BIRZEIT UNIVERSITY

COMP232
Data Structure

Lectures Note

Prepared by: **Dr. Mamoun Nawahdah**

2015

Math Review

$$
\begin{aligned}
&\textbf{1. } \log(nm) = \log n + \log m. \\
&\textbf{2. } \log(n/m) = \log n - \log m. \\
&\textbf{3. } \log(n^r) = r \log n. \\
&\textbf{4. } \log_a n = \log_b n / \log_b a.
\end{aligned}
$$

$$
\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.
$$

$$
\sum_{i=1}^{n} i^2 = \frac{2n^3 + 3n^2 + n}{6} = \frac{n(2n+1)(n+1)}{6}.
$$

$$
\sum_{i=1}^{\log n} n = n \log n.
$$

$$
\sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a - 1} \text{ for } a \neq 1.
$$

$$
\sum_{i=1}^{n} \frac{1}{2^i} = 1 - \frac{1}{2^n},
$$

and

$$
\sum_{i=0}^{n} 2^i = 2^{n+1} - 1.
$$

$$
\sum_{i=0}^{\log n} 2^i = 2^{\log n + 1} - 1 = 2n - 1.
$$

Finally,

$$
\sum_{i=1}^{n} \frac{i}{2^i} = 2 - \frac{n+2}{2^n}.
$$

Table of Contents

(Lecture 3) What is an Algorithm?

Definition:
- An algorithm is a way of solving WELL-SPECIFIED computational problems. *Cormen et al.*
- A finite set of rules that give a sequence of operations for solving a specific type of problem - *Knuth*
- **Algorithm** is a finite list of well-defined instructions for accomplishing some task that, given an initial state, will terminate in a defined end-state.

Euclid's Algorithm   (300BC)
- Used to find Greatest common divisor (**GCD**) of two positive integers.
- GCD of two numbers, the largest number that divides both of them without leaving a remainder.

Euclid's Algorithm:
   o Consider two positive integers '**m**' and '**n**', such that **m>n**
   o **Step1**: Divide **m** by **n**, and let the reminder be **r**.
   o **Step2**: if **r=0**, the algorithm ends, **n** is the GCD.
   o **Step3**: Set, **m→n**, **n→r** , go back to **step 1** .

Implement this iteratively and recursively

Why Algorithms?

   o Gives an idea (estimate) of running time.
   o Help us decide on hardware requirements.
   o What is feasible vs. what is impossible.
   o Improvement is a never ending process.

Correctness of an Algorithm

Must be proved (mathematically)

   Step1: statement to be proven.

   Step2: List all assumptions.

   Step3: Chain of reasoning from assumptions to the statement.

Another way is to check for **incorrectness** of an algorithm.

   Step1: give a set of data for which the algorithm does not work.

   Step2: usually consider small data sets.

   Step3: Especially consider borderline cases.

Analysis of Algorithms

Once an algorithm is given for a problem and decided (somehow) to be correct, an important step is to determine **how much in the way of resources**, such as **time** or **space**, the algorithm will require.

- **Space Complexity** ➔ memory and storage is very cheap nowadays. ✖
- **Time Complexity** ✓ Different platforms ➔ different time. Absolute time is hard to measure as it depends on many factors.

Example: moving between university buildings: it depends on who are walking, which way he/she use, etc. time is not good measurement. Number of steps is a better one.

Example:

- Consider the problem of summing
$$\sum_{k=1}^{n} k = 1 + 2 + 3 + ... + n$$

Come up with an algorithm to solve this problem.

Algorithm A	Algorithm B	Algorithm C

sum = 0
for i = 1 *to* n
  sum = sum + i
	sum = 0
	for i = 1 *to* n
	{
	    for j = 1 *to* i
	      sum = sum + 1
	}
		sum = n * (n + 1) / 2

Counting Basic Operations

- A **basic operation** of an algorithm is the most significant contributor to its total time requirement.

	Algorithm A	Algorithm B	Algorithm C
Additions	n	$n(n+1)/2$	1
Multiplications			1
Divisions			1
Total basic operations	n	$(n^2 + n)/2$	3

(Lecture 4) Analysis of Algorithms

- **Space Complexity** ✗
- **Time Complexity** ✓

How to calculate the time complexity?

- Measure execution time. ✗ Algorithm for small data size will take small time comparing to a large data.
- Calculate time required for an algorithm in terms of the size of input data. ✗ Does not work as the same algorithm over the same data will not take the same time.

<mark>Run summing code 2 times and compare time</mark>

- Determine order of **growth** of an algorithm with respect to the size of input data. ✓

Order of time or **growth of time**

<mark>Go back to summing result</mark>

```
      n,                    A,                   B,                   C
   ==========           ==========           ==========           ==========
        1 )                  7183,                7183,              820
       10 )                  2052,                4105,              102
      100 )  ,               7183,              155974,             1026
     1000 )  ,              66700,             2983004,             3079
    10000 )  ,             411484,           149256917,             2052
   100000 )  ,            1903500,         13209223813,             1027
```

Linear growth | Quadratic growth | Constant growth

In term of time complexity, we say that algorithm **C** is better than **A** and **B**

Types of Time Complexity

- Worst case analysis ✓
- Best case analysis   ✗
- Average case analysis ✗ too complex (statistical methods)

RAM model of computation

We assume that:

- We have infinite memory
- Each operation (+,-,*,/,=) takes 1 unit of time
- Each memory access takes 1 unit of time
- All data is in the RAM

6

Bubble sort

- **Rules**:
  - You can only pick one ball at a time.
  - Before picking up another ball, you have to drop the existing ball-in hand, in an empty basket.
  - You have to start from the left most basket and arrange the balls moving towards the right.
  - You can use a stick to keep track of the sorted part.

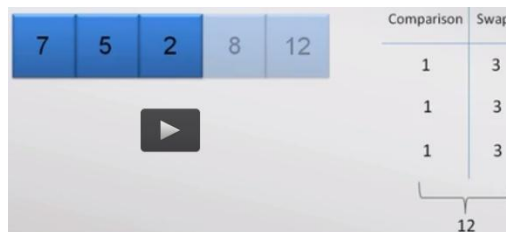Make a demo using the following data set

12	8	7	5	2

Worst case analysis

After 1st round:

After 2nd round:

For whole sorting algorithm: **16+12+8+4**   for a data size of 5 elements

= 4 (4 + 3 + 2 + 1)   = 4 (n-1 + n-2 + …. + 2 + 1) = 4 (n-1*n/2) =

2 * n * (n-1) ➔ **pn^2 + qn + r** ➔ **p**, **q**, and **r** are some constant.

<mark>Implement and test effectiveness of bubble sort algorithm</mark>

```
for(int i=0; i<n-1; i++){
    for(int j=0; j<n-1-i; j++){
        if(num[j+1] < num[j]){
            temp = num[j];
            num[j] = num[j+1];
            num[j+1] = temp;
        }
    }
}
```

$i = 0$	$0 \leq j < n-2$	$(n-1)$
$i = 1$	$0 \leq j < n-3$	$(n-2)$
$\vdots$	$\vdots$	$\vdots$
$i = n-2$	$0 \leq j < 0$	1

The Big O notation

Assume the order of time of an algorithm is a **quadratic** time as displayed in the graph. Our job is to find an **upper bond** for this function **T(n)**. Consider a function $c_1 n^2$ ← never over take **T(n)**

$C_2 n^2$ such that its greater than **T(n)** for **n>n₀** . in this case we say that $C_2 n^2$ is an upper bond of **T(n)**

<mark>But we can come up with many functions satisfy this condition. We need to be precise.</mark>
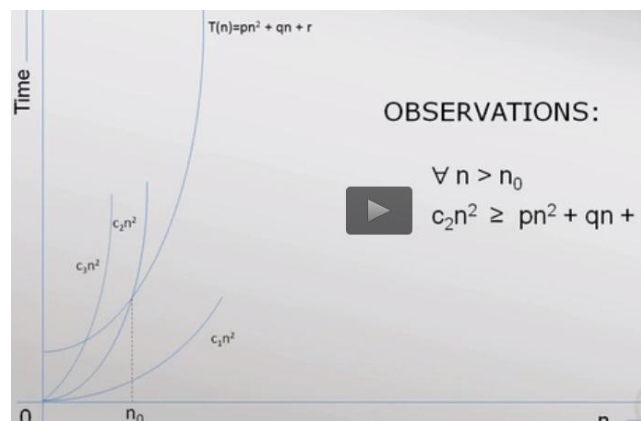
Big Oh **$O(n^2)$** : f(n): there exist positive constants **c** and **n₀** such that **0<= f(n) <= cn²** for all **n >= n₀**

In general

O(g(n)) : f(n): there exist positive constants **c** and **n₀** such that **0<= f(n) <= cg(n)**   for all **n >= n₀**

Example 1:

$5n^2 + 6 \in O(n^2)$??? ✔

Find cn^2 ➔   c=6 and n₀=3

➔   c=5.1 n0=8

Example 2:

$5n + 6 \in O(n^2)$??? ✔

Find cn^2 ➔   c=11 and n₀=1

8

Example 3:

$n^3 + 2n^2 + 4n + 8 \in O(n^2)$??? ✗

Find cn^2 $\geq n^3 + 2n^2 + 4n + 8$??? ✗

$$a_m n^m + a_{m-1}n^{m-1} - - - - - - - - - - + a_0 \in O(n^m)$$

$$\log n \leq \sqrt{n} \leq n \leq n\log n \leq n^2 \leq n^3 \leq 2^n \leq n!$$

What does it mean?

int [ ] a = { 1, 3, 7, 8, 9, 2}

| 1 | 3 | 7 | 8 | 9 | 2 |

a [4]

int [ ] b = { 5, 8, 1,..........25, 20 }100 Elements

| 5 | 8 | 1 | .... | 25 | 20 |

b[98]

Array element access: O(1) : Constant Time

| 1 | 12 | 7 | 8 | 9 |

$T_5 \sim 50ms$

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

Array element search: $T_{search} = O(n)$ $T_{10} \sim 100$ ms

| 2 | 5 | 7 | 8 | 9 | 10 | 12 |

A loop inside a loop in an algorithm usually represents a time complexity of $O(n^2)$
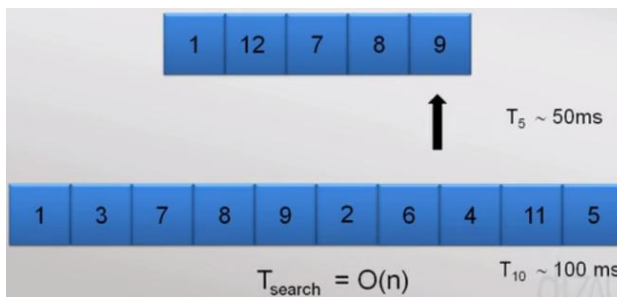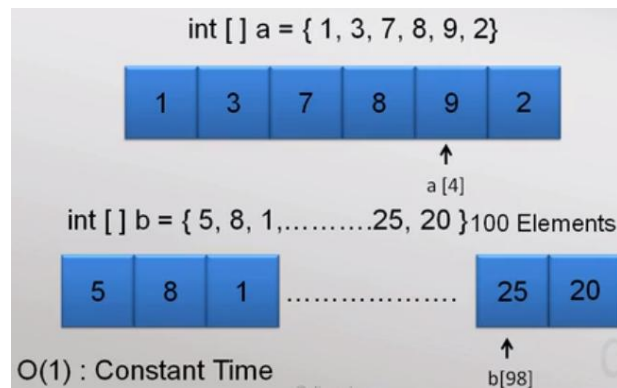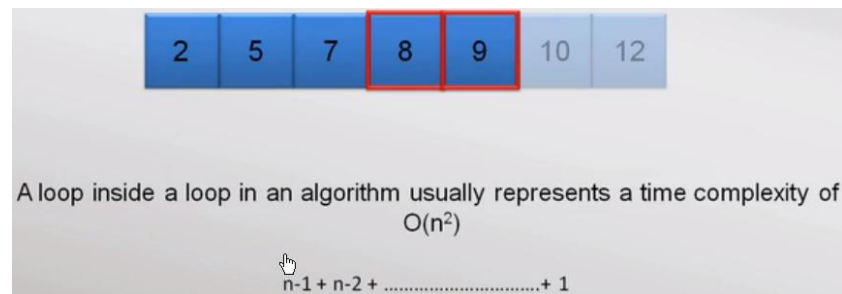
Bubble sort algorithm: n-1 + n-2 + ...+ 1

(Lecture 5) Asymptotic Analysis

Asymptotic analysis measures the efficiency of an algorithm as the input size becomes large.

It is actually an estimation technique. However, asymptotic analysis has proved useful to computer scientists who must determine if a particular algorithm is worth considering for implementation.

- The critical resource for a program is -most often- **running time**.
- The **growth rate** for an algorithm is the rate at which the cost of the algorithm grows as the size of its input grows.
    - cn (for c any positive constant) ➔ **linear** growth rate or running time.
    - n^2 ➔ **quadratic** growth rate
    - 2^n ➔ **exponential** growth rate.

Worst case? The advantage to analyzing the worst case is that you know for certain that the algorithm must perform at least that well.

Example:

Assume :   Algorithm A: time = **15n+93**

                Algorithm B: time = **2n^2+1**   **which is faster?**

Graph using Excel
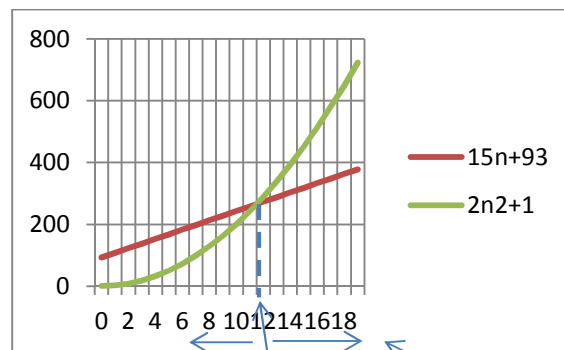
The "break-even point"

We are interested for large **n**

* for sufficiently large n, algorithm A is faster

* in the long run constants do not mater.

Upper bound for the growth of the algorithm's running time. It indicates the upper or highest growth rate that the algorithm can have. ➔ **big-O notation**.

> For **T(n)** a non-negatively valued function, **T(n)** is in set **O($f(n)$)** if there exist two positive constants c and n_0 such that **T(n) ≤ c$f(n)$** for all $n > n_0$.

* Prove that **15n+93** is **O(n)**

We must show +ve **c** and **n_0** such that **15n+93 <= cn** for **n >= n_0**

<provided **n= 93**> ➔   **15n+n** ➔   **16n <= cn** ➔   <provided **c = 16**>

So for **c=16**   and **n0 = 93**   ➔   **// proved**

<mark>Graph using Excel</mark>

Prove that **$2n^2+1 = O(n^2)$**

Must show +ve **c, n_0** such that **$2n^2+1 <= cn^2$ for n >= n_0**

$2n^2+1$ **<provided n=1>**

$2n^2+ n^2$ ➔   **$3n^2$**   **<provided c=3>**

$2n^2+1 <= 3n^2$

So, **c=3 , n_0=1**   // proved

<mark>Graph using Excel</mark>

Example 3.5 For a particular algorithm, $\mathbf{T}(n) = c_1 n^2 + c_2 n$ in the average case where c_1 and c_2 are positive numbers. Then, $c_1 n^2 + c_2 n \leq c_1 n^2 + c_2 n^2 \leq (c_1 + c_2)n^2$ for all $n > 1$. So, $\mathbf{T}(n) \leq cn^2$ for $c = c_1 + c_2$, and $n_0 = 1$. Therefore, $\mathbf{T}(n)$ is in $O(n^2)$ by the second definition.

The **lower bound** for an algorithm is denoted by the symbol **Ω**, pronounced "big-Omega" or just "Omega."

> For **T(n)** a non-negatively valued function, **T(n)** is in set **Ω($g(n)$)** if there exist two positive constants **c** and **n_0** such that **T(n) ≥ $cg(n)$** for all **$n > n_0$**.

* prove that **15n+93** is **Ω(n)**

We must show +ve **c** and **n_0** such that **15n+93 >= cn** for **n >= n_0**

<because 93 is +ve>   >= cn   ➔   <provided **c=15**>   ← so any n_0 >0 will do

So **c=15, n_0=1** // proved

<mark>Graph using Excel</mark>

* prove that **$2n^2+1$** is **Ω(n^2)**

must show +ve **c** and **n_0** such that **$2n^2+1 >= cn^2$** for **n >= n_0**

<because 1 is +ve>

So **c=2, n_0=1** // proved

<mark>Graph using Excel</mark>

Example 3.7 Assume $T(n) = c_1 n^2 + c_2 n$ for c_1 and $c_2 > 0$. Then,

$$c_1 n^2 + c_2 n \geq c_1 n^2$$

for all $n > 1$. So, $T(n) \geq cn^2$ for $c = c_1$ and $n_0 = 1$. Therefore, $T(n)$ is in $\Omega(n^2)$ by the definition.

When the **upper** and **lower bounds** are the same within a constant factor, we indicate this by using **Θ (big-Theta)** notation.

