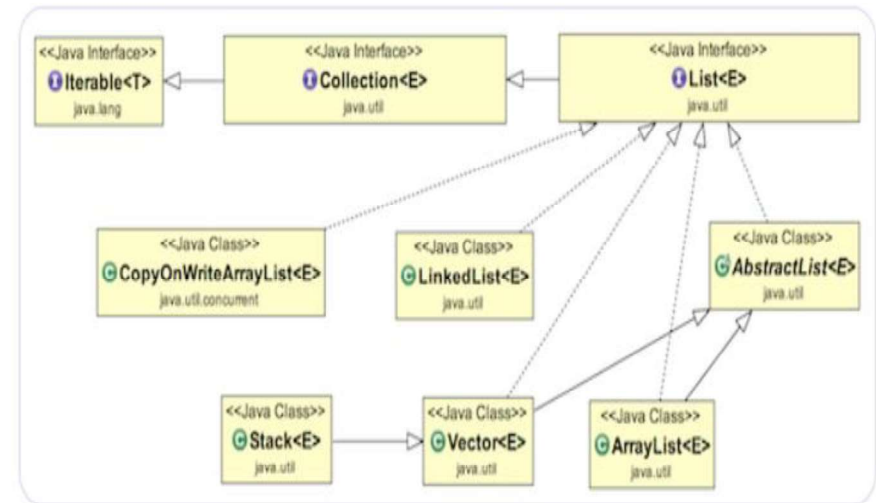
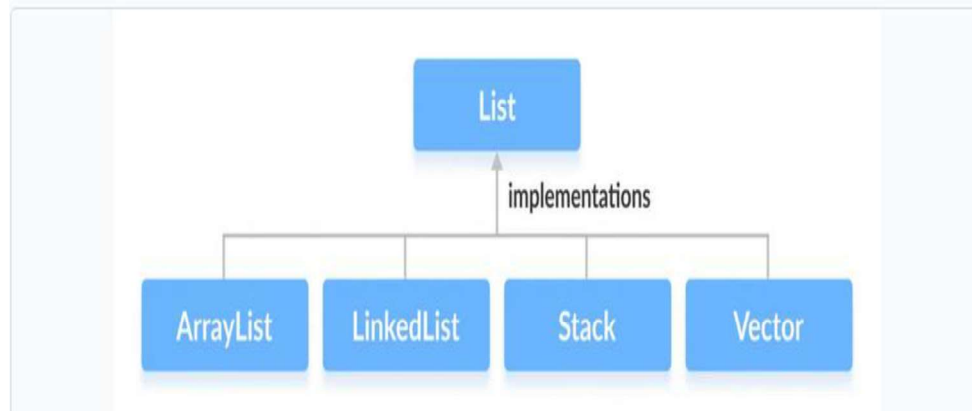
- A class work work for you Do Deletion operation Implementations in an Array.

Full Java Implementation insertion and Deletion

```
import java.util.Scanner;
public class ArrayDelete {
    public int size;
    int i=0;
    public int arr[]=new int[size];
    public ArrayDelete(int size, int[] arr) {
        super();
        this.size = size;
        this.arr = arr;
    }
    public int getSize() {
        return size;
    }
    public int[] getArr() {
        return arr;
    }
    public void addArray(int n)
    {if(i>=getSize()){
        System.out.println("Array is Full"); }
    else{
        arr[i]=n;
        i++;}}
```

```
public void DeleteArrayValue(){
    if(i<=0) {
        System.out.println("Array is Empty...");}
    else{
        i--;
        arr[i]=0;}}
    public void displayArrayValue(){
        if(i<=0){
            System.out.println("Array is Empty...");}
        else{
            for(int j=i-1;j>=0;j--){
                System.out.print(arr[j]+", "); }
            System.out.println();}}
```

- **public static void** main(String ar[]){
- Scanner **sc=new** Scanner(System.**in**);
- System.**out**.println("Enter Array Size");
- **int** ArraySize=sc.nextInt();
- **int** Arr[]=new int[ArraySize];
- ArrayDelete op=**new** ArrayDelete(ArraySize,Arr);
- **boolean** x=**true**;
- **while**(x){
- System.**out**.println("Chose Yor Options \n 1-->Add Value in Array \n 2--> Delete Value in Array \n 3--> Display Value in Array \n 10 --> End Program\n");
- **int** chose=sc.nextInt();
- **switch**(chose)
- {**case** 1: System.**out**.println("Enter Your Value");**int** selection=sc.nextInt();op.addArray(selection);**break**;
- **case** 2:op.DeleteArrayValue();**break**;
- **case** 3:op.displayArrayValue();}
- System.**out**.flush();
- } }



What is List in Java

- List Can contain Duplicate Value
- Data are store in some Order
- List is the Interface
- It is in Java.util package inherited collection interface
- A Java list controls where you can insert an element.
- You can access elements by their index and search for elements in the list.

Java List Methods

- **int size()**
Return to the size of the list or number of elements in the list. it is call as length
- **void clear()**
clear all elements in the list
- **void add(int index, Object element)**
index –which element need to add in the list
element-insert element in the list
Adds the given element to the list at the given index. The subsequent elements are shifted to the right.
- **boolean add (Object o)**
Element to be added to the list
Return Value: true=> Element successfully added
False- Add not successful
This method adds the given element at the end of the list.

Size and Clear method example code

```
import java.util.*;
public class sizeAndClear {
    public static void main(String[] args) {
        List<String> strList = new ArrayList<String>(); // Creating a list
        //add items in the list
        strList.add("Pascal");
        strList.add("C++");
        //Display size of list
        System.out.println("Size of list:" + strList.size());
        //add more items to list
        strList.add("Java");
        strList.add("Python");
        strList.add("PHP");
        //Display size of list again
        System.out.println("Size of list after adding more elements:" + strList.size());
        //call clear method
        strList.clear();
        System.out.println("after List calling clear() method:" + strList);
    }
}
```

AddAll method example code

```
import java.util.*;
public class AddAll {
    public static void main(String[] args) {
        List<String> sList = new ArrayList<String>(); // Creating a list
        sList.add("Pascal");
        sList.add("C++");
        //print the list
        System.out.println("List after adding two elements:" + sList);
        List<String> list = new ArrayList<String>(); // Create New list
        list.add("Java");
        list.add("Python");
        list.add("PHP");
        // addAll method - add list to strList
        sList.addAll(list);
        System.out.println("List after addAll:" + sList);
    }
}
```

Java List Methods(Con)

- **boolean addAll(Collection c)**

Collection whose elements are to be added to the list

Return is: true Method execution successful

The addAll method takes all the elements from collection **c** and appends them to the end of the list by maintaining the order that was set.

Java List Methods(Con)

- **boolean addAll(int index, Collection c)**

index- Position at which the collection is to be inserted.

c- Collection that is to be inserted in the list.

Return is: true -If collection elements are successfully added to the list.

The addAll method inserts all the elements in the specified collection into the list at the specified index. The subsequent elements are then shifted to the right. As in the case of the previous overload of addAll, the behavior is unspecified if the collection is altered when the operation is in progress.

Java List Methods(Con)

- **boolean contains(Object o)**

o-Element to be searched in the list.

Return is: true- If list contains the specified element.

The method 'contains' checks if the specified element is present in the list and returns a Boolean value true if the element is present. Otherwise, it returns false.

- **boolean containsAll(Collection c)**

c - Collection to be searched in the list.

Return is: true- If all elements in the specified collection are present in the list.

"containsAll" method checks if all the elements present in the specified collection are present in the list. If present it returns a true value and false otherwise.

Example code Java Contains & ContainsAll

```
import java.util.*;
public class ContainsContainsAll {
    public static void main(String[] args) {
        List<String> strList = new ArrayList<String>(); // Creating a list
        //add items in the list
        strList.add("Pascal");
        strList.add("C++");
        strList.add("Java");
        strList.add("Python");
        strList.add("PHP");
        if(strList.contains("C#")==true)
            System.out.println("Given list contains string 'C#'");
        else if(strList.contains("Java")==true)
            System.out.println("Given list contains string 'Java' but not string 'C#'");
        //Display size of list
        System.out.println("Size of list: " + strList.size());
        //create New List and Add New Items
        List<String> myList = new ArrayList<String>();
        myList.add("JavaScript");
        myList.add("HTML");
        strList.addAll(myList); //addAll methods send data to List into strlist
        if(strList.containsAll(myList)==true) //comprising two List
            System.out.println("List contains strings 'Java Script' and 'HTML'");
    }
}
```

Java List Methods(Con)

- **boolean equals(Object o)**

o - The object that is to be tested for equality.

Return is: true - If the given object is equal to the list.

This method is used to compare the given object with the list of equality. If the specified object is a list, then the method returns true. Both lists are said to be equal if and only if they are of the same size, and the corresponding elements in the two lists are equal and in the same order.

- **import java.util.LinkedList;**
- **import java.util.List;**
- **public class Main2 {**
- **public static void main(String[] args) {**
- **//define lists**
- **List<Integer> first_list= new LinkedList<>();**
- **List<Integer> second_list = new LinkedList<>();**
- **List<Integer> third_list = new LinkedList<>();**
- **//initialize lists with values**
- **for (int i=0;i<11;i++){**
- **first_list.add(i);**
- **second_list.add(i);**
- **third_list.add(i*i);**
- **}**
- **System.out.println("First list: " + first_list);**
- **System.out.println("Second list: " + second_list);**
- **System.out.println("Third list: " + third_list);**
- **}**

Java List Methods(Con)

```

• //use equals method
• if (first_list.equals(second_list) == true)
• System.out.println("\nfirst_list and second_list are equal.\n");
• else
• System.out.println("first_list and second_list are not equal.\n");
• if(first_list.equals(third_list))
• System.out.println("first_list and third_list are equal.\n");
• else
• System.out.println("first_list and third_list are not equal.\n");
• if(second_list.equals(third_list))
• System.out.println("second_list and third_list are equal.\n");
• else
• System.out.println("second_list and third_list are not equal.\n");
• }
• }
    
```

• Object get(int index)

index- position at which the element is to be returned.

Return is object- Element at the specified position.

The get() method returns the element at the given position. This method throws “`indexOutOfBoundsException`” if the index specified is out of the range of the list.

Java List Methods(Con)

```

import java.util.*;
public class Main2 {
    public static void main(String[] args) {
        //define list
        List list = new ArrayList();
        list.add("Java");
        list.add("C++");
        list.add("Python");
        //access list elements using index with get () method
        System.out.println("Element at index 0:" + list.get(0));
        //set element at index 1 to PHP
        list.set(1,"PHP");
        System.out.println("Element at index 1 changed to :"+ list.get(1) );
    }
}
    
```

hashCode	int hashCode ()	Returns the hash code value of the List.
indexOf	int indexOf (Object o)	Finds the first occurrence of the input element and returns its index
isEmpty	boolean isEmpty ()	Checks if the list is empty
lastIndexOf	int lastIndexOf (Object o)	Finds the last occurrence of the input element in the list and returns its index
remove	Object remove (int index)	Removes the element at the specified index
	boolean remove (Object o)	Removes the element at its first occurrence in the list
removeAll	boolean removeAll (Collection c)	Removes all elements contained in the specified collection from the list
retainAll	boolean retainAll (Collection c)	Opposite of removeAll. Retains the element specified in the input collection in the list.

Java List Methods(Con)

Set	Object set (int index, Object element)	Changes the element at the specified index by setting it to the specified value
subList	List subList (int fromIndex, int toIndex)	Returns sublist of elements between fromIndex(inclusive), and toIndex(exclusive).
sort	void sort (Comparator c)	Sorts the list element as per the specified comparator to give an ordered list
toArray	Object[] toArray ()	Returns array representation of the list
	Object [] toArray (Object [] a)	Returns the array representation whose runtime type is the same as a specified array argument
iterator	Iterator iterator ()	Returns an Iterator for the list
listIterator	ListIterator listIterator ()	Returns a ListIterator for the list
	ListIterator listIterator (int index)	Returns a ListIterator starting at the specified index in the list

```
import java.util.*;
public class Main2
{
    public static void main(String[] args) {
        List<Integer> mylist = new LinkedList<>();
        mylist.add(2);
        mylist.add(3);
        mylist.add(6);
        mylist.add(9);
        mylist.add(11);
        System.out.println("The list:" + mylist);
        //use hashCode() method to find hashcode of list
        int hash = mylist.hashCode();
        System.out.println("Hashcode for list:" + hash);
    }
}
```

Methods of Java List Iterator

void add(E e)	This method inserts the specified element into the list.
boolean hasNext()	This method returns true if the list iterator has more elements while traversing the list in the forward direction.
E next()	This method returns the next element in the list and advances the cursor position.
int nextIndex()	This method returns the index of the element that would be returned by a subsequent call to next()
boolean hasPrevious()	This method returns true if this list iterator has more elements while traversing the list in the reverse direction.
E previous()	This method returns the previous element in the list and moves the cursor position backward.
E previousIndex()	This method returns the index of the element that would be returned by a subsequent call to previous().
void remove()	This method removes the last element from the list that was returned by next() or previous() methods
void set(E e)	This method replaces the last element returned by next() or previous() methods with the specified element.

- **import** java.util.*;
- **public class** ListIterator1{
- **public static void** main(String args[]){
- List<String> list=new ArrayList<String>();
- list.add("Priyantha");
- list.add("Wasantha");
- list.add("Gamini");
- list.add(1,"Kamal");
- ListIterator<String> itr=list.listIterator();
- System.out.println("Traversing elements in forward direction");
- **while**(itr.hasNext()){
- System.out.println("index:"+itr.nextIndex()+" value:"+itr.next());}
- System.out.println("Traversing elements in backward direction");
- **while**(itr.hasPrevious()){
- System.out.println("index:"+itr.previousIndex()+" value:"+itr.previous());}
- } }




Advantages of ArrayList

- Since the size is resized during the running time, it is easy during implementation requirement changes.
- Because many predefined methods can be supported, it is very easy to store object during manipulation.
- Can store and delete data randomly
- Various types of Object can be entered.



Disadvantages of ArrayList

- If a data entry is added or removed from an array-based list, the data must be transferred to update the list.
- In the worst case, for an array-based list with n Data insertions, additions, and deletions are $O(n)$ Time.
- Also, all data in an array-based list must be stored in memory in order. Large lists require significant contiguous chunks of memory.
- Solutions is LinkedList –will discussed next week

- 
- https://www.tutorialspoint.com/data_structures_algorithms/array_data_structure.htm

HNDIT3032 Data Structures and Algorithms



Week 5-Linked List



Generics in Java

- You will have learned in the above lessons that we can use the collection framework to build a simple data structure concept in Java without algorithm implementation (add(), remove()).
- Likewise, we can do many implementations very easily using Java generics.

Generics in Java(Con)

- Java included support for writing generic class in J2SE 5.0 or JDK1.5 version
- The Generics framework allow us to define a class in term of a set of parameters that can be use as the type of declared variables , parameters , and return types
- Before Java SE 5 , Generics programming was heavily dependent upon the java Object class that the root of the hierarchy in a java programming language.
- Example :public class MyClass<T> {} This is a generic class that can be instantiated with any class type such as Integer , String , Double etc.

Generics in Java(Con)

- If you want to set the bubble sort algorithm, then you have to arrange two items next to each other in an array. But you have a problem if the data included in that array is of string type. You have to implement a separate method for that. example When your array is stored in the form of mango, apple, banana, you will have a problem when you want to sort mango and banana because it is not an integer. So you have to write a separate method for that. So by using Generics in Java facility you will be able to create a generic method **regardless** of data type.

