BIRZEIT UNIVERSITY

COMP232

Data Structure

Lectures Note 1

Prepared by: **Dr. Mamoun Nawahdah**

2016

Table of Contents

Math Review

1. $\log(nm) = \log n + \log m$.
2. $\log(n/m) = \log n - \log m$.
3. $\log(n^r) = r \log n$.
4. $\log_a n = \log_b n / \log_b a$.

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

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

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

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

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

and

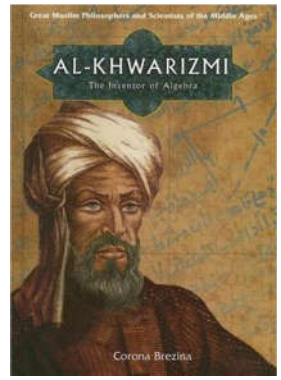
$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1.$$

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

Finally,

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

What is an Algorithm?



Born: Uzbekistan
Died: 850 AD, Baghdad, Iraq

Definition:

- **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 (300_{BC})

- 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:

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

Implement this iteratively and recursively

```
public static int iteratively (int m, int n){
   int r = m % n;
   while (r != 0) {
      m = n;
      n = r;
      r = m % n;
   }
   return n;
}
```

```
public static int recursively(int m, int n) {
      if (n==0)
         return m;
      return recursively(n, m % n);
   }
```

Why Algorithms?

- Gives an idea (estimate) of running time.
- Help us decide on hardware requirements.
- What is feasible vs. what is impossible.
- 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.

Recursion

Definition:

- A function that calls itself is said to be recursive.
- A function **f1** is also recursive if it calls a function **f2**, which under some circumstances calls **f1**, creating a cycle in the sequence of calls.
- The ability to invoke itself enables a recursive function to be repeated with different parameter values.
- You can use recursion as an alternative to iteration (looping).

The Nature of Recursion:

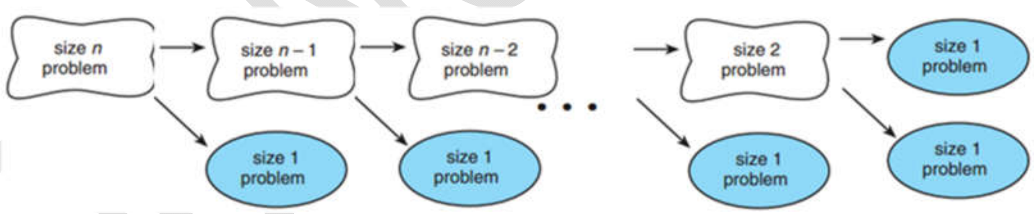
Problems that lend themselves to a recursive solution have the following characteristics:

- One or more simple cases of the problem have a straightforward, non-recursive solution.
- The other cases can be redefined in terms of problems that are closer to the simple cases.
- By applying this redefinition process every time the recursive function is called, eventually the problem is reduced entirely to the simple case(s), which are relatively easy to solve.

The recursive algorithms will generally consist of an "**if** statement" with the following form:

> **if** this is a **simple case**
>   ***solve it***
> **else**
>   ***redefine the problem using recursion***

Illustration:



Example:

Solve the problem of multiplying **6** by **3**, assuming **we only know addition**:

- **Simple case**: any number multiplied by 1 gives us the original number.
- The problem can be split into the two problems:

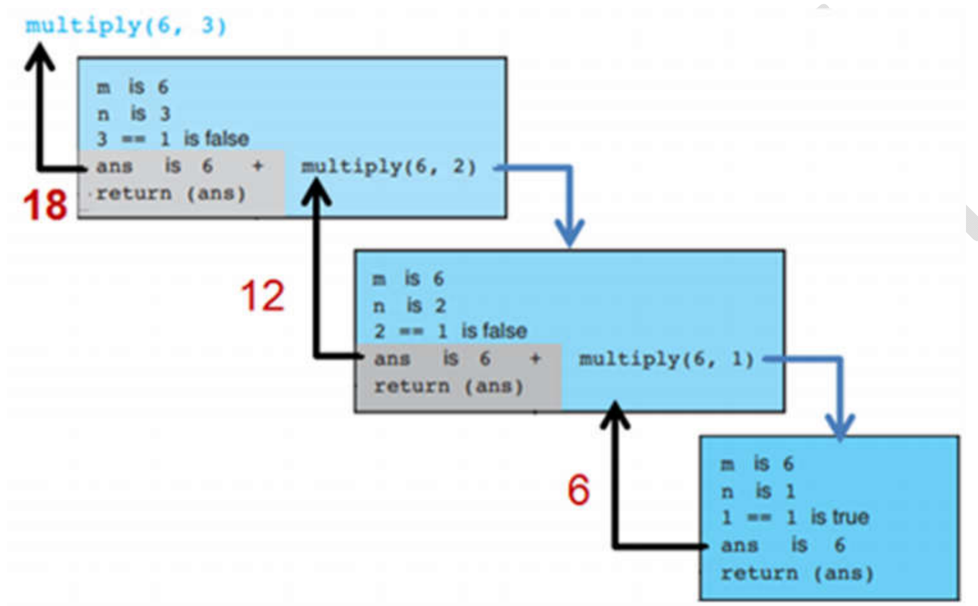
> **1. Multiply 6 by 2.**
>   **1.1 Multiply 6 by 1.**
>   **1.2 Add (Multiply 6 by 1) to the result of problem 1.1.**
> **2. Add (Multiply 6 by 1) to the result of problem 1.**

Implement this recursively

Tracing a Recursive Function:

- Tracing an algorithm's execution provides us with valuable insight into how that algorithm works.
- By drawing an **activation frame** corresponding to each call of the function.
- An activation frame shows the parameter values for each call and summarizes the execution of the call.

multiply(6, 3):

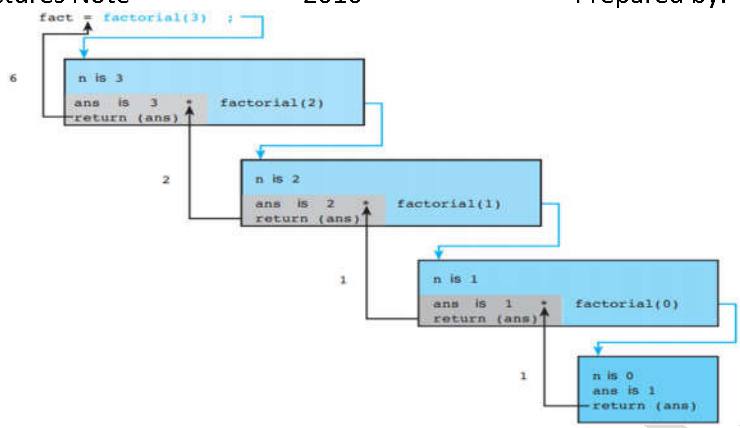


Recursive Mathematical Functions:

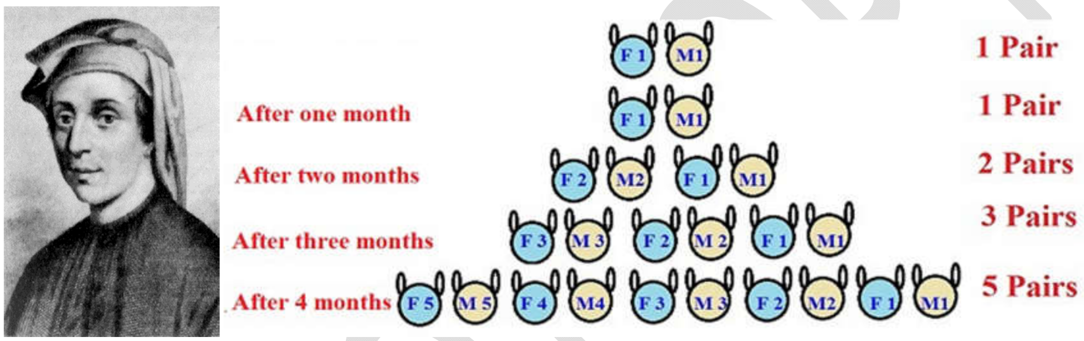
- Many mathematical functions can be defined recursively.
- An example is the factorial of n (**n!**):
  - **0!** is 1
  - **n!** is **n * (n 1)!** , for n> 0
- Thus **4!** is **4 *3!**, which means 4 *3 *2 *1, or 24.

Implement this iteratively and recursively

Tracing the recursive function



Fibonacci Numbers:



Leonardo **Bonacci** (1170 –1250)

- **Problem:**
  - How many pairs of rabbits are alive in month **n**?
- Recurrence relation:

rabbit(n) = rabbit(n-1) + rabbit(n-2)

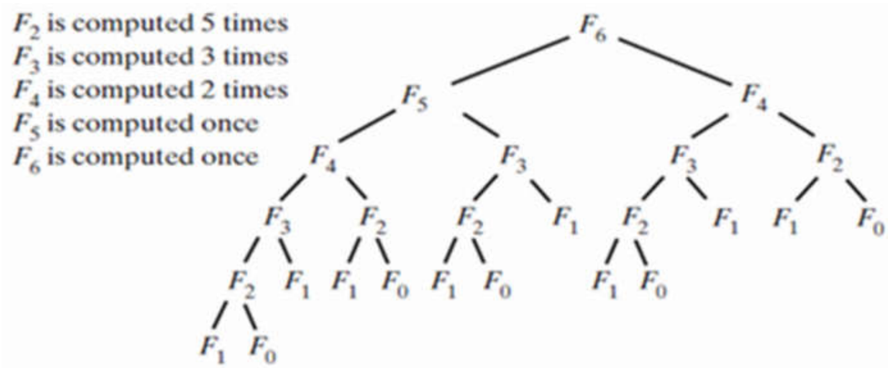
❖ The Fibonacci sequence is defined as:
- Fibonacci **0** is 1
- Fibonacci **1** is 1
- Fibonacci **n** is **Fibonacci n 2 + Fibonacci n 1**, for n>1

Implement this recursively

Poor Solution to a Simple Problem:

```
Algorithm Fibonacci(n)
if (n <= 1)
    return 1
else
    return Fibonacci(n - 1) + Fibonacci(n - 2)
```

Why is this inefficient? Try F_6



Self-Check:

- Write and test a recursive function that returns the value of the following recursive definition:
  - **f(x) = 0** **if x = 0**
  - **f(x) = f(x - 1) + 2** **otherwise**

What set of numbers is generated by this definition?

Design Guidelines:

- Method must be given an **input value**.
- Method definition must contain **logic** that involves this input, leads to different cases.
- One or more cases should provide solution that does not require recursion.
  - else **infinite recursion**
- One or more cases must include a recursive invocation.

Stack of Activation Records:

- Each call to a method generates an activation record.
- Recursive method **uses more memory** than an iterative method.
  - Each recursive call generates an activation record.
- If recursive call generates too many activation records, could cause **stack overflow**.

Recursively Processing an Array:

Starting with array[first]:

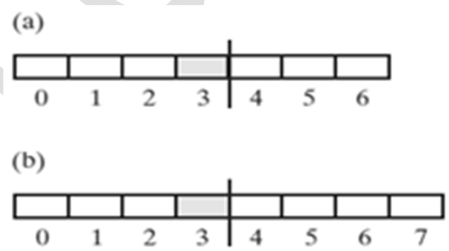
```
public static void displayArray(int array[], int first, int last)
{
   System.out.print(array[first] + " ");
   if (first < last)
      displayArray(array, first + 1, last);
} // end displayArray
```

Starting with array[last]:

```
public static void displayArray(int array[], int first, int last)
{
   if (first <= last)
   {
      displayArray(array, first, last - 1);
      System.out.print (array[last] + " ");
   } // end if
} // end displayArray
```

Processing array from middle:

```
int mid = (first + last) / 2;
```

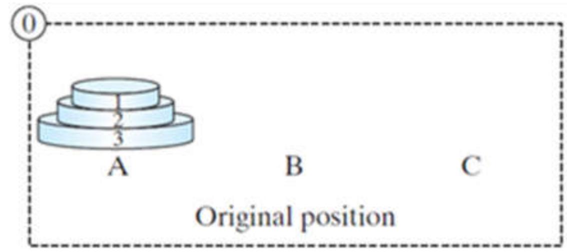



```
public static void displayArray(int array[], int first, int last)
{
   if (first == last)
      System.out.print(array[first] + " ");
   else
   {
      int mid = (first + last) / 2;
      displayArray(array, first, mid);
      displayArray(array, mid + 1, last);
   } // end if
} // end displayArray
```

Consider
first + (last – first) / 2
Why?

Tower of Hanoi

Simple Solution to a Difficult Problem:

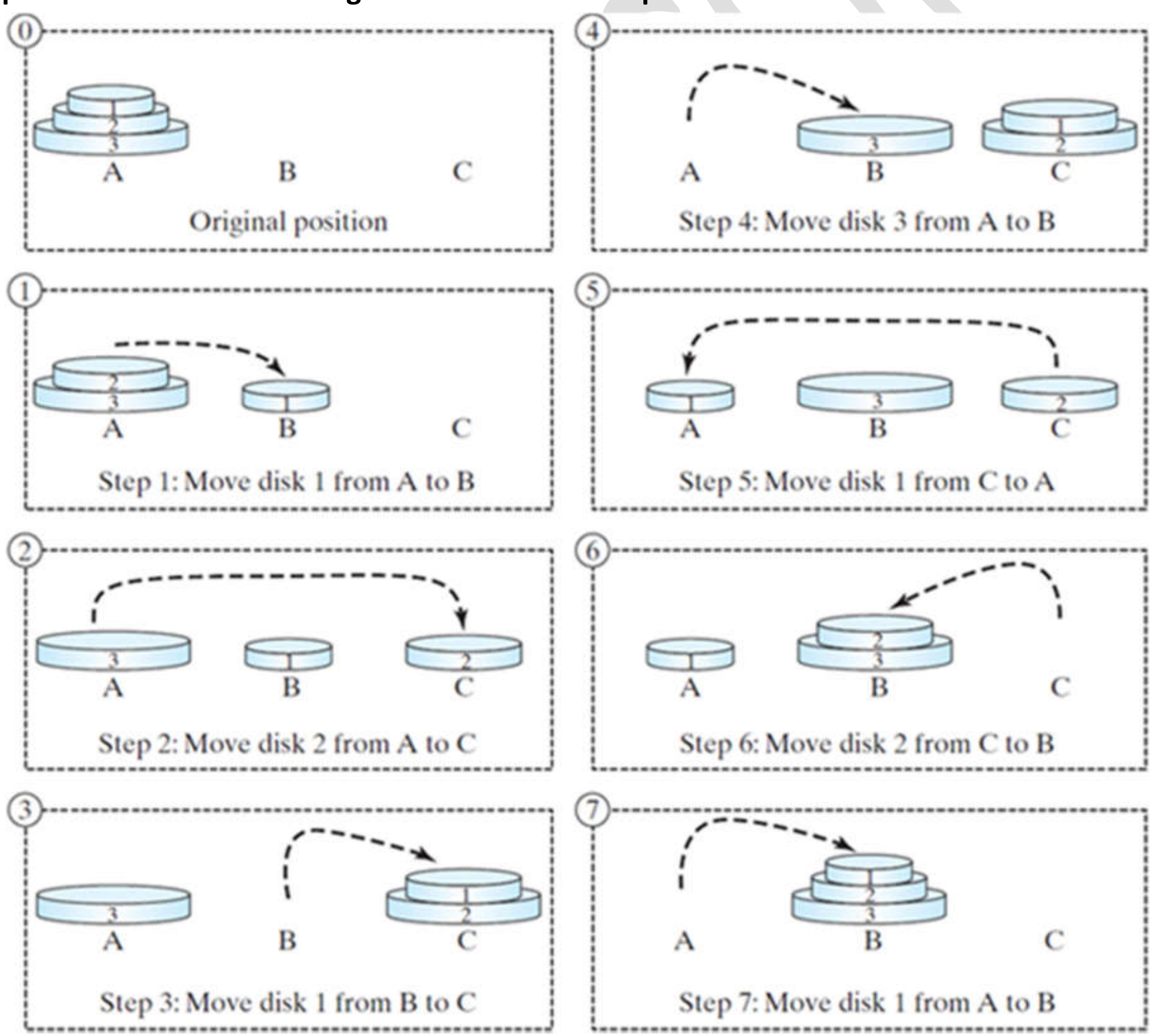


Rules:

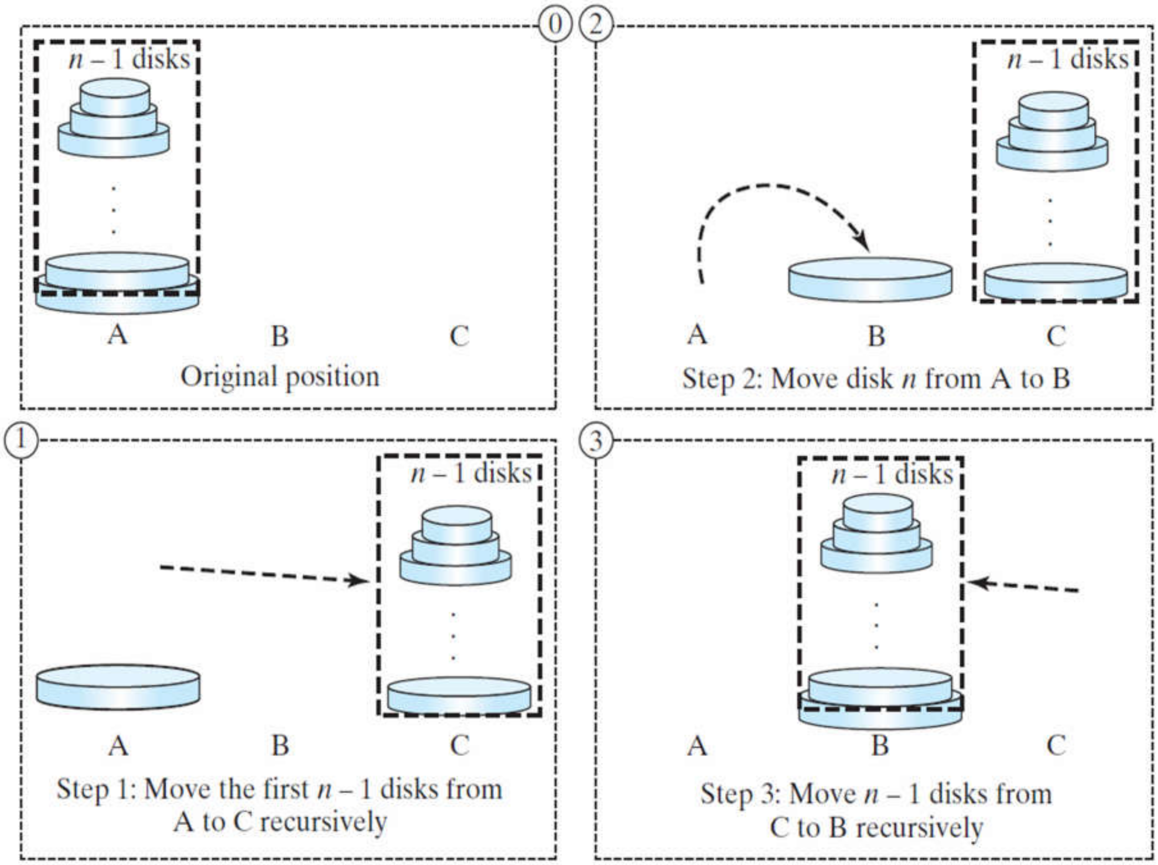
- Move one disk at a time. Each disk moved must be topmost disk.
- No disk may rest on top of a disk smaller than itself.
- You can store disks on the 2nd pole temporarily, as long as you observe the previous two rules.