T(n) = **Θ(g(n)) iff** T(n) = **O(g(n))**   **and** T(n) = Ω **(g(n))**

Example: Because the **sequential search algorithm** is both in **O(n)** and in **Ω(n)** in the average case, we say it is **Θ(n)** in the average case.

Examples:

f	g	Relations
n	$8n^2$	$f \in O(g)$
n^3	$12n^3 + 4n^2$	$f \in O(g), f \in \Omega(g), f \in \Theta(g)$
$2^{\log n}$	n	$f \in O(g), f \in \Omega(g), f \in \Theta(g)$
$n!$	$n^2 2^n$	$f \in \Omega(g)$

Simplifying Rules

1. If $f(n)$ is in $O(g(n))$ and $g(n)$ is in $O(h(n))$, then $f(n)$ is in $O(h(n))$.
2. If $f(n)$ is in $O(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $O(g(n))$.
3. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $f_1(n) + f_2(n)$ is in $O(\max(g_1(n), g_2(n)))$.
4. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $f_1(n)f_2(n)$ is in $O(g_1(n)g_2(n))$.

- Rule (2) is that you can ignore any multiplicative constants.
- Rule (3) says that given two parts of a program run in sequence, you need consider only the more expensive part.
- Rule (4) is used to analyze simple loops in programs.

Taking the first three rules collectively, you can ignore all constants and all lower-order terms to determine the asymptotic growth rate for any cost function.

Order of growth of some common functions

$$O(1) \leq O(\log_2 n) \leq O(n) \leq O(n \log_2 n) \leq O(n^2) \leq O(n^3) \leq O(2^n)$$
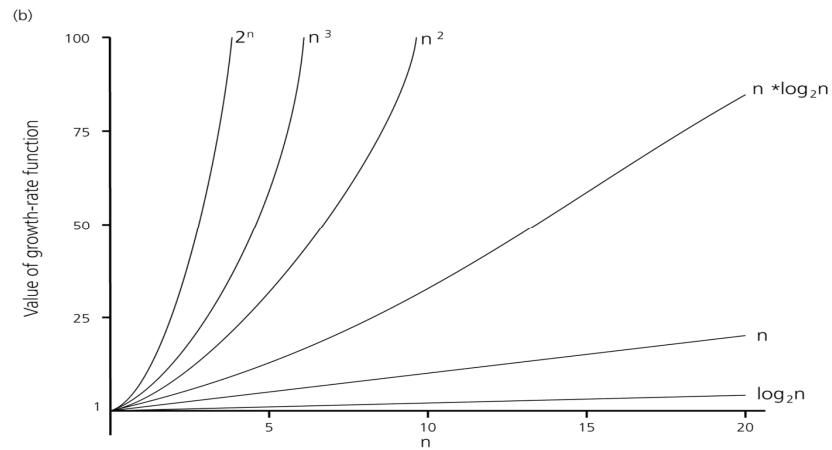
If the problem size is always small, you can probably ignore an algorithm's efficiency

Limitations of big-oh analysis

- Overestimate.
- Analysis assumes infinite memory.
- Not appropriate for small amounts of input.
- The constant implied by the Big-Oh may be too large to be practical   (**2*N* log *N*   *vs.*   1000*N*)

13

(Lecture 6) Analyzing algorithm examples

General Rules of analyzing algorithm code:

Rule 1—*for* loops.

The running time of a **for** loop is at most the running time of the statements inside the **for** loop (including tests) **times** the number of iterations.

Rule 2 — Nested loops.

Analyze these **inside out**. The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all the loops.

Rule 3—Consecutive Statements.

These just add (which means that the maximum is the one that counts;

Rule 4 —*if/else*.

```
if( condition )
    S1
else
    S2
```

The running time of an **if/else** statement is never more than the running time of the **test** plus the larger of the running times of **S1** and **S2**.

Rule 5 —*methods call*.

If there are method calls, these must be analyzed first.

Sorting Algorithm

1- **Bubble Sort (revision)** ➜ O(n^2)

```
for(int i=0; i<n-1; i++){
    for(int j=0; j<n-1-i; j++){
        if(num[j+1] < num[j]){
            temp = num[j];
            num[j] = num[j+1];
            num[j+1] = temp;
        }
    }
}
```

2- **Selection Sort (revision)** ➔ **O(n²)**   : named selection because every time we select the smallest item.

```
int temp, minIndx;
for(int i=0; i<num.length-1;i++){
    minIndx = i;
    for(int j=i+1; j<num.length;j++){
        if(num[j] < num[minIndx])
            minIndx=j;
    }
    if(i!= minIndx){
        temp = num[i];
        num[i] = num[minIndx];
        num[minIndx] = temp;
    }
}
```

3- **Insertion sort:**

Example:

```
int j, temp, current;
for(int i=1; i<n; i++){
    current = num[i];
    j = i-1;
    while(j>=0 && num[j]>current){
        num[j+1] = num[j];
        j--;
    }
    num[j+1] = current;
}
```

Pseudo code:

O(n^2) sorting algorithms comparison :

(run demo @ http://www.sorting-algorithms.com/)

Bubble Sort	Selection Sort	Insertion Sort
Very inefficient	Better than bubble sort	Relatively good for small lists
	Running time is independent of ordering of elements	Relatively good for partially sorted lists

Merge sort : recursive algorithm

Merge: take 2 sorted arrays and merge them together into one.

Example: merge method
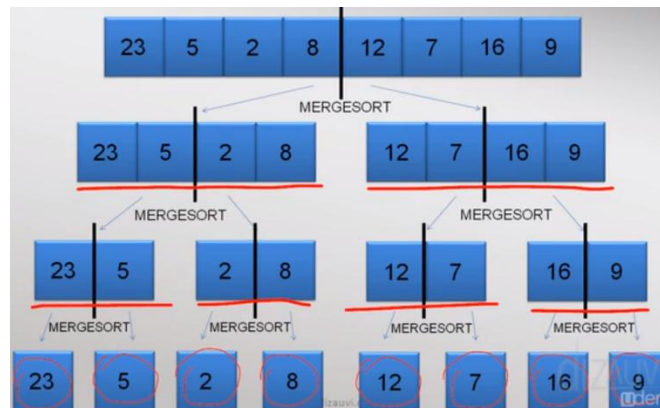
Example: merge sort

Pseudo-code :

MergeSort (A, start, end)   MergeSort (A, 0, 7) 🟥

   if start < end

      middle = Floor[(start + end)/2]   middle = 3

      MergeSort(A, start, middle)   MergeSort (A, 0, 3) 🟩

      MergeSort(A, middle+1, end)

Pseudo code:

      Merge(A, start, middle, end)

Pseudo-code (Merge) :

Merge (A, start, mid, end)

$n_1 = mid - start + 1$

$n_2 = end - mid$

Let left$[0..n_1]$ and right$[0..n_2]$ be new temp arrays

for i = 0 **to** n_1-1

   left [i] = A [start + i]

for j = 0 **to** n_2-1

   right [j] = A [mid + 1 + j]

i , j = 0

for k = start **to** end

   if left [i] ≤ right [j]

      A [k] = left [i]

      i = i + 1

   else A [k] = right [j]

      j = j + 1

==**Make sure of array boundaries**==

==H.W: implement merge sort your own==

Searching elements in an array:

a [2] = 5   : O(1)

find (8)   : O(n)

Case 1: unordered array: delete (item) : O(n)

3	7	20	32	45	55	60	75

find (60)

Finding Index

$\left\lfloor \dfrac{7+0}{2} \right\rfloor$ = 3 ⟶ a[3] = 32

$\left\lfloor \dfrac{7+3}{2} \right\rfloor$ = 5 ⟶ a[5] = 55

$\left\lfloor \dfrac{7+5}{2} \right\rfloor$ = 6 ⟶ a[6] = 60

Case 2: ordered array: -Binary search-

3	7	20	32	45	55	60	75

First Search   : n

Second Search   : $\dfrac{n}{2}$

Third Search   : $\dfrac{n}{4}$

$\vdots$

$(i\text{-}1)^{th}$ Search   : 2

i^{th} Search   : $1 = \dfrac{n}{2^{i-1}}$

$2^{i-1} = n \implies (i\text{-}1) = \log_2 n$

find (item) = $O(\log_2 n)$

n	$\log_2 n$
2	1
1024	10
1048576 (Million)	20
1099511627776 (Trillion)	40

Inserting and deleting items from ordered array

3	7	20	32	45	52	55	60	75

Insert (52)

Insert (item) = O (n)

Search (item) = O ($\log_2 n$)

3	7	20	32	45	52	60	75

Delete (55)

Delete (item) = O (n)

Algorithm - abstract way to perform computation tasks

Data Structure - abstract way to organize information

Linked List:
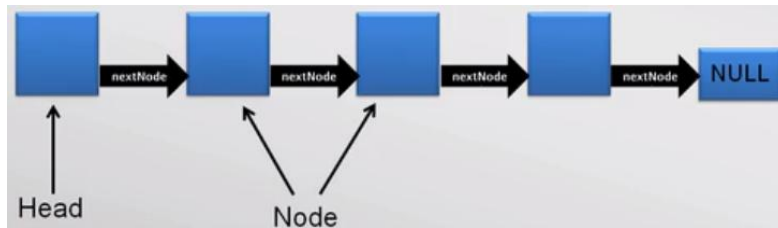
Node:
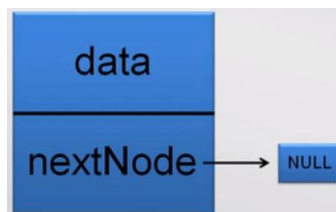
Node code:

```java
public class Node<T> {
        private T data;
        private Node<T> nextNode;

        public Node(T data) { this.data = data;          }

        public void setData(T data) {  this.data = data;  }
        public T getData() {  return data;  }

        public Node<T> getNextNode() {  return nextNode;  }
        public void setNextNode(Node<T> nextNode) {        this.nextNode = nextNode;  }
}
```

Linked List Code:

```java
public class LinkedList<T> {
        private Node<T> head;
}
```

Inserting a new node:
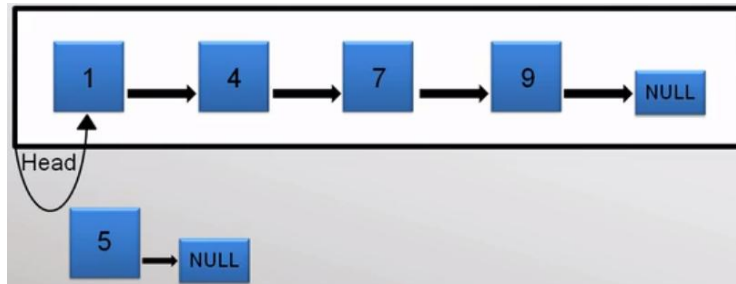
Connect Head → new node ?? we lose pointer to linked list
Order of connecting the node is very important
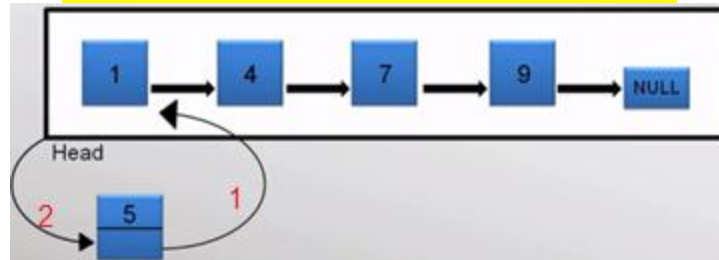
Insert code:

```java
public void addAtStart(T data) {
        Node<T> newNode = new Node<T>(data);
        newNode.setNextNode(this.head);    // step 1
        this.head = newNode;               // step 2
}
```

Create a driver class to test linked list classes.
Override the toString methods first

What's the time complexity of inserting an item to the head??   ➔   **O(1)**

Node toString:

```java
@Override
public String toString() {  return this.data.toString();  }
```

LinkedList toString:

```java
@Override
public String toString() {
        String res = "➔";
        Node<T> curr = this.head;
        while (curr != null) {
                res += curr + "➔ ";
                curr = curr.getNextNode();
        }
        return res + "NULL";
}
```

Length of Linked List?
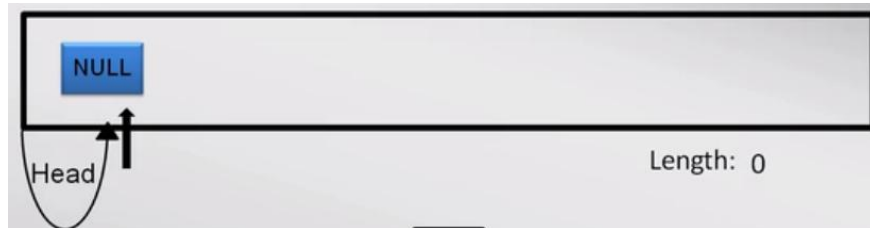
Case 1: If it's empty:

Case 2: If not: Make a pointer and move over all the nodes and maintain a counter
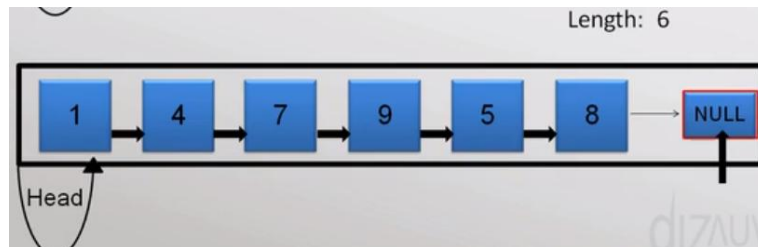
Length code: Time Complexity ➔ O(n)

```
public int length() {
        int length = 0;
        Node<T> curr = this.head;
        while (curr != null) {
                length++;
                curr = curr.getNextNode();
        }
        return length;
}
```

Deleting the head node:

Simply move the **head** to the **head.nextNode**
Now first Node has no reference to it ➔ Garbage
Time Complexity ➔   **O(1)**

Delete at head code: // make sure linked list is not empty

```
public Node<T> deleteAtStart() {
        Node<T> toDel = this.head;
        this.head = this.head.getNextNode();
        return toDel;
}
```

21

Searching for an Item in a Linked List:

Time Complexity: linear growth ➔ **O(n)**
Find code:

```
public Node<T> find(T data) {
        Node<T> curr = this.head;
        while (curr != null) {
                if (curr.getData() == data)   //  if (curr.getData().equals(data))
                        return curr;
                curr = curr.getNextNode();
        }
        return null;
}
```

How to use Java generics?? (Optional)
Provided by java, to be able to parameterize the Node and Linked List objects.

Doubly Ended Linked List:
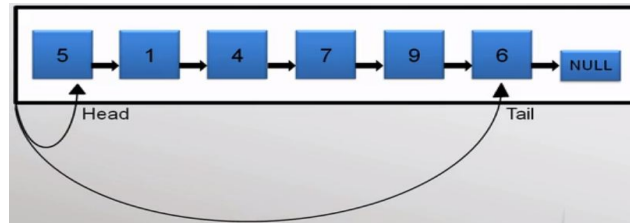
We have two pointers: one at **head** and one at **tail**
Therefore, we can add and delete at both ends.