Generics in Java(Con)

- // use < T> to specify Parameter type T is Specific type
- `class Gn<T> {`
- // object of type T is declared
- T obj;
- Gn(T obj) { `this.obj = obj;` } // constructor
- `public T getObject() { return this.obj; }`
- }
-
- //Driver class
- `class Generics1 {`
- `public static void main(String[] args)`
- {
- // instance of Integer type
- `Gn<Integer> iObj = new Gn<Integer>(15);`
- `System.out.println(iObj.getObject());`
- }
-
- // instance of String type
- `Gn<String> sObj`
- `= new Gn<String>("GeeksForGeeks");`
- `System.out.println(sObj.getObject());`
- `}}`

Generics in Java(Con)

Type Parameters

```
class Stack<T> { /* T is the type parameters *
```

- By convention, type parameter names are single, uppercase letters.
- Replace the type value with a concrete value (known as **parameterized** type
- Most commonly used type parameter names are:
 - ▶ E - Element (used extensively by the Java Collections Framework)
 - ▶ T - Type
 - ▶ N - Number
 - ▶ K - Key
 - ▶ V - Value
 - ▶ S,U,V etc. - 2nd, 3rd, 4th types

Generics in Java(Con)

Generics: Idea

Generic programming is a style of computer programming in which:

- algorithms are written in terms of types *to-be specified-later*, that are then
- *instantiated* when needed for specific types provided as *parameters*

Generics in Java(Con)

Tuple or pairs

Simple, yet useful idea: two values as one

Can be used to return more than two values from a function.

```
public class Pair <K,V>{ // Key (K) and Value (V)
    private K key;
    private V val;

    public Pair(K key, V val) {
        this.key = key;
        this.val = val;
    }

    public K getKey() { return this.key; }
    public V getVal() { return this.val; }
}
```

```

• public class Pairs <K,V>{ // Key (K) and Value (V)
•   int x,y;
•   private K key;
•   private V val;
•
•   public K getKey() { return this.key; }
•   public V getVal() { return this.val; }
•
•   public static Pairs dis()
•   {
•     Pairs p=new Pairs();
•     p.key="Amma";
•     p.val=90;
•     p.x=900;
•     p.y=800;
•     return p;
•   }
•   public static void main(String a[])
•   {
•     Pairs pi;
•     pi=Pairs.dis();
•     System.out.println(pi.val+" "+pi.key+" "+pi.x);
•   }
• }

```

Advantages of Generics

- Programs that use Generics has got many benefits over non-generic code.
- 1. Code Reuse: User can write a method/class/interface once and use it for any type user want
- 2. Type Safety: Generics make errors to appear compile time than at run time (It's always better to know problems in your code at compile time rather than making your code fail at run time). Suppose you want to create an ArrayList that store name of students, and if by mistake the programmer adds an integer object instead of a string, the compiler allows it. But, when we retrieve this data from ArrayList, it causes problems at runtime.

Return More Types in Method

```

• class Test<T> {
•   // An object of type T is declared
•   T obj;
•   Test(T obj) {
•     this.obj = obj;
•   } // constructor
•   public T getObject() { return this.obj; }
• }

• class TestRun {
•   public static void main(String[] args)
•   {
•     // instance of Integer type
•     Test<Integer> iObj = new Test<Integer>(15);
•     System.out.println(iObj.getObject());
•     // instance of String type
•     Test<String> sObj
•     = new Test<String>("GeeksForGeeks");
•     System.out.println(sObj.getObject());
•     //iObj = sObj; // This results an error
•   }
• }

```

Linked List

• Linked List:

To overcome the disadvantage of fixed size arrays, linked list were introduced.

A linked list consists of nodes of data which are connected with each other. Every node consist of two parts data and the link to other nodes. The nodes are created dynamically.

Linked list

Limitation of arrays:

- ▶ Need to know the size at the start! (at least some good estimate)
- ▶ Can not go beyond that!

Advantages of arrays:

- ▶ Contiguous in memory (good cache locality if accessed sequentially)
- ▶ Very limited bookkeeping is required (only need to know the start of the array and its size)

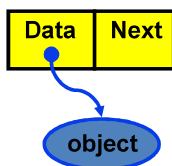
Alternative \Rightarrow Linked list Used when we do not know the size in advance and size varies a lot! (need to think before selecting)

Linked Lists

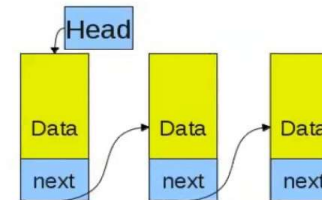
- Flexible space use
 - Dynamically allocate space for each element as needed
 - Include a pointer to the next item

← Linked list

- Each **node** of the list contains
 - the data item (an object pointer in our ADT)
 - a pointer to the next node



Linked list ...



- ▶ Head points to the first element (ideal for sequential)
- ▶ Add/remove items as required
- ▶ Each element contains a pointer to the next
- ▶ No limit on how many elements we can have
- ▶ Additional storage for pointers

In your program you need to store, on an average N integers. In the worst case this can be $2N$. Should you use an array or linked list? Explain. (assume 32bit addresses).

Linked Lists

- Collection structure has a pointer to the list **head**
 - Initially NULL

Collection



```
public class SampleLinked<T>{//Use generics

    class ListNode {
        public T data;
        public ListNode next;

        public ListNode(T data) {
            this.data = data;
            this.next = null;
        }
    } // end of node

    // now the linked List
    ListNode head;

    public SampleLinked() {
        this.head = null;
    }
}
```

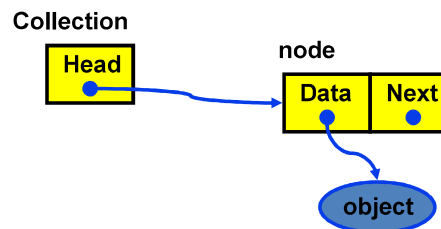
Linked list: Sample implementation

Main operations on a linked list:

- ▶ add: add a node to the list
- ▶ remove: remove an item from the list
- ▶ (relation between when an added item is removed is not specified)
- ▶ isEmpty: return true if the list is empty

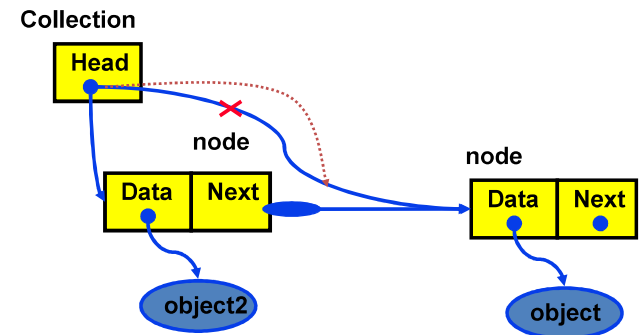
• Add first item **Linked Lists**

- Allocate space for node
- Set its data pointer to object
- Set Next to NULL
- Set Head to point to new node



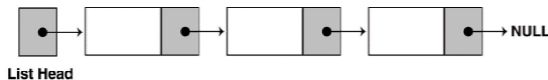
• Add second item **Linked Lists**

- Allocate space for node
- Set its data pointer to object
- Set Next to current Head
- Set Head to point to new node



The composition of a Linked List

- A linked list is called "linked" because each node in the series has a pointer that points to the next node in the list.



21

```

• public class Node<T> {
• //data to store
• private T data;
• //reference to the next
• private Node<T> next;
• //constructor set the object
• Node()
• {
• data=null;
• next=null;
• }
• //data store with in node object
• Node(T data)
• {
• this.data=data;
• this.next=null;
• }
• public T getData() {
• return data;
• }
• public void setData(T data) {
• this.data = data;
• }
• public Node<T> getNext() {
• return next;
• }
• public void setNext(Node<T> next) {
• this.next = next;
• }
• }
  
```

22

Declarations

- First you must declare a data structure that will be used for the nodes. For example, the following code could be used to create a list where each node holds a any data types

Declarations

- The next step is to declare a pointer to serve as the list head, as shown below.

```

public class List<T> {
//head of the List class
private Node<T> head;
//count variable
private int count;
//create an empty list
public List()
{
this.head=null;
this.count=0;
}
}
  
```

24

- Once you have declared a node data structure and have created a NULL head pointer, you have an empty linked list.
- The next step is to implement operations with the list.

Appending a Node to the List

- To append a node to a linked list means to add the node to the end of the list.
- The pseudocode is shown below.

```

Create a new node.
Store data in the new node.
If there are no nodes in the list
    Make the new node the first node.
Else
    Traverse the List to Find the last node.
    Add the new node to the end of the list.
End If.
    
```

Linked List Operations

- **Appending a Node to the List**
- **Traversing the List**
- **Inserting a Node**
- **Deleting a Node**

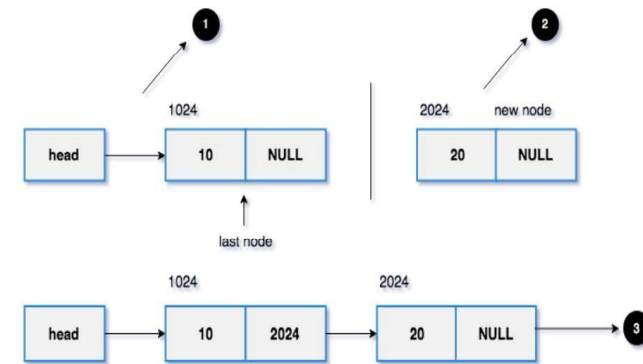
```

• public void addEnd(T data)
• {
• //check list is empty
• if(this.isEmpty())
• {
• this.addFront(data);
• }else {
• //create new node
• Node<T> node=new Node<>(data);
• //add to the end of list
• Node<T> current=head;
• while(current.getNext()!=null)
• {
• current=current.getNext();
• }
• //here we set the new node to next reference of last node
• current.setNext(node);
• node.setNext(null);
• count++;
• }
• }
    
```

Stepping Through the Program

- First, the list is checked for empty and if it is empty, the addFront(data) method has been called to add data to the front.
- Otherwise, a new node is created because data needs to be entered at the end of the list.
- The data to be entered into that node is also given.
- Then the while loop finds the last node because the last node is Null.
- Therefore, the loop finds the node by doing the next node until it is not null.
- By comparing the last found node with the next set node, our node set as the last node in the list.
- Last Node set be as Null

29



1. The head points to the memory address 1024 and it is the last node.
2. The new node with data as 20 and reference is NULL (address 2024).
set last node => next = new node. The new node added at the end of the linked list.
3. Finally, the new linked list.

Traversing the List

- The displayList member function traverses the list, displaying the value member of each node. The following pseudocode represents the algorithm.

Assign List head to node pointer.

While node pointer is not NULL

Display the value member of the node pointed to by node pointer.

Assign node pointer to its own next member.

End While.

```

• public void printData()
• {
•   if(this.isEmpty()) {
•     System.out.println("List is Empty:...");
•     return;
•   }
•   System.out.print("[");
•   if(!this.isEmpty())
•   {
•     System.out.printf("%s",head.getData());
•   }
•   Node<T> current=head.getNext();
•   while(current!=null)
•   {
•     System.out.printf(" , %s",current.getData());
•     current=current.getNext();
•   }
•   System.out.println("]");
• }
    
```

31

Stepping Through the Program

- First check if the list is empty and if it is empty then a message will be given as empty.
- If it is not empty, the head value is printed first.
- A new node will be created as current and equal to the after node of the head .
- The unfortunate node of a linked list is null, so the interaction is done by the while loop until the null node is found.

Create a new node.

Store data in the new node.

If there are no nodes in the list

Make the new node the first node.

Else

Find the first node whose value is greater than or equal the new value, or the end of the list (whichever is first).

Insert the new node before the found node, or at the end of the list if no node was found.

End If.

Inserting a Node

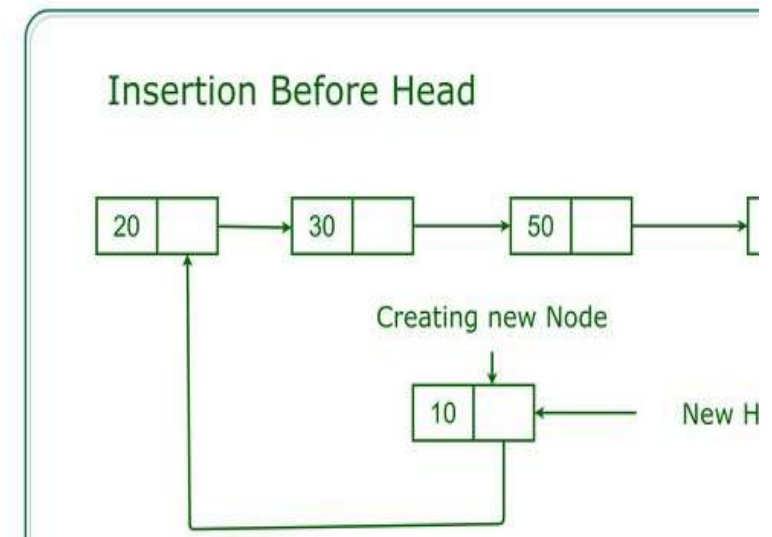
- Using the `linked List` structure again, the pseudocode on the next slide shows an algorithm for finding a new node's proper position in the list and inserting there.
- The algorithm assumes the nodes in the list are already in order.

```
public void addFront(T data)
{
    //create new node
    Node<T> node=new Node<>(data);
    if(this.isEmpty())
    {
        head=node;
    }
    else//if the list not empty
    {
        node.setNext(head);//set the head to the next pointer of new node
        head=node;//make it first node
    }
    count++;
}
```

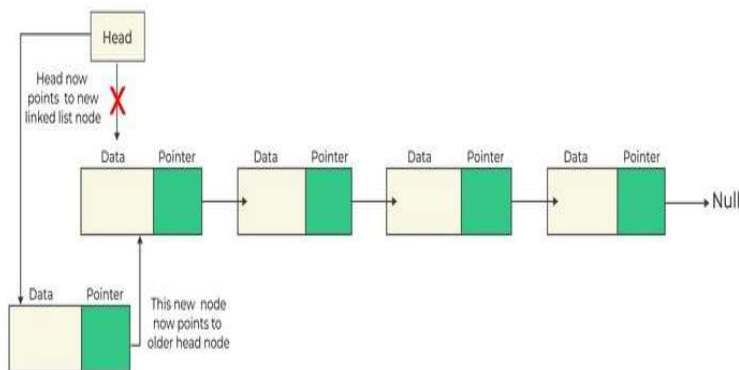
Stepping Through the Program

- First a node is created and the data is given to it
- The Linked List is checked for empty, and if it is empty, the created node is equal to the head.
- Since the new node must be entered first in the Linked list, the head is equal to the next node and the new node is equal to the head.