Tower of Hanoi flash @ https://www.mathsisfun.com/games/towerofhanoi.html

Sequence of moves for solving the Towers of Hanoi problem with three disks:



The Tower of Hanoi problem can be decomposed into three sub-problems.



- Move the first **n-1** disks from **A** to **C** with the assistance of tower **B**.
- Move disk **n** from **A** to **B**.
- Move **n-1** disks from **C** to **B** with the assistance of tower **A**.

Solutions:

```
Algorithm solveTowers(numberOfDisks, startPole, tempPole, endPole)
if (numberOfDisks == 1)
    Move disk from startPole to endPole
else
{
    solveTowers(numberOfDisks - 1, startPole, endPole, tempPole)
    Move disk from startPole to endPole
    solveTowers(numberOfDisks - 1, tempPole, startPole, endPole)
}
```

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 are 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 $\sum_{k=1}^{n} k = 1 + 2 + 3 + \ldots + n$

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

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.

Run summing code 2 times and compare time

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

Order of time or **growth of time:**

Go back to summing result

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

- Best case analysis ✗ too optimistic
- Average case analysis ✗ too complex (statistical methods)
- Worst case analysis ✓ it will not exceed this

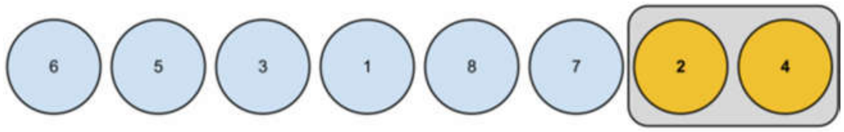
RAM model of computation

We assume that:

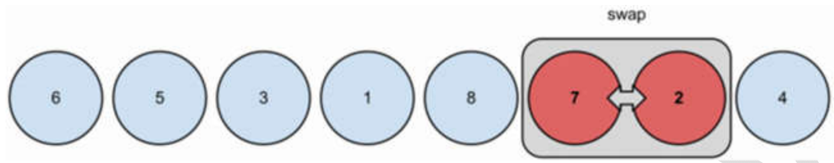
- We have infinite memory
- Each operation (+,-,*,/,=) takes 1 unit of time
- Each memory access takes 1 unit of time
- All data is in the RAM

Bubble Sort:

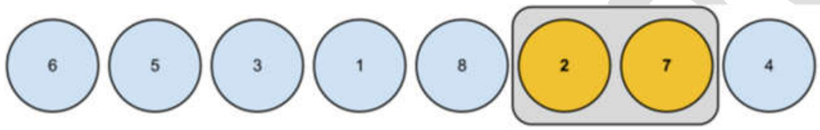
1. Each two adjacent elements are compared:

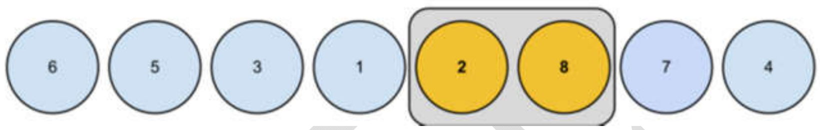


2. Swap with larger elements:

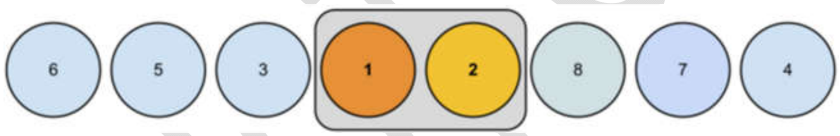


3. Move forward and swap with each larger item:

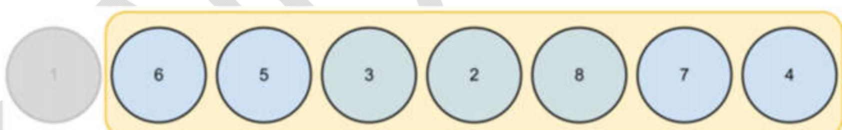




4. If there is a lighter element, then this item begins to bubble to the surface:



5. Finally the smallest element is on its place:



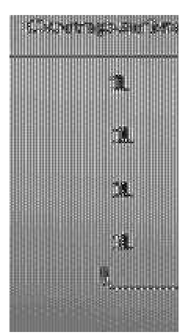
Make a demo using the following data set

12	8	7	5	2

Worst case analysis

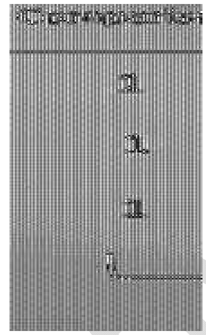
After 1st round:

8	7	5	2	12



After 2nd round:

7	5	2	8	12



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

$$= 4\,(4 + 3 + 2 + 1) = 4\,(n\text{-}1 + n\text{-}2 + \ldots + 2 + 1) = 4\,(n\text{-}1*n/2) =$$

$$2 * n * (n\text{-}1) \rightarrow pn^2 + qn + r \rightarrow$$ **p**, **q**, and **r** are some constant.

Implement and test effectiveness of bubble sort algorithm

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

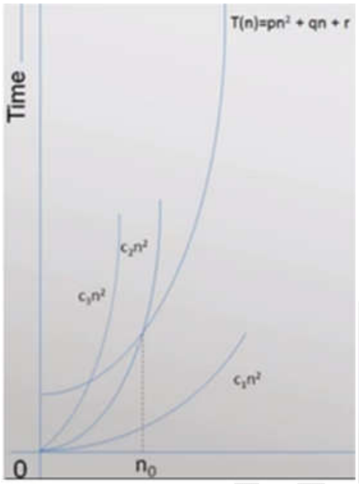
i=0	j=n-1	n-1
i=1	j=n-2	n-2
:	:	:
:	:	:
i=n-1	j=0	1

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_1n^2 ← never over take **T(n)**

C_2n^2 such that its greater than **T(n)** for $n>n_0$. In this case we say that C_2n^2 is an upper bond of **T(n)**

But we can come up with many functions satisfy this condition. We need to be precise.



OBSERVATIONS:

$\forall\, n > n_0$

$c_2n^2 \ge pn^2 + qn + r$

Big Oh $O(n^2)$: **f(n)**: there exist positive constants **c** and n_0 such that $0 \le f(n) \le cn^2$ for all $n \ge n_0$

In general

$O(g(n))$: **f(n)**: there exist positive constants **c** and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$

Example 1:

$5n^2 + 6 \in O(n^2)$??? ✓

Find cn^2 → $c=6$ and $n_0=3$

→ $c=5.1$ $n_0=8$

Example 2:

$5n + 6 \in O(n^2)$??? ✓

Find cn^2 → $c=11$ and $n_0=1$

Example 3:

$n^3 + 2n^2 + 4n + 8 \in O(n^2)$??? ✗

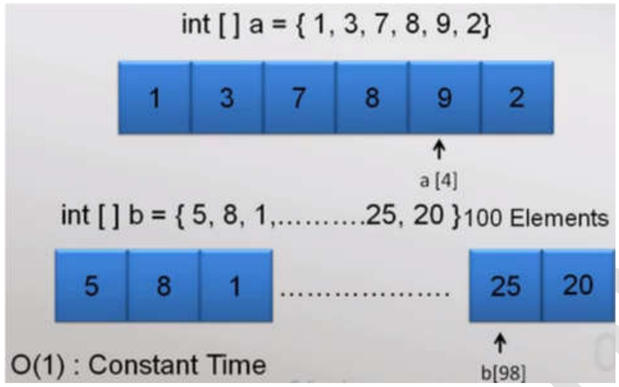
Find $cn^2 \ge n^3 + 2n^2 + 4n + 8$??? ✗

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

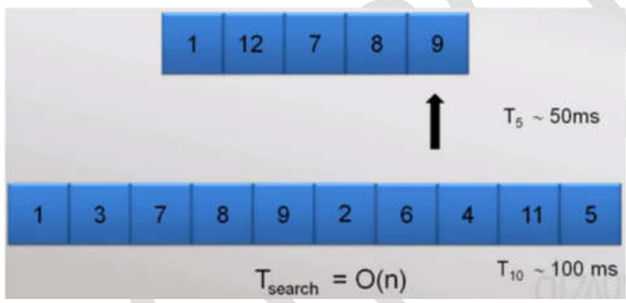
$$\log n \le \sqrt{n} \le n \le n\log n \le n^2 \le n^3 \le 2^n \le n!$$

What does it mean?

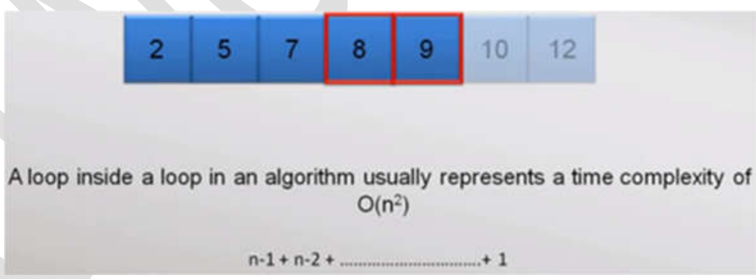
Array element access:



Array element search:



Bubble sort algorithm:



Asymptotic Analysis

Asymptotic (مقارب) 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.

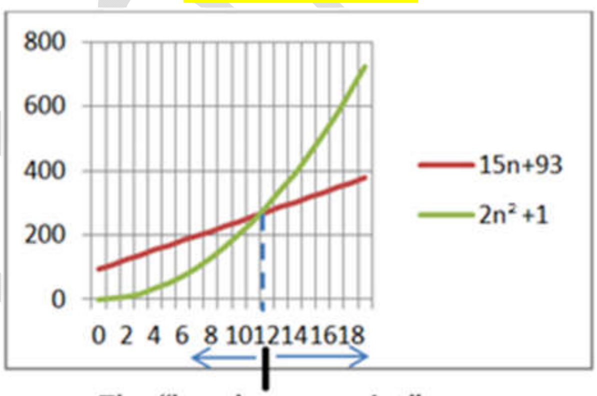
- The critical resource for a program is -most often- **running time**.
- The **growth rate** for an algorithm is the rate at which the cost of the algorithm grows as the size of its input grows.
  - ***cn*** (for ***c*** any positive constant) ➔ **linear** growth rate or running time.
  - n^2 ➔ **quadratic** 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 = $\mathbf{2n^2 + 1}$ **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*) ≤ *cf*(*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 ≤ c(n)** for **n ≥ n_0**

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

  So for **c=16** and n_0 **= 93** ➔ **// proved**

  Graph using Excel

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

  Must show +ve **c**, n_0 such that $2n^2+1 \le c(n^2)$ **for** $n \ge n_0$

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

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

  $2n^2+1 \le 3n^2$

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

  Graph using Excel

Example 3.5 For a particular algorithm, $\mathbf{T}(n) = c_1n^2 + c_2n$ in the average case where c_1 and c_2 are positive numbers. Then, $c_1n^2 + c_2n \le c_1n^2 + c_2n^2 \le (c_1 + c_2)n^2$ for all $n > 1$. So, $\mathbf{T}(n) \le cn^2$ 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 ≥ c(n)** for **n ≥ n_0**

  <because 93 is +ve> ≥ **c(n)** ➔ <provided **c=15**> ← so any $n_0 > 0$ will do

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

  Graph using Excel

- Prove that $2n^2+1$ is $\Omega(n^2)$

  Must show +ve **c** and n_0 such that $2n^2+1 \ge cn^2$ for $n \ge n_0$

  <because 1 is +ve>

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

Graph using Excel

Example 3.7 Assume $\mathbf{T}(n) = c_1 n^2 + c_2 n$ 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, $\mathbf{T}(n) \geq cn^2$ for $c = c_1$ and $n_0 = 1$. Therefore, $\mathbf{T}(n)$ 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.

Simplifying Rules

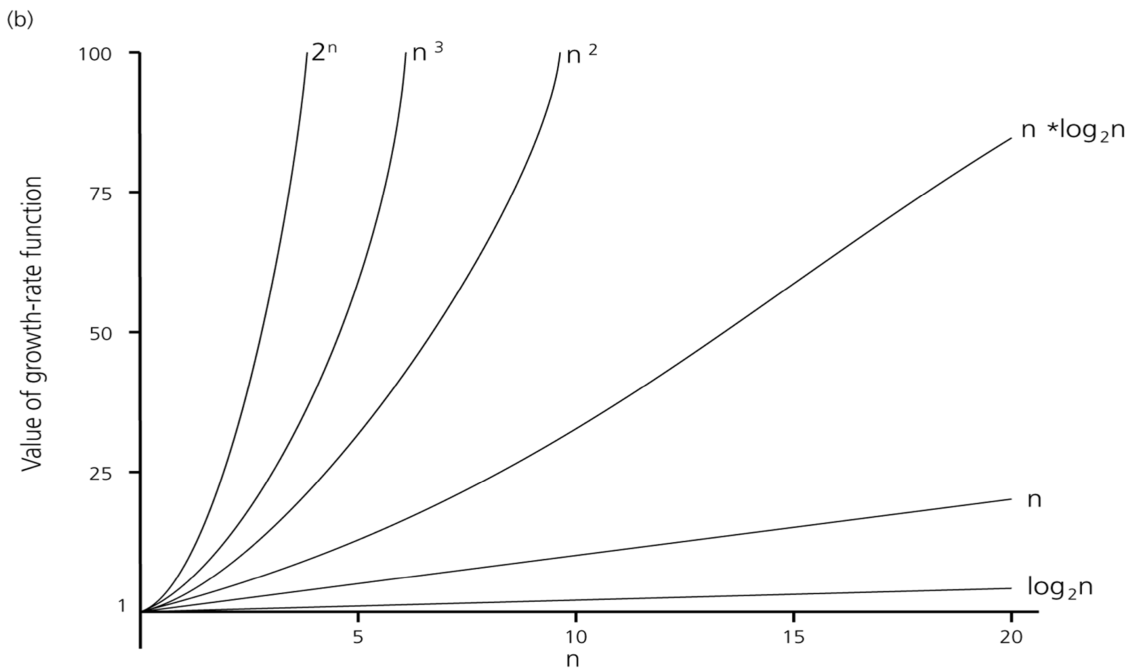
1. If $f(n)$ is in $O(g(n))$ and $g(n)$ is in $O(h(n))$, then $f(n)$ is in $O(h(n))$.
2. If $f(n)$ is in $O(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $O(g(n))$.
3. If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $f_1(n) + f_2(n)$ is in $O(\max(g_1(n), g_2(n)))$.
4. If $f_1(n)$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

$$O(1) \le O(\log_2 n) \le O(n) \le O(n \log_2 n) \le O(n^2) \le O(n^3) \le O(2^n)$$



If the problem size is always small, you can probably ignore an algorithm's efficiency

Limitations of big-O 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 ignored (**2*N* log *N*** *vs.* **1000*N***)

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)

```java
public static void bubble(int[] arr){
   int temp;
   for (int i = 0; i < arr.length-1; i++) {
      for (int j = 0; j <arr.length-i-1 ; j++) {
         if(arr[j+1]<arr[j]){
            temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
         }
      }
   }
}
```

2- **Selection Sort (revision) → $O(n^2)$**: named selection because every time we select the smallest item.

```
public static void selection (int[] arr){
   int temp, minIndex;
   for (int i = 0; i < arr.length-1; i++) {
      minIndex = i;
      for (int j = i+1; j <arr.length ; j++) {
         if(arr[j]<arr[minIndex]){
            minIndex=j;
         }
      }
      if(i!= minIndex){
         temp = arr[i];
         arr[i] = arr[minIndex];
         arr[minIndex] = temp;
      }
   }
}
```

3- Insertion sort → $O(n^2)$:

```
public static void insertion (int[] arr){
   int j, temp, current;
   for (int i = 1; i < arr.length; i++) {
     current = arr[i];
      j=i-1;
      while (j>=0 && arr[j]>current){
         arr[j+1] = arr[j];
         j--;
      }
      arr[j+1]=current;
   }
}
```

$O(n^2)$ sorting algorithms comparison:

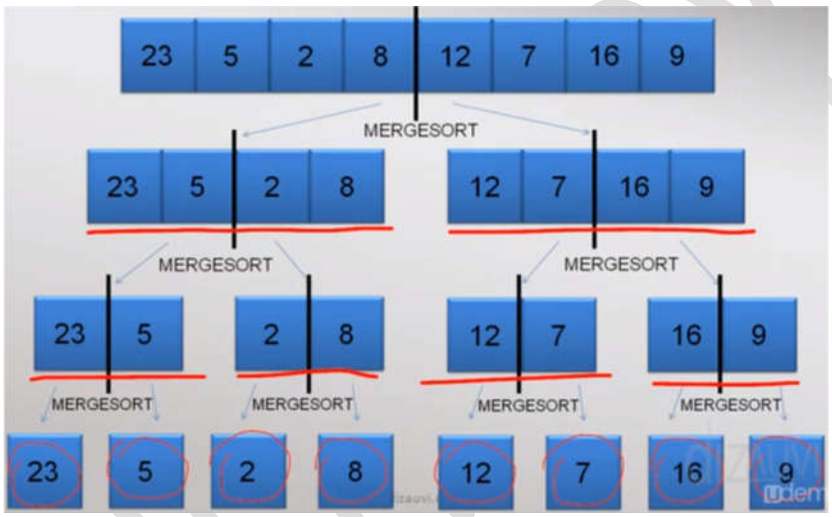
(run demo @ http://www.sorting-algorithms.com/)

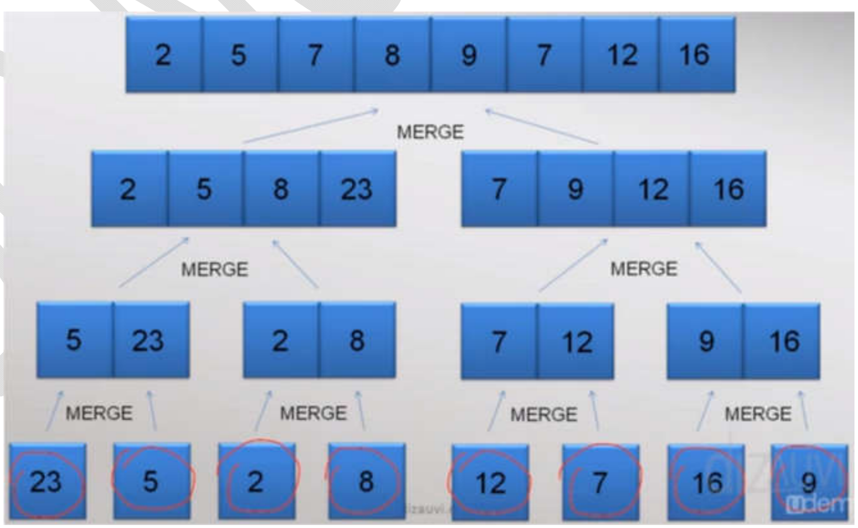
Bubble Sort	Selection Sort	Insertion Sort
Very inefficient	• Better than bubble sort • Running time is independent of ordering of elements	• Relatively good for small lists • Relatively good for partially sorted lists

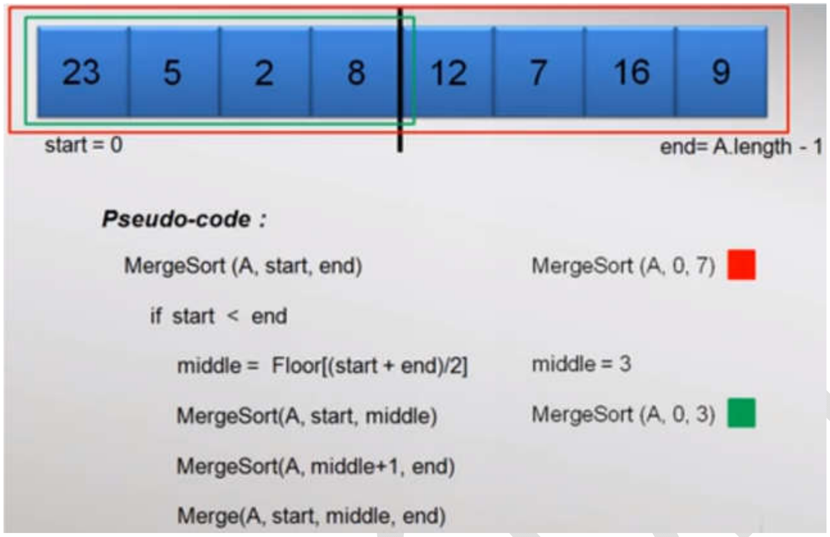
Merge sort: recursive algorithm

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

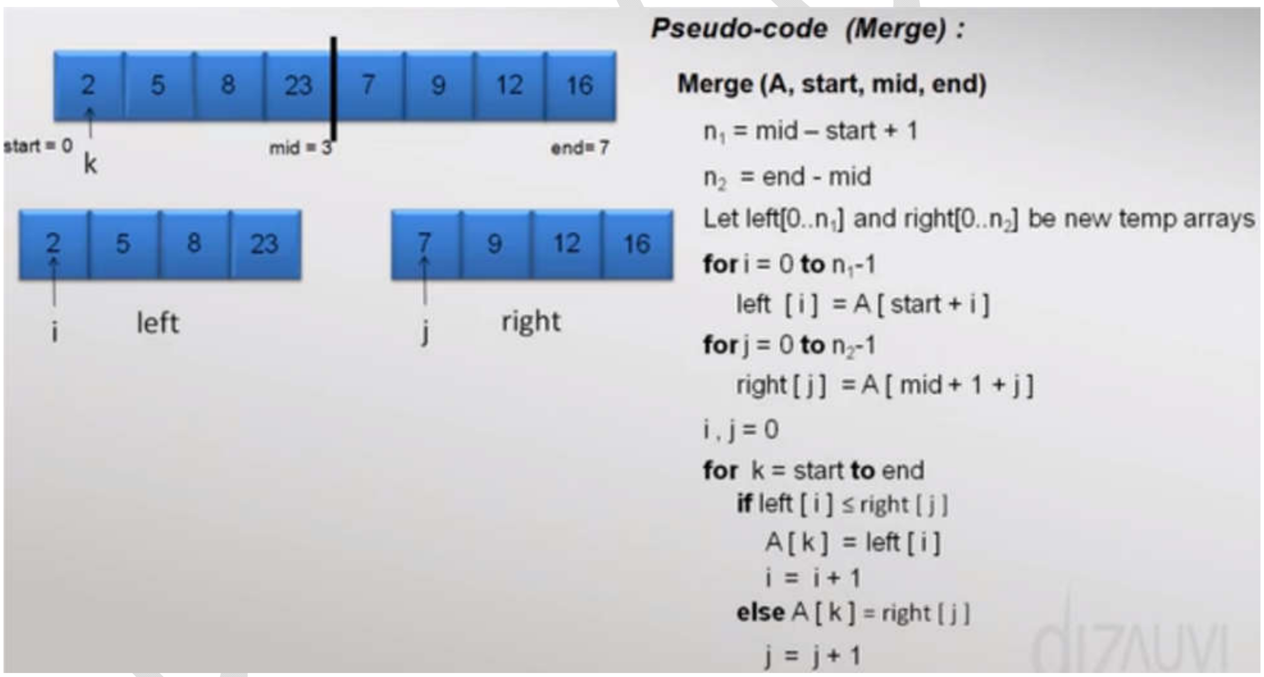
Example:







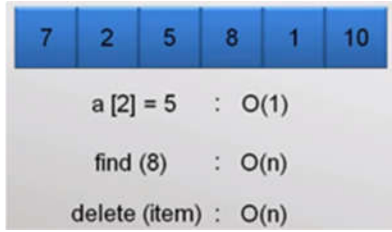
Pseudo code:



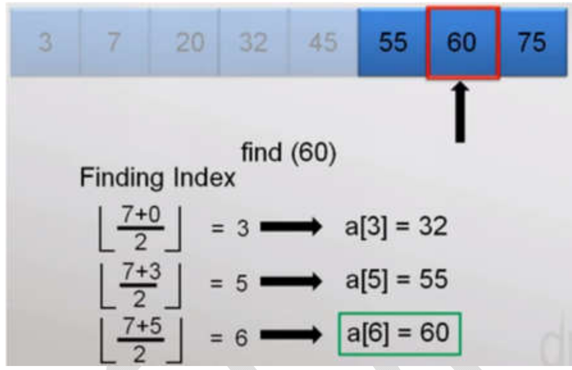
Make sure of array boundaries

H.W: implement merge sort your own

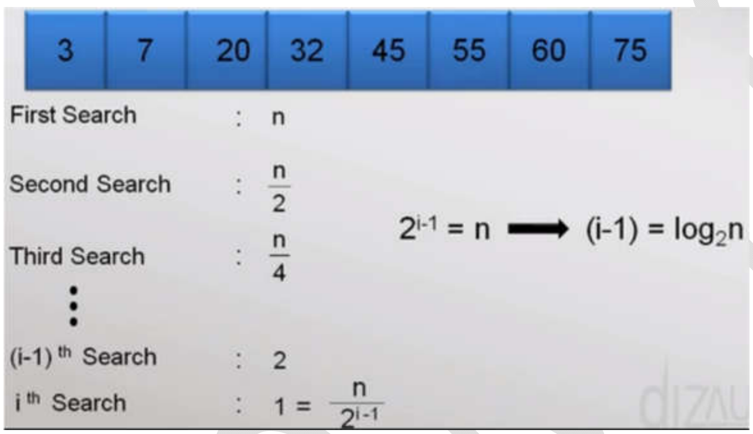
Searching elements in an array:

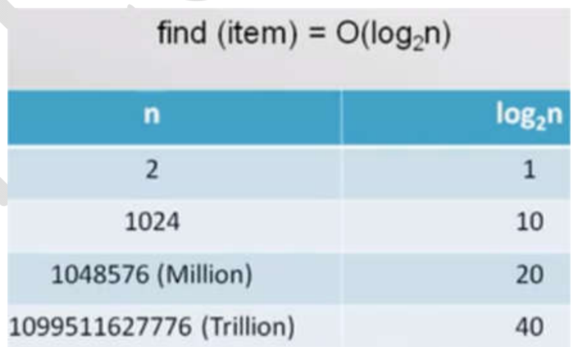


Case 1: unordered array:



Case 2: ordered array: -Binary search-

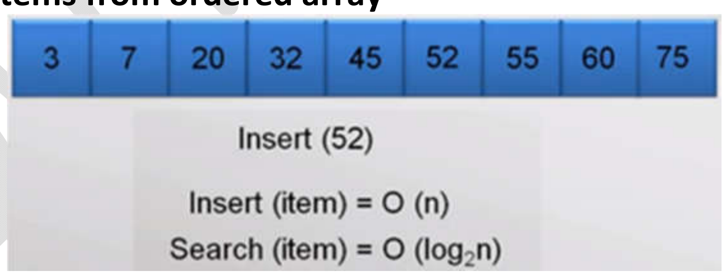


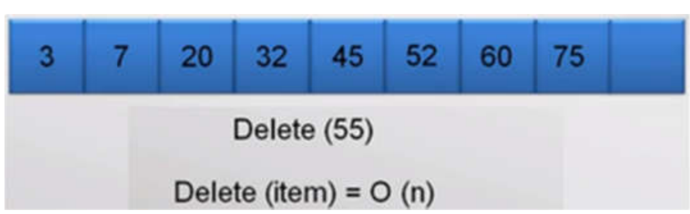


find (item) = $O(\log_2 n)$

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

Inserting and deleting items from ordered array

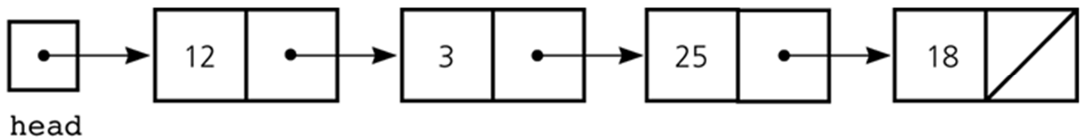




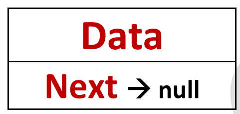
Linked List

Algorithm - abstract way to perform computation tasks
Data Structure - abstract way to organize information

Linked List:



Node:



Node code:

```
public class Node<T> {
        private T data;
        private Node<T> next;

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

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

        public Node<T> getNext() {  return next;  }
        public void setNext(Node<T> next) {  this.next = next;  }
}
```

Linked List Code:

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

Inserting a new node:

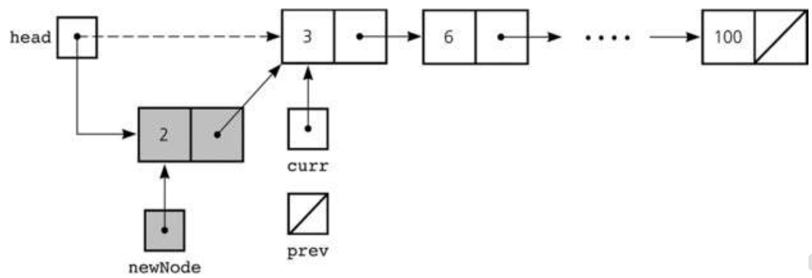
Inserting a Node into a Specified Position of a Linked List:

Three steps to insert a new node into a linked list

- Determine the point of insertion
- Create a new node and store the new data in it
- Connect the new node to the linked list by changing references

Case 1: To insert a node at the beginning of a linked list: (**curr == head**)

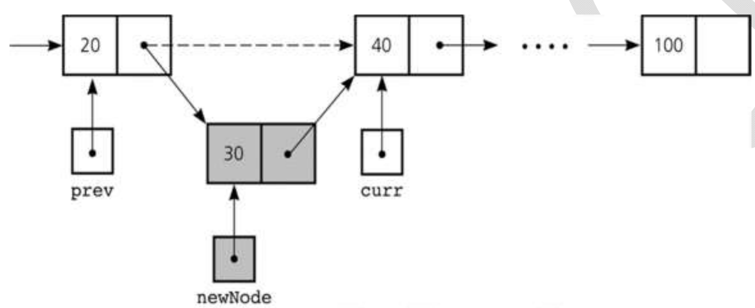
newNode.next = head;
head = newNode;



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

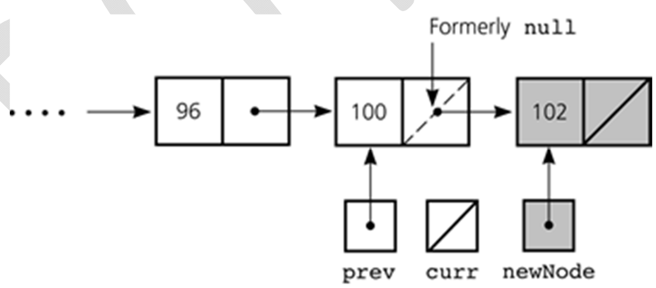
Case 2: To insert a node between two nodes:

newNode.next = curr;
prev.next = newNode;



Case 3: Inserting at the end of a linked list is a special case if **curr** is **null:**

newNode.next = curr;
prev.next = newNode;



Time Complexity ➔ **O(n)**

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

Determining **curr** and **prev**

Determining the point of insertion or deletion for a sorted linked list of objects

```
for ( prev = null , curr = head;
      (curr != null)  &&  (newValue.compareTo(curr.item) > 0);
```

```
prev = curr , curr = curr.next ) ;   // end for
```

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

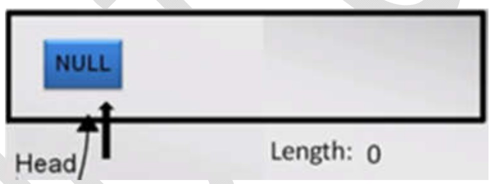
Node toString:

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

LinkedList toString:

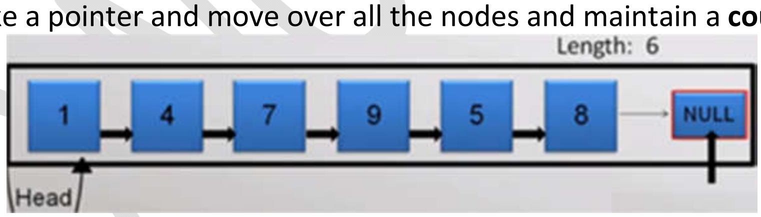
```
@Override
public String toString() {
        String res = "➔";
        Node<T> curr = head;
        while (curr != null) {
                res += curr + "→ ";
                curr = curr.next;
        }
        return res + “NULL”;
}
```

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**

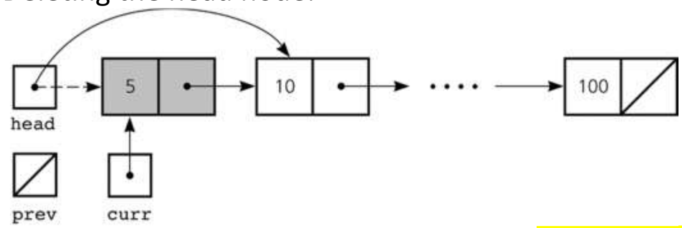


Length code: Time Complexity ➔ O(n)

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

Deleting Nodes:

Case 1: Deleting the head node:



Simply move the **head** to the **head.next**: head = head.next;

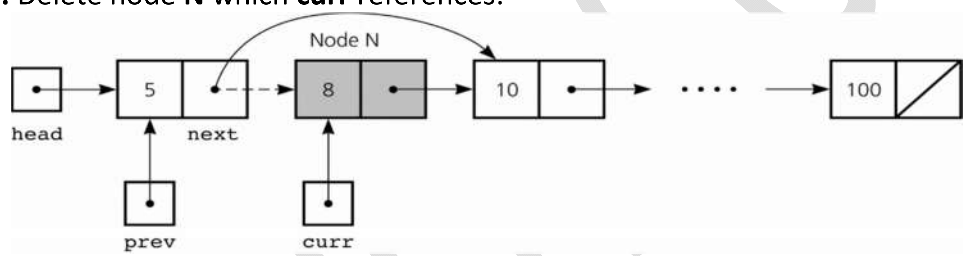
Now first Node has no reference to it ➔ **Garbage**