Doubly Ended list code:

```java
public class DoubleEndedList<T> extends LinkedList<T> {
        private Node<T> tail;

        public Node<T> getTail() {      return this.tail;   }

        public void addAtEnd(T data) {
                Node<T> newNode = new Node<T>(data);
                if (this.head == null) {  // empty
                        this.head = newNode;
                        this.tail = newNode;
                }
                else {
                        this.tail.setNextNode(newNode);
                        this.tail = newNode;
                }
        }
}
```

Make sure to override addAtStart to set the tail pointer correctly:

```java
@Override
public void addAtStart(T data) {
    Node<T> newNode = new Node<T>(data);
    if (this.head == null) {  // empty
      this.head = newNode;
      this.tail = newNode;
    }
    else{
      newNode.setNextNode(this.head);
      this.head = newNode;
    }
}
```

Inserting new Node to a sorted linked list:
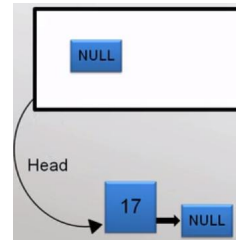
Case 1: empty linked list: in this case we added as first element.
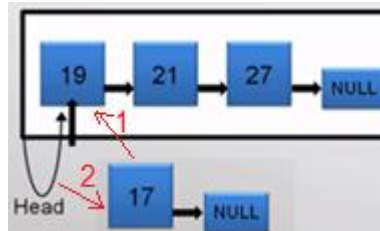
Case 2: adding first to a sorted linked list:

Case 3: adding in the middle in a sorted linked list:
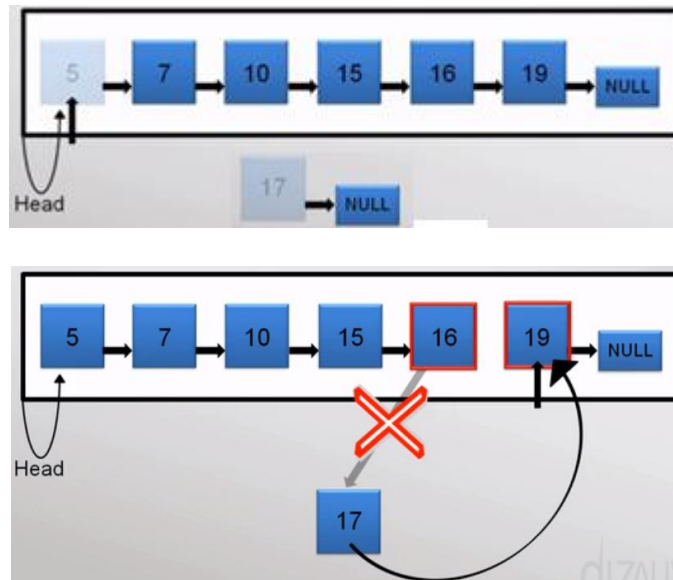
However we can access the next node from the current node.
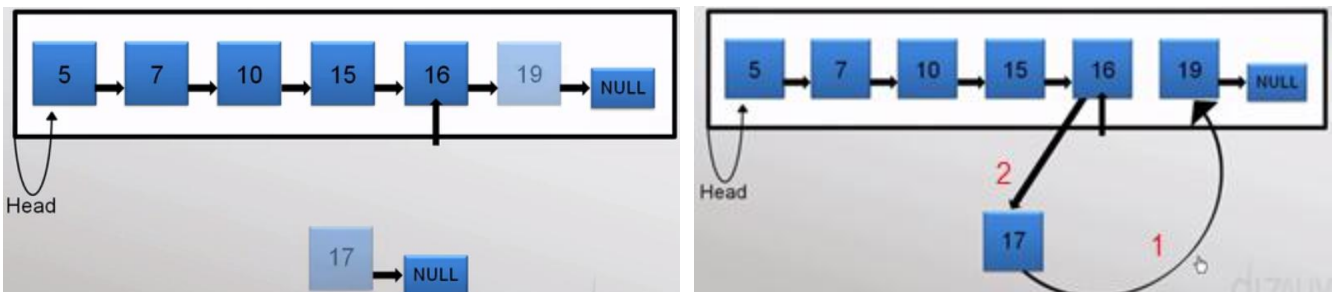
Time Complexity ➔ **O(n)**

H.W. ➔   implement insert into a sorted linked list

(Lecture 8) Doubly Linked List

Node:
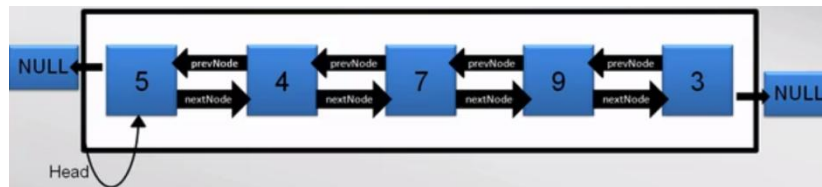
Doubly Linked List:

Doubly Node Code:

```java
public class DNode {
        private int data;
        private DNode nextNode;
        private DNode prevNode;

        public DNode(int data) { this.data = data; }
        public int getData() { return data;  }
        public DNode getNextNode() { return nextNode;     }
         public DNode getPrevNode() { return prevNode; }

        public void setNextNode(DNode nextNode) {   this.nextNode = nextNode; }
        public void setPrevNode(DNode prevNode) {   this.prevNode = prevNode; }

        @Override
        public String toString() { return this.data+""; }
}
```

Doubly Linked List code:

```java
public class DLinkedList {
        private DNode head;
}
```

Insert a new Element
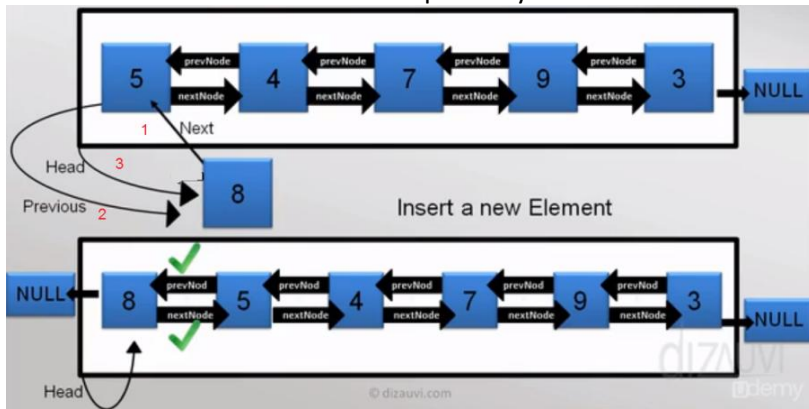
Insert a new node at head:
Insert at head code:

```
public void insertAtHead(int data) {
        DNode newNode = new DNode(data);
        newNode.setNextNode(this.head);
        if (this.head != null)              // make sure it's not empty
                this.head.setPrevNode(newNode);
        this.head = newNode;
}
```

Length of a doubly linked list code:

```
public int length() {
        int length = 0;
        DNode curr = this.head;
        while (curr != null) {
                length++;
                curr = curr.getNextNode();
        }
        return length;
}
```

Override toString method code:

```
@Override
public String toString() {
        StringBuilder sb = new StringBuilder("head ->");
        DNode n = this.head;
        while (n != null) {
                sb.append("["+n+"]");
                n = n.getNextNode();
                if(n!=null)
                        sb.append("<=>");
        }
        sb.append("->NULL");
        return sb.toString();
}
```

Student Activity: insert at last:

```
public void insertAtEnd(int data) {
        DNode newNode = new DNode(data);
        if (this.head == null)
                this.head = newNode;
        else  {                              // find last node
                DNode last = head;
                while(last.nextNode != null)   last = last.nextNode;
                last.nextNode = newNode;
                newNode.prevNode = last;
        }
}
```

Insertion Sort using doubly linked list:

<mark>Review insertion sort logic and point to problem of insertion and time needed to shift the items</mark>
<mark>Worst case if the array is reverse sorted</mark>

Example: assume we need to sort the following doubly linked list:
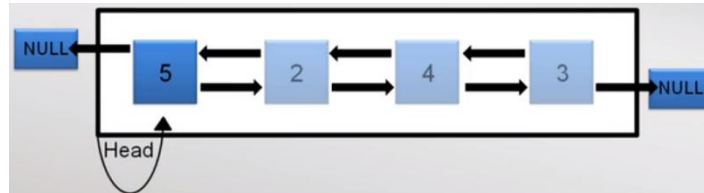
Assumption: 1st node is sorted. We start from the 2nd element:
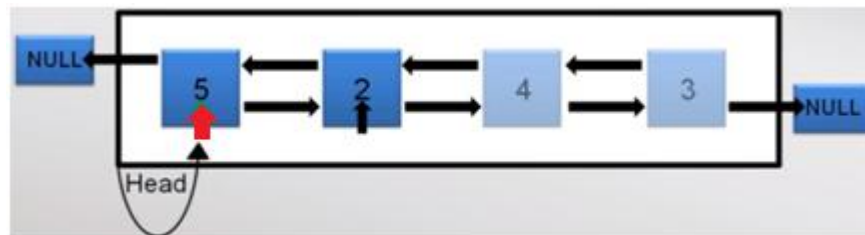
Here:

* The **black** pointer points to the **current** node to be sorted.
* The **red** pointer points to previous node of **current** node to be sorted.
* The **green** pointer points to next node of **current** node to be sorted.

Step 1: The **red** pointer keeps move backward until it reaches a node which has a value **smaller** than the

current node **or** reach **NULL**.

Step 2: the **current** item will be inserted after **red** pointer as follow:

<mark>Make sure you maintain references correctly.</mark>
<mark>To do so draw the expected outcome and follow the steps to change the pointers:</mark>
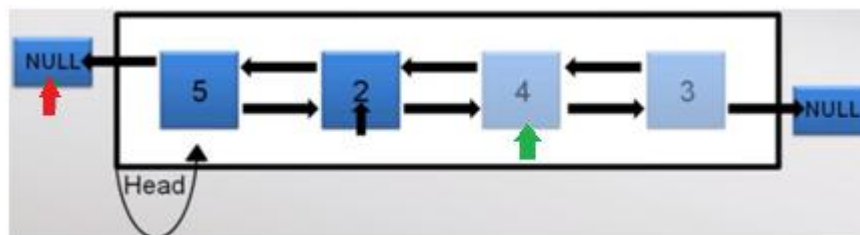
Initial state:

Final state:

28
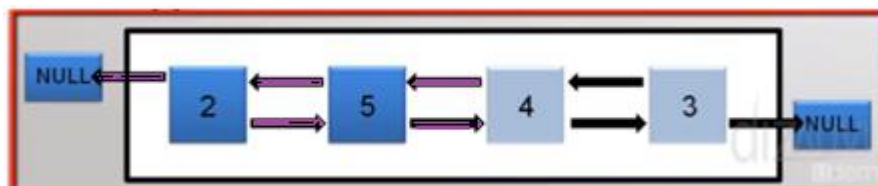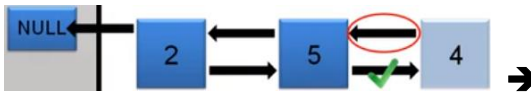
Case 1: insert to head

Step 2.0: make new **green** pointer = **black.nextNode**

Step 2.1: ➜ **black.prevNode.nextNode = green**
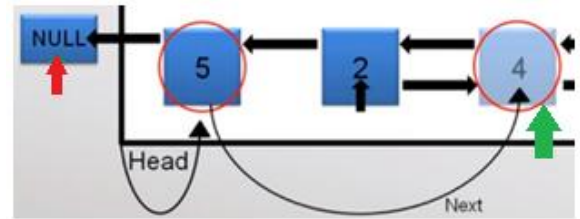
Step 2.2: ➜

if (green != null) green.prevNode = black.prevNode
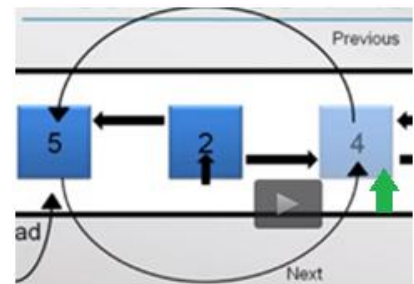
Step 2.3: ➜ **black.prevNode = red**
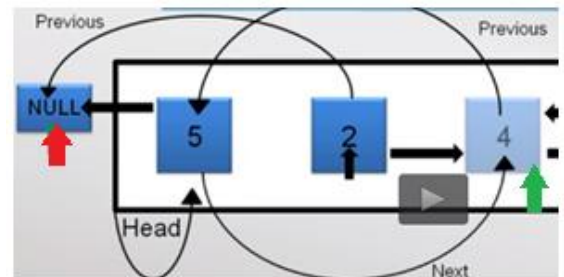
Step 2.4: ➜

if(red==null) black.nextNode = black.nextNode.prevNode
else   black.nextNode = red.nextNode
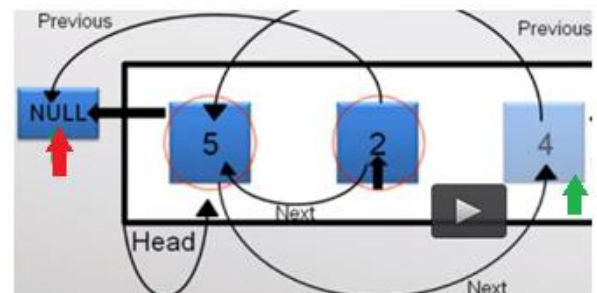
Step 2.5:

      If (red == null) black.nextNode.prevNode = black
      else   red.NextNode. PrevNode = black
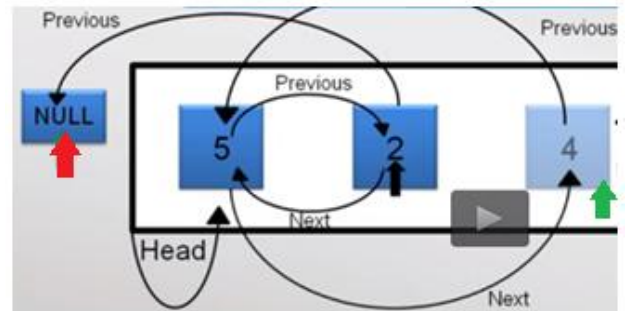
Step 2.6:

      if (red == NULL)   head = black
      else   red.setNextNode = black;
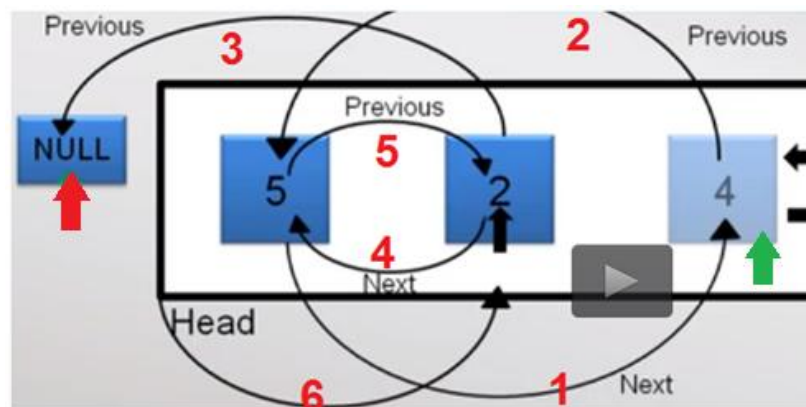
Step 2.7: black = green

Case 2: insert 4 in the middle

                                        Practice yourself

Case 3: insert last element
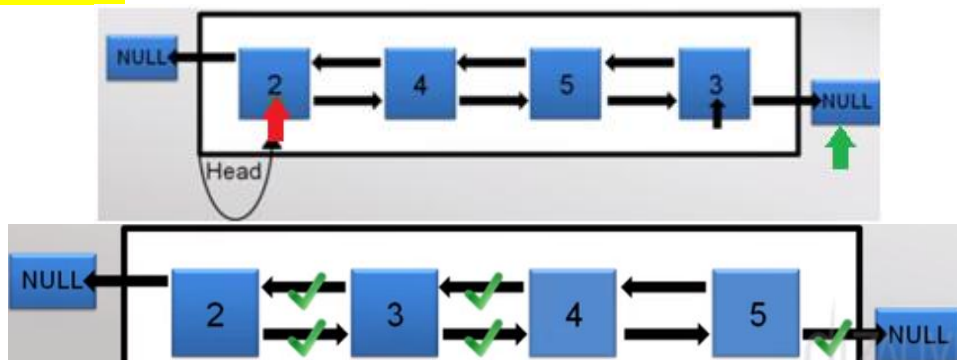
Insertion Sort Code:

```java
public void sort() {
    DNode black = this.head;
    while (black != null) {
        DNode red = black.getPrevNode();
        while (red != null && (red.getData() > black.getData())) {
            red = red.getPrevNode();
        }
        DNode green = black.getNextNode();  // step 2.0
        if (red != null || (head != black)) {
            black.getPrevNode().setNextNode(green);  // step 2.1
            if (green != null) {
                green.setPrevNode(black.getPrevNode()); // step 2.2
            }
            black.setPrevNode(red);  // step 2.3
        }
        if (red == null) {   // set the black as head
            if (head != black) {
                black.setNextNode(this.head);  // step 2.4
                black.getNextNode().setPrevNode(black);  // step 2.5
                head = black;  // step 2.6
            }
        } else {  // red is not null
            black.setNextNode(red.getNextNode()); // step 2.4
            red.getNextNode().setPrevNode(black);  // step 2.5
            red.setNextNode(black);  // step 2.6
        }
        black = green;
    }
}
```

Circular Double Linked List:

Doubly Linked Circular list
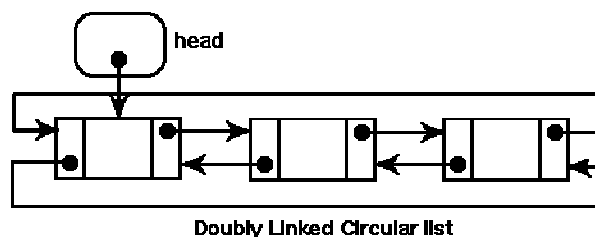
(Lecture 9) Analyzing the Complexity of Merge Sort

$$2^{h-1} = n$$

$$h = 1 + \log_2 n$$

$$O(n \log_2 n)$$
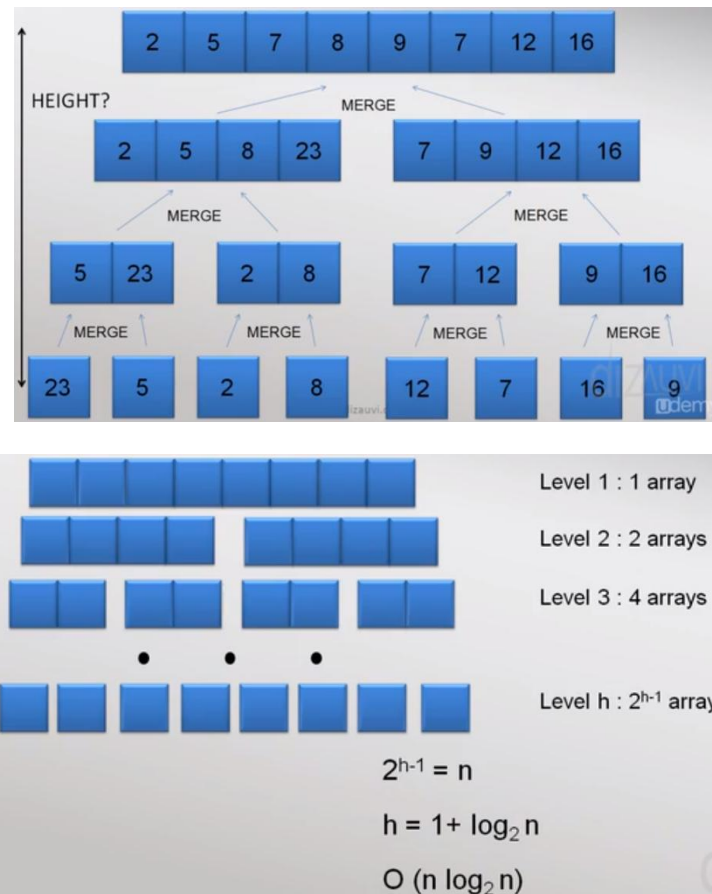
In Place vs. Not in Place Sorting

In place sorting algorithms are those, in which we sort the data array, without using any additional memory.