37



Insertion at Beginning



Deleting a Node

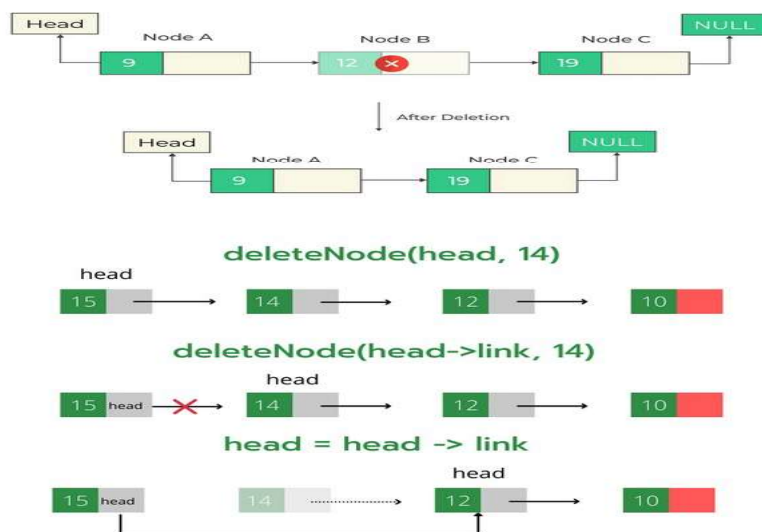
- Deleting a node from a linked list requires two steps:
 - Remove the node from the list without breaking the links created by the next pointers
 - Deleting the node from memory
- The `remove()` function begins on the next slide.

```

• public boolean remove(T data)
• { //case 1 check the list is empty
• if(this.isEmpty())
• {return false;
• }
• if(head.getData().equals(data))
• {
• head=head.getNext();
• count--;
• return true;}
• else {
• //node having the between the node or node is last node
• Node<T> current=head;
• Node<T> pre=null;
• while(current!=null )
• { if( current.getData().equals(data))
• { pre.setNext(current.getNext()); //skip the current node
• count--;
• return true;
• } //update navigate pointer
• pre=current;
• current=current.getNext();
• }
• return false; //node not found data}

```

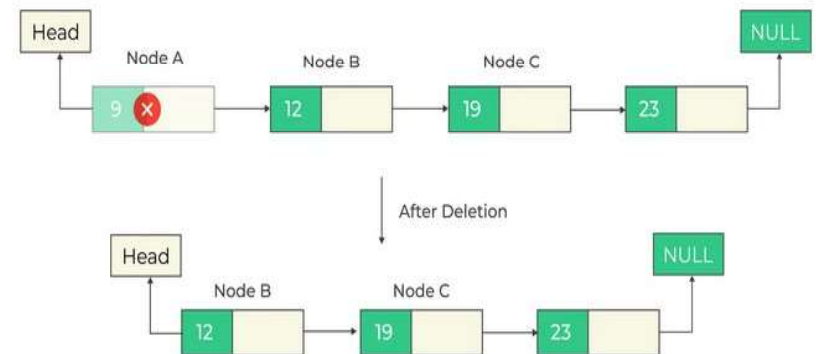
Deletion At Middle



Stepping Through the Program

- Check linked list is empty ,if it is empty return false
- If it is not empty , The given data should be found in the linked list. First, it is checked whether the data is the same as the head. The head node is equal to the next node.
- Otherwise, while loop is used to check all other nodes until null. If we find the node related to our data while checking, it should be deleted.
- The node found is equal to the first empty node and it is deleted. Then the empty node is equal as the (pre)current node.
- Finally If there is no node according to the given data, it will be returned as false.

Deletion At Beginning



Exercise:

- 1.) Update the program given to you and display the data in reverse order in the linked list.
- 2.) Write a short note on the following topics
 - 1.) Linked List vs Arraylist
 - 2.) Linked List vs Array

Stacks

- Stack is Last in First Out (LIFO)
- A stack is a linear data structure which can be accessed only at one of its ends for storing and retrieving data.
- There are two ways of implementing a stack Array (Static) and linked list (dynamic).

HNDIT3032 Data Structures and Algorithms

Week 7-Stack



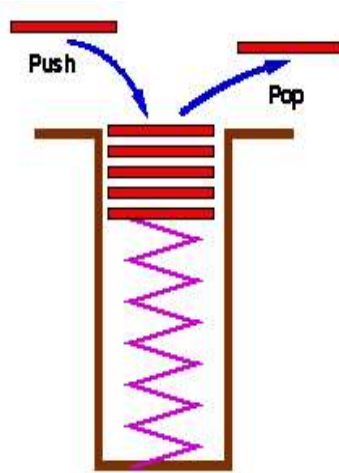
Static Application (Array Based)

- The array implementation of our collection has one serious drawback: you must know the maximum number of items in your collection when you create it.
- This presents problems in programs in which this maximum number cannot be predicted accurately when the program starts up.
- Fortunately, we can use a structure called a linked list to overcome this limitation.

Dynamic application (Linked list based)

- The linked list is a very flexible dynamic data structure: items may be added to it or deleted from it at will.
- A programmer need not worry about how many items a program will have to accommodate.

Examples



Array implementation of Stacks

- A stack can be implemented with an array and an integer. The integer **top** (Top of stack) provides the array index of the top element of the stack.
- Thus if **top** is -1 , the stack is empty.

- A stack is generally implemented with only two principle operations (apart from a constructor and destructor methods):

Push :adds an item to a stack

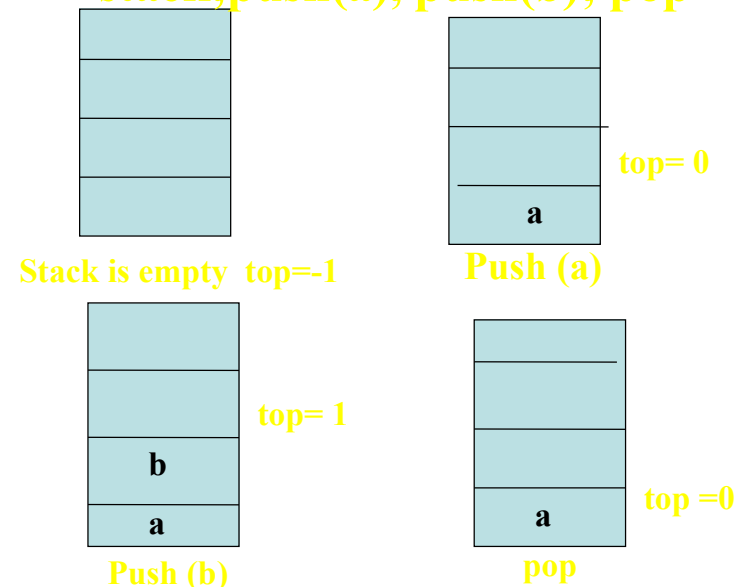
Pop :extracts the most recently pushed item from the stack

Other methods such as

top: returns the item at the top *without removing it*

isempty :determines whether the stack has anything in it

How the stack routines work:empty stack;push(a), push(b); pop



Array implementation

Public operations

- void **push(x)** – Insert x
- void **pop()** – Remove most recently inserted item
- boolean **isEmpty()** – Return true if empty, false otherwise
- void **display()** – Display all items
- Boolean **isFull()** -Return true is Stack is Full, false otherwise

Stack Algorithms

Void Push(item)

```
{  
    If (stack is full) print “ stack over flow”  
    else  
        Increment top ;  
        Stack [top]= item;  
}
```

10

Void Pop()

```
{  
    If( stack is empty) print” stack under flow”  
    Else  
        Stk[top]=NULL  
        Decrement top  
}
```

Void Display()

```
{  
    If ( stack is empty) print” no element to display”  
    else  
        for i= top to 0 step -1  
            Print satck[i];  
}
```

-

11

12

- `boolean isEmpty() {`
- `if(top<=-1){`
- `return true;`
- `}`
- `else`
- `return false;`
- `}`

13

- `boolean isFull() {`
- `if(top>=SizeofArray-1)`
- `return true;`
- `else`
- `return false;`
- `}`

14

Implementation

- `public class ArrayStack {`
- `public int size;`
- `int top;`
- `public int stk[]=new int[size];`
- `public ArrayStack(int size, int[] arr) {`
- `super();`
- `this.size = size;`
- `this.stk = arr;`
- `top=-1;}`

15

- `public int getSize() {`
- `return size;}`
- `public int[] getStk() {`
- `return stk;}`

16

- **public void** push(int n)
- {
- **if**(isFull()==**true**) {
- System.**out**.println("Stack is Full");
- }
- **else**{
- top++;
- stk[top]=n;
- }}

17

- **public void** pop(){
- **if**(isEmpty()==**true**) {
- System.**out**.println("Stack is Empty...");}
- **else**{
- System.**out**.println("Stack top value Deleted "+stk[top]);
- stk[top]=0;
- top--;}}

18

- **public void** display(){
- **if**(isEmpty()==**true**){
- System.**out**.println("Stack is Empty...");}
- **else**{
- **for**(int j=0;j<=top;j++){
- System.**out**.print(stk[j]+","); }
- System.**out**.println();}}

19

```

public boolean isFull()
{
    if(top>=getSize()-1)
        return true;
    else
        return false;
}

public boolean isEmpty() {
    if(top<=-1){
        return true;
    }
    else
        return false;
}
    
```

20

- **public static void** main(String ar[]){
- Scanner **sc=new** Scanner(System.**in**);
- System.**out**.println("Enter `Stack Size");
- **int** StackSize=sc.nextInt();
- **int** Stk[]=**new int**[StackSize];
- ArrayStack op=**new** ArrayStack(StackSize,Stk);
- **boolean** x=**true**;

21

```
+ "1-->Add Value in Stack \n "
+ "2--> Delete Value in Stack \n "
+ "3--> Display Value in Stack\n "
+ "4 --> End Program\n");
int chose=sc.nextInt();
```

22

```
switch(chose)
{case 1:System.out.println("Enter
Your Stack Value");
int selection=sc.nextInt();
op.push(selection);break;
case 2:op.pop();break;
case 3:op.display();break;
case 4:System.out.flush();}
} } }
```

23

OUTPUT

```
Enter Stack Size
4
Chose Yor Options
1-->Add Value in Stack
2--> Delete Value in Stack
3--> Display Value in Stack
4 --> End Program
1
Enter Your Stack Value
11
Chose Yor Options
1-->Add Value in Stack
2--> Delete Value in Stack
3--> Display Value in Stack
4 --> End Program
3
11,
```

```
Chose Yor Options
1-->Add Value in Stack
2--> Delete Value in Stack
3--> Display Value in Stack
4 --> End Program
1
Enter Your Stack Value
23
Chose Yor Options
1-->Add Value in Stack
2--> Delete Value in Stack
3--> Display Value in Stack
4 --> End Program
3
11,23,
Chose Yor Options
1-->Add Value in Stack
2--> Delete Value in Stack
3--> Display Value in Stack
4 --> End Program
2
Stack top value Deleted 23
Chose Yor Options
1-->Add Value in Stack
2--> Delete Value in Stack
3--> Display Value in Stack
4 --> End Program
3
11,
```

Stack Operation in Collection Framework

push(object element)-push (insert) element in stack top of the stack
 pop()-remove the top of element in stack
 peek()-return the top of element of stack but dose not remove
 empty()- It returns true if nothing is on the top of the stack. Else, returns false.
 search (object element)-It determines whether an object exists in the stack. If the element is found, It returns the position of the element from the top of the stack. Else, it returns -1.

```

// Remove element stacks
String element = animals.pop();
System.out.println("Removed Element: " + element);
//return top element in stack
String element1 = animals.peek();
System.out.println("Element at top: " + element1);
//search value in stack index
int position = animals.search("Horse");
System.out.println("Position of Horse: " + position);
    
```

Implementation

```

import java.util.Stack;
public class ArrayStack {
    public static void main(String[] args) {
        Stack<String> animals= new Stack<>();
        // Add elements to Stack
        animals.push("Dog");
        animals.push("Horse");
        animals.push("Cat");
        System.out.println("Initial Stack: " + animals);
    }
}
    
```

```

//check stack is empty or not
boolean result = animals.empty();
System.out.println("Is the stack empty? " + result);
}
    
```


Output

Initial Stack: [Dog, Horse, Cat]

Removed Element: Cat

Element at top: Horse

Position of Horse: 1

Is the stack empty? false

Thanks

HNDIT3032 Data Structures and Algorithms



Queues

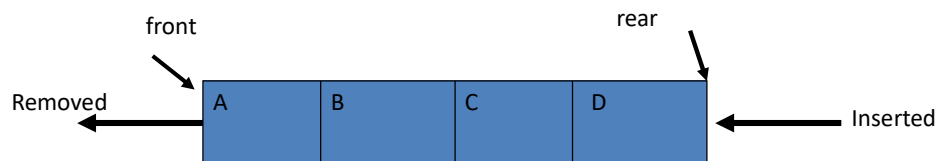


Week 8-Queue



Queue Definition

- A queue is an ordered collection of items from which items may be deleted at one end (called front of the queue) and into which items may be inserted at the other end (called the rear of the queue)



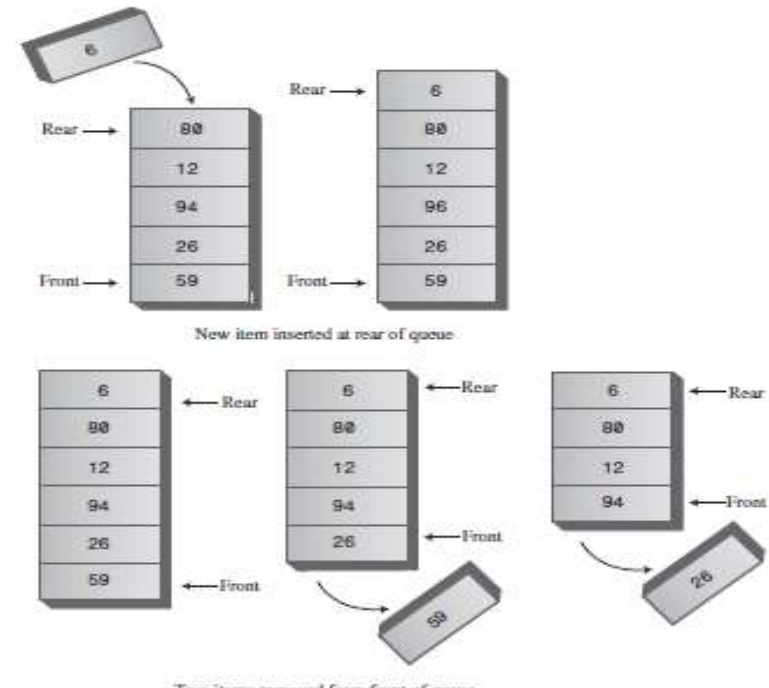
The queue data structure

- A queue is used in computing in much the same way as it is used in every day life: allow a sequence of items to be processed on a **first-come-first-served basis**.

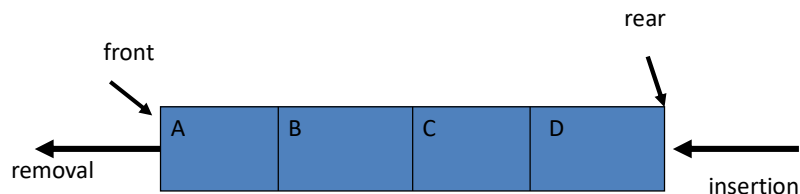
Eg: In most computer installations, for example, one printer is connected to several different several machines. So that more than one user can submit printing jobs to the same printer, it maintains a queue.

The queue data structure-applications

- Virtually every real-life line (supposed to be) queue. For instance, lines at ticket counters are queues, because service is first come first served.
- In computer networks. there are many network setups of personal computers in which the disk is attached to one machine. known as the file server. because service is first come first served.