Time Complexity ➔ **O(1)**

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

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

Case 2: Delete node **N** which **curr** references:



Set **next** in the node that precedes **N** to reference the node that follows **N**

```
prev.next = curr.next;  // prev.next = prev.next.next;
```

Searching for an Item in a Linked List:

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

Find code:

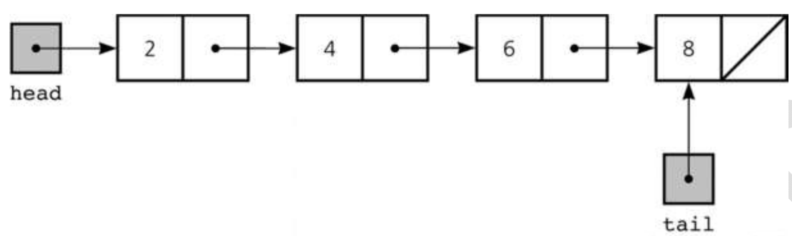
```
public Node<T> find(T data) {
        Node<T> curr =  head;
        while (curr != null) {
                if (curr.getData() == data)    //  if (curr.getData().equals(data))
                        return curr;
                curr = curr.next;
        }
        return null;
}
```

Variations of the Linked List:

1- Tail References (Doubly Ended Linked List)

- Remembers where the end of the linked list is.
- Therefore, we can add and delete at both ends.
- To add a node to the end of a linked list

tail.next = new Node(request, null);



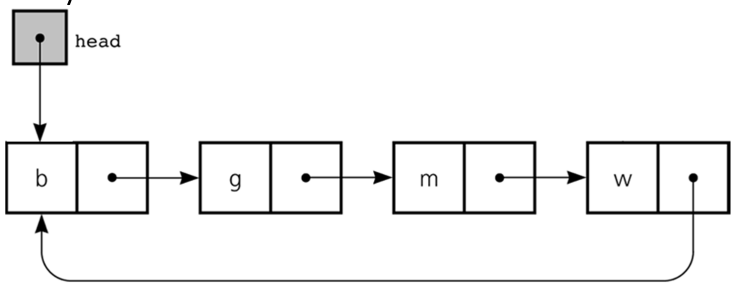

```
public class DoubleEndedList<T> extends LinkedList<T> {
        private Node<T> tail;
        public Node<T> getTail() {        return tail;   }

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

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

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

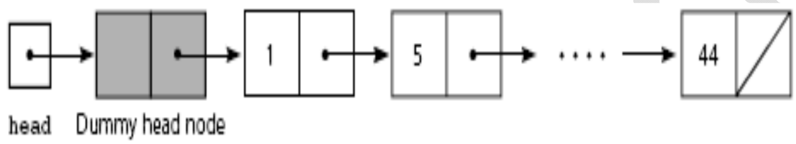
2- **Circular Linked List**

- Last node references the first node
- Every node has a successor



3- **Dummy Head Nodes**

- Always present, even when the linked list is empty
- Insertion and deletion algorithms initialize **prev** to reference the dummy head node, rather than **null**



Processing Linked Lists Recursively:

- **Traversal**
  - Recursive strategy to display a list
    Write the first node of the list
    Write the list minus its first node

```
public static void traversList(Node curr) {
  if(curr == null)
    System.out.println("NULL");
  else {
     System.out.print("[" + curr + "]-->");
     traversList(curr.next);
  }
}
```

  - Recursive strategies to display a list backward
    - writeListBackward strategy
      Write the last node of the list
      Write the list minus its last node backward

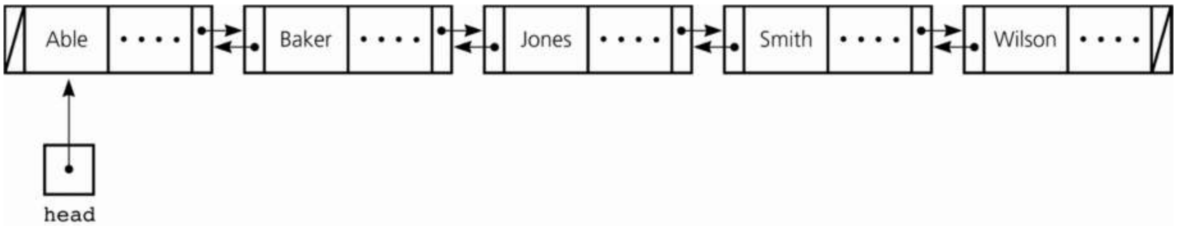
```
public static void traversListBackward(Node curr) {
  if(curr == null)
    System.out.print("NULL");
  else {
    traversListBackward(curr.next);
    System.out.print("<--[" + curr + "]");
  }
}
```

Doubly Linked List

Node:

Date
Next → null
null← Prev

Doubly Linked List: Each node references both its predecessor and its successor:



Doubly Node Code:

```
public class DNode <T extends Comparable<T>>{
   T data;
   DNode next;
   DNode prev;

   public DNode(T data) { this.data = data; }
   public T getData() { return data;  }
   public DNode getNext () { return next; }
   public DNode getPrev () { return prev; }

   public void setNext(DNode next) {   this.next = next; }
   public void setPrev(DNode prev) {   this.prev = prev; }
   public String toString() { return this.data.toString();  }
}
```

Doubly Linked List code:

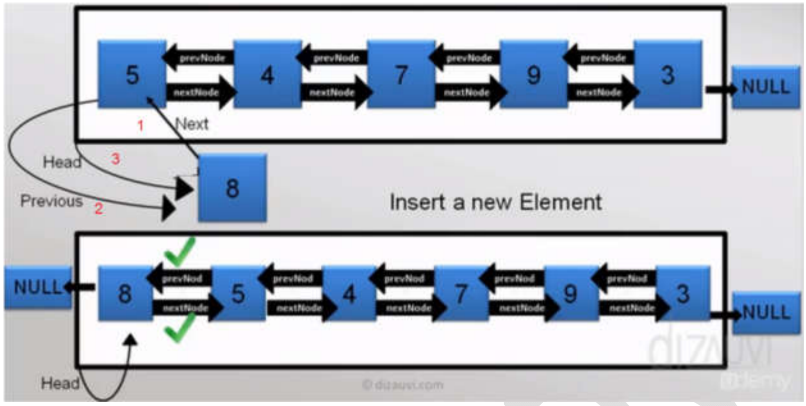
```
public class DLinkedList <T extends Comparable<T>>{
   DNode head;
}
```

Override toString method code:

```
public String toString() {
   String res = "Head-->";
   DNode<T> curr = this.head;
   while (curr != null) {
      res += "["+curr + "]";
      curr = curr.getNext();
      if(curr!=null)
         res +="<=>";
   }
   return res + "-->NULL";
}
```

Insert a new node (not sorted)

Case 1: Insert at head:




```
public void insertAtHead(T data) {
   DNode<T> newNode = new DNode(data);
   if(head==null) // empty linkedlist
      head = newNode;
   else {
      newNode.setNext(this.head);
      head.setPrev(newNode);
      head = newNode;
   }
}
```

Case 2: Insert at end:

Student Activity: insert at last

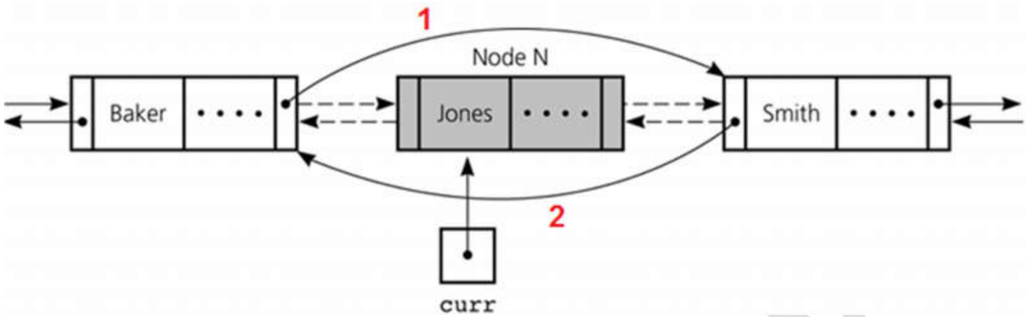
```
public void insertAtEnd(T data) {
   DNode<T> newNode = new DNode(data);
   if (head == null) // empty linkedlist
      head = newNode;
   else  {  // find last node
      DNode<T> last = head;
      while(last.getNext() != null)
         last = last.getNext();
      last.setNext(newNode);
      newNode.setPrev(last);
   }
}
```

Length of a doubly linked list code:

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

Delete a node:

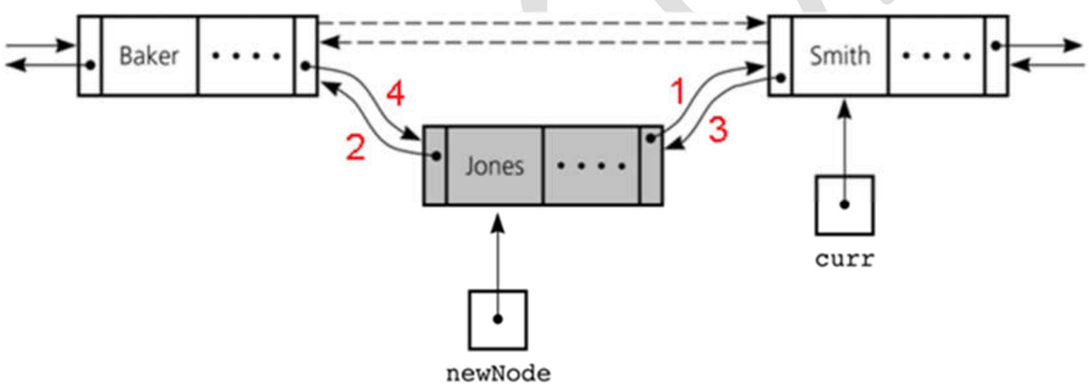
- To delete the node that **curr** pointer references



curr.prev.next = curr.next;
curr.next.prev = curr.prev;

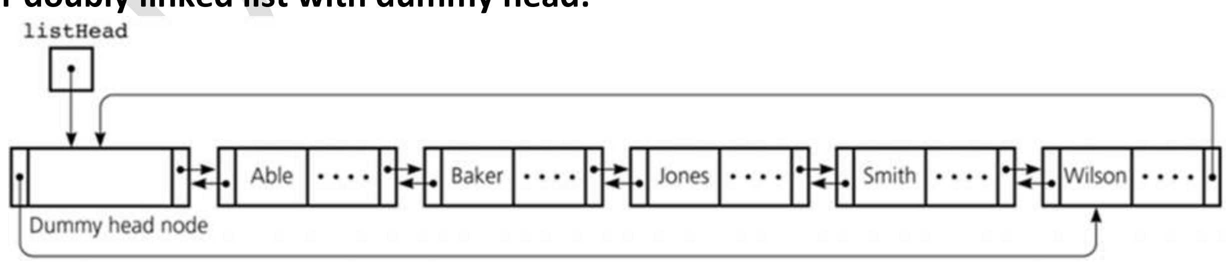
Insert a new Node (Sorted):

- To insert a new node that **newNode** references before the node referenced by **curr**



newNode.next = curr;	// 1
newNode.prev = curr.prev;	// 2
curr.prev = newNode;	// 3
newNode.prev.next = newNode;	// 4

Circular doubly linked list with dummy head:

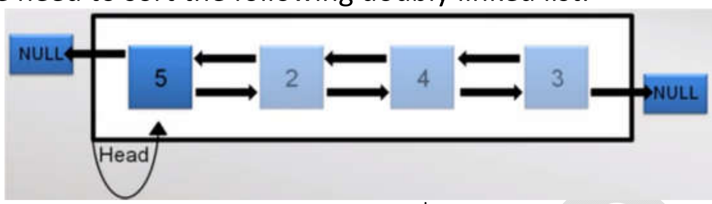


- Preceding reference of the dummy head node references the last node.
- next reference of the last node references the dummy head node.
- **Eliminates special cases for insertions and deletions.**

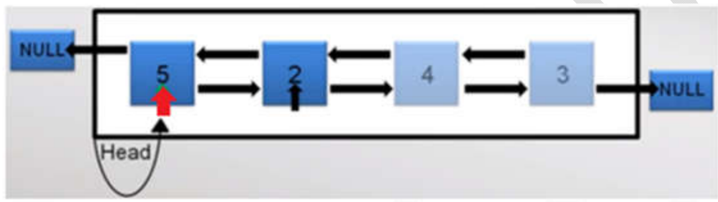
Case Study: Insertion Sort using doubly linked list (Using NO extra space):

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

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.

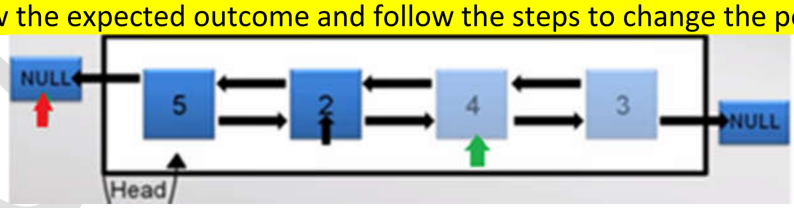
Step 1: The **red** pointer keeps move backward until it reaches a node which has a value **smaller** than the **current** node **or** reach **NULL**.

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

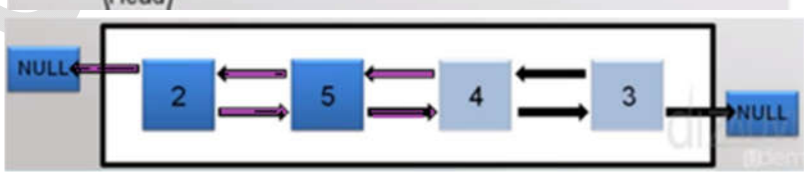
Make sure you maintain references correctly.

To do so draw the expected outcome and follow the steps to change the pointers:

Initial state:



Final state:

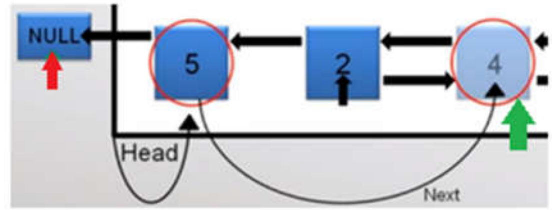


Case 1: insert to head

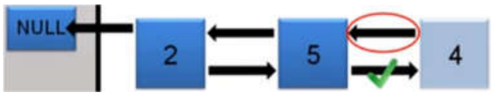
Step 2.0: **make new green pointer = black.next**

Step 2.1:

➔ **black.prev.next = green**

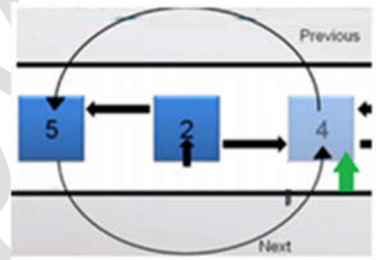


Step 2.2:

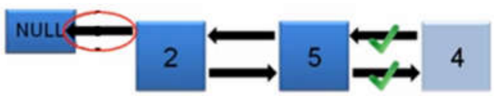


➔

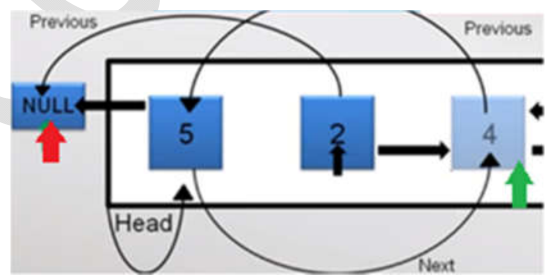
if (green != null) green.prev = black.prev



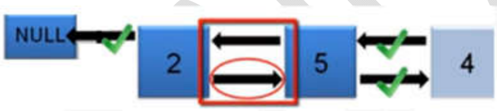
Step 2.3:



➔ black.prev = red

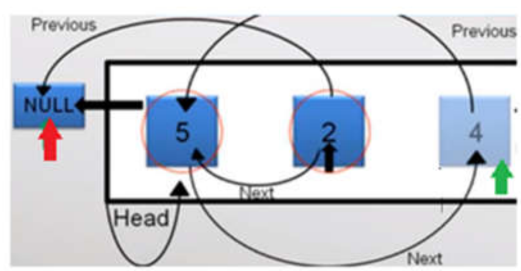


Step 2.4:

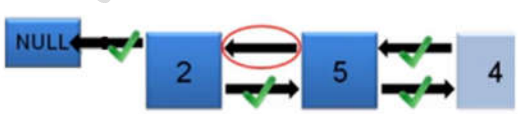


➔

if(red==null) black.next = black.next.prev
else black.next = red.next

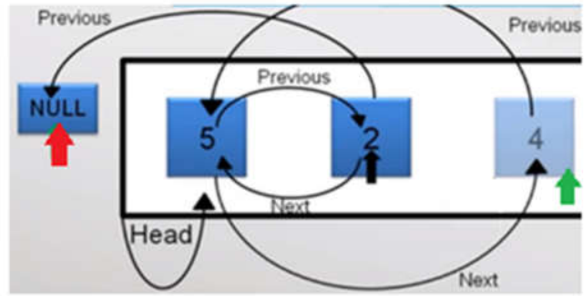


Step 2.5:



➔

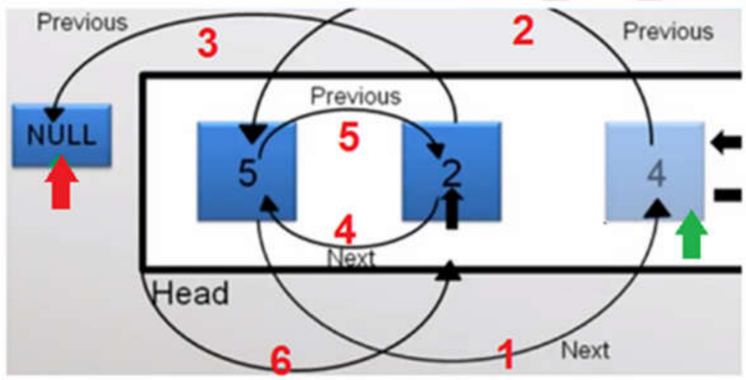
If (red == null) black.next.prev = black
else red.next.prev = black



Step 2.6:

if (red == NULL) head = black

else red.next = black;

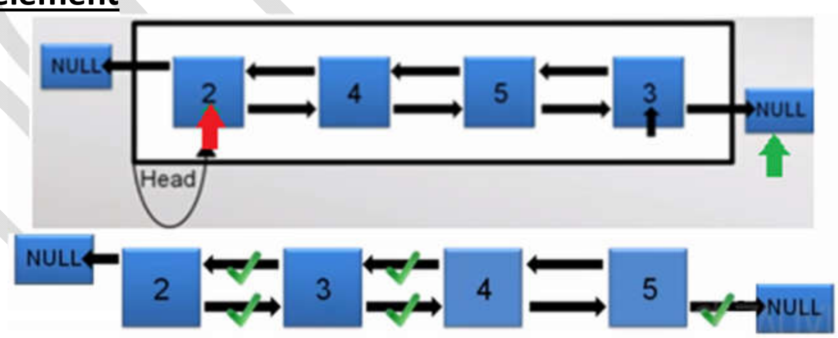


Step 2.7: black = green

Case 2: insert 4 in the middle

Practice yourself

Case 3: insert last element



Insertion Sort Code:

```
// Insertion Sort of a Doubly Linked List
public void sort() {
   DNode black = head.next;
   while (black != null) {
      DNode red = black.prev;
      while (red != null && (red.data.compareTo(black.data) > 0 ))  // step 1.0
         red = red.prev;

      DNode green = black.next;             // step 2.0
      if (red != null || (head != black)) {
         black.prev.next = green;           // step 2.1
         if (green!= null) {
            green.prev = black.prev;        // step 2.2
         }
         black.prev = red;                  // step 2.3
      }
      if (red == null) {    // set the black as head
         if (head != black) {
            black.next = head;              // step 2.4
            black.next.prev = black;        // step 2.5
            head = black;                   // step 2.6
         }
      } else {   // red is not null
         black.next = red.next;             // step 2.4
         red.next.prev = black;             // step 2.5
         red.next = black;                  // step 2.6
      }
      black = green;  // step 2.7
   }
}
```

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**.

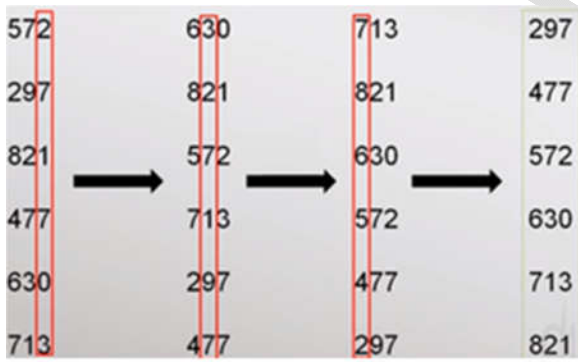
Example Radix Sort:

Step 1: take the least significant digits (LSD) of the values to be sorted.

Step 2: sort the list of elements based on that digit.

Step 3: take the 2nd LSD and repeat step 2.

Then the 3rd LSD and so on.



Radix Sort Algorithm using linked lists:

- Consider the following array:

9	179	139	38	10	5	36

- Create an array of **10** linked lists as follow:
  - **0** to **9** refer to actual numbers.
  - With input numbers, we will start with **mod 10** then **divide** the resulted number by **1**.

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

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

- In this case add **Arr[0]** to the **10th** linked list
- Repeat for remaining array elements.

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

- If we reach the end of array: make a new array by removing data from the head of each linked list in order:

10	5	36	38	9	179	139

Is this sorted? NO

- **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	→
6	→
7	→
8	→
9	→

Result:

5	9	10	36	38	139	179

Is this sorted? What is the time complexity?

HW: implement Radix sort using Doubly Linked List

Cursor Implementation of Linked Lists

- **Reason 1**: Many Languages do not support pointers (e.g. Basic, Fortran).
  - If linked lists are required and pointers are not available, then an alternate implementation must be used.
  - The alternate method we will describe here is known as a **cursor implementation**.
- **Reason 2**: If data max length is **known**, using Array is **faster**.

Two features present in a pointer implementation of linked lists:

1. The data are stored in **array** are nodes, each array element (node) contains **data** and a **pointer** to the next node.
2. A new node can be obtained from the system's global memory by a call to ***malloc*** (**m**emory **alloc**ation) and released by a call to ***free*** methods.

Our cursor implementation must be able to simulate these two features:

- The logical way to satisfy 1st feature is to have a global array of nodes. For any cell in the array, its array index can be used in place of an address. The following gives the type declarations for a cursor implementation of linked lists:

```
public class Node<T extends Comparable<T>>  {
   T data;
   int next;

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

   public void setData(T data) { this.data = data;   }
   public T getData() {  return data;    }
   public int getNext() {  return next;   }
   public void setNext(int next) { this.next = next;  }

   public String toString() { return "["+ data+ " , " + next + "]"; }
}
```

- We must now simulate 2nd feature by allowing the equivalent of ***malloc*** and ***free*** for nodes in the array.
  - To do this, we will keep a list (the **freelist**) of nodes that are not in any list. The list will use **node 0** as a header. The initial configuration is shown in the following figure: →→→→
- A value of **next** is the equivalent of a pointer to next node.
- The following code to create an array of free nodes:

```
Node<T>[ ]  cursorArray  =  new Node[11];
```

i	data	next
0	null	1
1	null	2
2	null	3
3	null	4
4	null	5
5	null	6
6	null	7
7	null	8
8	null	9
9	null	10
10	null	0

- The initialization of **cursorArray** is a straightforward loop:

```
public int initialization(){
   for(int i=0;i<cursorArray.length-1;i++)
      cursorArray[i] = new Node<>(null, i+1);
   cursorArray[cursorArray.length-1] = new Node<>(null, 0);
   return 0;
}
```

- To perform an ***malloc***, the first element (after the header) is removed from the **freelist**:

```
public int malloc() {
   int p = cursorArray[0].next;
   cursorArray[0].next = cursorArray[p].next;
   return p;
}
```

- To perform a ***free***, we place the cell at the front of the **freelist**:

```
public void free(int p){
   cursorArray[p] = new Node(null, cursorArray[0].next);
   cursorArray[0].next = p;
}
```

- The following are a list of functions to test whether a linked list is **null**, **empty**, or whether a specific node is the **last**:

```
public boolean isNull(int l){
   return cursorArray[l]==null;
}

public boolean isEmpty(int l){
   return cursorArray[l].next == 0;
}

public boolean isLast(int p){
   return cursorArray[p].next == 0;
}
```

- To create a new linked list, first you have to allocate one free node using **malloc** function, then make a new point that next points to **0** as follow:

```
public int createList(){
   int l = malloc();
   if(l==0)
      System.out.println("Error: Out of space!!!");
   else
      cursorArray[l] = new Node("-",0);
   return l;
}
```

- The following code is used to add a new data to a specific linked list:

```
public void insertAtHead(T data, int l){
    if(isNull(l)) // list not created
        return;
    int p = malloc();
    if(p!=0){
        cursorArray[p] = new Node(data, cursorArray[l].next );
        cursorArray[l].next = p;
    }
    else
        System.out.println("Error: Out of space!!!");
}
```

- The following code is used to travers a linked list:

```
public void traversList(int l) {
    System.out.print("list_"+l+"-->");
    while(!isNull(l) && !isEmpty(l)){
        l=cursorArray[l].next;
        System.out.print(cursorArray[l]+"-->");
    }
    System.out.println("null");
}
```

- The following code is used to find a specific data in a linked list:

```
public int find(T data, int l){
    while(!isNull(l) && !isEmpty(l)){
        l=cursorArray[l].next;
        if(cursorArray[l].data.equals(data))
            return l;
    }
    return -1;  // not found
}
```

- Sometimes you need the previous location of a specific data in a linked list:

```
public int findPrevious(T data, int l){
    while(!isNull(l) && !isEmpty(l)){
        if(cursorArray[cursorArray[l].next].data.equals(data))
            return l;
        l=cursorArray[l].next;
    }
    return -1;  // not found
}
```

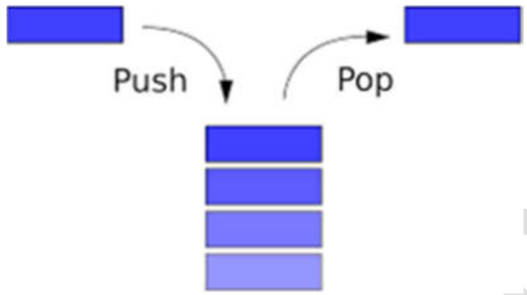
- The following code is used to delete some data from a linked list:

```
public Node delete(T data, int l){
    int p = findPrevious(data, l);
    if(p!=-1){
        int c = cursorArray[p].next;
        Node temp = cursorArray[c];
        cursorArray[p].next = temp.next;
        free(c);
    }
    return null;
}
```

Stacks

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**

ABSTRACT DATA TYPE: STACK

DATA

- A collection of objects in reverse chronological order and having the same data type

OPERATIONS

PSEUDOCODE	UML	DESCRIPTION
`push(newEntry)`	`+push(newEntry: T): void`	Task: Adds a new entry to the top of the stack. Input: `newEntry` is the new entry. Output: None.
`pop()`	`+pop(): T`	Task: Removes and returns the stack's top entry. Input: None. Output: Returns the stack's top entry. Throws an exception if the stack is empty before the operation.
`peek()`	`+peek(): T`	Task: Retrieves the stack's top entry without changing the stack in any way. Input: None. Output: Returns the stack's top entry. Throws an exception if the stack is empty.
`isEmpty()`	`+isEmpty(): boolean`	Task: Detects whether the stack is empty. Input: None. Output: Returns true if the stack is empty.
`clear()`	`+clear(): void`	Task: Removes all entries from the stack. Input: None. Output: None.

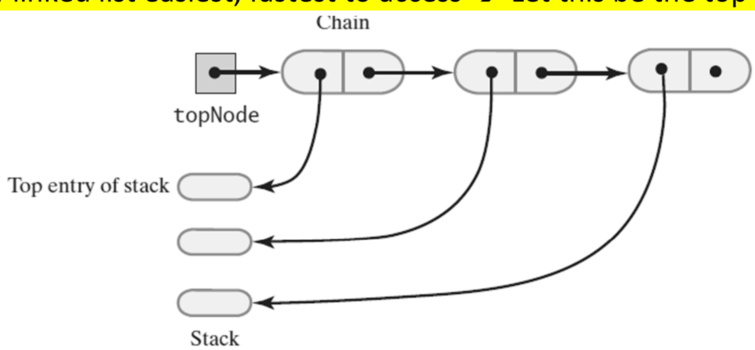
Single Linked List Implementation:

Each of the following operation involves top of stack

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

Head or Tail for topNode??

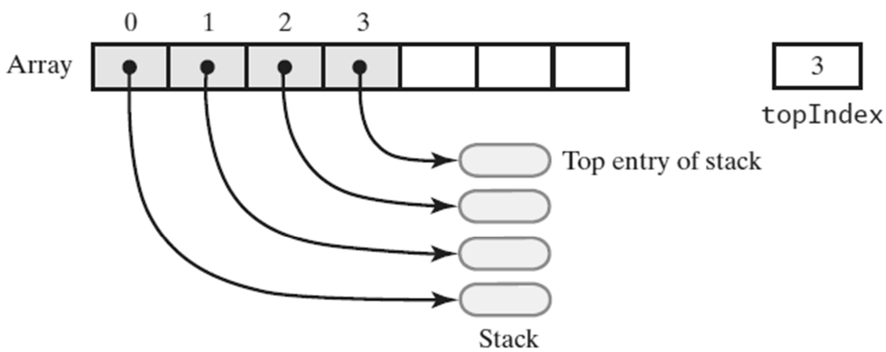
Head of linked list easiest, fastest to access → Let this be the top of the stack




```
public class LinkedStack<T extends Comparable<T>> {
   private Node<T> topNode;

   public void push(T data) {
      Node<T> newNode = new Node<T>(data);
      newNode.setNext(topNode);
      topNode = newNode;
   }

   public Node<T> pop() {
      Node<T> toDel = topNode;
      if(topNode != null)
         topNode = topNode.getNext();
      return toDel;
   }

   public Node<T> peek() {   return topNode; }

   public int length() {
      int length = 0;
      Node<T> curr = topNode;
      while (curr != null) {
         length++;
         curr = curr.getNext();
      }
      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




```
public class ArrayStack <T> {
   private Object[] s;
   private int n=-1;

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

   public boolean isEmpty(){ return n ==-1;}
   public int getN(){ return n;}

   public void push(T data){
      s[++n] = data;
   }

   public Object pop(){
      if(!isEmpty())
         return s[n--];
      return null;
   }

   public String toString() {
      String res = "Top-->";
      for(int i=n; i>=0;i--)
         res+="["+s[i]+"]-->";
      return res+"Null";
   }
}
```

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.

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

Iterable interface

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

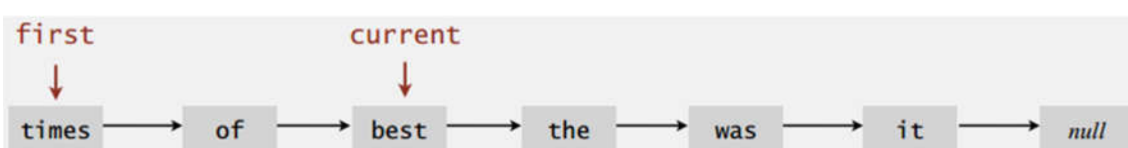
Q. What is an `Iterator` ?
A. Has methods `hasNext()` and `next()`.
Q. Why make data structures `Iterable` ?
A. Java supports elegant client code.

Iterator interface

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

```
import java.util.Iterator;
public class LinkedStack<T extends Comparable<T>> implements Iterable<T> {
   :
   public Iterator<T> iterator(){
      return new ListIterator();
   }

   private class ListIterator implements Iterator<T>{
      private Node<T> curr = topNode;
      public boolean hasNext(){return curr!=null;}
      public void remove(){}
      public T next(){
         T t = curr.data;
         curr = curr.next;
         return t;
      }
   }
}
```

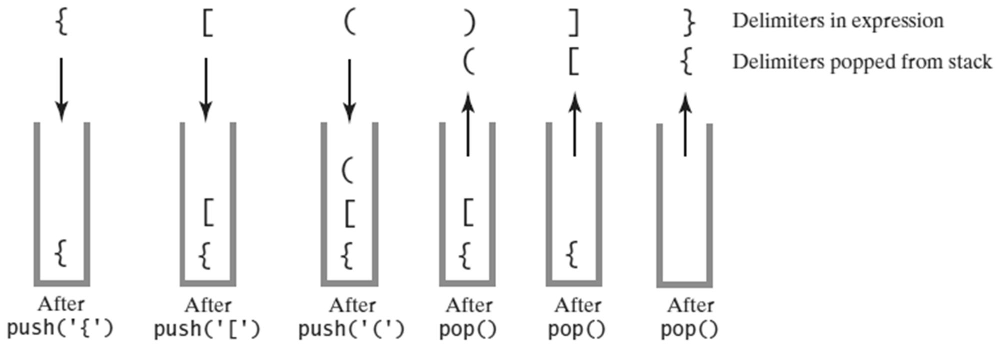
Iterator<String> itt = ls.iterator();
while (itt.hasNext())
   System.*out*.println(itt.next());
	for(String s: ls)
	   System.*out*.println(s);



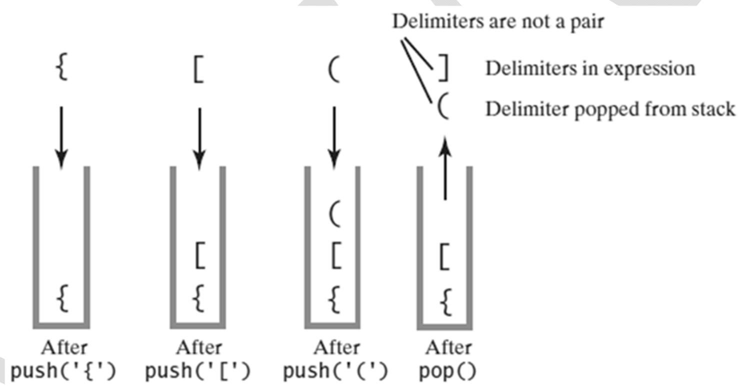
Balanced Delimiters

Problem: Find out if delimiters ("[{(]})") are paired correctly ➔ Compilers

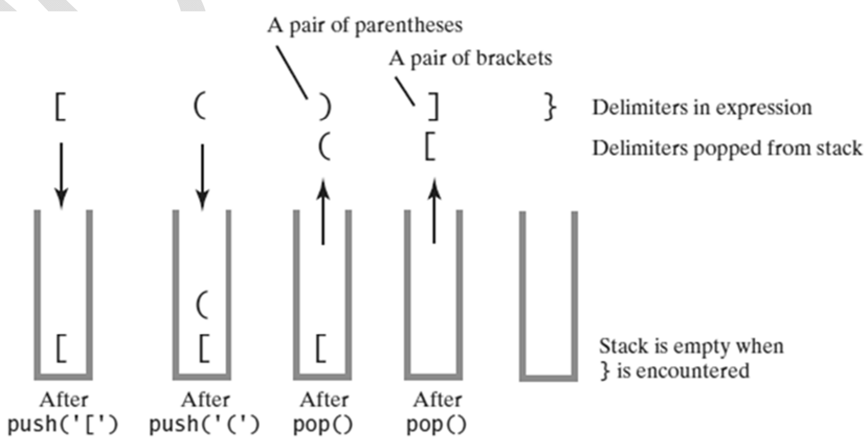
Example 1: The contents of a stack during the scan of an expression that contains the **balanced delimiters { [( )] }**