What about selection, bubble, insertion algorithms?
Well, our implementation of these algorithms is **IN PLACE**. The thing is, if we use a **constant** amount of extra memory (like one temporary variable/s), the sorting is **In-Place**.

But in case extra memory (merging sort), which is **proportional** to the input data size, is used, then it is **NOT IN PLACE sorting**.

But because memory these days is so cheap, that we usually don't bother about using extra memory, if it makes the program run faster.

Stable vs. Unstable Sort
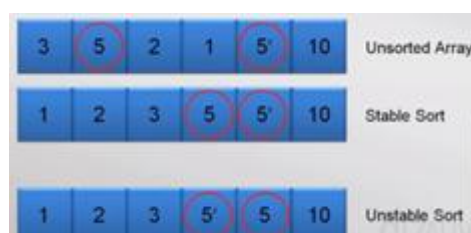
Example: Insertion Sort Code:

```java
public void sort(int[] data) {
    for (int i =0; i < data.length; i++) {
        int current = data[i];
        int j = i-1;
        while (j >=0 && data[j] > current) {
            data[j+1] = data[j];
            j--;
        }
        data[j+1] = current;
    }
}
```

```java
public void sort(int[] data) {
    for (int i =0; i < data.length; i++) {
        int current = data[i];
        int j = i-1;
        while (j >=0 && data[j] >= current) {
            data[j+1] = data[j];
            j--;
        }
        data[j+1] = current;
    }
}
```

Example:

$O(n^2)$ ➔ selection sort, bubble sort, insertion sort

$O(n \log n)$ ➔ merge sort

$O(n)$ ➔ (Sorting in linear time) ??

If we know some information about data to be sorted (e.g. students' marks -Range 50 to 99 –), we can achieve linear time sorting

Counting Sort:

Example: assume data range from 1 to 10
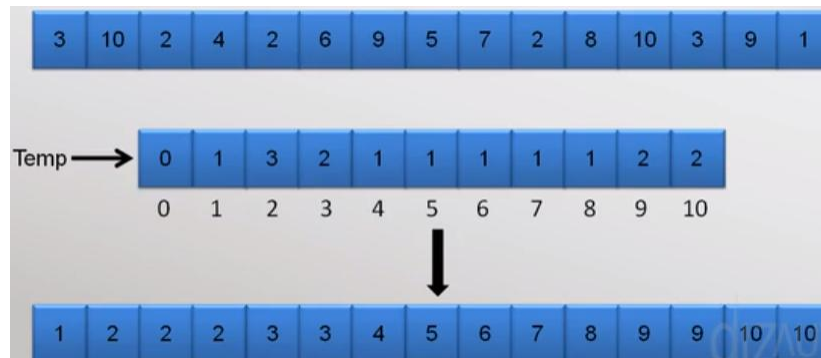
Time analysis:

Note: K is typically small comparing to n

Bad Situation: what if K is larger than n ??
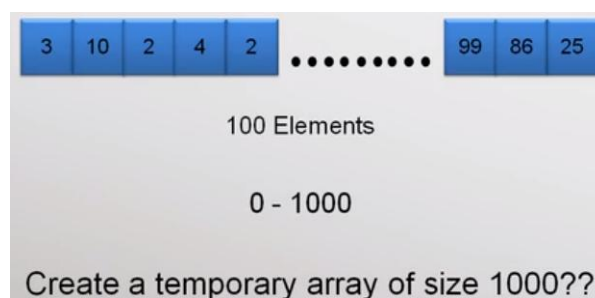
Is counting sort is In-Place or Not-In-Place ??   why?

Radix Sort:

What is Radix? The **radix** or **base** is the number of unique digits, including zero, used to represent numbers in a positional numeral system.

For example, for the decimal system: radix is 10 ,   Binary system: radix is 2

Example Radix Sort:

Step 1: take the least significant digits of the values to be sorted.
Step 2: sort the list of elements based on that digit
Step 3: take the 2nd least significant digits and repeat step 2
Then the 3rd LSD and so on

<mark>How to implement Radix Sort:</mark>

Radix Sort Algorithm using linked list:

Consider the following array

9	179	139	38	10	5	36

Create an array of linked lists as follow:

0
1
2
3
4
5
6
7
8
9

- Total of 10 linked lists
- 0 to 9 refer to actual numbers
- With input numbers, we will start with mod 10 then divide the resulted number by 1

Code:
- m=10 ➜ mod operation
- n=1; ➜ find the specific digit at that column

e.g.   Arr[0] = 9
        9 % m = 9
                9 / n = 9

0 | → 10
1
2
3
4
5 | → 5
6 | → 6
7
8 | → 38
9 | → 9 → 179 → 139

- If we reaches the end of array.
- Make a new array by removing data from the head of each linked list in order.

Result:

10	5	36	38	9	179	139

Is this sorted?

Next step: consider the **2nd** significant digit from the previous resulted array:

Code:

$m = m * 10 = 100$

$n = n * 10 = 10$

e.g. Arr[0] = 10

10 % m = 10

10 / n = 1

0 | → 5 → 9
1 | → 10
2 | →
3 | → 36 → 38 → 139
4 | →
5 | →
6 | →
7 | → 179
8 | →
9 | →

Result:

5	9	10	36	38	139	179

Is this sorted? Yes in this case but we are not done yet

Next step: consider the **3rd** significant digit from the previous array:

Code:

$m = m * 10 = 1000$

$n = n * 10 = 100$

e.g. Arr[0] = 5

5 % m = 5

5 / n = 0

0 | → 5 → 9 → 10 → 36 → 38
1 | → 139 → 179
2 | →
3 | →
4 | →
5 | →

:

Result:

5	9	10	36	38	139	179

Is this sorted? What is the time complexity

HW: implement Radix sort using Doubly Linked List

36

(Lecture 10) Stacks 1

stack is an abstract data type that serves as a collection of elements, with two principal operations:

- **push** adds an element to the collection;
- **pop** removes the last element that was added.
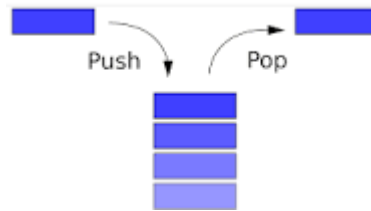
- **Last In, First Out ➔ LIFO**

UML	DESCRIPTION
+push(newEntry: T): void	Task: Adds a new entry to the top of the stack.
+pop(): T	Task: Removes and returns the stack's top entry.
+peek(): T	Task: Retrieves the stack's top entry without changing the stack in any way.
+isEmpty(): boolean	Task: Detects whether the stack is empty.
+clear(): void	Task: Removes all entries from the stack.

Linked Implementation:

Each of the following operation involves top of stack

- **push**
- **pop**
- **peek**

Head or Tail for topNode??

Head of linked list easiest, fastest to access ➔ Let this be the top of the stack
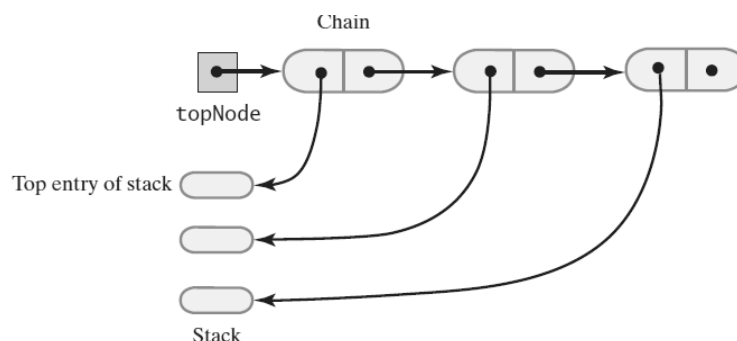
```java
public class LinkedStack<T> {
        private Node<T> topNode;
        public void push(T data) {
                Node<T> newNode = new Node<T>(data);
                newNode.setNextNode(topNode);
                topNode = newNode;
        }
        public Node<T> pop() {
                Node<T> toDel = topNode;
                assert topNode!=null : "Empty Stack" ;
                topNode = topNode.getNextNode();
                return toDel;
        }

        public Node<T> peek() {   return topNode;    }
        public int length() {
                int length = 0;
                Node<T> curr = topNode;
                while (curr != null) {
                        length++;
                        curr = curr.getNextNode();
                }
                return length;
        }
        public boolean isEmpty() {    return (topNode == null);        }
        public  void clear {    topNode == null;        }
}
```

Array-Based Implementation
- End of the array easiest to access
  - Let this be top of stack
  - Let first entry be bottom of stack
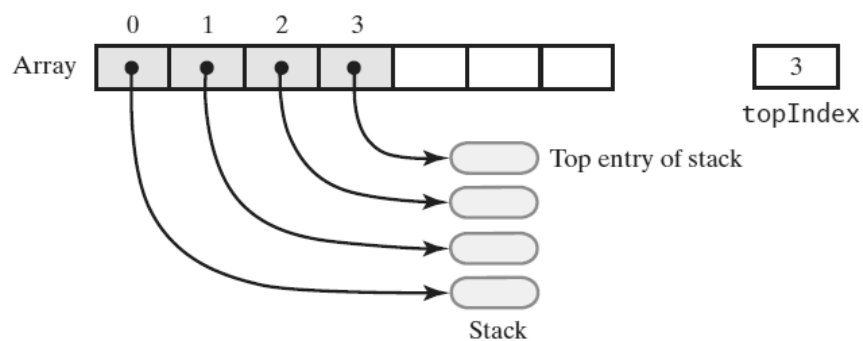
H.W. implement array based stack

Balanced Expressions

Delimiters paired correctly ➜ compilers

Example 1: The contents of a stack during the scan of an expression that contains the **balanced delimiters** `{[()]}`
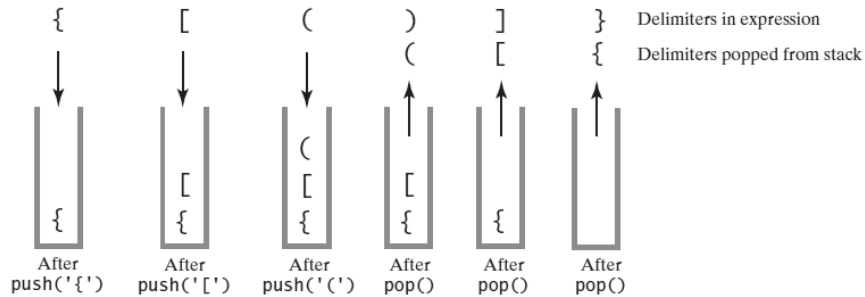
Example 2: The contents of a stack during the scan of an expression that contains the **unbalanced delimiters** `{[()}`
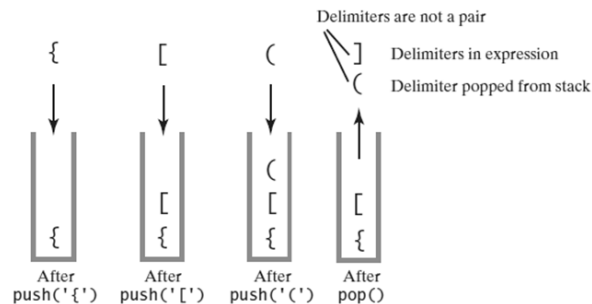
Example 3: The contents of a stack during the scan of an expression that contains the **unbalanced delimiters** `[()]}`
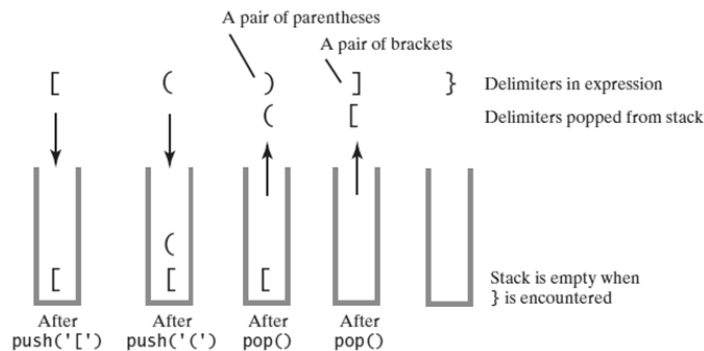
Example 4: The contents of a stack during the scan of an expression that contains the **unbalanced delimiters** `{[()]`
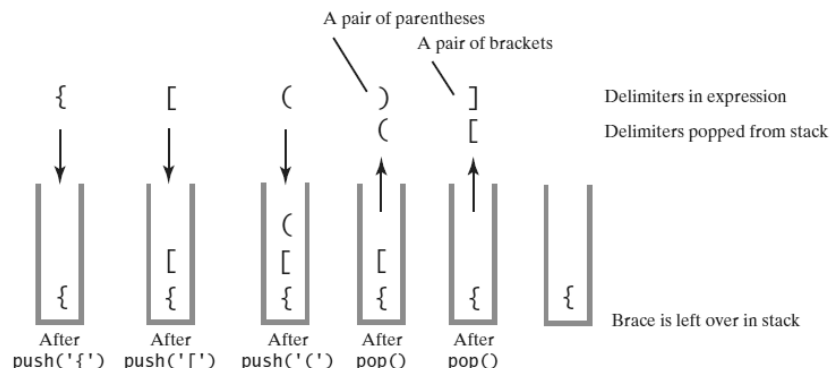
Algorithm to process for balanced expression:

```
Algorithm checkBalance(expression)
// Returns true if the parentheses, brackets, and braces in an expression are paired correctly.

isBalanced = true
while ((isBalanced == true) and not at end of expression)
{
    nextCharacter = next character in expression
    switch (nextCharacter)
    {
        case '(': case '[': case '{':
            Push nextCharacter onto stack
            break

        case ')': case ']': case '}':
            if (stack is empty)
                isBalanced = false
            else
            {
                openDelimiter = top entry of stack
                Pop stack
                isBalanced = true or false according to whether openDelimiter and
                             nextCharacter are a pair of delimiters
            }
            break
    }
}

if (stack is not empty)
    isBalanced = false
return isBalanced
```

H.W. implement check balance algorithm using linked/array stacks

Generic stack: array implementation

```java
public class FixedCapacityStack<Item>
{
    private Item[] s;
    private int N = 0;

    public FixedCapacityStack(int capacity)
    {  s = (Item[]) new Object[capacity];  }

    public boolean isEmpty()
    {  return N == 0;  }

    public void push(Item item)
    {  s[N++] = item;  }

    public Item pop()
    {  return s[--N];  }
}
```

(Lecture 11) Stacks 2
Processing Algebraic Expressions

- **Infix**: each binary operator appears between its operands *a + b*
- **Prefix**: each binary operator appears before its operands   *+ a b*
- **Postfix***: each binary operator appears after its operands *a b +*

Arithmetic expression evaluation

Evaluate infix expressions.