Basic array implementation of queues



Queue initialisation

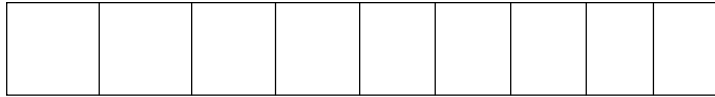
- For an empty queue



For an empty queue rear must be initialised to rear-1,
Front=-1, rear=-1 and size=0

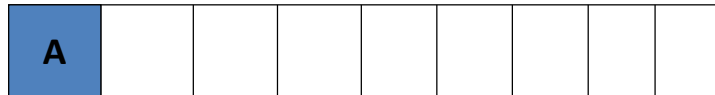
Array implementation of queues

Eg:



Size=0,front=-1,rear=-1

Enqueue(A)



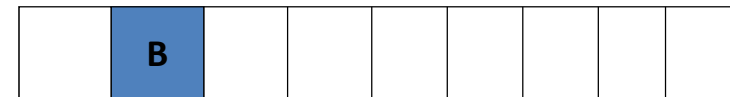
Size=1,front=-1,rear=0

Enqueue(B)



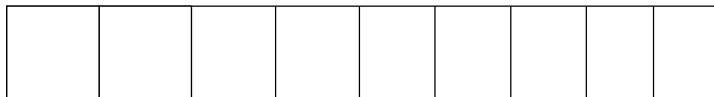
Size=2,front=-1,rear=1

Dequeue()



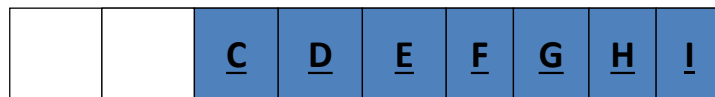
Size=1,front=0,rear=1

Dequeue



Size=0,front=1,rear=1

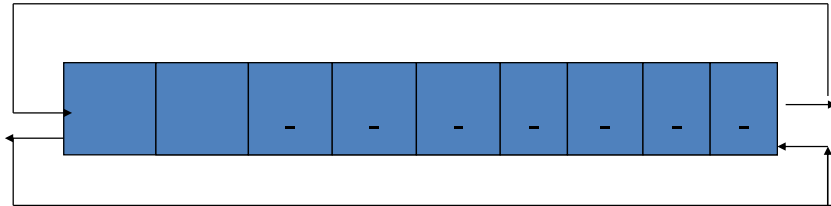
After 7 Enqueues



Size=7,front=1,rear=8

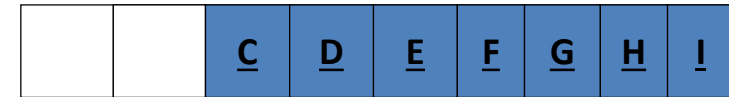
- There is plenty of extra space :All the positions before the front are unused and can thus be recycled. When either rear or front reaches the end of the array, we reset it to the beginning. This operation is called a circular array implementation.

Circular array implementation



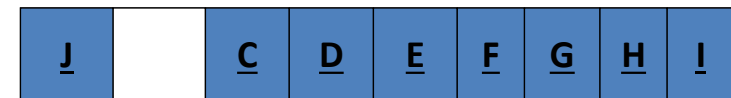
Circular array implementation (Array implementation of the queue with wraparound)

After 7 Enqueue



Size=7,front=1,rear=8

Enqueue(J)



Size=8,front=1,rear=0

Queue implementation.

- The queue class has four data fields
 - A dynamically expanding array
 - The number of items currently in the queue
 - The array index of the front item
 - The array index of the rear item

Queue Algorithms

Insert (item)

```
{
  If rear = max -1 then print " queue is full"
  else
  {
    Increment rear
    Queue [rear]=item;
  }
}
```

Delete()

```
{
  If front = rear print "queue is empty"
else
  Increment front
  Array[front]= NULL;
}
```

Display()

```
{
  If front=rear print "queue is empty "
else
  For i =front to rear
  Print queue[i];
}
```

Queue Implementation

- **import** java.util.Scanner;
- **public class** ArrayQueue {
- **public int** size;
- **int** front,rear;
- **public int** que[]=new int[size];
- **public** ArrayQueue(**int** size, **int**[] arr) {
- **super**();
- **this.size** = size;
- **this.que** = arr;
- front=-1;
- rear=-1;}
- **public int** getSize() {
- **return** size;}
- **public int**[] getQue() {
- **return** que;}

- **public void** add(**int** n)
- {
- **if**(isFull()==**true**) {
- System.**out**.println("Queue is Full");
- }
- **else**{
- front++;
- que[front]=n;
- }}

- **public void** delet(){
- **if**(isEmpty()==**true**) {
- System.**out**.println("Queue is Empty...");}
- **else**{
- rear++;
- System.**out**.println("Queue first value Deleted "+que[rear]);
- que[rear]=0;
- }

- **public boolean** isFull()
- {
- **if**(front>=getSize()-1)
- **return true**;
- **else**
- **return false**;
- }
- **public boolean** isEmpty() {
- **if**(front==rear){
- **return true**;
- }
- **else**
- **return false**;
- }

- **public void** display(){
- **if**(isEmpty()==**true**) {
- System.**out**.println("Queue is Empty...");}
- **else**{
- **for**(int j=rear+1;j<=front;j++){
- System.**out**.print(que[j]+","); }
- System.**out**.println();}}

- **public static void** main(String ar[]){
- Scanner sc=**new** Scanner(System.**in**);
- System.**out**.println("Enter Queue Size");
- **int** ArraySize=sc.nextInt();
- **int** Arr[]=**new int**[ArraySize];
- ArrayQueue op=**new** ArrayQueue(ArraySize,Arr);
- **boolean** x=**true**;

- **while**(x){
- System.**out**.println("Chose Yor Options \n "
- + "1-->Add Value in Quequ \n "
- + " 2--> Delete Value in Quequ \n "
- + "3--> Display Value in Quequ \n "
- + "4 --> End Program\n");
- **int** chose=sc.nextInt();

- **switch**(chose)
- {**case** 1: System.**out**.println("Enter Your Quequ Value");
- **int** selection=sc.nextInt();
- op.add(selection);**break**;
- **case** 2:op.delet();**break**;
- **case** 3:op.display();**break**;
- **case** 4: System.**out**.flush();}
- } }

- Thanks

HNDIT3032 Data Structures and Algorithms



Week 9-Tree



Trees

Tree is one of the most important non-linear data structures in computing. It allows us to implement faster algorithms(compared with algorithms using linear data structures).

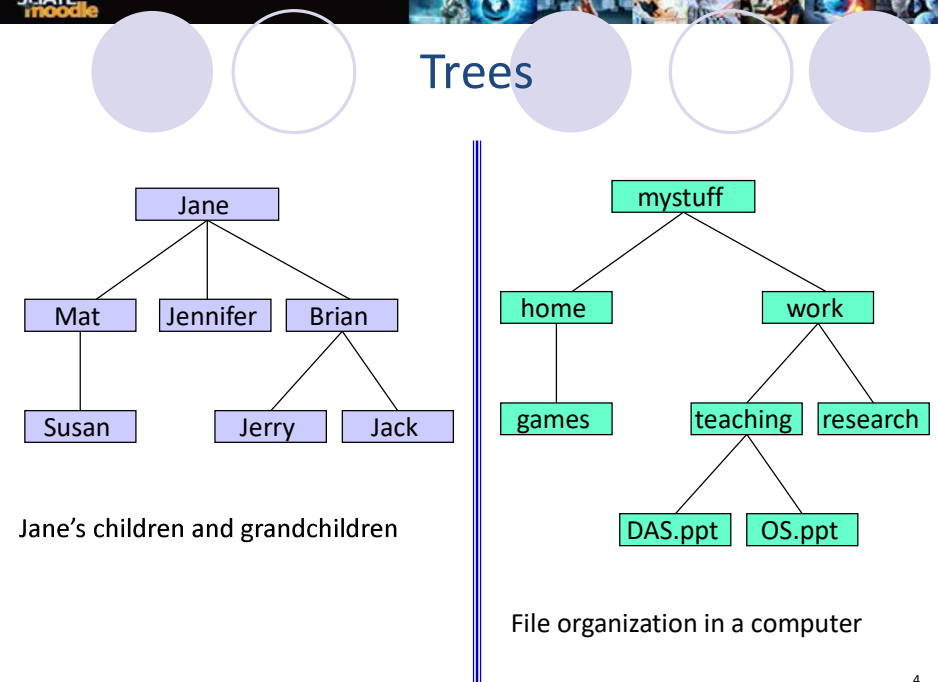
Application areas are :

- Almost all operating systems store files in trees or tree like structures.
- Compiler Design/Text processing
- Searching Algorithms
- Evaluating a mathematical expression.
- Analysis of electrical circuits.

2

An application :File system

- There are many applications for trees. A popular one is the directory structure in many common operating systems, including VAX/VMX, Unix and DOS.



Tree Types

- Binary tree — each node has at most two children
- General tree — each node can have an arbitrary number of children.
- Binary search trees
- AVL trees

5

Tree Terminology and Basic Properties

- **Definition:** A Tree is a set of nodes storing elements in a parent-child relationship with the following properties:
 - It has a special node called **root**.
- Each node different from the **root** has a **parent** node.
 - **Parent** — the parent of a node is the node linked above it.

6

Tree Terminology and Basic Properties

- Sibling
- Ancestor
- Descendant
- Leaf.
- Subtree
- Path of two nodes
- Length of a path

7

General Trees.

Trees can be defined in two ways :

- Recursive
- Non- recursive

One natural way to define a tree is recursively

8

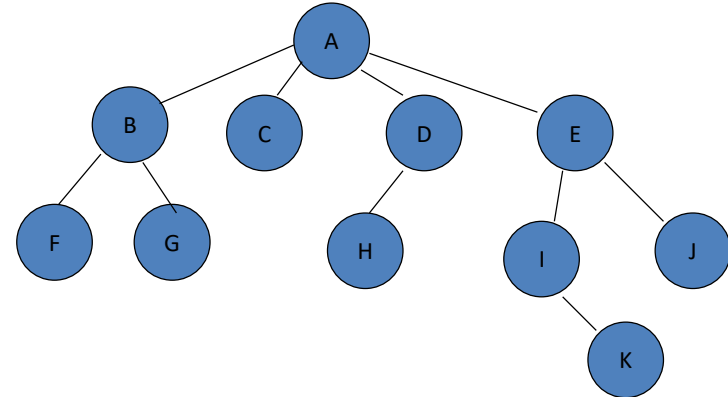
General trees-Definition

- A tree is a collection of nodes. The collection can be empty; otherwise a tree consists of a distinguish node r , called root, and zero or more non-empty (sub)trees $T_1, T_2, T_3, \dots, T_K$. Each of whose roots are connected by a directed edge from r .

9

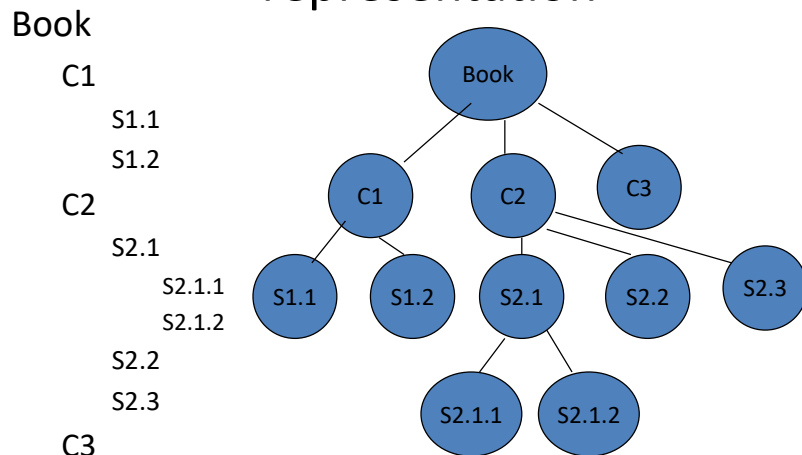
General trees-Definition

- A tree consists of set of nodes and set of edges that connected pair of nodes.



10

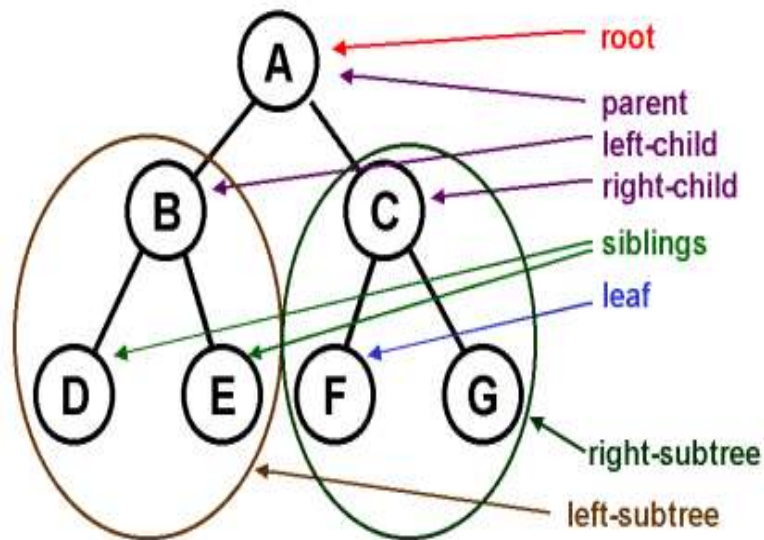
Eg. A table of contents and its tree representation



11

- Degree : The number of sub tree of a node is called its degree.
Eg. Degree of book $\rightarrow 3$, $C1 \rightarrow 2$, $C3 \rightarrow 0$
- Nodes that have degree 0 is called Leaf or Terminal node. Other nodes called non-terminal nodes. Eg. Leaf nodes : $C3, S1.1, S1.2$ etc.
- Book is said to be the father (parent) of $C1, C2, C3$ and $C1, C2, C3$ are said to be sons (children) of book.
- Children of the same parent are said to be siblings. Eg. $C1, C2, C3$ are siblings (Brothers)
- Length : The length of a path is one less than the number number of nodes in the path. (Eg path from book to $s1.1 = 3 - 1 = 2$)

12



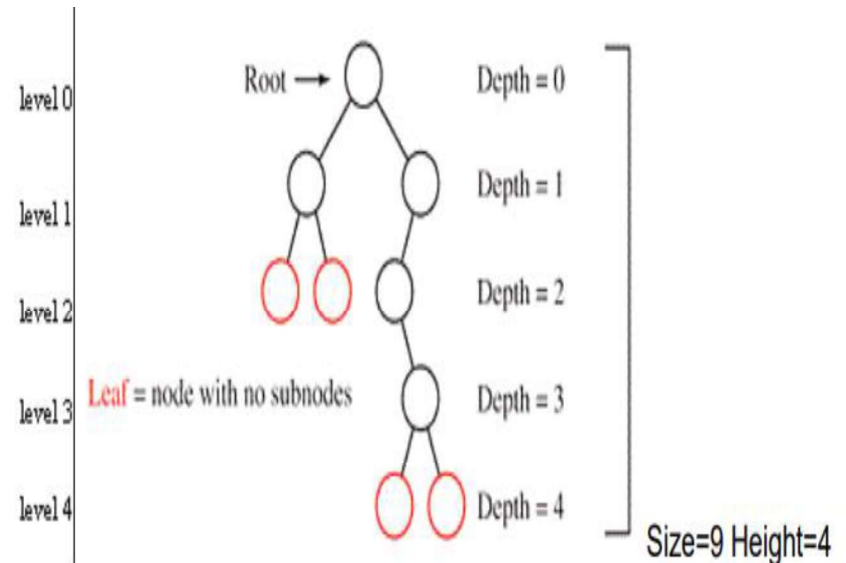
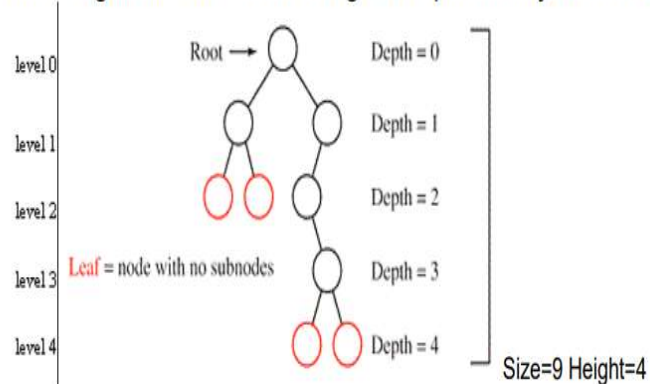
- If there is a path from node a to node b , then a is an ancestor of b and b is descendent of a.
- In above example, the ancestor of S2.1are itself,C2 and book, while it descendent are itself, S2.1.1 and S2.1.2.
- An ancestor or descendent of a node, other than the node itself is called a proper ancestor or proper descendent.
- Height of a tree — the maximum depth of a leaf node. ...[In above example height=3]
- Depth of a node — the length of the path between the root and the node.[In above example node C1 has Depth 1, node S2.1.2 has Depth 3.etc.]