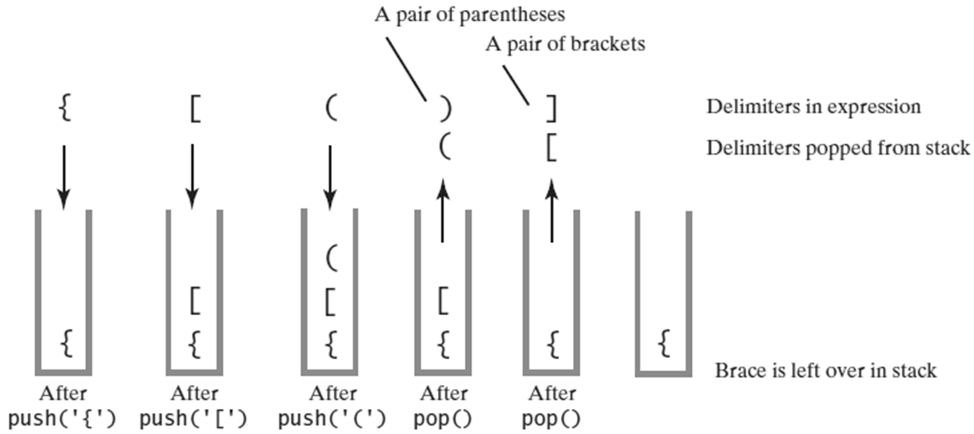
Example 2: The contents of a stack during the scan of an expression that contains the **unbalanced delimiters { [( ]) }**



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 balanced expression:

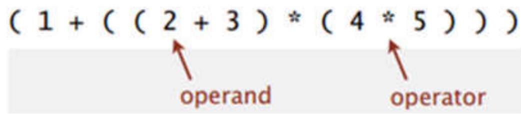
```
Algorithm checkBalance(expression)
// Returns true if the parentheses, brackets,
// and braces in an expression are paired correctly.
isBalanced = true
while ((isBalanced == true) and not at end of expression) {
   nextCharacter = next character in expression
   switch (nextCharacter) {
      case '(': 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 list/array stacks

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 +*

Evaluate infix expressions:



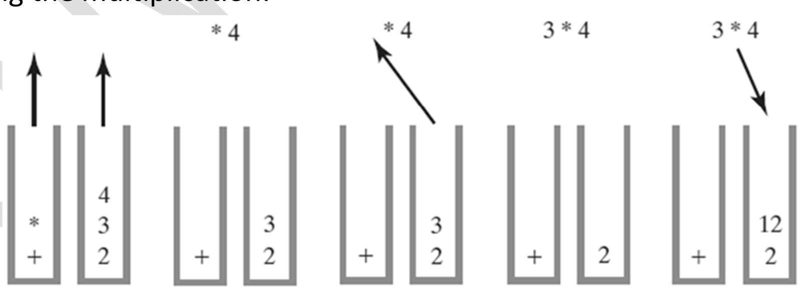
Two-stack algorithm. [E. W. Dijkstra]

- 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.

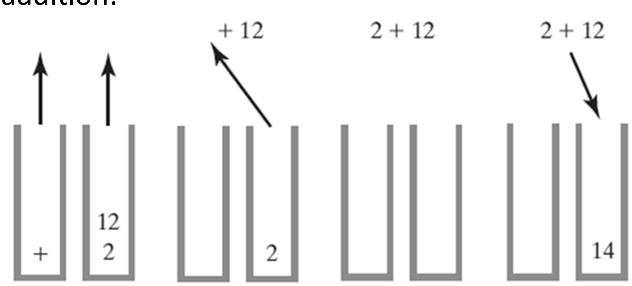
Example: evaluate a + b * c when a is 2, b is 3, and c is 4:

Step 1: Fill the two stacks until reaching the end of the expression:

Step 2: performing the multiplication:



Step 3: performing the addition:



Algorithm to evaluate infix expression:

```
Algorithm evaluateInfix(infix)
operatorStack = a new empty stack
valueStack = a new empty stack
while (infix has characters left to process) {
   nextCharacter = next nonblank character of infix
   switch (nextCharacter) {
      case variable:
         valueStack.push(value of the variable nextCharacter)
         break
      case '^' :
         operatorStack.push(nextCharacter)
         break
      case '+' : case '-' : case '*' : case '/' :
         while (!operatorStack.isEmpty() and
               precedence of nextCharacter <= precedence of operatorStack.peek()) {
            // Execute operator at top of operatorStack
            topOperator = operatorStack.pop()
            operandTwo = valueStack.pop()
            operandOne = valueStack.pop()
            result = the result of the operation in topOperator and its operands
                     operandOne and operandTwo
            valueStack.push(result)
         }
         operatorStack.push(nextCharacter)
         break
      case '(' :
         operatorStack.push(nextCharacter)
         break
      case ')' : // Stack is not empty if infix expression is valid
         topOperator = operatorStack.pop()
         while (topOperator != '(') {
            operandTwo = valueStack.pop()
            operandOne = valueStack.pop()
            result = the result of the operation in topOperator and its operands
                     operandOne and operandTwo
            valueStack.push(result)
            topOperator = operatorStack.pop()
         }
         break
      default: break // Ignore unexpected characters
   }
}
while (!operatorStack.isEmpty()) {
  topOperator = operatorStack.pop()
  operandTwo = valueStack.pop()
  operandOne = valueStack.pop()
  result = the result of the operation in topOperator and its operands
           operandOne and operandTwo
  valueStack.push(result)
}
return valueStack.peek()
```

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 **a + b * c** 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: Successive Operators with Same Precedence: **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: Successive Operators with Same Precedence: **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 − + *	

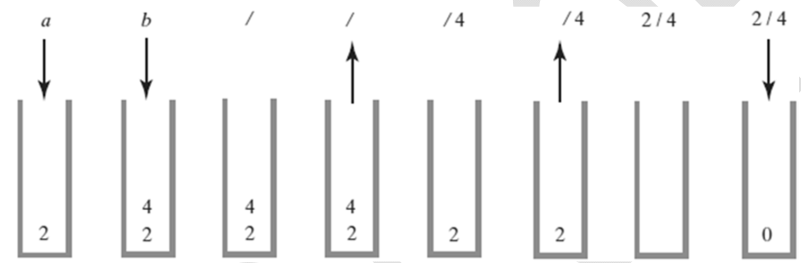
Infix-to-postfix Algorithm:

```
Algorithm convertToPostfix(infix)
 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
```

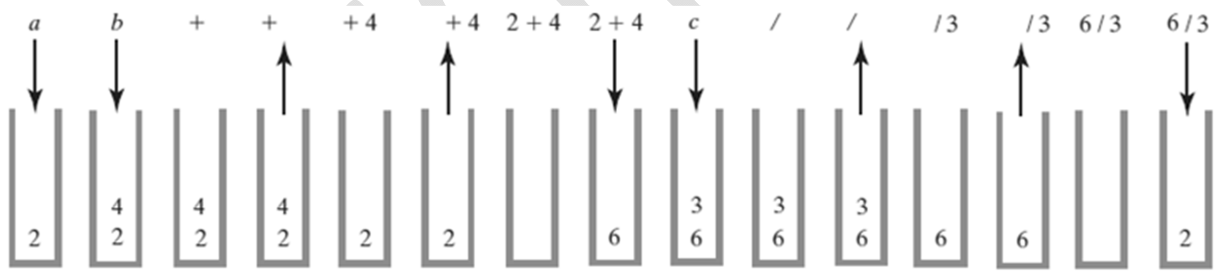
Evaluating Postfix Expressions

- 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 (right hand side) **rhs** parameter to the binary operator and that the **2nd** item popped is the (left hand side) **lhs** parameter; thus **parameters are popped in reverse order**.
  - For addition and 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***



Self exercises:

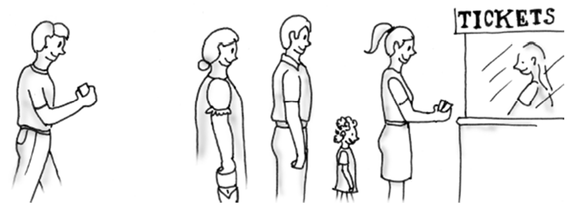
- **2 3 4 + * 6 -** ➔ **8.0**
- **2 3 + 7 9 / -** ➔ **4.222**
- **10 2 8 * + 3 -** ➔ **23.0**
- **1 2 - 4 5 ^ 3 * 6 * 7 2 2 ^ ^ / -** ➔ **-8.67**

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
    }
}
```

Queues

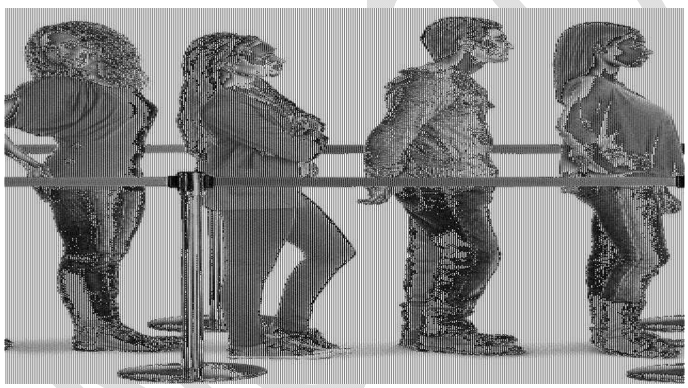
- A **queue** is another name for a **waiting line**:



- Used within operating systems and to simulate real-world events.
  - Come into play whenever processes or events must wait
- Entries organized **first-in**, **first-out**.

Terminology

- Item added first, or earliest, is at the front of the queue
- Item added most recently is at the back of the queue
- Additions to a software queue must occur at its back.
- Client can look at or remove only the entry at the front of the queue



Tail
Last
Back

FIFO: First In First Out

Head
First
Front

The ADT Queue

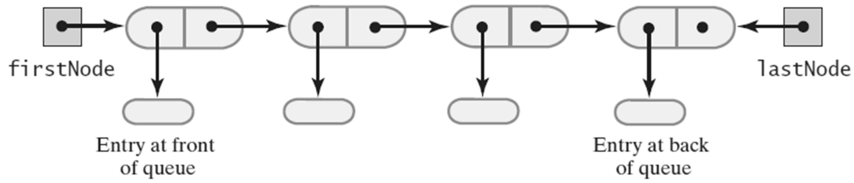
DATA

- A collection of objects in chronological order and having the same data type

OPERATIONS

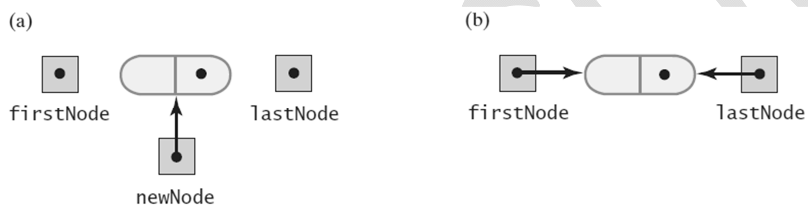
PSEUDOCODE	UML	DESCRIPTION
`enqueue(newEntry)`	`+enqueue(newEntry: integer): void`	Task: Adds a new entry to the back of the queue.
`dequeue()`	`+dequeue(): T`	Task: Removes and returns the entry at the front of the queue.
`getFront()`	`+getFront(): T`	Task: Retrieves the queue's front entry without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

Linked-list Representation of a Queue

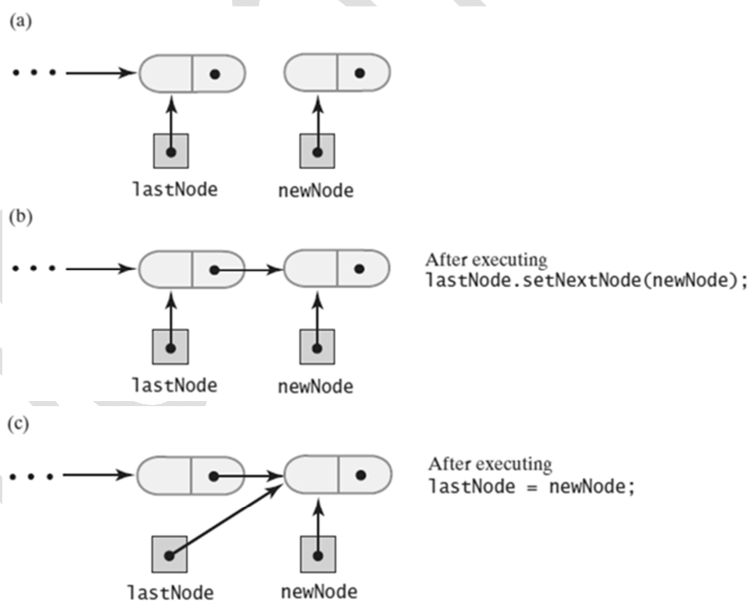



```java
public class linkedQueue <T extends Comparable<T>> {
    private Node<T> first;
    private Node<T> last;

    public boolean isEmpty(){   return (first==null) && (last==null);   }
    public void clear(){
        first = null;
        last = null;
    }
}
```

- The definition of **enqueue** Performance is **O(1)**:
  - Adding a new node to an empty chain



  - Adding a new node to the end of a nonempty chain that has a **tail** reference

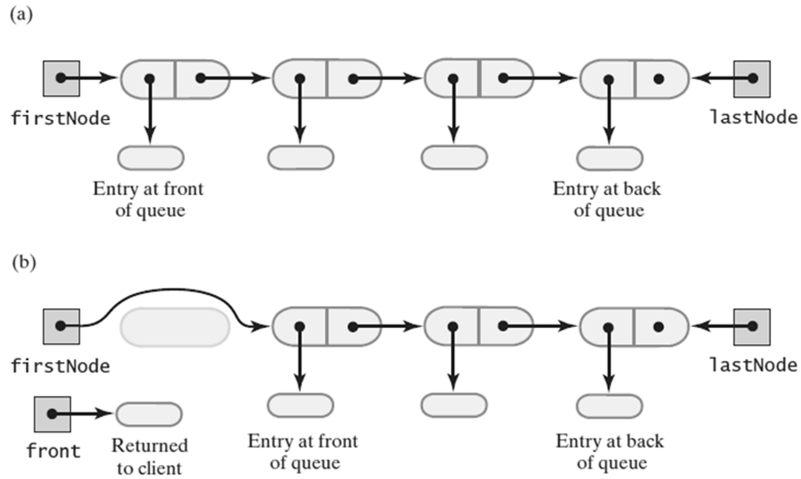



```java
public void enqueue(T data){
    Node<T> newNode = new Node<T>(data);
    if(isEmpty())
        first=newNode;
    else
        last.next = newNode;
    last = newNode;
}
```

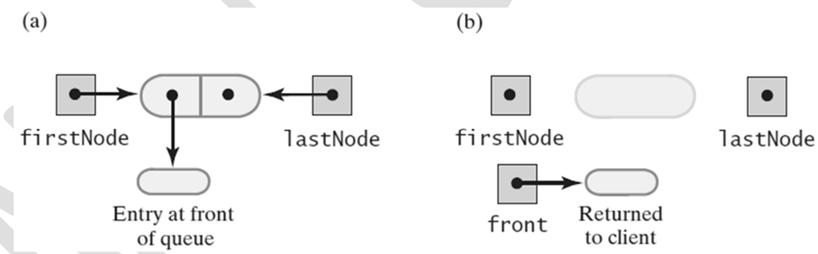
- Retrieving the front entry:

```
public T getFront(){
   if(!isEmpty())
      return first.data;
   return null;
}
```

- Removing the front entry (**dequeue**):
  - A queue of more than one entry:



  - A queue of one entry:

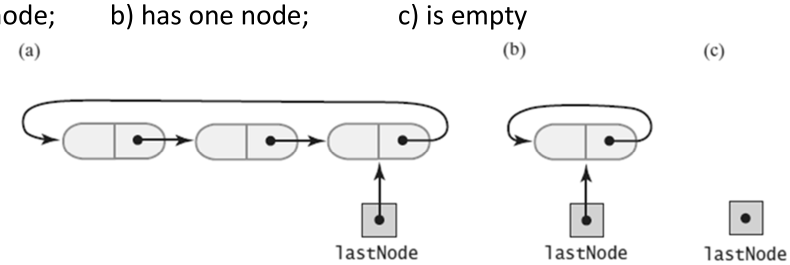



```
public T dequeue(){
   T front = getFront();
   if(!isEmpty())
        first = first.next;
   if(first==null)
        last = null;
   return front;
}
```

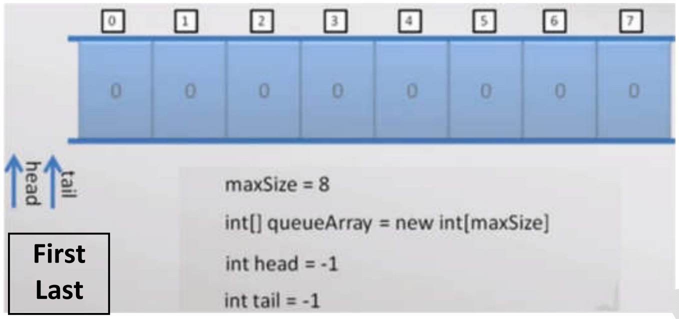
Circular Linked Implementations of a Queue

A circular linked chain with an external reference to its last node that

a) has more than one node; b) has one node; c) is empty

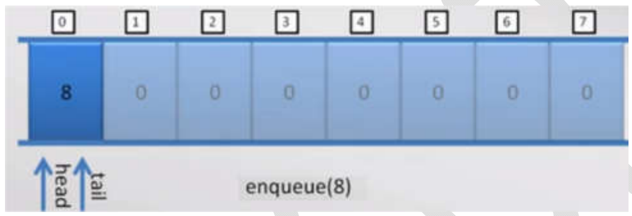


Array implementation of a Queue

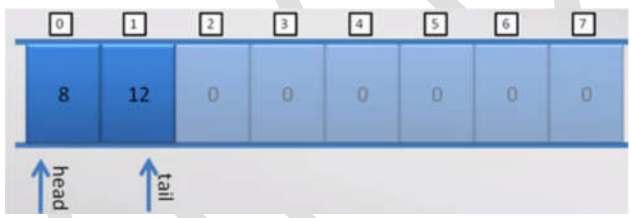


- **enqueue**(): add new item at after **last** (**tail**).
- **dequeue**(): remove item from **first** (**head**).