Two-stack algorithm. [E. W. Dijkstra]
- Value: push onto the value stack.
- Operator: push onto the operator stack.
- Left parenthesis: ignore.
- Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

Example:

41
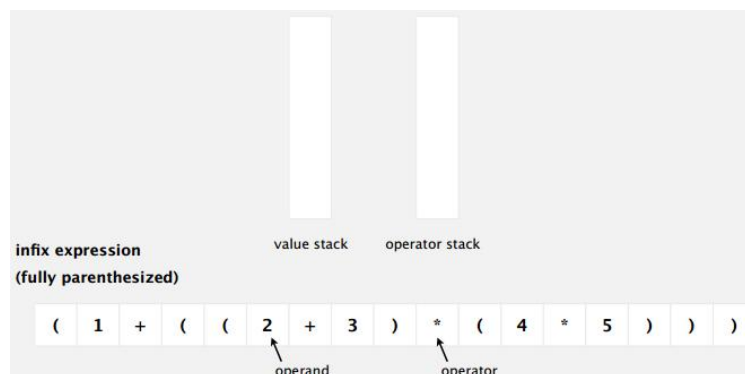
```java
public class Evaluate
{
   public static void main(String[] args)
   {
      Stack<String> ops  = new Stack<String>();
      Stack<Double> vals = new Stack<Double>();
      while (!StdIn.isEmpty()) {
         String s = StdIn.readString();
         if      (s.equals("("))                  ;
         else if (s.equals("+"))     ops.push(s);
         else if (s.equals("*"))     ops.push(s);
         else if (s.equals(")"))
         {
            String op = ops.pop();
            if      (op.equals("+")) vals.push(vals.pop() + vals.pop());
            else if (op.equals("*")) vals.push(vals.pop() * vals.pop());
         }
         else vals.push(Double.parseDouble(s));
      }
      StdOut.println(vals.pop());
   }
}
```

```
% java Evaluate
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0
```

Infix to Postfix

Infix-to-postfix Conversion:

• Operand	Append each operand to the end of the output expression.
• Operator ^	Push ^ onto the stack.
• Operator +, -, *, or /	Pop operators from the stack, appending them to the output expression, until the stack is empty or its top entry has a lower precedence than the new operator. Then push the new operator onto the stack.
• Open parenthesis	Push (onto the stack.
• Close parenthesis	Pop operators from the stack and append them to the output expression until an open parenthesis is popped. Discard both parentheses.

Example 1: Converting the **infix** expression <u>a + b * c</u> to **postfix** form

Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	a	
+	a	+
b	a b	+
*	a b	+ *
c	a b c	+ *
	a b c *	+
	a b c * +	

Example 2: Converting an infix expression to postfix form: **a - b + c**

Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	a	
−	a	−
b	a b	−
+	a b −	
	a b −	+
c	a b − c	+
	a b − c +	

Example 3: Converting an infix expression to postfix form: **a ^ b ^ c**

Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	a	
^	a	^
b	a b	^
^	a b	^ ^
c	a b c	^ ^
	a b c ^	^
	a b c ^ ^	

Example 4: The steps in converting the infix expression **a / b * (c + (d - e))** to postfix form

Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	a	
/	a	/
b	a b	/
*	a b /	
	a b /	*
(	a b /	* (
c	a b / c	* (
+	a b / c	* (+
(	a b / c	* (+ (
d	a b / c d	* (+ (
−	a b / c d	* (+ (−
e	a b / c d e	* (+ (−
)	a b / c d e −	* (+ (
	a b / c d e −	* (+
)	a b / c d e − +	* (
	a b / c d e − +	*
	a b / c d e − + *	

Infix-to-postfix Algorithm

```
Algorithm convertToPostfix(infix)
// Converts an infix expression to an equivalent postfix expression.

operatorStack = a new empty stack
postfix = a new empty string
while (infix has characters left to parse)
{
    nextCharacter = next nonblank character of infix
    switch (nextCharacter)
    {
        case variable:
            Append nextCharacter to postfix
            break

        case '^' :
            operatorStack.push(nextCharacter)
            break

        case '+' : case '-' : case '*' : case '/' :
            while (!operatorStack.isEmpty() and
                    precedence of nextCharacter <= precedence of operatorStack.peek())
            {
                Append operatorStack.peek() to postfix
                operatorStack.pop()
            }
            operatorStack.push(nextCharacter)
            break

        case '(' :
            operatorStack.push(nextCharacter)
            break

        case ')' : // Stack is not empty if infix expression is valid
            topOperator = operatorStack.pop()
            while (topOperator != '(')
            {
                Append topOperator to postfix
                topOperator = operatorStack.pop()
            }
            break

        default: break // Ignore unexpected characters
    }
}
while (!operatorStack.isEmpty())
{
    topOperator = operatorStack.pop()
    Append topOperator to postfix
}
return postfix
```

Infix: 1 - 2 ^ 3 ^ 3 - (4 + 5 * 6) * 7
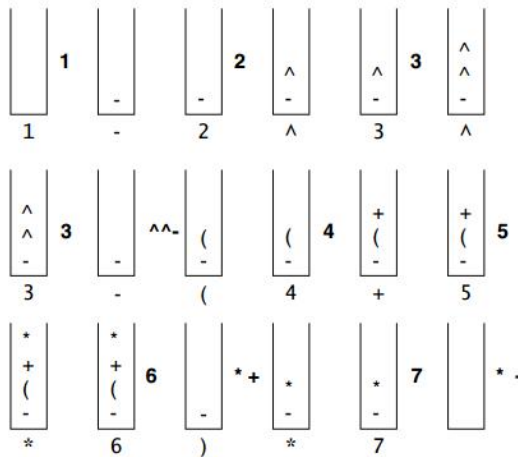
Postfix: 1 2 3 3 ^ ^ - 4 5 6 * + 7 * -

H.W. example:

Evaluating Postfix Expressions

- When an **operand** is seen, it is **pushed** onto a stack.
- When an **operator** is seen, the appropriate numbers of **operands** are **popped** from the stack, the operator is **evaluated**, and the result is **pushed** back onto the stack.
    - Note that the **1ˢᵗ** item popped becomes the **rhs** parameter to the binary operator and that the **2ⁿᵈ** item popped is the **lhs** parameter; thus **parameters are popped in reverse order**. For multiplication, the order does not matter, but for subtraction and division, it does.
- When the complete postfix expression is evaluated, the result should be a single item on the stack that represents the answer.

Example 1: The stack during the evaluation of the postfix expression *a b /* when *a* is **2** and *b* is 4
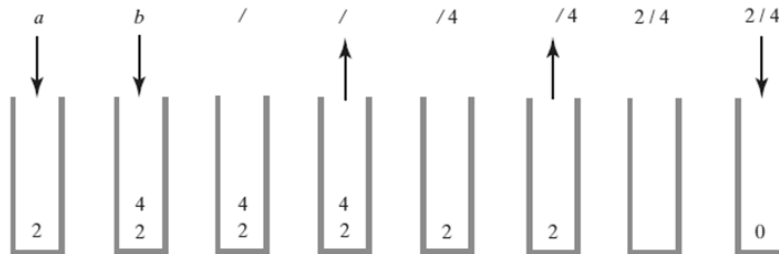
Example 2: The stack during the evaluation of the postfix expression *a b + c /* when *a* is *2*, *b* is *4*, and *c* is *3*

45
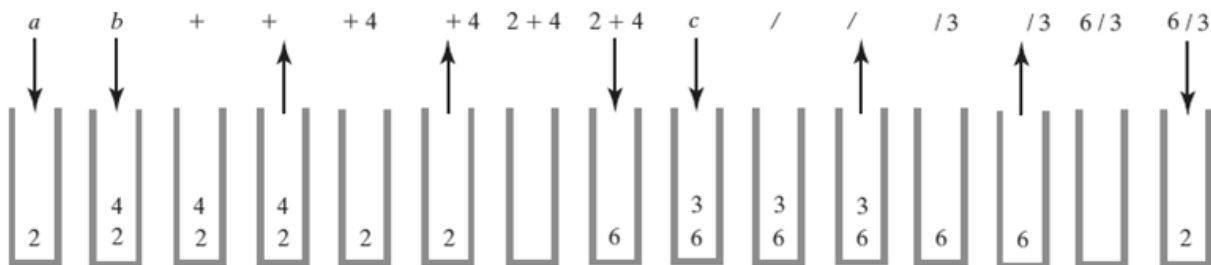
Algorithm for evaluating postfix expressions.

```
Algorithm evaluatePostfix(postfix)
//  Evaluates a postfix expression.

valueStack = a new empty stack
while (postfix has characters left to parse)
{
    nextCharacter = next nonblank character of postfix
    switch (nextCharacter)
    {
      case variable:
          valueStack.push(value of the variable nextCharacter)
          break
      case '+' : case '-' : case '*' : case '/' : case '^' :
          operandTwo = valueStack.pop()
          operandOne = valueStack.pop()
          result = the result of the operation in nextCharacter and its operands
                     operandOne and operandTwo
          valueStack.push(result)
          break
      default: break // Ignore unexpected characters
    }
}
```

H.W. Example:

Postfix Expression: 1 2 - 4 5 ^ 3 * 6 * 7 2 2 ^ ^ / -

Iteration (optional)

- **Design challenge**. Support iteration over stack items by client, without revealing the internal representation of the stack.
- **Java solution**. Make stack implement the **java.lang.Iterable** interface.

Iterable interface

Q. What is an `Iterable` ?
A. Has a method that returns an `Iterator`.

```
public interface Iterable<Item>
{
    Iterator<Item> iterator();
}
```

Iterator interface

Q. What is an `Iterator` ?
A. Has methods `hasNext()` and `next()`.

Q. Why make data structures `Iterable` ?
A. Java supports elegant client code.

```
public interface Iterator<Item>
{
    boolean hasNext();
    Item next();
    void remove();        optional; use
}                                    at your own risk
```

"foreach" statement (shorthand)

```
for (String s : stack)
    StdOut.println(s);
```

equivalent code (longhand)

```
Iterator<String> i = stack.iterator();
while (i.hasNext())
{
    String s = i.next();
    StdOut.println(s);
}
```

```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    ...

    public Iterator<Item> iterator() { return new ListIterator(); }

    private class ListIterator implements Iterator<Item>
    {
        private Node current = first;

        public boolean hasNext() {  return current != null;  }
        public void remove()     {  /* not supported */       }
        public Item next()
        {
            Item item = current.item;
            current    = current.next;
            return item;
        }
    }
}
```

throw UnsupportedOperationException

throw NoSuchElementException
if no more items in iteration

first   current

times → of → best → the → was → it → null

(Lecture 12) Queues

HEAD
First Front

TAIL
Last Back

FIFO : First In First Out
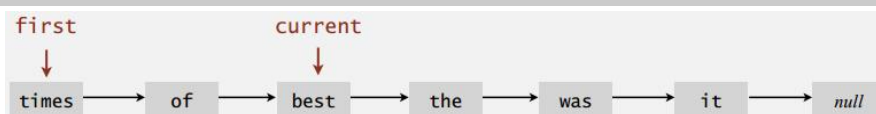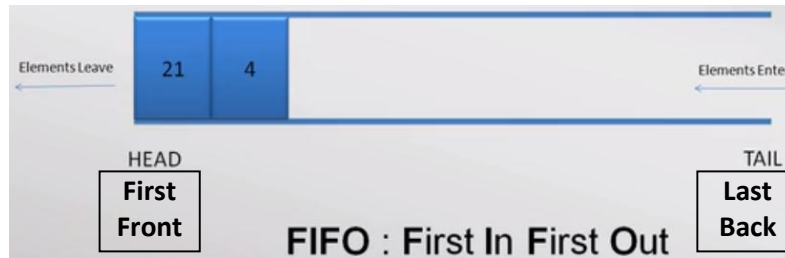
ENQUEUE

Inserts an element in the queue from the tail towards the head

DEQUEUE

Removes an element in the queue from the head of the queue

UML	DESCRIPTION
+enqueue(newEntry: integer): void	Task: Adds a new entry to the back of the queue.
+dequeue(): T	Task: Removes and returns the entry at the front of the queue.
+getFront(): T	Task: Retrieves the queue's front entry without changing the queue in any way.
+isEmpty(): boolean	Task: Detects whether the queue is empty.
+clear(): void	Task: Removes all entries from the queue.

Linked-list Representation of a Queue

Maintain pointer to first (head) and last (tail) nodes in a linked list;
insert/remove from opposite ends.
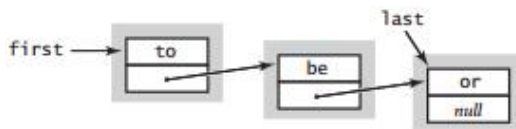
Delete dequeue:
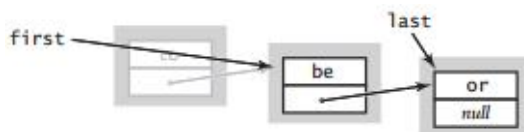
Add enqueue:

save a link to the last node

Node oldlast = last;

48
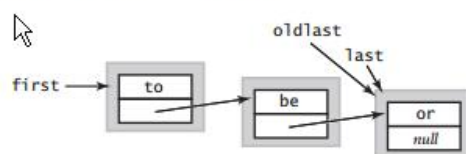
create a new node for the end

```
last = new Node();
last.item = "not";
```
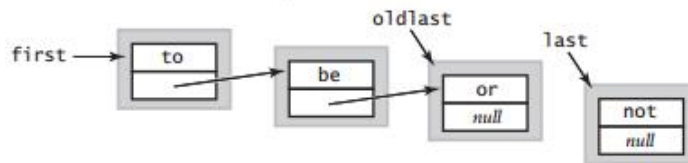

link the new node to the end of the list

```
oldlast.next = last;
```
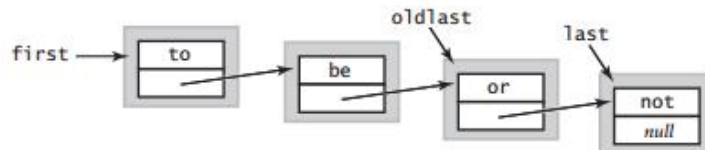


```java
public class LinkedQueueOfStrings
{
   private Node first, last;

   private class Node
   {  /* same as in StackOfStrings */  }

   public boolean isEmpty()
   {  return first == null;  }

   public void enqueue(String item)
   {
      Node oldlast = last;
      last = new Node();
      last.item = item;
      last.next = null;
      if (isEmpty()) first = last;
      else           oldlast.next = last;
   }

   public String dequeue()
   {
      String item = first.item;
      first       = first.next;
      if (isEmpty()) last = null;
      return item;
   }
}
```

special cases for
empty queue

Array implementation of a Queue.
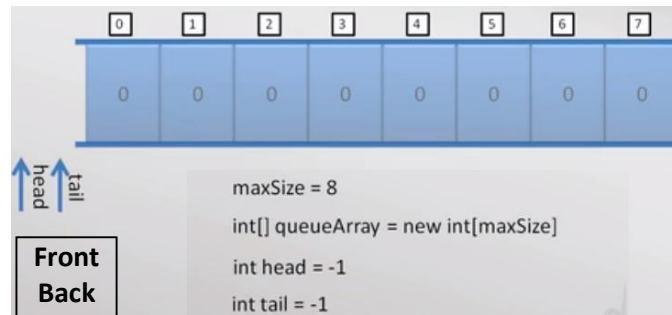
- **enqueue**(): add new item at q[tail] .
- **dequeue**(): remove item from q[head] .

enqueue(8)

enqueue (12)

After a number of enqueues:

dequeue(): returns the item pointed by head and advances head pointer

enqueue (27) ?? how to advance tail?? We have space at the beginning?? Shift??
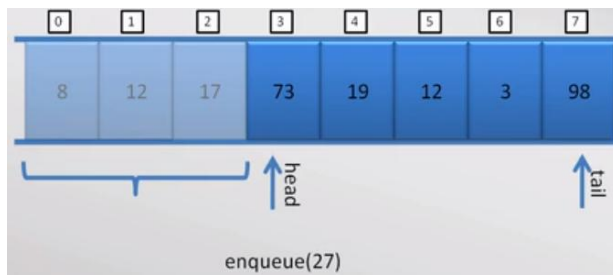
How to find free spaces??

$$\text{Math.abs(Tail Index} - \text{Head Index)} \; < \; \text{Length of array} \; -1$$

So, if tail at max index and we have free spaces, we move tail to 1st index. ➔ **Circular Queue**
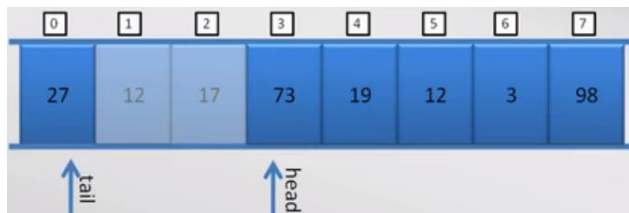
enqueue (9) ??
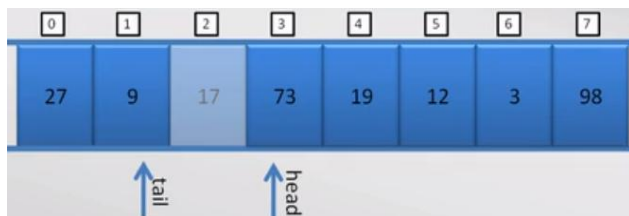
Circular Queue
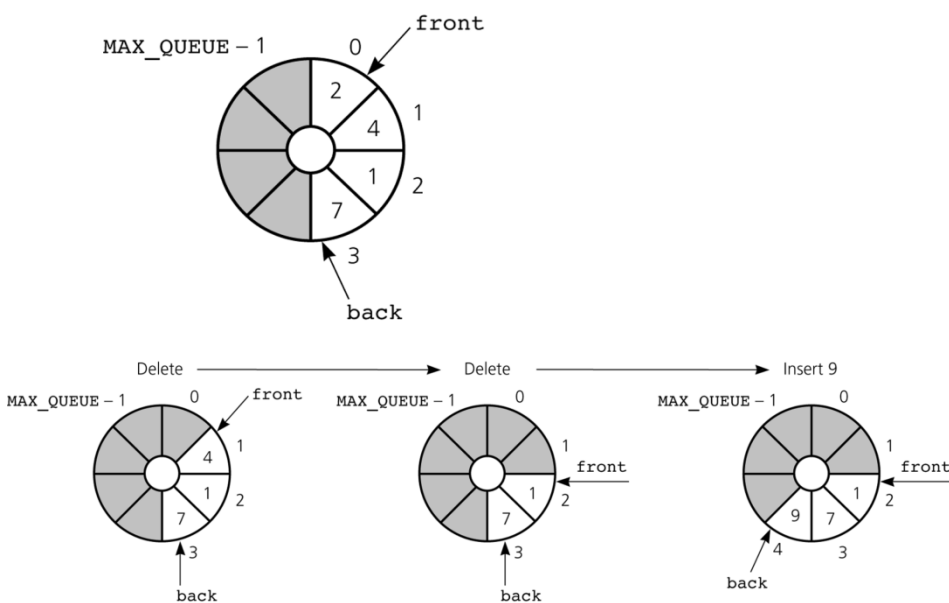
Queue with single item ──────► Delete item—queue becomes empty