Size, depth, height and level

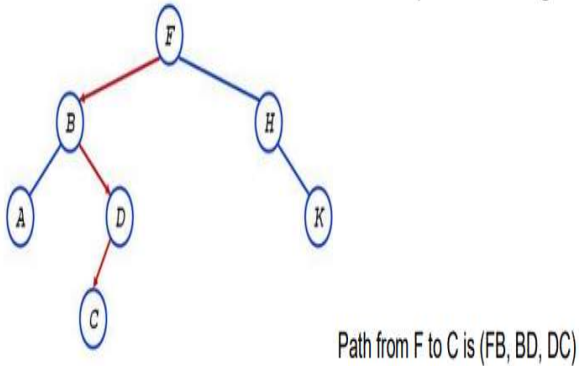
The size of a tree is the number of nodes that it contains.

The depth of a node is the number of edges from the root to the node.

The height of a tree is the largest depth of any of its nodes.



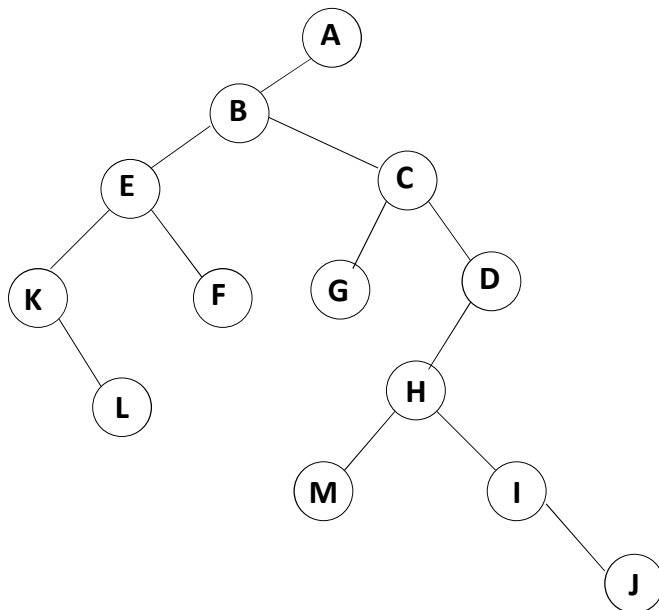
Path between two nodes in a tree is a sequence of edges which connect those nodes.



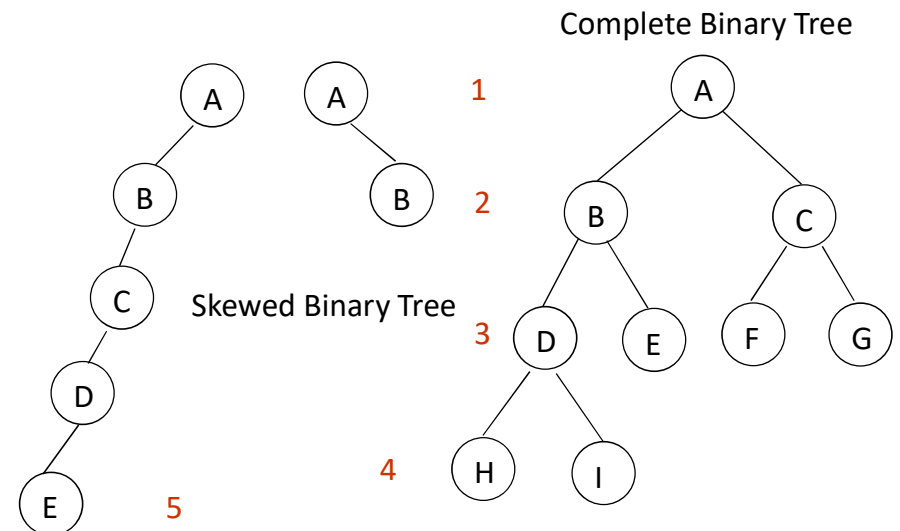
Binary Trees

- A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called *the left subtree* and *the right subtree*.
- Any tree can be transformed into binary tree.
 - by left child-right sibling representation
- The left subtree and the right subtree are distinguished.

***Figure 5.6:** Left child-right child tree representation of a tree (p.191)



Samples of Trees

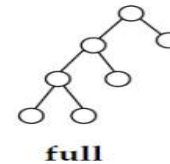


Full Binary Tree

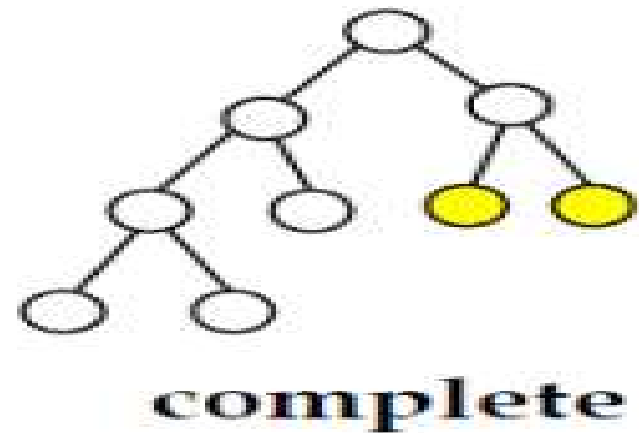
- A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.
- It is also known as **a proper binary tree**.

Prove by induction.

$$\sum_{i=1}^k 2^{i-1} = 2^k - 1$$



- A complete binary tree is a binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left.
- A complete binary tree is just like a full binary tree, but with two major differences
- All the leaf elements must lean towards the left.
- The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.



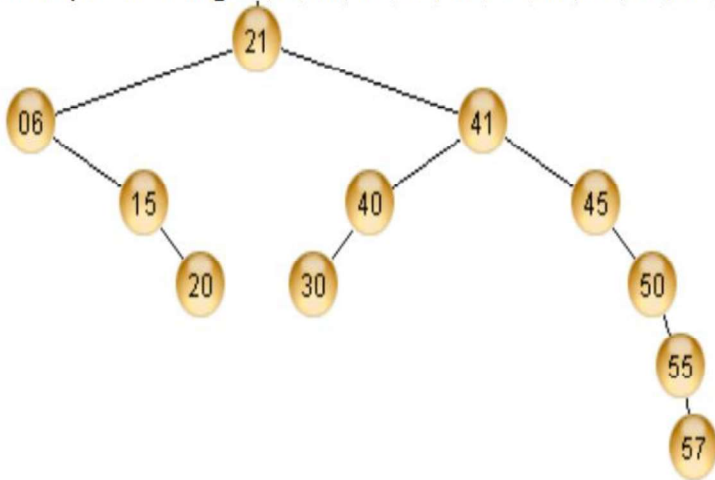
Perfect Binary Tree

- A **perfect binary tree** is a special type of binary tree in which all the leaf nodes are at the same depth, and all non-leaf nodes have two children. In simple terms, this means that all leaf nodes are at the maximum depth of the tree, and the tree is completely filled with no gaps.
- **Total number of nodes:** A tree of height h has total nodes = $2^{h+1} - 1$. Each node of the tree is filled. So total number of nodes can be calculated as $2^0 + 2^1 + \dots + 2^h = 2^{h+1} - 1$.

Binary Search Tree

- It is a binary tree such that for every node N in the tree:
- All keys in N 's left sub tree are less than the key in N , and
- All keys in N 's right sub tree are greater than the key in N .
- Note: if duplicate keys are allowed, then nodes with values that are equal to the key in node N can be either in N 's left sub tree or in its right sub tree (but not both). In these notes, we will assume that duplicates are not allowed.

Example: Inserting 21, 6, 41, 45, 50, 55, 57, 40, 50, 15, 20, 30 in a BST.

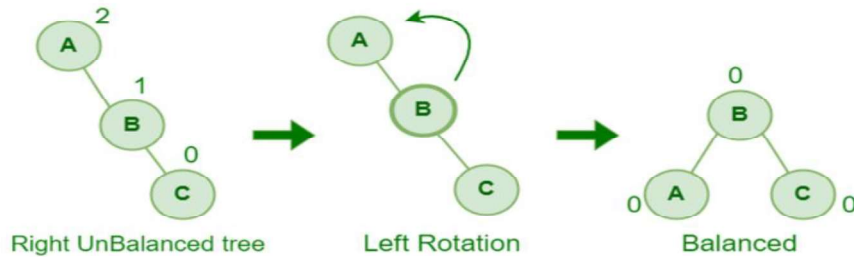


AVL Tree

- An **AVL tree** defined as a self-balancing BST where the difference between heights of left and right sub trees for any node cannot be more than one.
- The difference between the heights of the left sub tree and the right sub tree for any node is known as the **balance factor** of the node.
- The AVL tree is named after its inventors, Georgy Adelson-Velsky and Evgenii Landis, who published it in their 1962 paper "An algorithm for the organization of information"

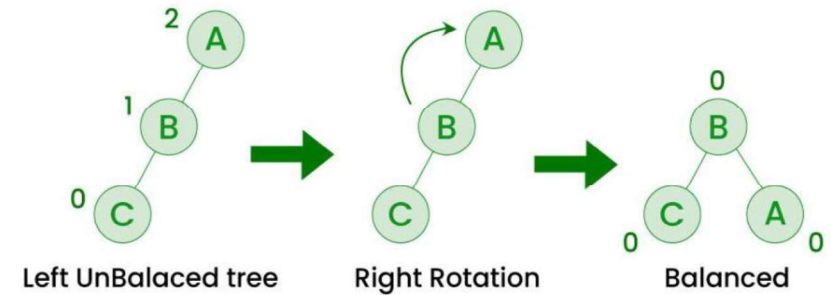
balance factor

- Rotating the sub trees in an AVL Tree: Left Rotation:



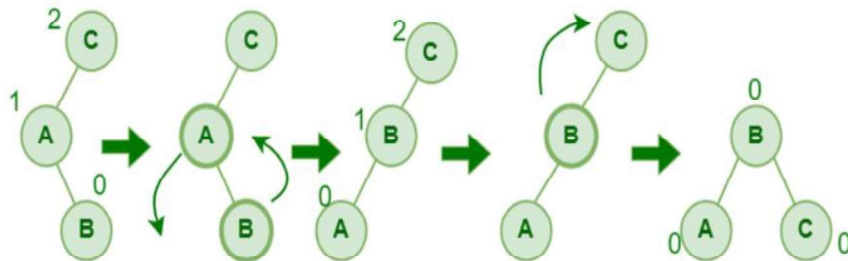
balance factor(con)

- Rotating the sub trees in an AVL Tree: right Rotation:



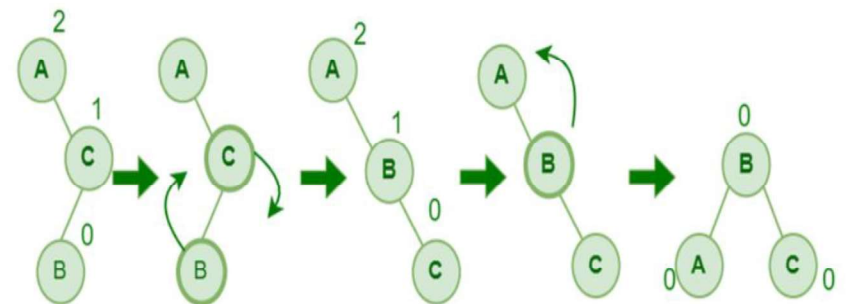
balance factor(con)

- Left-Right Rotation:



balance factor(con)

- Right-Left Rotation:



HNDIT3032 Data Structures and Algorithms



Week 9-BST

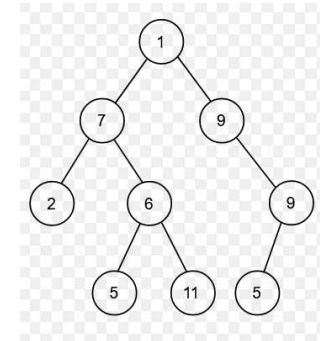


Types of Binary Tree

- Full Binary Tree/Strictly Binary Tree
- Complete Binary Tree /Perfect Binary Tree
- Almost Complete Binary Tree/in Complete Binary Tree
- Left Skewed Binary Tree
- Right Skewed Binary Tree

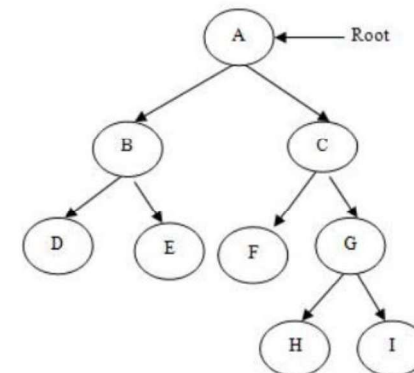
Binary Tree

- Every Node in a tree should have at most two children.