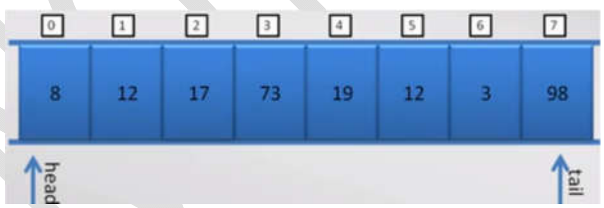
enqueue(8)



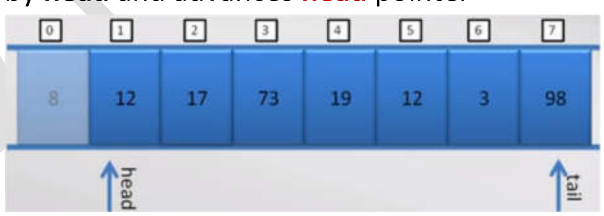
enqueue (12)



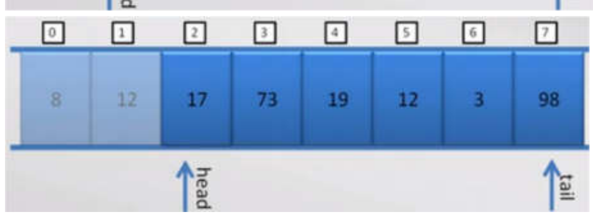
After a number of enqueues:



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

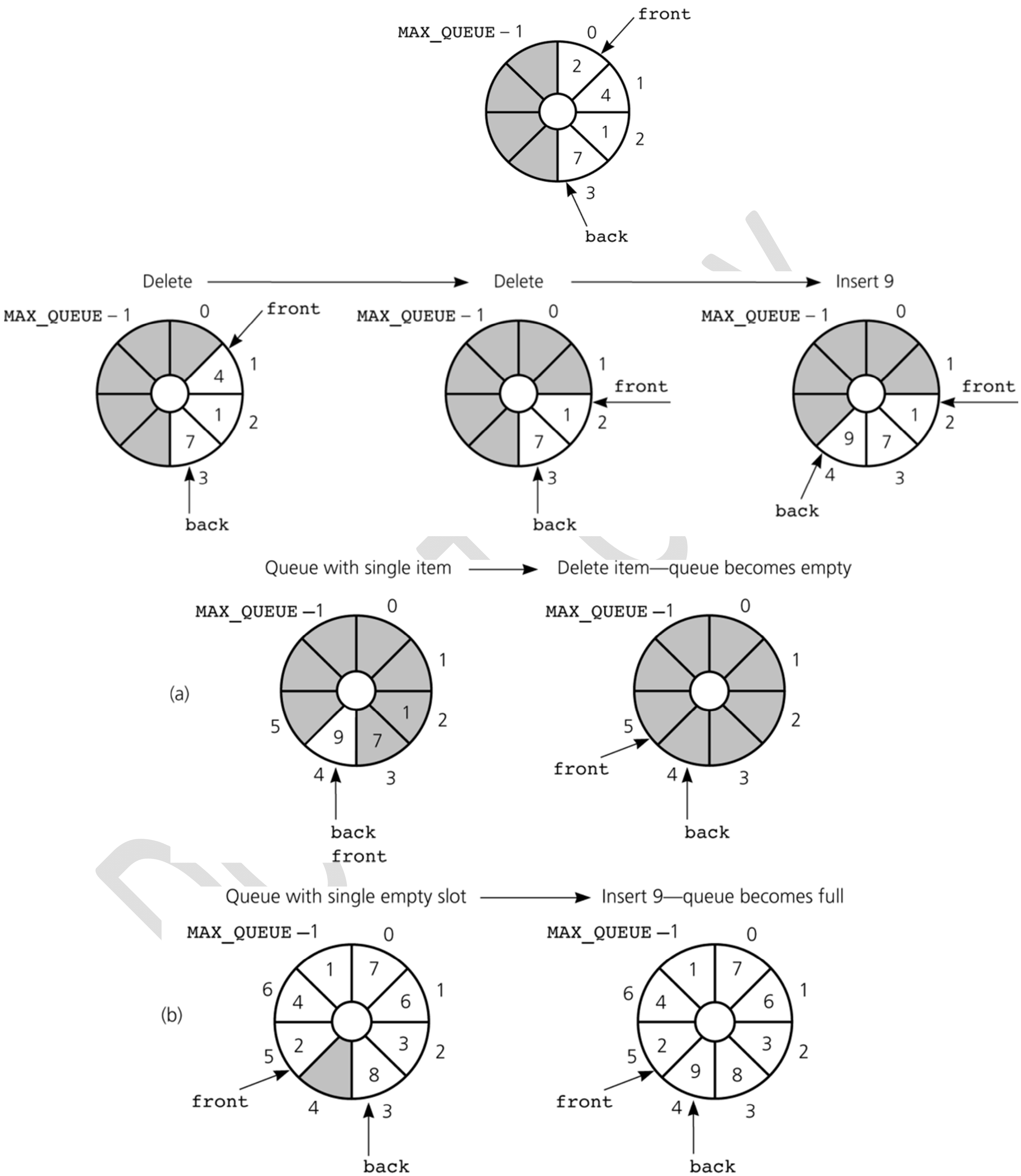


dequeue()



enqueue (27) ?? How to advance **tail**?? We have space at the beginning?? Shift??

Circular Queue



- **To detect circular queue-full and queue-empty conditions**
  - Keep a **count** of the queue items

- **To initialize the circular queue, set:**
  - front to -1
  - back to -1
  - count to 0

- **Inserting into a circular queue:**

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

- **Deleting from a circular queue:**

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


HW: Queue implementations using linked List and Arrays.

DE Queue (Double Ended Queue)

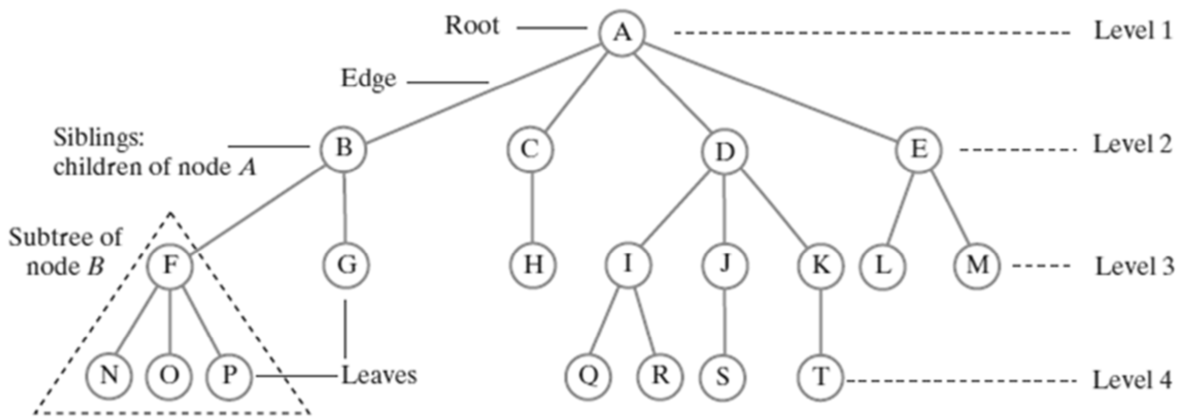
Allows add/remove elements from both head/tail.

Trees

Revision:

	Sorted Arrays	Sorted Linked List
Search	**Fast O(log n)**	**Slow O(n)**
Insert	**Slow O(n)**	**Slow O(n)**
Delete	**slow O(n)**	**Slow O(n)**

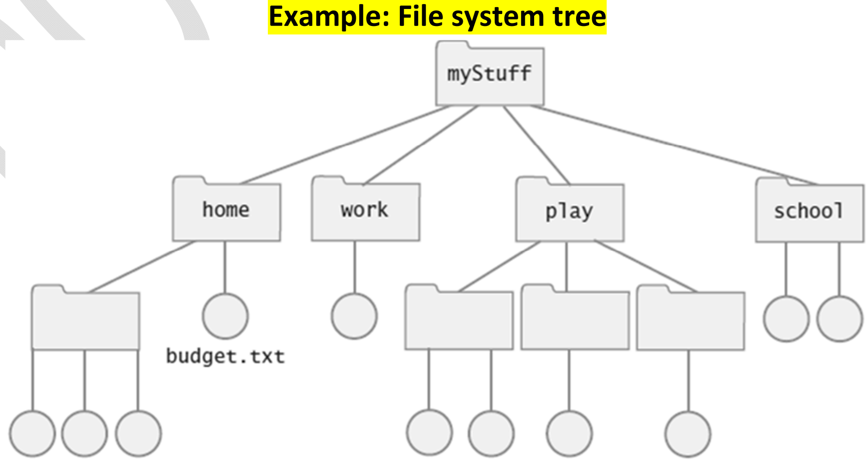
Tree



- 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.

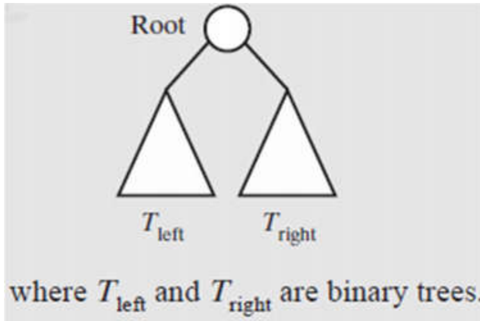
Example: Family Trees (one parent)

Example: File system tree

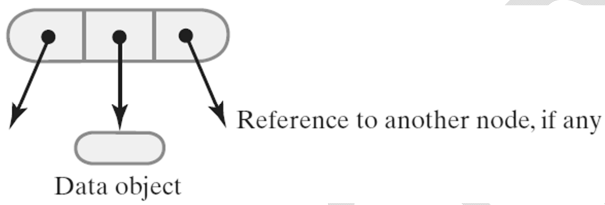


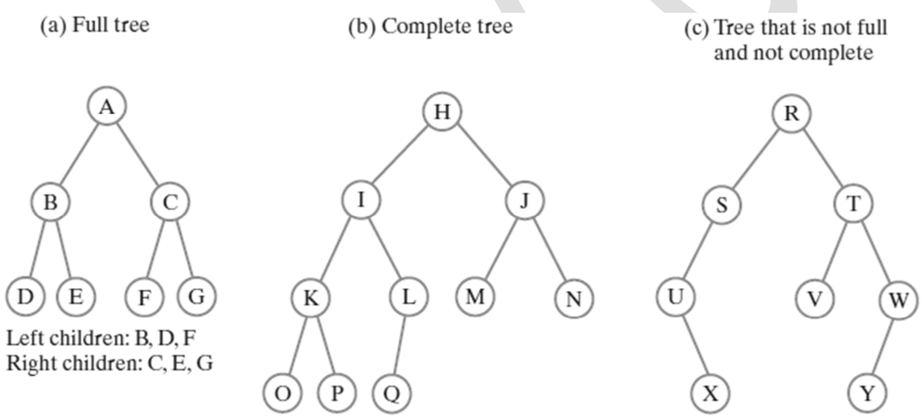
Binary Trees

- A **binary tree** is a tree in which no node can have more than **two** children:



- Binary Tree **Node**:



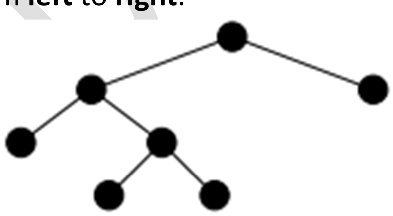


(a) Each node in a **full binary tree** is either:

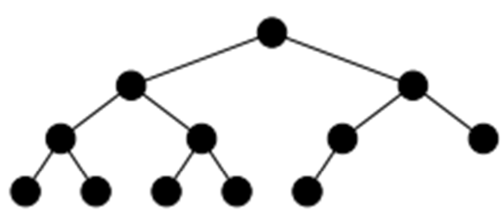
(1) an internal node with exactly two non-empty children or

(2) a leaf.

(b) A **complete binary tree** has a restricted shape obtained by starting at the root and filling the tree by levels from **left** to **right**.

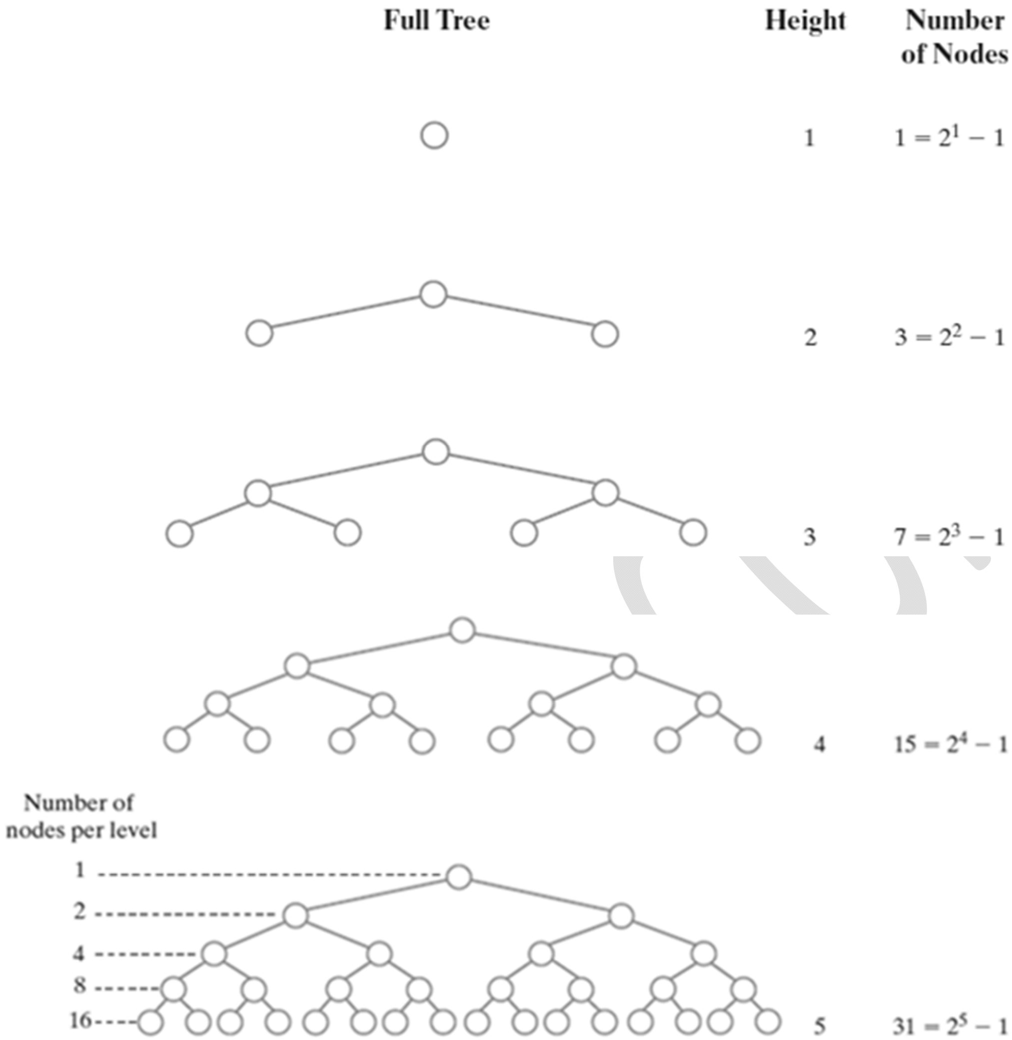


(a) This tree is full (but not complete).



(b) This tree is complete (but not full).

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



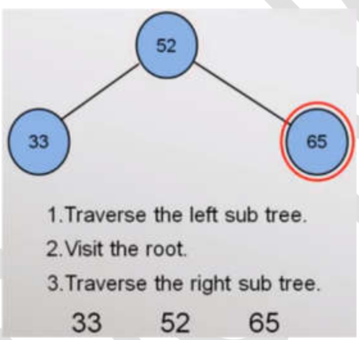
Implementation:

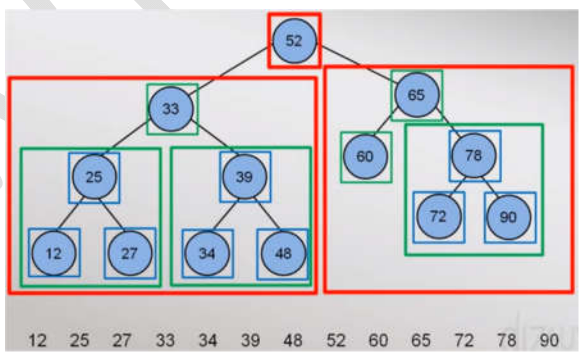
```
public class TNode<T extends Comparable<T>> {
   T data;
   TNode left;
   TNode right;

   public TNode(T data) {    this.data = data; }
   public void setData(T data)   {  this.data=data;  }
   public T getData()    {  return data;  }
   public TNode getLeft() {  return left;  }
   public void setLeft(TNode left) {  this.left = left; }
   public TNode getRight() {  return right;  }
   public void setRight(TNode right) { this.right = right;}
   public boolean isLeaf(){   return (left==null && right==null);   }
   public boolean hasLeft(){  return left!=null;  }
   public boolean hasRight(){  return right!=null;  }
   public String toString() {      return "[" + data + "]";   }
}
```

Tree Traversal

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

- **In-Order Traversal:** Visit **root** of a binary tree between visiting nodes in root's subtrees.