(a)
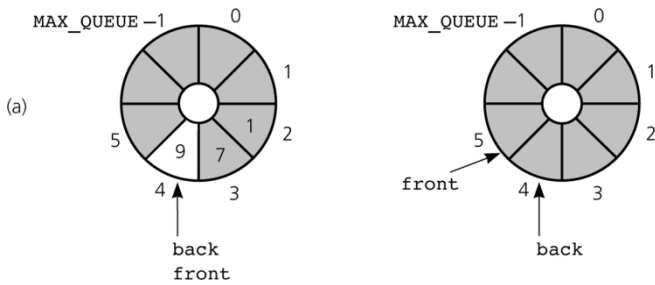
Queue with single empty slot ──────► Insert 9—queue becomes full

(b)
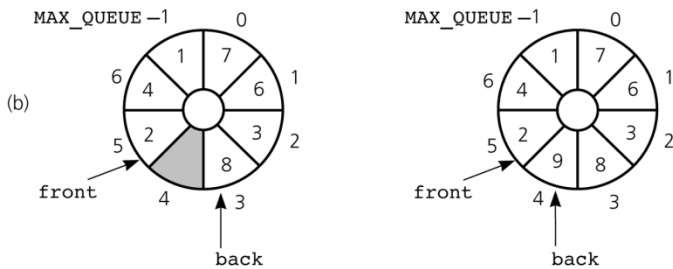
- **To detect queue-full and queue-empty conditions**
  − Keep a count of the queue items
- **To initialize the queue, set**
  − front to -1
  − back to -1
  − count to 0

Inserting into a queue

        If(count < MAX_QUEUE) // free
                back = (back+1) % MAX_QUEUE;
                items[back] = newItem;
                ++count;
                If(count==1) // first item
                        front = back;

Deleting from a queue

        If(count > 0) // not empty
                front = (front+1) % MAX_QUEUE;
                --count;
                If(count==0) // empty
                        front = back = -1

DE Queue (Double Ended Queue)

Allows add / remove elements from both head/tail.

==**HW This of implementations using linked List and Arrays.**==

(Lecture 13) Cursor Implementation of Linked Lists

Many Languages do not support pointers.

If data max length is known, using Array is faster

Solution ➔ Cursor Implementation

2 features present in a pointer implementation of linked lists:

- The data are stored in array, each array element contains data and a pointer to the next structure.
- A new structure can be obtained from the system's global memory by a call to *malloc* and released by a call to *free*.

To Be Completed

(Lecture 14) Trees

Sorted Arrays	Linked List
Search : Fast (O(logn))	Search : Slow (O(n))
Insert : Slow (O(n))	Insert : Fast (O(1))
Delete : Slow (O(n))	Delete : Fast (O(1))

Tree
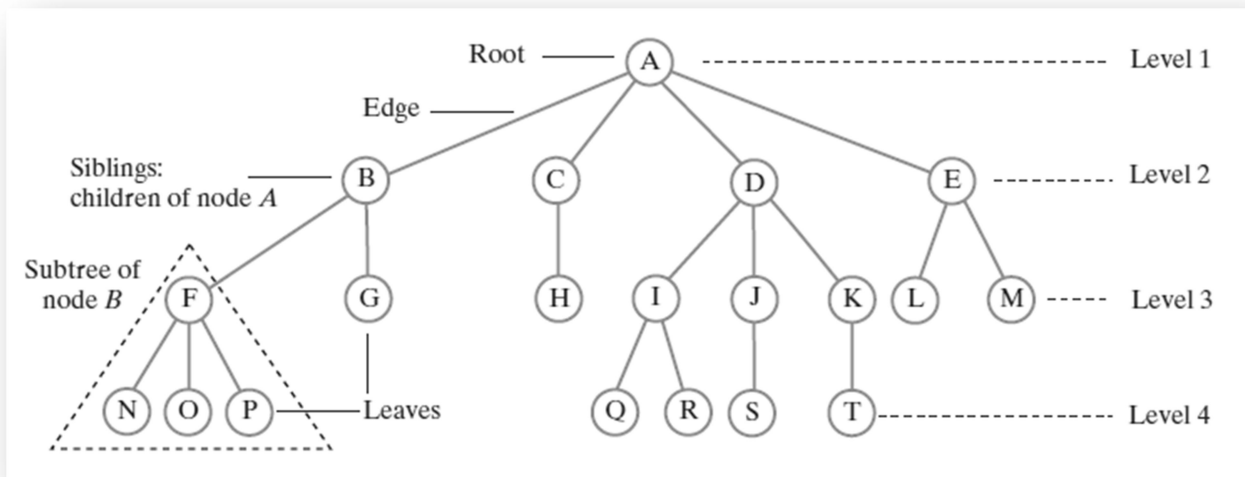
- A **tree** is a collection of *N* **nodes**, one of which is the **root**, and *N* − 1 **edges**.
- Every node except the **root** has one **parent**.
- Nodes with no children are known as **leaves**.
- An **internal node (parent)** is any node that has at least one non-empty child.
- Nodes with the same parent are **siblings**.
- The *depth of a node* in a tree is the length of the path from the **root** to the node.
- The *height* of a tree is the number of levels in the tree.

<mark>Example: Family Trees (one parent)</mark>
<mark>Example: file system tree</mark>

Binary Trees

- A **binary tree** is a tree in which no node can have more than __two__ children.
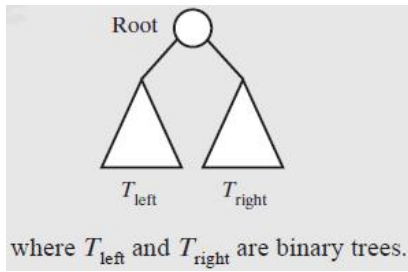
where T_{left} and T_{right} are binary trees.   **Binary Tree Node:**
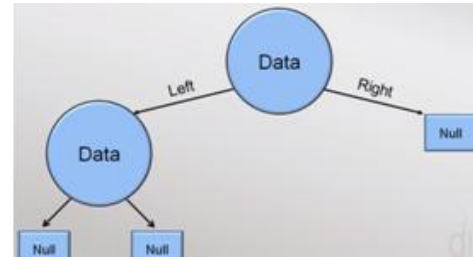
- Each node in a **full binary tree** is either:

  (1) an internal node with exactly two non-empty children or

  (2) a leaf.

- A **complete binary tree** has a restricted shape obtained by starting at the root and filling the tree by levels from **left** to **right**.
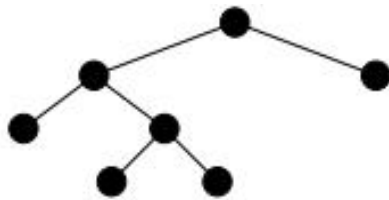
(a) This tree is full
(but not complete).
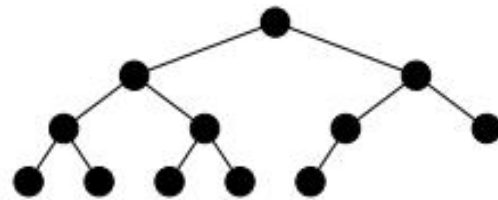
(b) This tree is complete
(but not full).

- The max. number of nodes in a full binary tree as a function of the tree's height **$= 2^h - 1$**

Full Tree	Height	Number of Nodes
	1	$1 = 2^1 - 1$
	2	$3 = 2^2 - 1$
	3	$7 = 2^3 - 1$

Implementation:

```java
public class TreeNode {
    private Integer data;
    private TreeNode leftChild;
    private TreeNode rightChild;
    public TreeNode(Integer data)          {     this.data = data;        }
    public Integer getData()               {     return data;             }
    public TreeNode getLeftChild()         {     return leftChild;        }
    public void setLeftChild(TreeNode left)   {     this.leftChild = left;    }
    public TreeNode getRightChild()        {     return rightChild;       }
    public void setRightChild(TreeNode right)  {     this.rightChild = right;}
}
public class BinaryTree {
    private TreeNode root;
    public void insert(Integer data)       {   }
    public TreeNode find(Integer data)  {     return null;   }
    public void delete(Integer data)       {   }
}
```

Tree Traversal

Definition: visit, or process, each data item exactly once.

In-Order Traversal:

1. Traverse the left sub tree.
2. Visit the root.
3. Traverse the right sub tree.

33   52   65

12   25   27   33   34   39   48   52   60   65   72   78   90
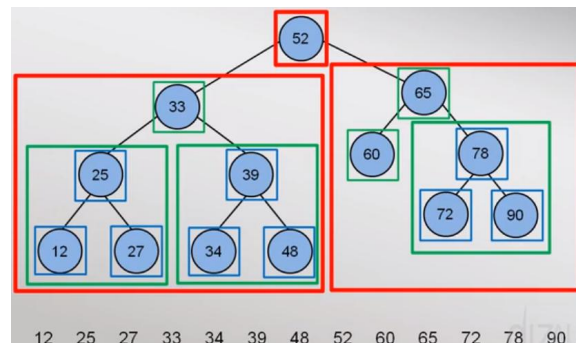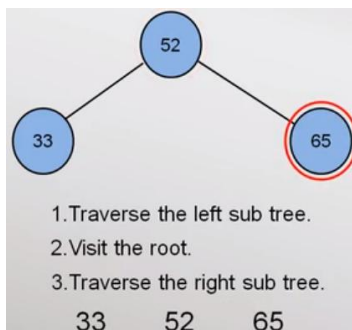
@ TreeNode

```java
public void traverseInOrder() {
    if (this.leftChild != null)
        this.leftChild.traverseInOrder();
    System.out.print(this + " ");
    if (this.rightChild != null)
        this.rightChild.traverseInOrder();
}
```

@BinarySerachTree

```java
public void traverseInOrder() {
    if (this.root != null)
        this.root.traverseInOrder();
    System.out.println();
}
```

Pre-Order Traversal

1. Visit the root.
2. Traverse the left sub tree.
3. Traverse the right sub tree.

52   33   65

52   33   25   12   27   39   34   48   65   60   78   72   90

Post-Order Traversal

1. Traverse the left sub tree.
2. Traverse the right sub tree.
3. Visit the root.

33   65   52

12   27   25   34   48   39   33   60   72   90   78   65
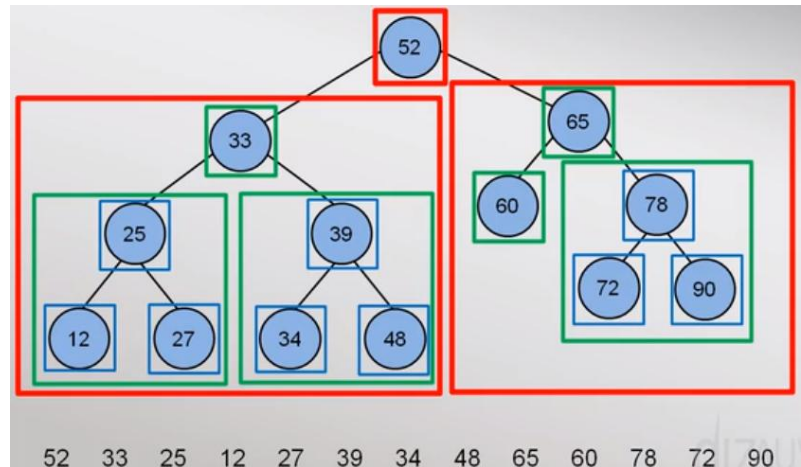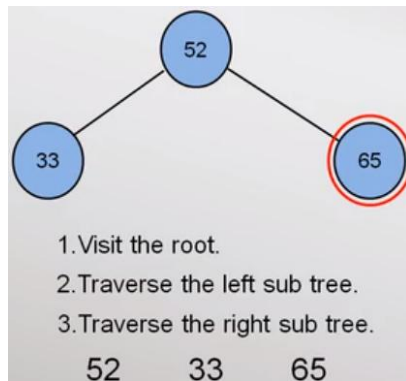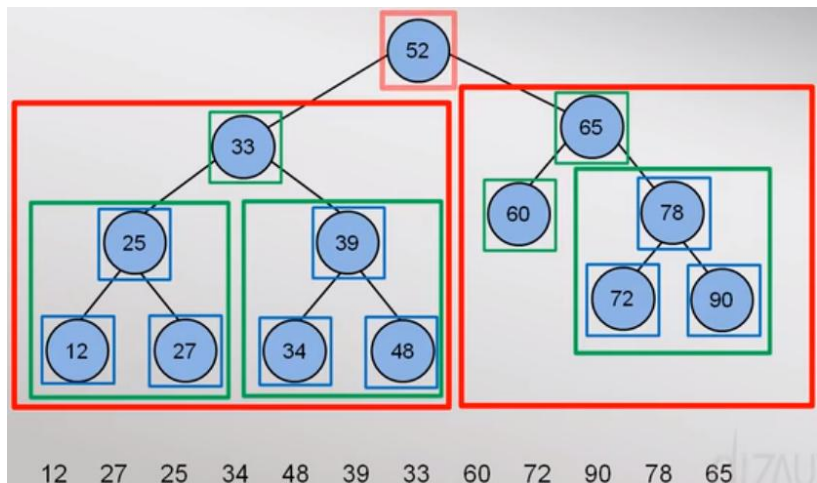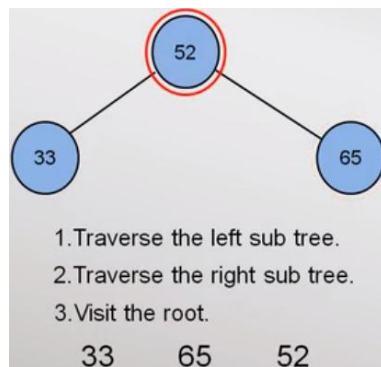
Level-Order Traversal   (Optional)

- Begin at root and visit nodes one level at a time
- Level-order traversal is implemented via a **queue**.
- The traversal is a breadth-first search.

HW: implement level-order traversal

(Lecture 15) Expression Trees

(a) a / b (b) $a * b + c$ (c) $a * (b + c)$ (d) $a * (b + c * d) / e$
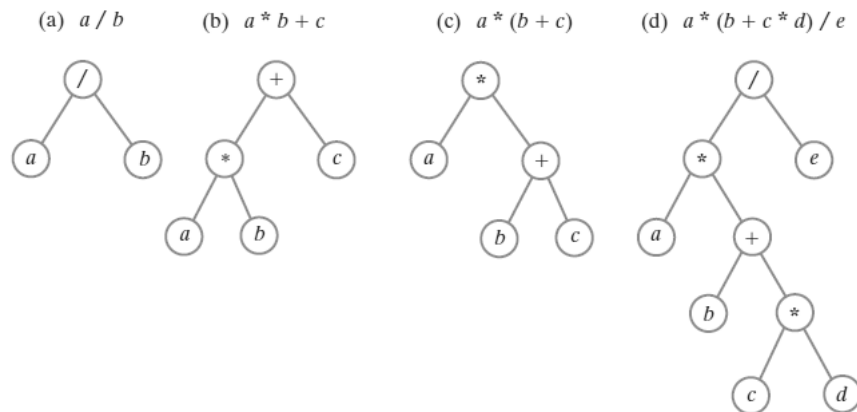
- The leaves of an expression tree are **operands**, such as **constants** or **variable** names, and the other nodes contain **operators**.
- It is also possible for a node to have only one child, as is the case with the **unary minus** operator.
- We can evaluate an expression tree by applying the **operator** at the **root** to the values obtained by recursively evaluating the **left** and **right** subtrees.

Algebraic expressions:

- Algebraic expression trees represent expressions that contain numbers, variables, and unary and binary operators.
- Some of the common operators are × (multiplication), ÷ (division), **+** (addition), **−** (subtraction), **^** (exponentiation), and **-** (negation).