Full Binary Tree/Strictly Binary Tree

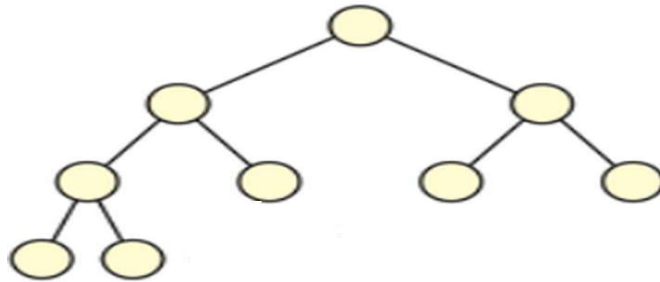
- Every Node must have two children except the leaf nodes



Complete Binary Tree

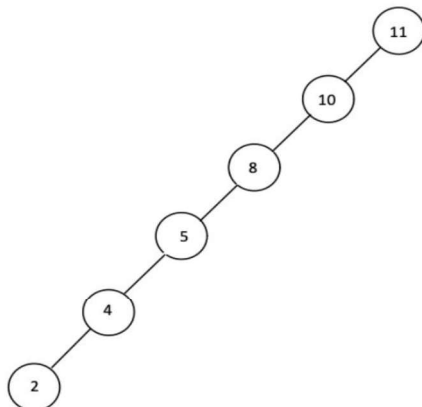
- Every Node must have two children in levels except in last level but filled from left to right

Almost Complete Binary Tree



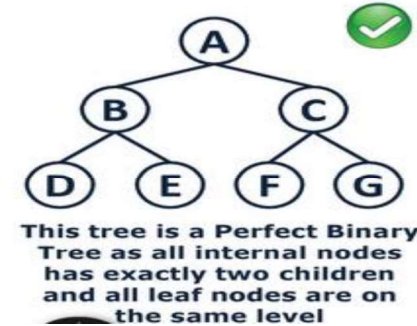
Left Skewed Binary Tree

- Tree only content left children



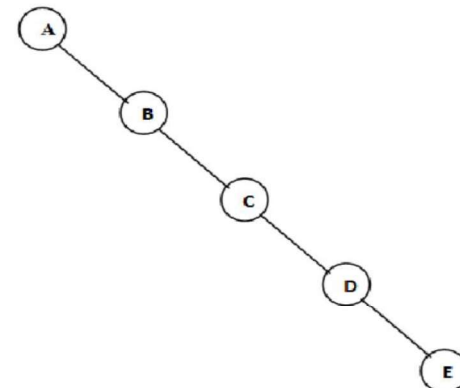
Complete Binary Tree /Perfect Binary Tree

- Every Node must have two children in a all levels. each level there must be 2^n nodes. n is a level



Right Skewed Binary Tree

- Tree only content right children

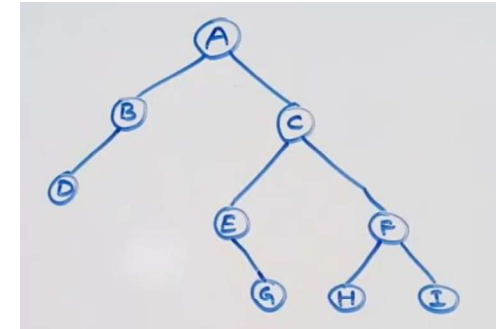


Tree Traversals

- Inorder Traversals
- Preorder Traversals
- Postorder Traversals

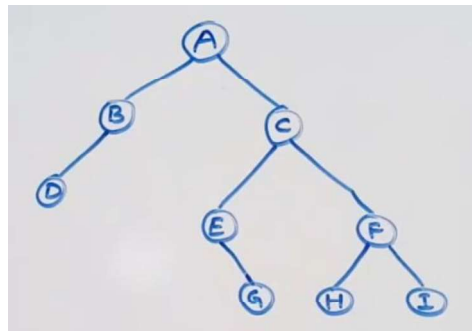
Inorder Traversals

- Left child – Root Node – Right Child
DBAEGCHFI



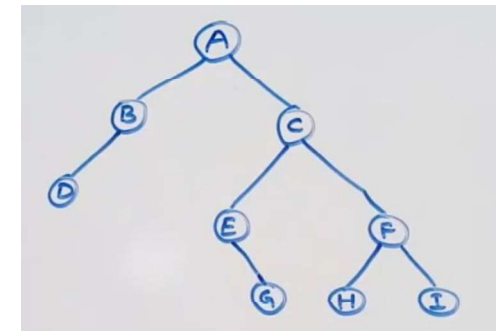
Preorder Traversals

- Root Node – Left Child – Right Child
ABDCEGFHI



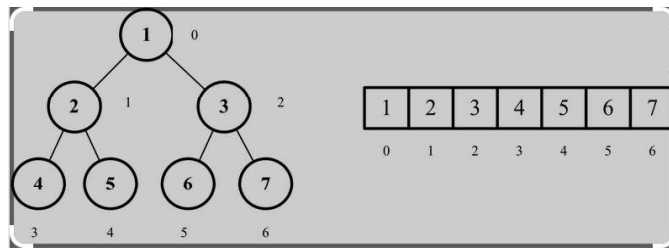
Postorder Traversals

- Left Child – Right Child – Root Node
DBGEHIFCA



Binary Tree Representation at Array

- Consider the Root Node at index 0
- Left child is placed at $2*i+1$ where i is position of parent
- Right child is placed at $2*i+2$ where i is position of parent

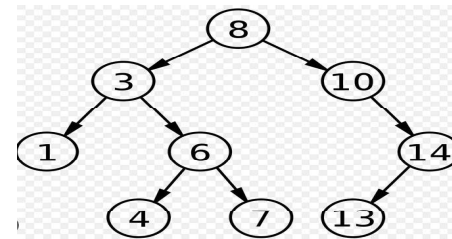


BST operation

- Insertion
- Deletion (0 children, 1 children, 2 children)
- Searching
- Minimum
- Maximum

Binary Search Tree

- Every node must have at most (0,1,2) two children.
- All the element of the left subtree must have less than root node
- All the element of the right subtree must have greater than root node



Insertion

- Algorithm
 - If the tree is empty, then consider element as a root
 - If the element greater than the root insert to right side
 - If the element less than the root insert to left side

(insert this number set in BST-
50,30,10,60,80,20,70,55,35,5)

Deletion

Three Ways

- **Deleting a node but with no children(leaf node)**- If the node to be deleted is a leaf node, then delete it.
- **Delete a node having one Children** -If the node to be deleted has one child then, delete the node and place the child of the node at the position of the deleted node.
- **Delete a Node Having Two Children** -If the node to be deleted has two children then, find the inorder successor or inorder predecessor of the node according to the nearest capable value of the node to be deleted. Delete the inorder successor or predecessor using the above cases. Replace the node with the inorder successor or predecessor.

Searching

- Searching in BST involves the comparison of the key values. If the key value is equal to root key then, search successful, if lesser than root key then search the key in the left subtree and if the key is greater than root key then search the key in the right subtree.

Searching in BST algorithm

- Check if tree is NULL, if the tree is not NULL then follow the following steps.
- Compare the key to be searched with the root of the BST.
- If the key is lesser than the root then search in the left subtree.
- If the key is greater than the root then search in the right subtree.
- If the key is equal to root then, return and print search successful.
- Repeat step 3, 4 or 5 for the obtained subtree.

Advantages BST

- BST is fast in insertion and deletion when balanced. It is fast with a time complexity of $O(\log n)$.
- BST is also for fast searching, with a time complexity of $O(\log n)$ for most operations.
- BST is efficient. It is efficient because they only store the elements and do not require additional memory for pointers or other data structures.
- We can also do range queries – find keys between N and M ($N \leq M$).
- BST code is simple as compared to other data structures.
- BST can automatically sort elements as they are inserted, so the elements are always stored in a sorted order.
- BST can be easily modified to store additional data or to support other operations. This makes it flexible.



Disadvantages BST

- The main disadvantage is that we should always implement a balanced binary search tree. Otherwise the cost of operations may not be logarithmic and degenerate into a linear search on an array.
- They are not well-suited for data structures that need to be accessed randomly, since the time complexity for search, insert, and delete operations is $O(\log n)$, which is good for large data sets, but not as fast as some other data structures such as arrays or hash tables.
- A BST can be imbalanced or degenerated which can increase the complexity.
- Do not support some operations that are possible with ordered data structures.
- They are not guaranteed to be balanced, which means that in the worst case, the height of the tree could be $O(n)$ and the time complexity for operations could degrade to $O(n)$.

Thanks

Next Topic is Analyze of Algorithms

HNDIT3032 Data Structures and Algorithms



Week 11-Analyze Algorithm



Definition of Algorithm

- The word **Algorithm** means " A set of finite rules or instructions to be followed in calculations or other problem-solving operations "
- Or
- " A procedure for solving a mathematical problem in a finite number of steps that frequently involves recursive operations".

Definition of Algorithm

- An Algorithm is a set of instructions to perform a task or to solve a given problems.

Why need Algorithm Analysis

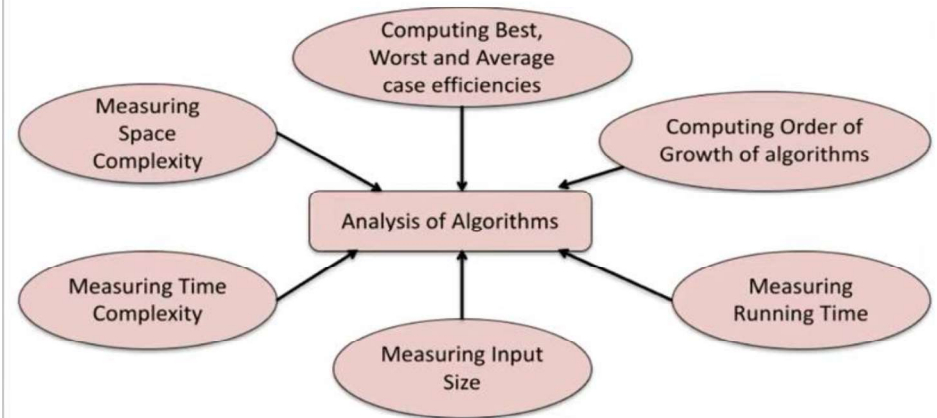
- Generally there are multiple approaches/method/algorithms to solve one problems statement. Algorithm analysis is performed to figure out which is the better /optimum approaches/method/algorithms out of this options.

What is better algorithm

- Faster (Less execution Time)-time complexity
- Less Memory (Space Complexity)
- Easy to read
- Less line of code
- Less H/W and S/W need

Analysis Framework

- It is a systematic approach applied for analyzing any given algorithm.



Analyzes of Algorithm

- Analyzes of Algorithm deals in finding best algorithm which runs fast and take in less memory.
- Example-Fine sum of first n Natural Numbers.
n=6 output is 21
n=7 output is 28
Two programmer Nimal and Sunil

Analyzes of Algorithm

- Nimal

```
Public int findSum(int n){  
Return n*(n+1)/2;}
```

- Sunil

```
Public int findSum(int n){  
Int sum=0;  
For(int i=1;i<=n;i++)  
{  
Sum=sum+i;}  
Return sum;}
```


How to Determine Best Algorithm

1. Time Complexity

2. Space Complexity

Time Complexity

- It is amount of time taken by algorithm to run
- The input processed by an algorithm helps in determine the time complexity.
- Consider Nimal and Sunil Program Sunil Program will take a time according to the input size n.

```

1 package com.hubberspot.algorithms.analysis;
2
3 public class TimeComplexityDemo {
4
5     public static void main(String[] args) {
6         double now = System.currentTimeMillis();
7
8         TimeComplexityDemo demo = new TimeComplexityDemo();
9         System.out.println(demo.findSum(99999));
10
11         System.out.println("Time taken - " + (System.currentTimeMillis() - now) + " millisecs.");
12     }
13
14     public int findSum(int n) {
15         return n * (n + 1) / 2;
16     }
17
18     // public int findSum(int n) {
19     //     int sum = 0;
20     //     for(int i = 1; i <= n; i++) {
21     //         sum = sum + i;
22     //     }
23     //     return sum;
24     // }

```

```

1 package com.hubberspot.algorithms.analysis;
2
3 public class TimeComplexityDemo {
4
5     public static void main(String[] args) {
6         double now = System.currentTimeMillis();
7
8         TimeComplexityDemo demo = new TimeComplexityDemo();
9         System.out.println(demo.findSum(99999));
10
11         System.out.println("Time taken - " + (System.currentTimeMillis() - now) + " millisecs.");
12     }
13
14     // public int findSum(int n) {
15     //     return n * (n + 1) / 2;
16     // }
17
18     public int findSum(int n) {
19         int sum = 0;
20         for(int i = 1; i <= n; i++) {
21             sum = sum + i;
22         }
23         return sum;
24     }
25 }

```

Space Complexity

- It is amount of memory or space taken by algorithm to run.
- The memory required to process the input by an algorithm helps in determining the space complexity.
- Consider Nimal and Sunil Program Sunil Program will take a more space according to the defined variables.

- For any algorithm, memory is required for the following purposes
 - To store program instructions
 - To store constant values
 - To store variable values
 - And for few other things like function calls, jumping statement etc
- Auxiliary Space: is the temporary space (excluding input size) allocated by your algorithm to solve the problem, with respect to input size.
- Space complexity includes both Auxiliary space and space used by input

Fundamentals of the Analysis of Algorithm efficiency

- Already discussed efficiency of algorithm depends on time and space. the algorithm efficiency can be analyzed by the following ways.
- Asymptotic Notations and its properties.

Asymptotic Analysis

- Asymptotic analysis helps in evaluating performance of an algorithm in terms of input size and its increase.
- Using Asymptotic analysis we do not measure actual running time of algorithm.
- It helps in determining how time and space taken by algorithm increases with size.

Asymptotic Notations

- Asymptotic Notations are the mathematical tools used to describe the running time of an algorithm in terms of input size.
- Example –performance of vehicle in one litre of petrol
Highway (No Traffic)-25km/litre.
city(max traffic) -15km/litre
City+highway(average traffic)-20km/litre

Asymptotic Notations help us in determining

1. Best Case
2. Average Case
3. Worst Case

Types of Asymptotic Notations

- Omega(Ω) Notation
- Big O (O) Notation
- Theta (Θ) Notation

Omega(Ω) Notation

- Lower bound of an algorithm's on Running time.
- Lower bound means for any given input this notation determines best amount of time an algorithm can take to complete.

For example –

-If we say certain algorithm takes 100 secs as best amount of time. so 100 secs will be lower bound of that algorithm . the algorithm can take more than 100 secs but it will not take less than 100 secs

Big O (O) Notation

- Upper bound of an algorithm's running time.
- Upper bound means for any given input this notation determines longest amount of time an algorithm can take to complete.

For example

– If we say certain algorithm takes 100 secs as longest amount of time. so 100 secs will be upper bound of that algorithm . the algorithm can take less than 100 secs but it will not take more than 100 secs

Theta (Θ) Notation

- Upper and Lower bound of an algorithm's running time.
- By Lower and Upper bound means for any given input this notation determines average amount of time an algorithm can take to complete.
- For example –
 - If we run certain algorithm and it takes 100 secs for first run, 120 secs for second run, 110 for third run and so on. so theta notation gives an average of running time of that algorithm

Rules of Big O (O) Notation

- Running PC it has single processor assume
- It performs sequential Execution of statements
- Assignment operation takes 1 unit of time
- Return statement takes 1 unit of time
- Arithmetical operation takes 1 unit of time
- logical operation takes 1 unit of time
- Drop lower order terms $T = n^4 + 3n + 4$
Big O (O) ---- $O(n^4)$
- Drop constant multiples $T = 3n^4 + 3n + 4$
Big O (O) ---- $O(n^4)$

Calculating Time complexity of a Constant Algorithm (Big O)

1	public int sum(int x, int y) {	line no.	operations	unit time
2	int result = x + y;	2	1+1+1+1	4
3	return result;	3	1+1	2
4	}			

$$T = 4 + 2 = 6$$

$$T \approx C \text{ (constant)}$$

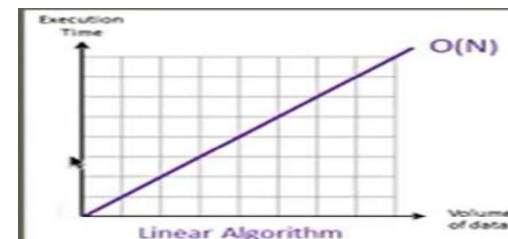
Calculating Time complexity of a Linear Algorithm (Big O)

1	public void findSum(int n) {	line no.	operations	unit time
2	int sum = 0; // 1 step	2	1	1
3	for(int i = 1; i <= n; i++) {	3	1 + 3n + 3 + 3n	6n + 4
4	sum = sum + i; // n steps	4	n(1 + 1 + 1 + 1)	4n
5	}	6	1 + 1	2
6	return sum; // 1 step			
7	}			

$$T = 1 + 6n + 4 + 4n + 2$$

$$T = 10n + 7$$

$$\text{Time Complexity} = O(n)$$



Calculating Time complexity of a Polynomial Algorithm (Big O)

```

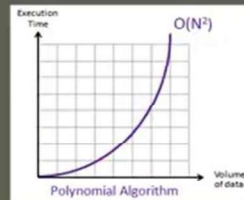
1 public void print(int n) {
2     for(int i = 1; i <= n; i++) {
3         for(int j = 1; j <= n; j++) {
4             s.o.p("i = " + i + ", j = " + j);
5         }
6         s.o.p("End of inner loop");
7     }
8     s.o.p("End of outer loop");
9 }
    
```

line no.	operations	unit time
2	$1 + 3n + 3 + 3n$	$6n + 4$
3	$n(1 + 3n + 3 + 3n)$	$6n^2 + 4n$
4	$n^2(1 + 1 + 1)$	$3n^2$
6	$n(1)$	n
8	1	1

$$T = 6n + 4 + 6n^2 + 4n + 3n^2 + n + 1$$

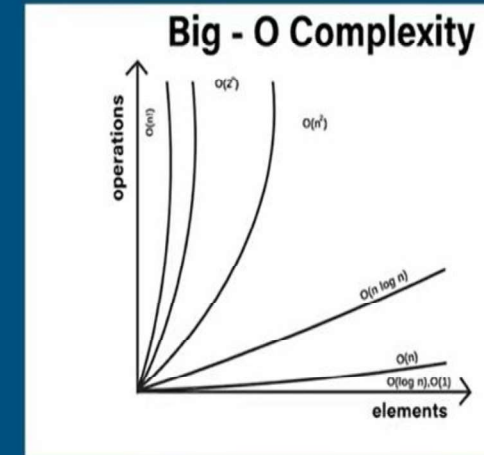
$$T = 9n^2 + 11n + 5$$

Time Complexity = $O(n^2)$



Big O Notation

- $O(1)$: Constant Time
- $O(\log n)$: Logarithmic Time
- $O(n)$: Linear Time
- $O(n \log n)$: Linearithmic Time
- $O(n^2)$: Quadratic Time
- $O(2^n)$: Exponential Time
- $O(n!)$: Factorial Time



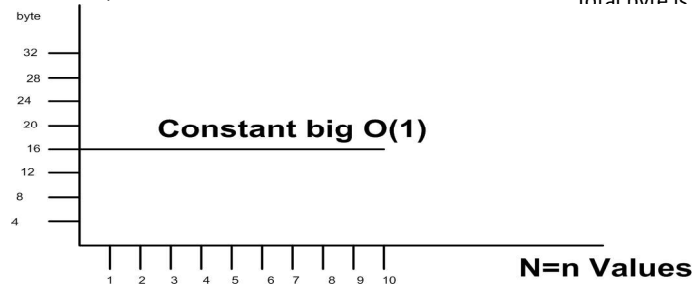
Space Complexity calculate

space complexity= input size+ auxiliary space

```

public int add(int n1,int n2) {
    int sum =n1+n2;
    return sum;
}
    
```

$n1=4\text{byte}$
 $n2=4\text{byte}$
 $\text{sum}=4\text{ byte}$
 Auxiliary Space for return 4byte
 constant
 Total byte is 16



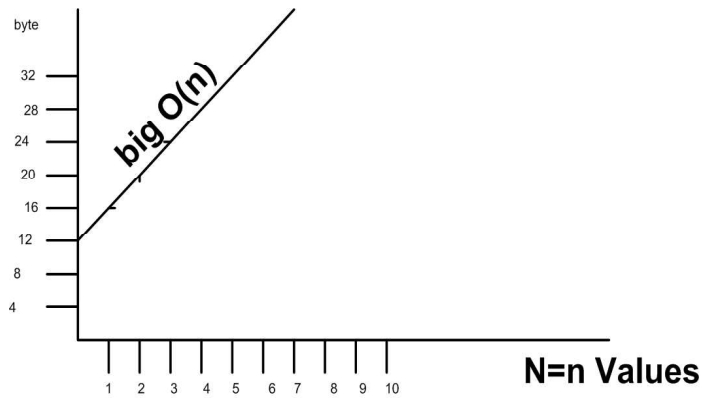
Line Space complexity

- space complexity= input size+ auxiliary space

```

1 public void sumofNo(int x[],int n)
2 {int sum =0;
3 for(int i=0;i<n;i++)
4 {
5     sum=sum+x[i];
6 }
7 System.out.println(sum);
8 }
    
```

Array $n*4$ bytes
 Sum 4 byte
 i 4 byte
 Aux 4 bytes
 Total $4n+12=4n+c$
 Big $O(n)$

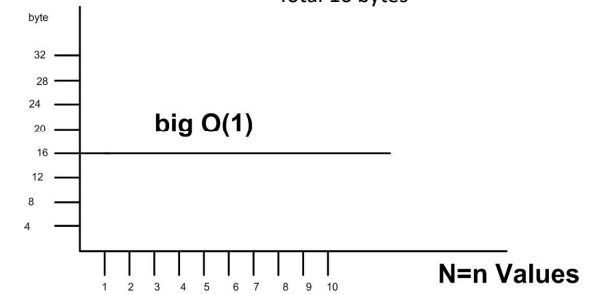


Integrative space complexity

- space complexity= input size+ auxiliary space

```
public void fact(int n)
{
    int fac=1;
    for(int i=1;i<=n;i++)
    {
        fac=fac*i;
    }
    System.out.println(fac);
}
```

Assume n=5
 Fact---4 byte
 n ---4 byte
 i 4 byte
 Aux 4 byte
 Total 16 bytes



Recursive space complexity

- space complexity= input size+ auxiliary space

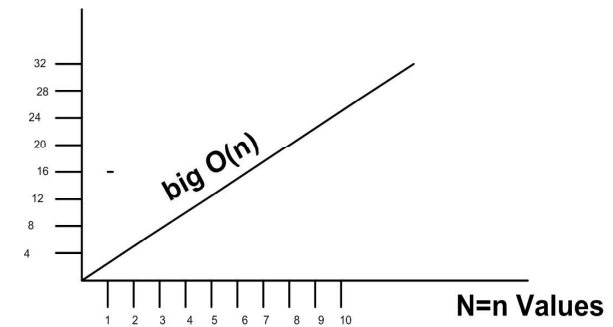
```
public int fact(int n)
{
    if(n<=1)
    return 1;
    else
    return n*fact(n-1);
}
```

Assume n=5
 n take 4 bytes
 Function take 4 bytes one time function
 work two time .it take 8 byte . but it call n
 times .this mean 8n byte taken.
 Total bytes =input size+aux
 T=4+8n
 Big O(n)

1	
2*f(1)	2
3*f(2)	3*2
4*f(3)	6*4
5*f(4)	24*5

Take fixed size bytes

Recursive space complexity



Thanks

HNDIT3032 Data Structures and Algorithms



Week 12/13-Searching Algorithm



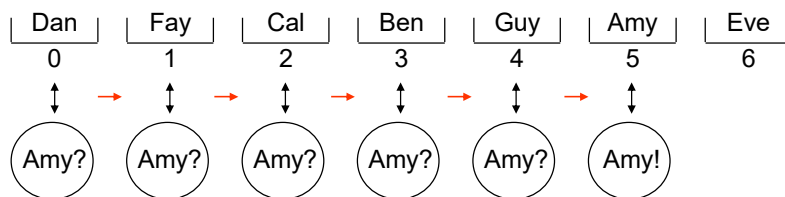
Sequential Search (cont'd)

Pseudo code:
 public int sequentialSearch(a[],n,t)
 for i = 0 to n-1
 if (a[i]=t)
 return i;
 next i
 return -1;

For primitive data types it is
if (value == arr [i])

Sequential Search

- Examines each element of a list in sequence until it finds the target value or reaches the end of the list.



Sequential Search (cont'd)

- The average number of comparisons (assuming the target value is equal to one of the elements of the array, randomly chosen) is about $n / 2$ (where $n = \text{arr.length}$).
- Worst case: n comparisons.
- Also n comparisons are needed to establish that the target value is not in the array.
- We say that this is an $O(n)$ (order of n) algorithm.

Efficiency of a Sequential Search

- Best case $O(1)$
 - Locate desired item first
- Worst case $O(n)$
 - Must look at all the items
- Average case $O(n)$
 - Must look at half the items
 - $O(n/2)$ is still $O(n)$

5

```
public class Seqsearch {
    static int sequentialSearch(int a[], int n, int t)
    {
        int i;
        for (i = 0; i < n; i++)
            if (a[i]==t) return i;
        return (-1);
    }

    public static void main(String a[])
    {
        int num[] = {4, 65, 2, -31, 0, 99, 2, 83, 782, 1};
        int n=num.length;
        int t=99;
        if(sequentialSearch(num,n,t)==-1)
            {System.out.println("Can not Fine Value in Array");
            }
        else
            {System.out.println("Fine Value in Array "+sequentialSearch(num,n,t));
            }
    }
}
```

Searching a Sorted Array

- A sequential search can be more efficient if the data is sorted



6

Binary search algorithm

- An algorithm that searches for a value in a sorted list by repeatedly eliminating half the list from consideration.
 - Can be written iteratively or recursively
 - implemented in Java as method `Arrays.binarySearch` in `java.util` package
- The elements of the list must be arranged in ascending (or descending) order.
- The target value is always compared with the middle element of the remaining search range.
- We must have random access to the elements of the list (an array or ArrayList are OK).

8

Binary search pseudocode

binary search array a for value i :

- if all elements have been searched,
result is -1.
- examine middle element $a[mid]$.
- if $a[mid]$ equals i ,
result is mid .
- if $a[mid]$ is greater than i ,
binary search lower half of a for i .
- if $a[mid]$ is less than i ,
binary search upper half of a for i .

9

Binary search example

searching for value 16

0	4	← min
1	7	
2	16	
3	20	← mid (too big!)
4	37	
5	38	
6	43	← max

11

Binary Search of Sorted Array

- Algorithm for a binary search

```
Algorithm binarySearch(a, first, last, desiredItem)
mid = (first + last)/2 // approximate midpoint
if (first > last)
    return false
else if (desiredItem equals a[mid])
    return true
else if (desiredItem < a[mid])
    return binarySearch(a, first, mid-1, desiredItem)
else // desiredItem > a[mid]
    return binarySearch(a, mid+1, last, desiredItem)
```

10

Binary search example

searching for value 16

0	4	← min
1	7	← mid (too small!)
2	16	← max
3	20	
4	37	
5	38	
6	43	

12

Binary search example

searching for value 16

0	4	
1	7	
2	16	Min,max,mid (found it!)
3	20	
4	37	
5	38	
6	43	

13

Binary Search (cont'd)

- Iterative implementation:

```
public int binarySearch (int [ ] arr, int value, int left, int right)
{
    while (left <= right)
    {
        int middle = (left + right) / 2;
        if ( value == arr [middle] )
            return middle;
        else if ( value < arr[middle] )
            right = middle - 1;
        else // if ( value > arr[middle] )
            left = middle + 1;
    }
    return -1; // Not found
}
```

Binary Search (cont'd)

- Recursive implementation:

```
public int binarySearch (int [ ] arr, int value, int left, int right)
{
    if (right < left)
        return -1; // Not found

    int middle = (left + right) / 2;
    if (value == arr [middle] )
        return middle;
    else if (value < arr[middle])
        return binarySearch (arr, value, left, middle - 1);
    else // if ( value > arr[middle])
        return binarySearch (arr, value, middle + 1, right);
}
```

Binary Search (cont'd)

- We say that this is an $O(\log n)$ algorithm.

Efficiency of a Binary Search

- Best case $O(1)$
 - Locate desired item first
- Worst case $O(\log n)$
 - Must look at all the items
- Average case $O(\log n)$

Choosing a Search Method

	Best Case	Average Case	Worst Case
Sequential search (unsorted data)	$O(1)$	$O(n)$	$O(n)$
Sequential search (sorted data)	$O(1)$	$O(n)$	$O(n)$
Binary Search (sorted array)	$O(1)$	$O(\log n)$	$O(\log n)$

- Iterative search saves time

17

- Thanks –Next Topic Sorting

Efficiency of the Search Algorithms (Best, Worst and Average Cases):

Searching Technique	Best case	Average Case	Worst Case
Sequential Search	$O(1)$	$O(n)$	$O(n)$
Binary Search	$O(1)$	$O(\log n)$	$O(\log n)$

The difference between $O(\log(N))$ and $O(N)$ is extremely significant when N is large.
 For example, suppose your array contains 2 billion values, the sequential search would involve about a billion comparisons; binary search would require only 32 comparisons!

HNDIT3032 Data Structures and Algorithms



Week 14/15-Sorting Algorithm



Some important factors that must be considered are:

- **Speed** : the simplest algorithms are $O(N^2)$ while more advanced ones are $O(N \log N)$. No algorithm can make less than $O(N \log N)$ comparisons between keys.
- **Storage** : algorithms that sort in place are the best, needing memory $O(N)$. Those are using linked list representation need extra N words of memory for references and those that work on a copy of the file needed $O(2N)$.

Sorting Methods

- Arranging things into ascending or descending order is called sorting.
i.e. Sorting is the process of arranging items in order.

- **Simplicity** : the simpler algorithms are often easiest to implement and often perform more sophisticated ones for small problem.
- **Stability** : an algorithm is stable if it preserves the relative order of records with equal keys. there are ways to convert unstable implementation into stable ones.

Sorting Techniques

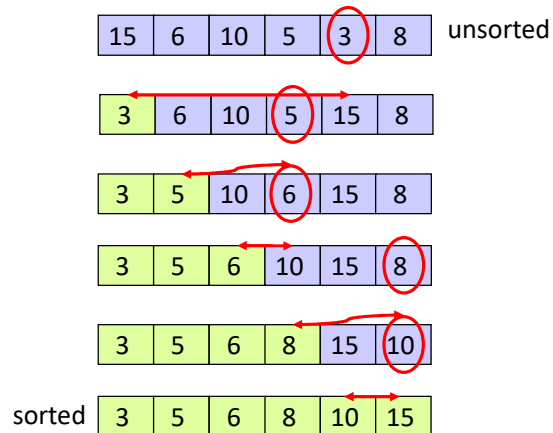
- There are three major sorting techniques. Such as
 1. Sorting by selection
 2. Sorting by insertion
 3. Sorting by exchange

Selection Sort

- In terms of an array A, the selection sort finds the smallest element in the array and exchanges it with A[0]. Then, ignoring A[0], the sort finds the next smallest and swaps it with A[1] and so on.

Selection Sort

- An example of selection sort



Selection Sort Algorithm:

Here we repeatedly find the next largest (or smallest) element in the array and move it to its final position in the sorted array.

Example: Sort the numbers 6, 7, 72, 4, 32, 65, 9, 56 using selection sort.