Example: **((5 + z) / -8) * (4 ^ 2)**
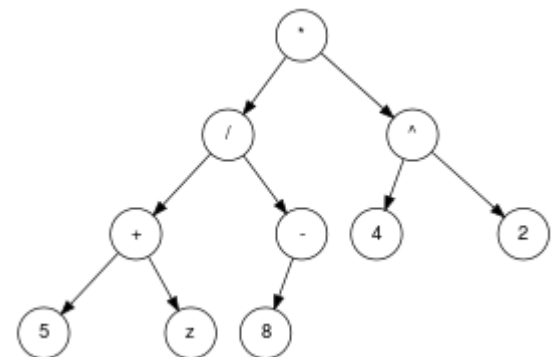
Boolean expressions:

- Boolean expressions are represented very similarly to algebraic expressions, the only difference being the specific values and operators used.
- Boolean expressions use **true** and **false** as constant values, and the operators include **∧ (AND), ∨ (OR), ~ (NOT).**
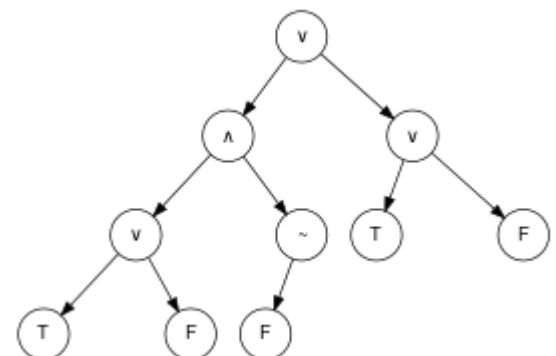
Algorithm for evaluation of an expression tree:

```
Algorithm evaluate(expressionTree)
if (expressionTree is empty)
    return 0
else
{
    firstOperand = evaluate(left subtree of expressionTree)
    secondOperand = evaluate(right subtree of expressionTree)
    operator = the root of expressionTree
    return the result of the operation operator and its operands firstOperand
            and secondOperand
}
```

Constructing an expression tree:

The construction of the expression tree takes place by reading the **postfix** expression one symbol at a time:

- If the symbol is an **operand**, one-node tree is created and a pointer is pushed onto a **stack**.
- If the symbol is an **operator**,
  - Two pointers trees T1 and T2 are popped from the stack
  - A new tree whose root is the **operator** and whose left and right children point to T2 and T1 respectively is formed .
  - A pointer to this new tree is then pushed to the Stack.

Example: (a b + c d e + * *)

- Since the first two symbols are operands, one-node trees are created and pointers are pushed to them onto a stack.
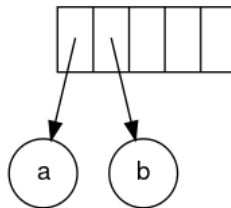
- The next symbol is a '**+**'. It pops two pointers, a new tree is formed, and a pointer to it is pushed onto to the stack.
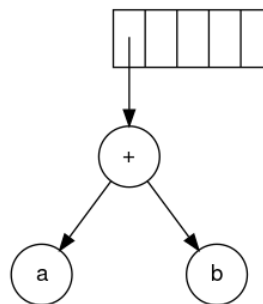
- Next, **c**, **d**, and **e** are read. A one-node tree is created for each and a pointer to the corresponding tree is pushed onto the stack.
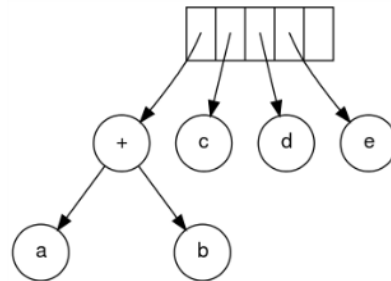
- Continuing, a '**+**' is read, and it merges the last two trees.
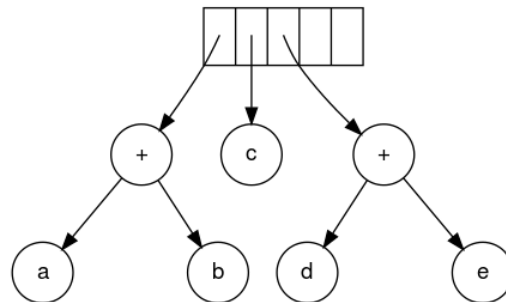
- Now, a '*****' is read. The last two tree pointers are popped and a new tree is formed with a '*****' as the root.
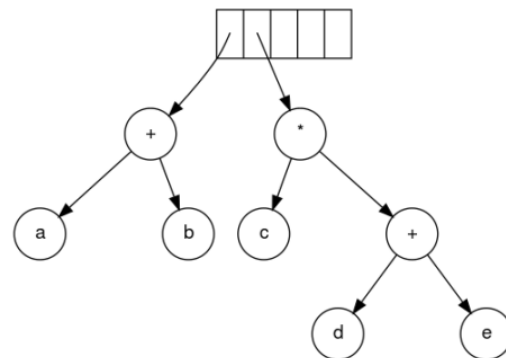
- Finally, the last symbol is read. The two trees are merged and a pointer to the final tree remains on the stack.
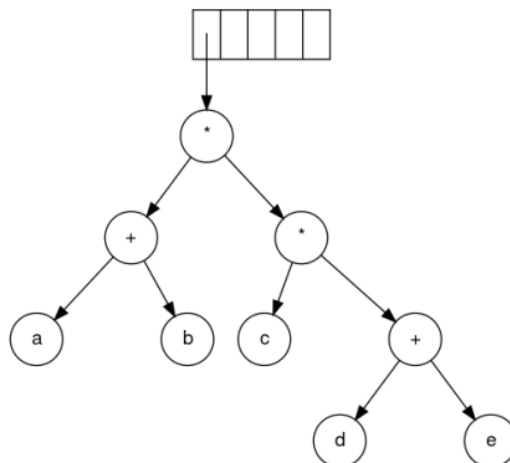
(Lecture 16) Binary Search Trees BST
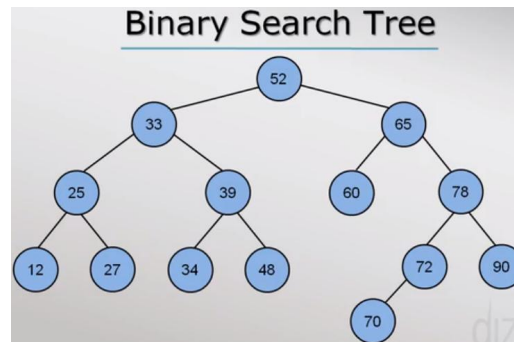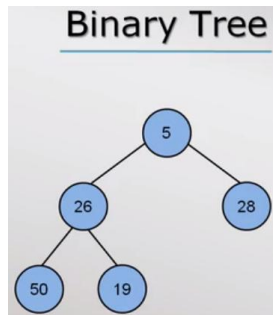
- In a **binary search tree** for every node , **X**, in the tree, the values of all the items in its **left subtree** are smaller than the item in **X**, and the values of all the items in its **right subtree** are larger (<u>**or equal**</u>) than the item in **X**.
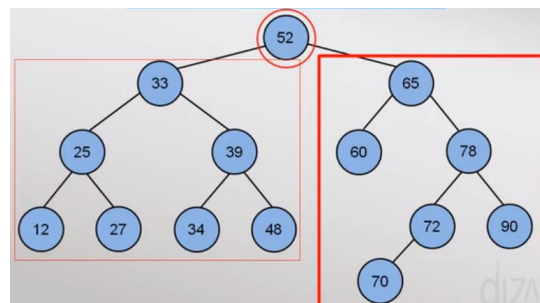
Search for an item: **Find(52)**   **,**   **Find(39)**   **,**   **Find(35)**

@ TreeNode

```
public TreeNode find(Integer data) {
        if (this.data == data)
                return this;
        if (data < this.data && leftChild != null)
                return leftChild.find(data);
        if (rightChild != null)
                return rightChild.find(data);
        return null;
}
```

@BinarySerachTree

```
public TreeNode find(Integer data) {
        if (root != null)
                return root.find(data);
        return null;
}
```

Efficiency of a search: Searching a binary search tree of height **h** is **O(h)**

To make searching a binary search tree as efficient as possible … Tree must be as **short** as possible.

Finding Max and Min Values
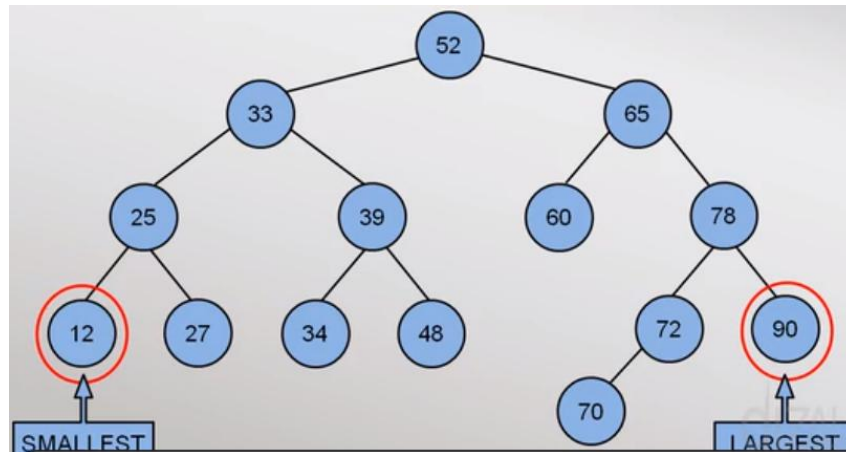
- The find **Min** operation is performed by following left nodes as long as there is a left child.
- The find **Max** operation is similar.

@TreeNode

```
public Integer largest() {
        if (this.rightChild == null)
                return this.data;
        return this.rightChild.largest();
}


public Integer smallest() {
        if (this.leftChild == null)
                return this.data;
        return this.leftChild.smallest();
}
```

@BinarySerachTree

```
public Integer largest() {
        if (this.root != null)
                return root.largest();
        return null;
}

public Integer smallest() {
        if (this.root != null)
                return root.smallest();
        return null;
}
```

Insert in Binary Search Tree

Insert(63)
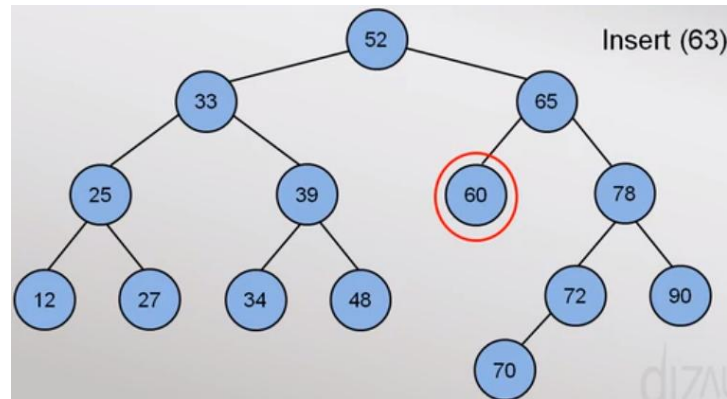
@TreeNode

```java
public void insert(Integer data) {
        if (data >= this.data) {          // insert in right subtree
                if (this.rightChild == null)
                        this.rightChild = new TreeNode(data);
                else
                        this.rightChild.insert(data);
        } else {          // insert in left subtree
                if (this.leftChild == null)
                        this.leftChild = new TreeNode(data);
                else
                        this.leftChild.insert(data);
        }
}
```

@BinarySerachTree

```java
public void insert(Integer data) {
        if (root == null)
                this.root = new TreeNode(data);
        else
                root.insert(data);
}
```

Deleting a Node

Case 1: Node to be deleted is a leaf.
Case 2: Node to be deleted has one child.
Case 3: Node to be deleted has two children.
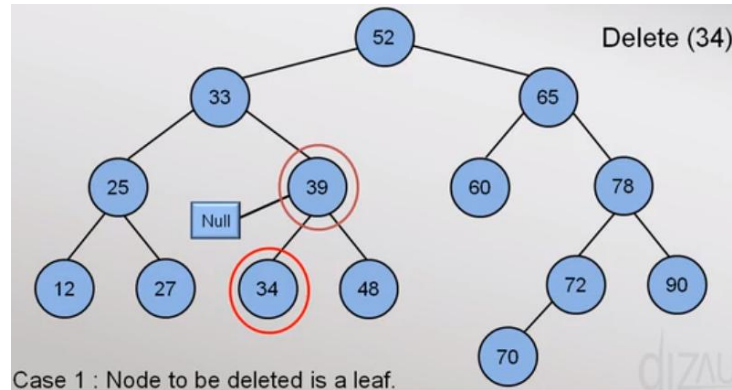
@BinarySerachTree

```
public void delete(Integer data) {
        TreeNode current = this.root;
        TreeNode parent = this.root;
        boolean isLeftChild = false;

        if (current == null)     return;     // tree is empty

        while (current != null && current.getData() != data) {
                parent = current;
                if (data < current.getData()) {
                        current = current.getLeftChild();
                        isLeftChild = true;
                } else {
                        current = current.getRightChild();
                        isLeftChild = false;
                }
        }
        if (current == null)     return;    // node to be deleted not found

        // this is case 1
        if (current.getLeftChild() == null && current.getRightChild() == null) {
                if (current == root) {   root = null;  // no elements in tree now
                } else {
                        if (isLeftChild)
                                parent.setLeftChild(null);
                        else
                                parent.setRightChild(null);
                }
        }
    }
```
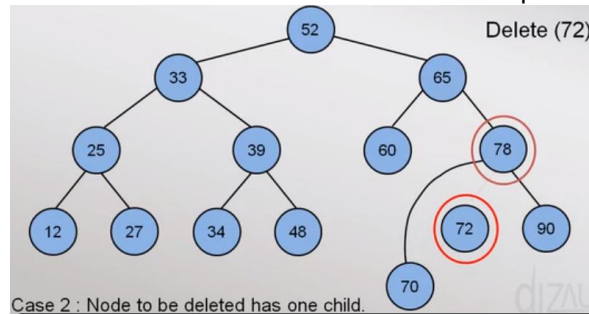
If a node has one child, it can be removed by having its parent bypass it.
Note: The **root** is a special case because it does not have a parent.

@BinarySerachTree

```
// This is case 2 broken down further into 2 separate cases
else if (current.getRightChild() == null) {      // current has left child
    if (current == root) {
        root = current.getLeftChild();
    } else if (isLeftChild) {
        parent.setLeftChild(current.getLeftChild());
    } else {
        parent.setRightChild(current.getLeftChild());
    }
} else if (current.getLeftChild() == null) {      // current has right child
    if (current == root) {
        root = current.getRightChild();
    } else if (isLeftChild) {
        parent.setLeftChild(current.getRightChild());
    } else {
        parent.setRightChild(current.getRightChild());
    }
}
```
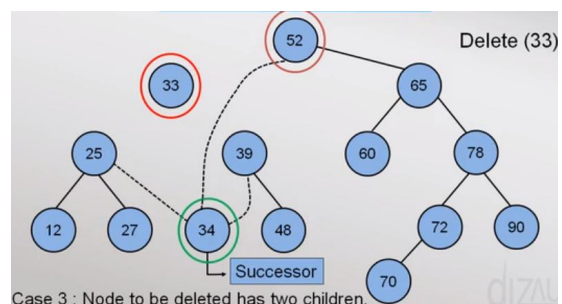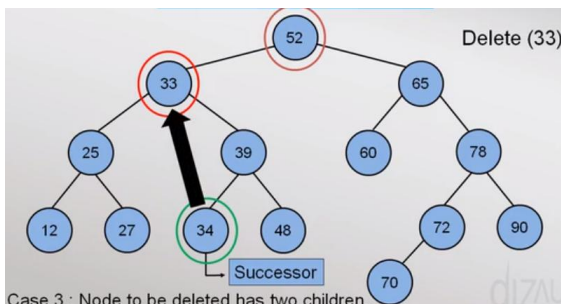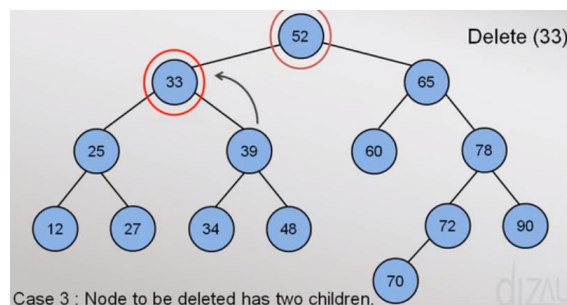
A node with two children is replaced by using the **smallest** item in the right subtree (**Successor**). Then another node is removed.
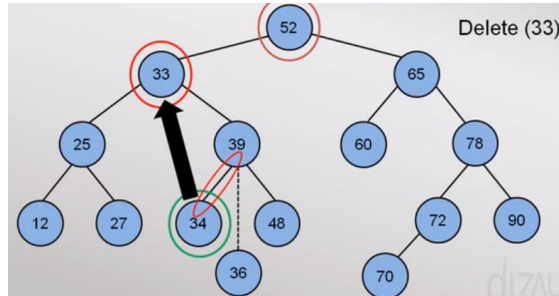
What if **34** has a right child?