	0	1	2	3	4	5	6	7	
Pass0	6	7	72	4	32	65	9	56	Original
Pass1	4	7	72	6	32	65	9	56	
Pass2	4	6	72	7	32	65	9	56	
Pass3	4	6	7	72	32	65	9	56	
Pass4	4	6	7	9	32	65	72	56	
Pass5	4	6	7	9	32	65	72	56	
Pass6	4	6	7	9	32	56	72	65	
Pass7	4	6	7	9	32	56	65	72	Sorted

Iterative Selection Sort

- Iterative selection sort algorithm

//Sort the first n elements of an array

Input: array A, int n

Output:

```
selectionSort1(A, n){
    for(index = 0; index<n-1; index++){
        indexOfNextSmallest = the index of the smallest value
        among A[index], A[index+1], ...A[n-1]
        interchange the value of A[index] and A[indexOfNextSmallest]
    }
}
```

9

1.Iterative Selection Sort

- Implementing iterative selection sort in Java

```
public static void selectionSort1(int[] A, int n){
    for(int i = 0; i<n-1; i++){
        int indexOfNextSmallest = i;
        int smallest = A[i];
        for(int j=i+1; j<n; j++){
            if(smallest>A[j]){
                indexOfNextSmallest=j;
                smallest=A[j];
            }
        }
        A[indexOfNextSmallest]=A[i];
        A[i]=smallest;
    }
}
```

10

2. Selection Sort

- Implementing iterative selection sort in Java

```
public void selectionSort()
{
    int out, in, min;
    for(out=0; out<nElems-1; out++) // outer loop
    {
        min = out; // minimum
        for(in=out+1; in<nElems; in++) // inner loop
        if(a[in] < a[min] ) // if min greater,
        min = in; // we have a new min
        swap(out, min); // swap them
    } // end for(out)
} // end selectionSort()
```

Swap Method:

```
private void swap(int one, int two)
{
    long temp = a[one];
    a[one] = a[two];
    a[two] = temp;
}
```


Implementing the Java Programme

```
class ArraySel
{
private long[] a; // ref to array a
private int nElems; // number of data items
//-----
-
public ArraySel(int max) // constructor
{
a = new long[max]; // create the array
nElems = 0; // no items yet
}
//-----
```

```
public void insert(long value) // put element into array
{
a[nElems] = value; // insert it
nElems++; // increment size
}
//-----
public void display() // displays array contents
{
for(int j=0; j<nElems; j++) // for each element,
System.out.print(a[j] + " "); // display it
System.out.println("");
}
//-----
```

```
public void selectionSort()
{
int out, in, min;
for(out=0; out<nElems-1; out++) // outer loop
{
min = out; // minimum
for(in=out+1; in<nElems; in++) // inner loop
if(a[in] < a[min] ) // if min greater,
min = in; // we have a new min
swap(out, min); // swap them
} // end for(out)
} // end selectionSort()
```

```
private void swap(int one, int two)
{
long temp = a[one];
a[one] = a[two];
a[two] = temp;
}
//-----
----
} // end class ArraySel
////////////////////////////////////
////////////////////////////////////
```



```
class SelectSortApp
{
    public static void main(String[] args)
    {
        int maxSize = 100; // array size
        ArraySel arr; // reference to array
        arr = new ArraySel(maxSize); // create the array
        arr.insert(77); // insert 10 items
        arr.insert(99);
        arr.insert(44);
        arr.insert(55);
        arr.insert(22);
        arr.insert(88);
        arr.insert(11);
        arr.insert(00);
        arr.insert(66);
        arr.insert(33);
```

```
        arr.display(); // display items
        arr.selectionSort(); // selection-sort them
        arr.display(); // display them again
    } // end main()
} // end class Select
```

Running Time Analysis

- Selection sort is $O(n^2)$ regardless of the initial order of the elements in an array.

Insertion Sort

- An insertion sort algorithm starts by considering the first two element of the data array.
- Which is data[0] and data[1]. If they are out of order, then shift data[0] by one position and data[1] can insert into empty position.
- Then consider the third element ,(ie data[2]). If this element is smaller than data[1] and data[0] shift data[0] and data[1] by one position and insert data[2] into that location and so-on

Insertion Sort

- An insertion sort of an array partitions the array into two parts.
 - One part is sorted and initially contains just the first element in the array.
 - The second part contains the remaining elements.
- The sort inserts one by one the elements in the unsorted part of the array into their proper location within the sorted part of the array.

21

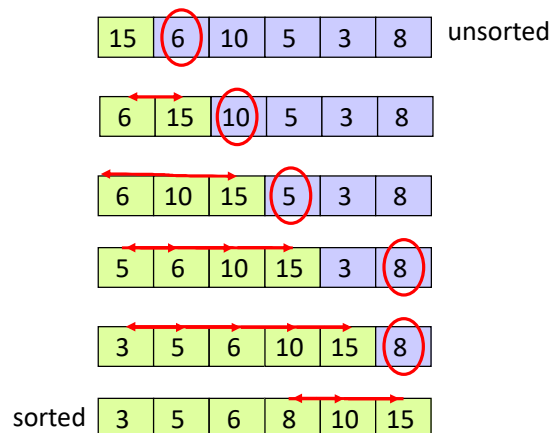
Analysis of Insertion Sort

- Maximum number of comparisons is $O(n^2)$
- In the best case, number of comparisons is $O(n)$
- The number of shifts performed during an insertion is one less than the number of comparisons or, when the new value is the smallest so far, the same as the number of comparisons
- A shift in an insertion sort requires the movement of only one item whereas in a bubble or selection sort an exchange involves a temporary item and requires the movement of three items

22

Insertion Sort

- An example of Insertion sort



23

Pseudo code algorithm for insertion sort

```
Insertionsort (data[]){  
  for (i=0;i<data.length-1;i++)  
    temp=data[i];  
    Move all elements data[j] greater than temp by  
    one position;  
    Place temp in its proper position;
```

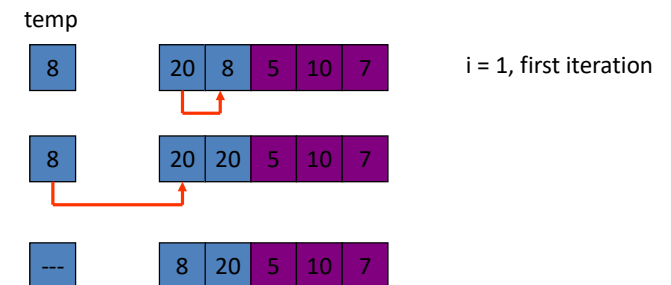
Each element $data[i]$ is inserted into its proper position j such that $0 \leq j \leq i$

Insertion Sort (cont'd)

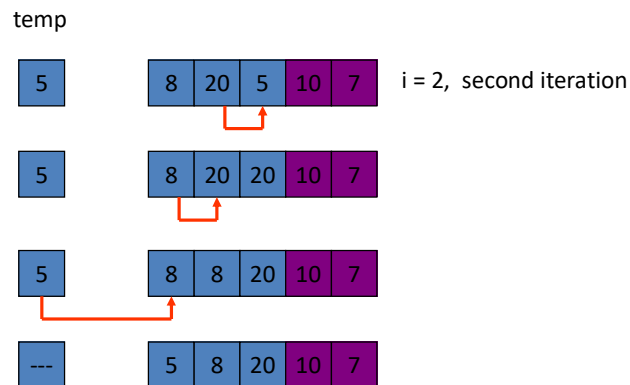
```
public void insertionSort (double [ ] arr, int n)
{
    for (int k = 1 ; k < n; k++)
    {
        double temp = arr [ k ];
        int i = k;
        while (i > 0 && arr [i-1] > temp)
        {
            arr [i] = arr [i - 1];
            i --;
        }
        arr [i] = temp;
    }
}
```

shift to the
right

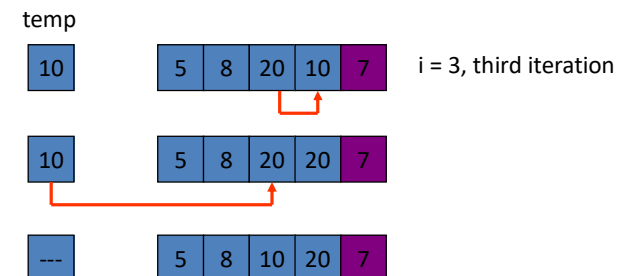
Insert Action: i=1



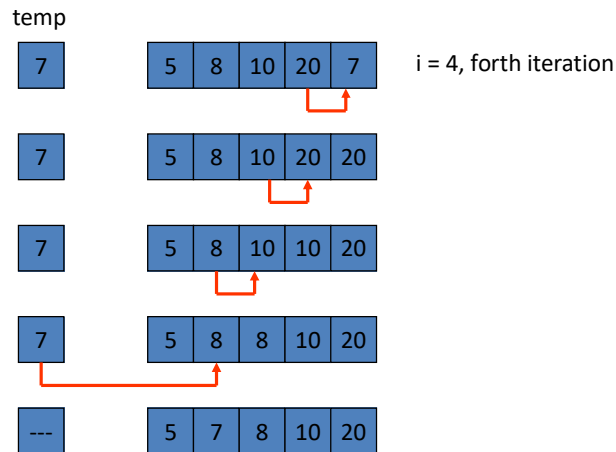
Insert Action: i=2



Insert Action: i=3



Insert Action: i=4



Insertion Sort

Running time analysis

- Insertion sort is at best $O(n)$ and at worst $O(n^2)$.
- The closer an array is to sorted order, the less work an insertion sort does

30

Comparison of Quadratic Sorts

- None of the algorithms are particularly good for large arrays

Comparison of Quadratic Sorts

	Number of Comparisons		Number of Exchanges	
	Best	Worst	Best	Worst
Selection sort	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n)$
Bubble sort	$O(n)$	$O(n^2)$	$O(1)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n)$	$O(n^2)$

Comparison of Sort Algorithms

	Number of Comparisons		
	Best	Average	Worst
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Shell sort	$O(n^{7/6})$	$O(n^{5/4})$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

31

Sorting by Exchange

- Eg : Bubble sort
- One of the simplest sorting is algorithm is known as bubble sort. The algorithm as follows.
- Beginning at the last element in the list
- Compare each element with the previous element in the list. If an element is less than its predecessor, swap these two element.
- To completely sort the list, you need to perform this process $n-1$ times on a list of length n

BubbleSort

- Most frequently used sorting algorithm
- Algorithm:
 - for $j=n-1$ to 1 $O(n)$
 - for $i=0$ to j $O(j)$
 - if $A[i]$ and $A[i+1]$ are out of order, swap them
(that's the bubble) $O(1)$
- Analysis
 - Bubblesort is $O(n^2)$
- Appropriate for small arrays
- Appropriate for nearly sorted arrays

- Eg :
- (a) Run through the bubble sort algorithm by hand on the list :44,55,12,42,94,18,06,67
- (b) Write a Java program to implementing the bubble sort algorithm on an array.
- The basic idea underlying the bubble sort is to pass through the file sequentially several times.
- Each pass consists of comparing each element in the file with its predecessor $[x[j]]$ and $x[j-1]$ and interchanging the two elements if they are not proper order.

- The following comparisons are made on the first pass
- 44,55,12,42,94,18,06,67
- Data[7] with Data[6] (67,06) No interchange
- Data[6] with Data[5] (06,18) interchange
- Data[5] with Data[4] (06,94) interchange
- Data[4] with Data[3] (06,42) interchange
- Data[3] with Data[2] (06,12) interchange
- Data[2] with Data[1] (06,55) interchange
- Data[1] with Data[0] (06,44) interchange
- after the first pass, the smallest element is in its proper positions.
- **06,44,55,12,42,94,18,67**

- Original -> 44,55,12,42,94,18,06,67
- Pass 1 -> 06,44,55,12,42,94,18,67
- Pass 2 -> 06,12,44,55,18,42,94,67
- Pass 3 -> 06,12,18,44,55,42,67,94
- Pass 4 -> 06,12,18,42,44,55,67,94
- Pass 5 -> 06,12,18,42,44,55,67,94
- Pass 6 -> 06,12,18,42,44,55,67,94
- Pass 7 -> 06,12,18,42,44,55,67,94
- Sorted file->06,12,18,42,44,55,67,94

Pseudo code Algorithm

- bubblesort(data[])
- for (i=0,i<data.length-1;i++)
- for (j=data.length-1;j>i;- - j)
- swap elements in positions j and j-1 if they out of order;

```
public void bubbleSort()
{
    int out, in;
    for(out=nElems-1; out>1; out--) { // outer loop (backward)
        for(in=0; in<out; in++) { // inner loop (forward)
            if( a[in] > a[in+1] ) // out of order?
                swap(in, in+1); // swap them
        }
    }
} // end bubbleSort()

private void swap(int one, int two)
{
    long temp = a[one];
    a[one] = a[two];
    a[two] = temp;
}
```

```
public void bubbleSort()
{
    int out, in;
    for(out=nElems-1; out>1; out--) // outer loop (backward)
    {
        for(in=0; in<out; in++) // inner loop (forward)
        {
            if( a[in] > a[in+1] ) // out of order?
                long temp = a[in]; // swap them
                a[in] = a[in+1];
                a[in+1] = temp;
        }
    }
} // end bubbleSort()
```

Summary

- “Bubble Up” algorithm will **move largest value to its correct location** (to the right)
- Repeat “Bubble Up” until all elements are correctly placed:
 - **Maximum of N-1 times**
 - Can finish early if **no swapping** occurs
- We reduce the number of elements we compare each time one is correctly placed

Time complexity of bubble sort

- The number of comparison is same in each case (best, average and worst case) and equals the total number of iterations of the inner for loops:

$$\sum_{i=0}^{n-2} (n-1-i) = n(n-1)/2 = O(n^2)$$

Bubble Sort Algorithm:

Here we repeatedly move the largest element to the highest index position of the array.

Example: Sort the numbers 6, 7, 72, 4, 32, 65, 9, 56 using bubble sort.

	0	1	2	3	4	5	6	7	
Pass0	6	7	72	4	32	65	9	56	Original
Pass1	6	7	4	32	65	9	56	72	
Pass2	6	4	7	32	9	56	65	72	
Pass3	4	6	7	9	32	56	65	72	
Pass4	4	6	7	9	32	56	65	72	
Pass5	4	6	7	9	32	56	65	72	Sorted

Selection sort algorithm than bubble sort

Sorting Algorithm	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$
Selection sort algorithm	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(1)$

- Bubble sort algorithm is considered to be the most simple and inefficient algorithm, but selection sort algorithm is efficient as compared to bubble sort. Bubble sort also consumes additional space for storing temporary variable and needs more swaps.

Efficiency of the Sort Algorithms (Best, Worst and Average Case Comparison):

Name	Best	Average	Worst
Quicksort	$n \log n$	$n \log n$	n^2
Merge sort	$n \log n$	$n \log n$	$n \log n$
In-place merge sort	—	—	$n (\log n)^2$
Heapsort	$n \log n$	$n \log n$	$n \log n$
Insertion sort	n	n^2	n^2
Introsort	$n \log n$	$n \log n$	$n \log n$
Selection sort	n^2	n^2	n^2
Timsort	n	$n \log n$	$n \log n$
Shell sort	n	$n (\log n)^2$ or $n^{3/2}$	Depends on gap sequence; best known is $n (\log n)^2$
Bubble sort	n	n^2	n^2
Binary tree sort	n	$n \log n$	$n \log n$

- THANKS