@BinarySerachTree

```
// This is case 3 - Most complicated (node to be deleted has 2 children)
else {
        TreeNode successor = getSuccessor(current);
        if (current == root)
                root = successor;
        else if (isLeftChild) {
                parent.setLeftChild(successor);
        } else {
                parent.setRightChild(successor);
        }
        successor.setLeftChild(current.getLeftChild());
}



private TreeNode getSuccessor(TreeNode node) {
        TreeNode parentOfSuccessor = node;
        TreeNode successor = node;
        TreeNode current = node.getRightChild();
        while (current != null) {
                parentOfSuccessor = successor;
                successor = current;
                current = current.getLeftChild();
        }
        if (successor != node.getRightChild()) {
                parentOfSuccessor.setLeftChild(successor.getRightChild());
                successor.setRightChild(node.getRightChild());
        }
        return successor;
}
```

Soft Delete (lazy deletion): When an element is to be deleted, it is left in the tree and merely **marked** as being deleted.
- If a deleted item is reinserted, the overhead of allocating a new cell is avoided.

Tree Height

```
@BinarySerachTree
public int height() {
        if (this.root == null) return 0;
        return this.root.height();
}


@TreeNode
public int height() {
        if (isLeaf()) return 1;
        int left = 0;
        int right = 0;
        if (this.leftChild != null)
                left = this.leftChild.height();
        if (this.rightChild != null)
                right = this.rightChild.height();
        return (left > right) ? (left + 1) : (right + 1);
}
```

Efficiency of Operations

- For tree of height **h**
  - The operations **add**, **remove**, and **getEntry** are **O(h)**
- If tree of **n** nodes has height **h = n**
  - These operations are **O(n)**
- Shortest tree is **full**
  - Results in these operations being **O(log n)**

Unbalanced Tree
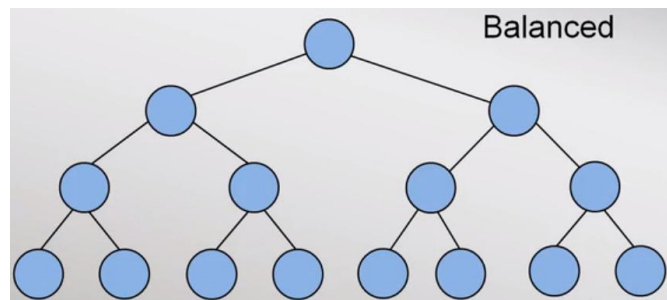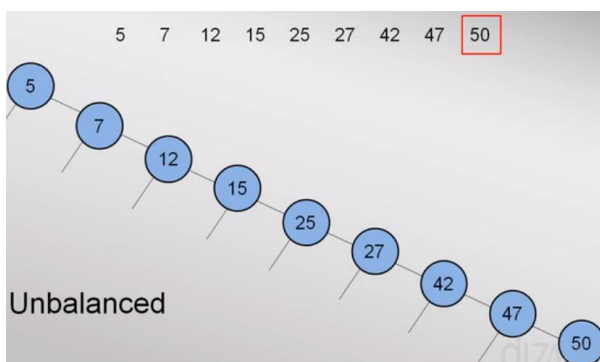
- The order in which you add entries to a binary search tree affects the shape of the tree.

- An **AVL tree** is a **BST** with the additional **balance** property that, for any node in the tree, the height of the **left** and **right** subtrees can differ by at most **1**.
- **Complete** binary trees are **balanced.**

Single Rotations

Example: After inserting (a) 60; (b) 50; and (c) 20 into an initially empty **BST**, the tree is **not balanced**;
(d) a corresponding **AVL** tree rotates its nodes to restore balance
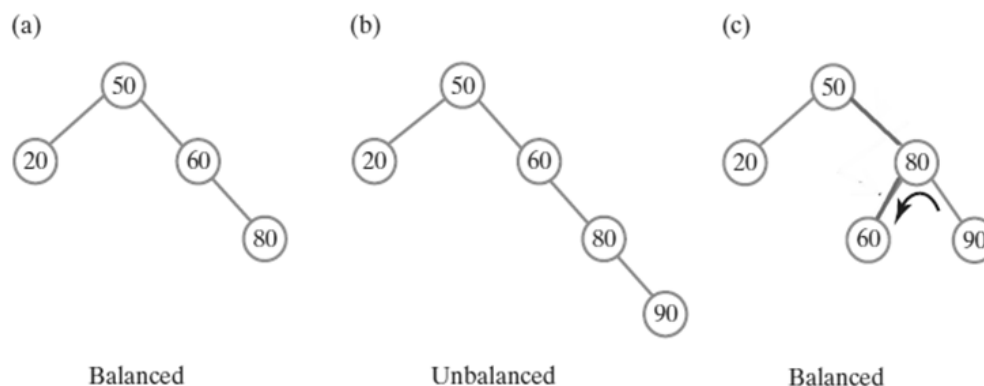
Example: (a) Adding 80 to the tree does not change the balance of the tree;
(b) a subsequent addition of 90 makes the tree **unbalanced** ;
(c) a left rotation restores its balance

Case 1: Single Right Rotation
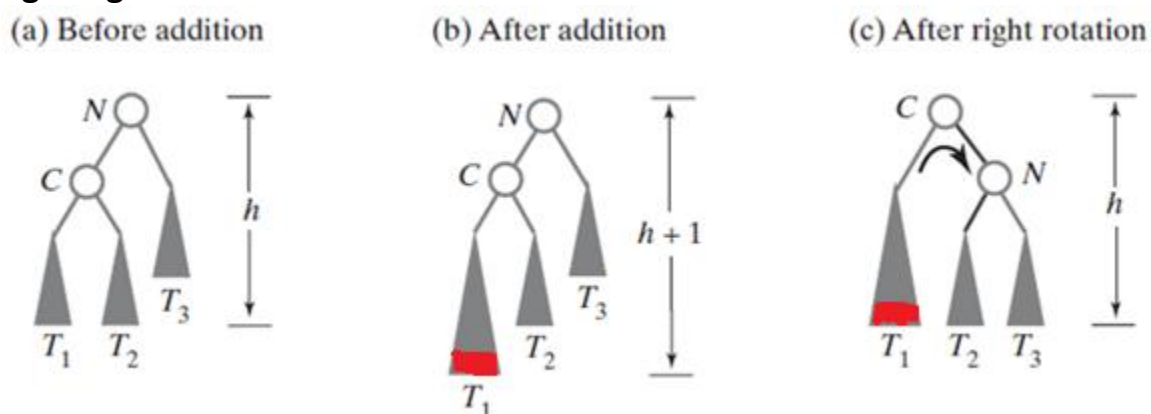
Before and after an addition to an **AVL** subtree that requires a **right rotation** to maintain its balance.

68
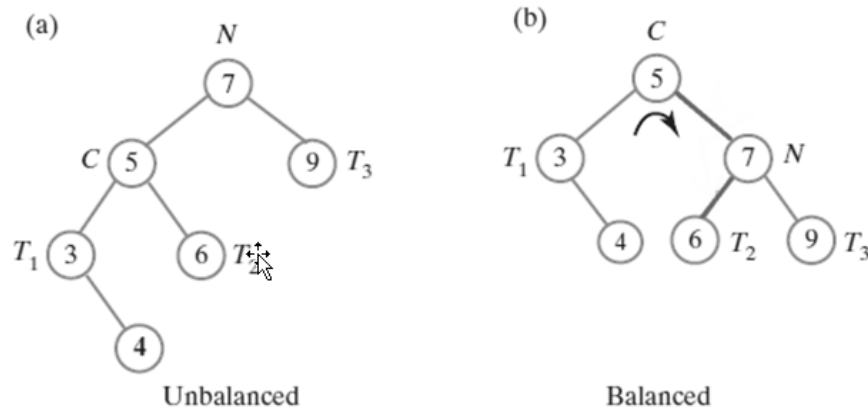
Example: Before and after a **right rotation** restores balance to an **AVL** tree

```
Algorithm rotateRight(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the left subtree of nodeN's left child.

nodeC = left child of nodeN
Set nodeN's left child to nodeC's right child
Set nodeC's right child to nodeN
return nodeC
```
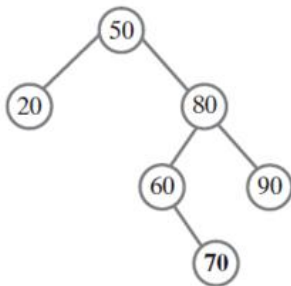
Case 2: Single Left Rotation

Before and after an addition to an **AVL** subtree that requires a **left rotation** to maintain its balance

```
Algorithm rotateLeft(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the right subtree of nodeN's right child.

nodeC = right child of nodeN
Set nodeN's right child to nodeC's left child
Set nodeC's left child to nodeN
return nodeC
```

Double Rotations

A **double rotation** is accomplished by performing two single rotations:
1. A rotation about node **N's grandchild G** (its child's child)
2. A rotation about node **N's new child**

Case 3: Right-Left Double Rotations

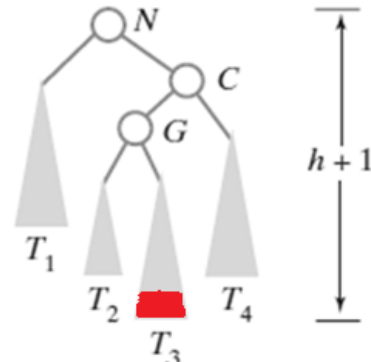(a) After adding 70   (b) After right rotation   (c) After left rotation
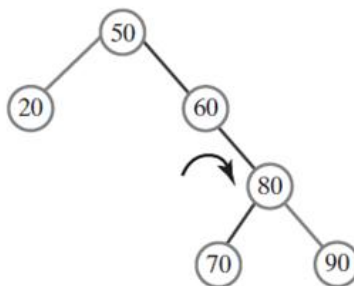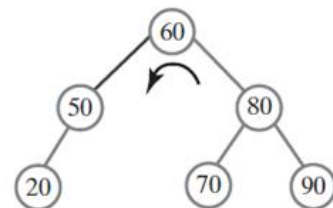
Example: (a) Adding 70 destroys tree's balance; to restore the balance, perform both
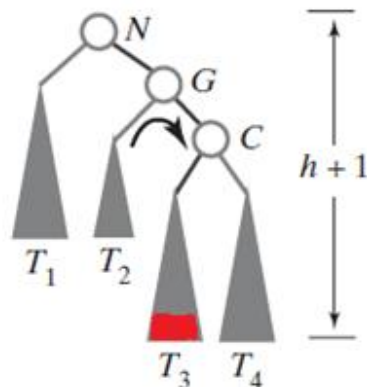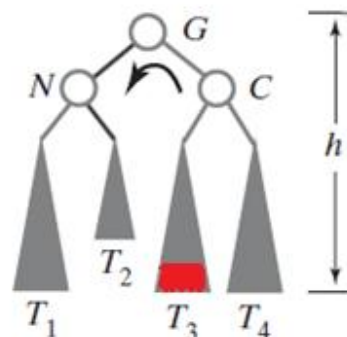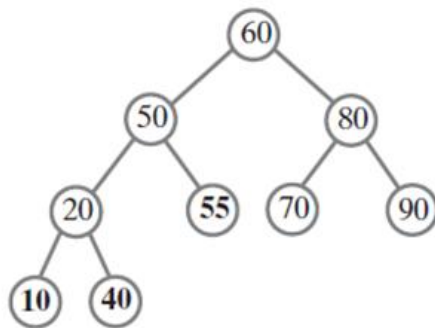(b) a **right rotation** and (c) a **left rotation**
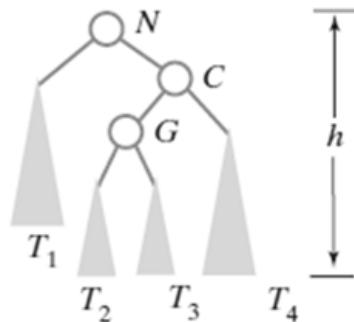
(a) Before addition   (b) After addition

(c) After right rotation   (d) After left rotation

Before and after an addition to an **AVL** subtree that requires both
a **right rotation** and a **left rotation** to maintain its balance

```
Algorithm rotateRightLeft(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the left subtree of nodeN's right child.

nodeC = right child of nodeN
Set nodeN's right child to the node returned by rotateRight(nodeC)
return rotateLeft(nodeN)
```
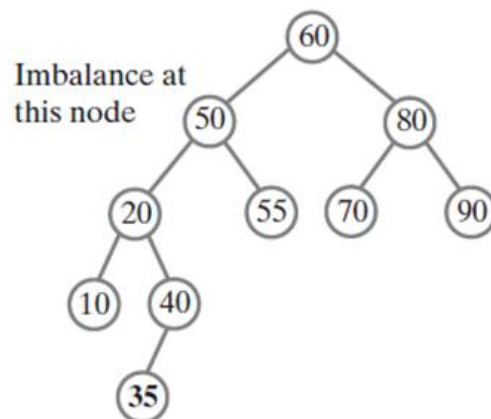
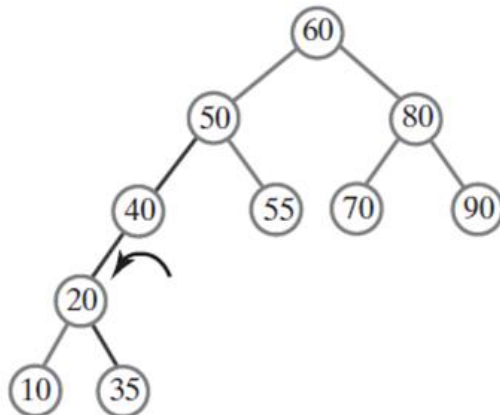## Case 4: Left-Right Double Rotations
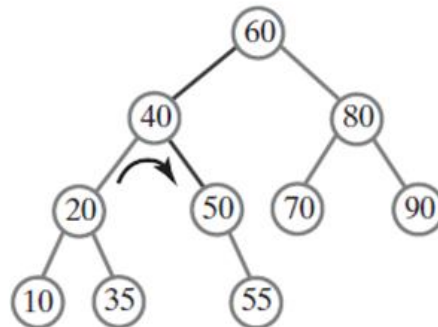**Example:**

(a) After adding 55, 10, and 40

(b) After adding 35

(c) After left rotation about 40

(d) After right rotation about 40

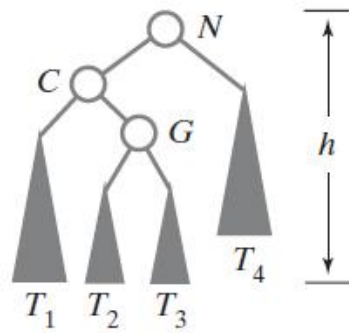(a) The **AVL** tree after additions that maintain its balance;
(b) after an addition that destroys the balance;
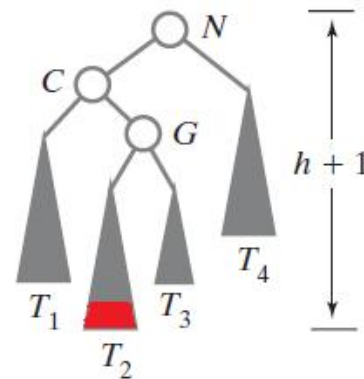(c) after a **left rotation**;
(d) after a **right rotation**
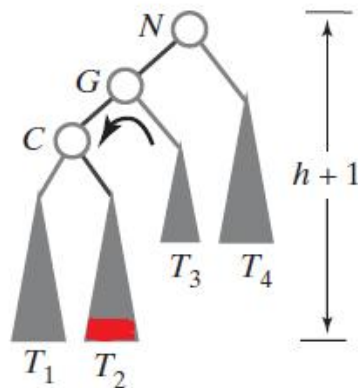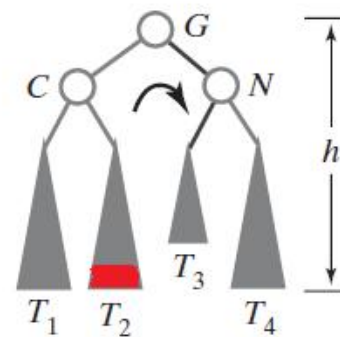
71

(a) Before addition   (b) After addition

(c) After left rotation   (d) After right rotation

Before and after an **addition** to an **AVL** subtree that requires both
a **left rotation** and a **right rotation** to maintain its balance

Algorithm rotateLeftRight(nodeN)
// *Corrects an imbalance at a given node* nodeN *due to an addition*
// *in the right subtree of* nodeN*'s left child.*

nodeC = *left child of* nodeN
Set nodeN*'s left child to the node returned by* rotateLeft(nodeC)
return rotateRight(nodeN)

- Four rotations cover the only four possibilities for the cause of the imbalance at node **N**
- The addition occurred at:
  - The left subtree of **N**'s left child (case 1: right rotation)
  - The right subtree of **N**'s left child (case 4: left-right rotation)
  - The left subtree of **N**'s right child (case 3: right-left rotation)
  - The right subtree of **N**'s right child (case 2: left rotation)