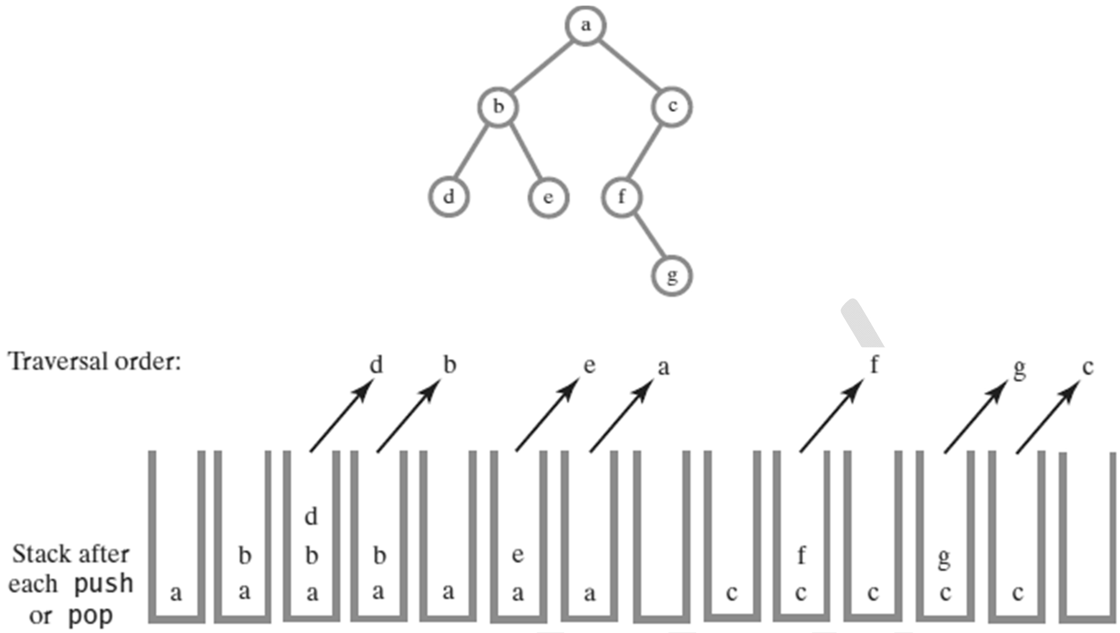




  - **Recursive implementation:**

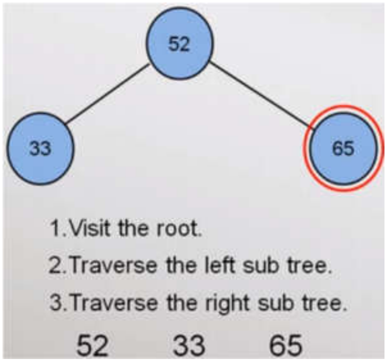
```
public void traverseInOrder() {  traverseInOrder(root);   }
public void traverseInOrder(TNode node) {
   if (node != null) {
      if (node.left != null)
         traverseInOrder(node.left);
      System.out.print(node + " ");
      if (node.right != null)
         traverseInOrder(node.right);
   }
}
```

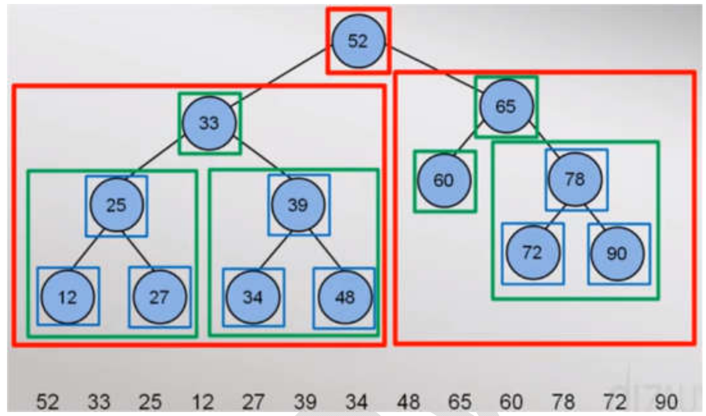
- **Using a stack to perform an in-order traversal iteratively: (Optional)**



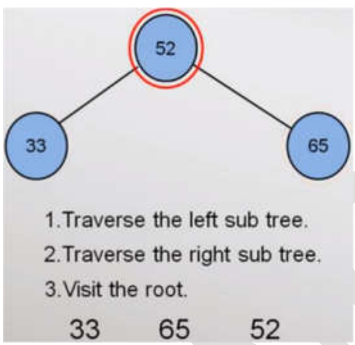

```
public void iterativeInorderTraverse()
{
   StackInterface<BinaryNodeInterface<T>> nodeStack = new LinkedStack<>();
   BinaryNode<T> currentNode = root;
   while (!nodeStack.isEmpty() || (currentNode != null))
   {
      // Find leftmost node with no left child
      while (currentNode != null)
      {
         nodeStack.push(currentNode);
         currentNode = currentNode.getLeftChild();
      } // end while
      // Visit leftmost node, then traverse its right subtree
      if (!nodeStack.isEmpty())
      {
         BinaryNode<T> nextNode = nodeStack.pop();
         assert nextNode != null;  // Since nodeStack was not empty
                                   // before the pop
         System.out.println(nextNode.getData());
         currentNode = nextNode.getRightChild();
      } // end if
   } // end while
} // end iterativeInorderTraverse
```

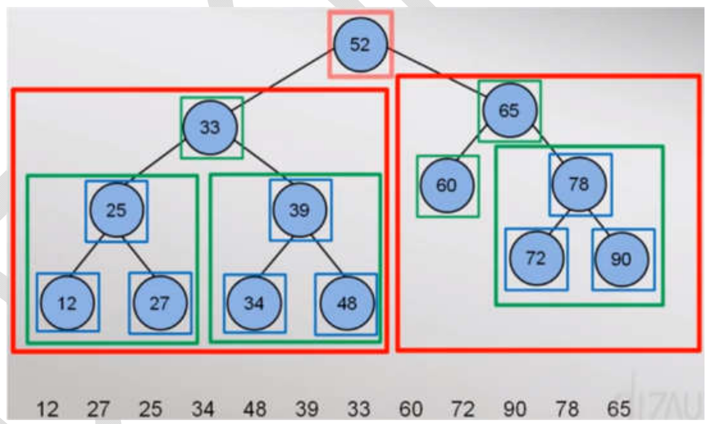
- **Pre-Order Traversal:** Visit **root** before we visit root's subtrees.



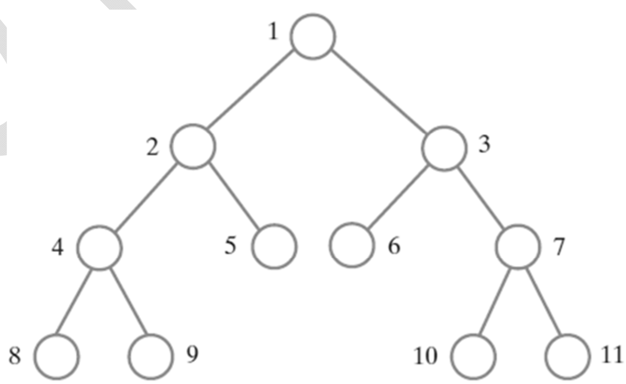


- **Post-Order Traversal:** Visit **root** of a binary tree after visiting nodes in root's subtrees.





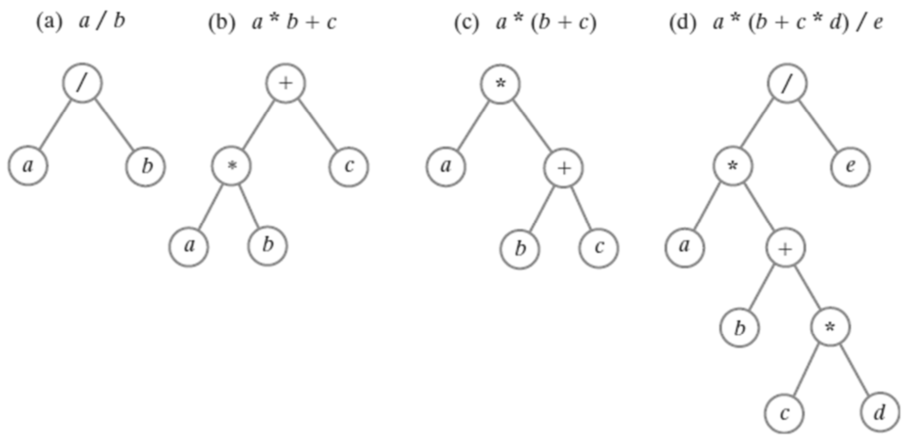
- **Level-Order Traversal:** Begin at **root** and visit nodes one level at a time.
  - The visitation order of a level-order traversal:



  - Level-order traversal is implemented via a **queue**.
  - The traversal is a breadth-first search.

HW: implement level-order traversal

Expression Trees



- 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.

Algorithm for evaluation of an expression tree:

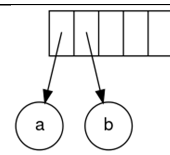
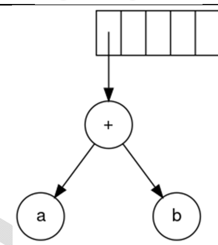
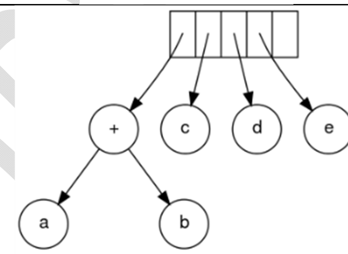
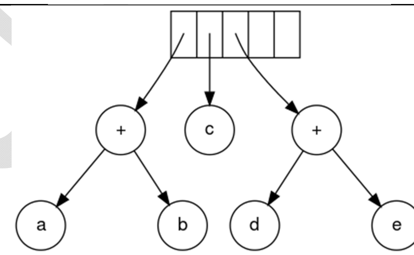
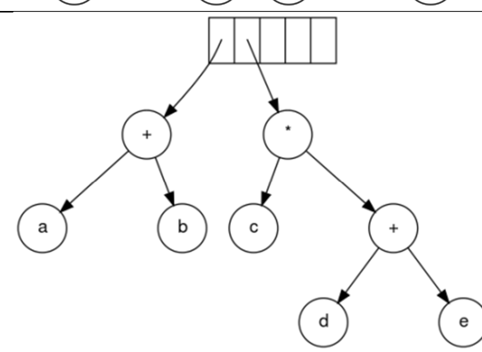
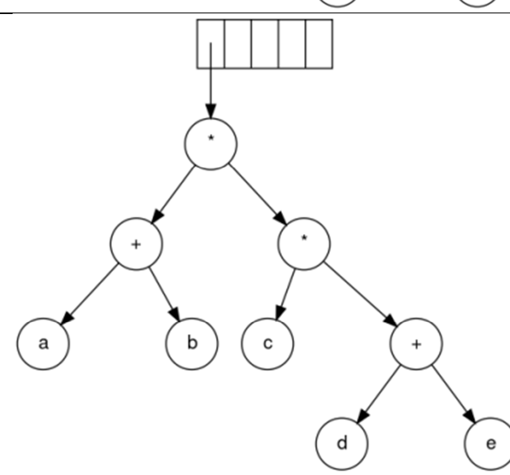
```
Algorithm evaluate(expressionTree)
if (expressionTree is empty)
    return 0
else
{
    firstOperand = evaluate(left subtree of expressionTree)
    secondOperand = evaluate(right subtree of expressionTree)
    operator = the root of expressionTree
    return the result of the operation operator and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

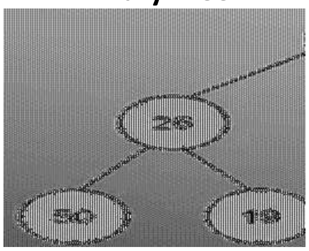
Example: (a b + c d e + * *)

• 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.	

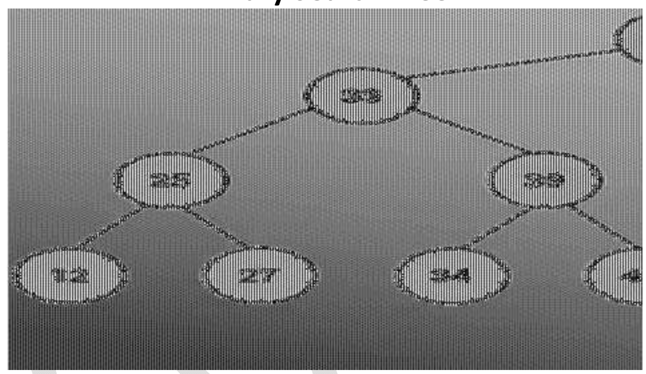
Binary Search Trees (BST)

- **Problem**: searching in binary tree takes **O(n)**.
- **Solution**: forming a binary search tree.
- 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 (***or equal if duplication is allowed***) than the item in **X**.

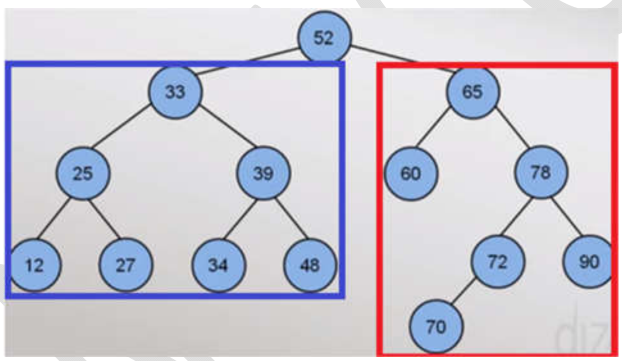
Binary Tree



Binary Search Tree



- Every node in a binary search tree is the root of a binary search tree.



- **Search for an item:**

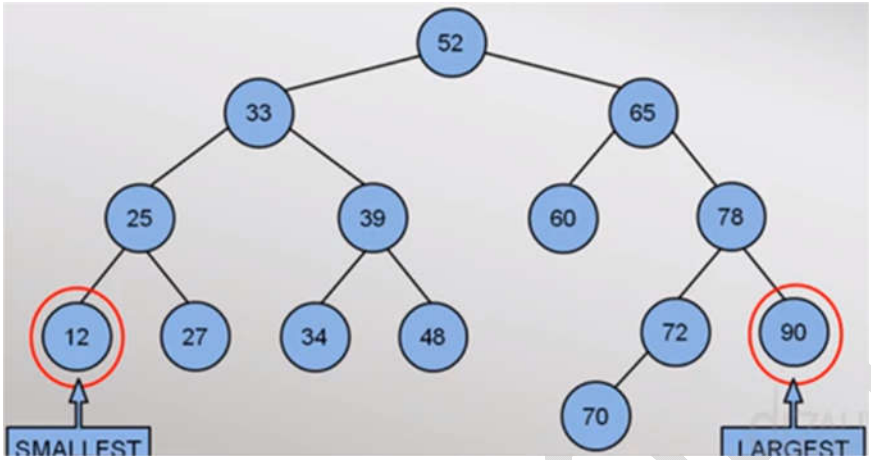
Example: find(52) , find(39) , find(35)

```
public TNode find(T data) {  return find(data, root);   }
public TNode find(T data, TNode node) {
   if (node!= null) {
      int comp = node.data.compareTo(data);
      if (comp == 0)
         return node;
      else if (comp > 0 && node.hasLeft())          return find(data, node.left);
      else if (comp < 0 && node.hasRight())         return find(data, node.right);
   }
   return null;
}
```

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

However, to make searching a binary search tree as efficient as possible, tree must be as **short** as possible.

Finding Max and Min Values:

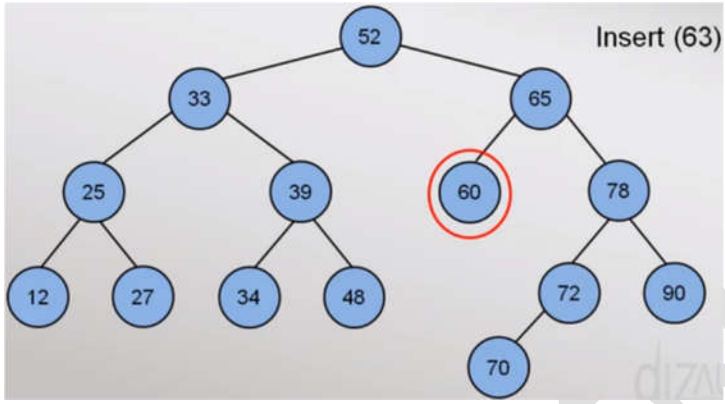


- The find **Min** operation is performed by following left nodes as long as there is a **left** child.
- The find **Max** operation is similar.

```
public TNode largest() {  return largest(root);   }
public TNode<T> largest(TNode node) {
   if(node!= null){
      if(!node.hasRight())
         return (node);
      return largest(node.right);
   }
   return null;
}

public TNode smallest() {  return smallest(root);   }
public TNode<T> smallest(TNode node) {
   if(node!= null){
      if(!node.hasLeft())
         return (node);
      return smallest(node.left);
   }
   return null;
}
```

Insert in Binary Search Tree:

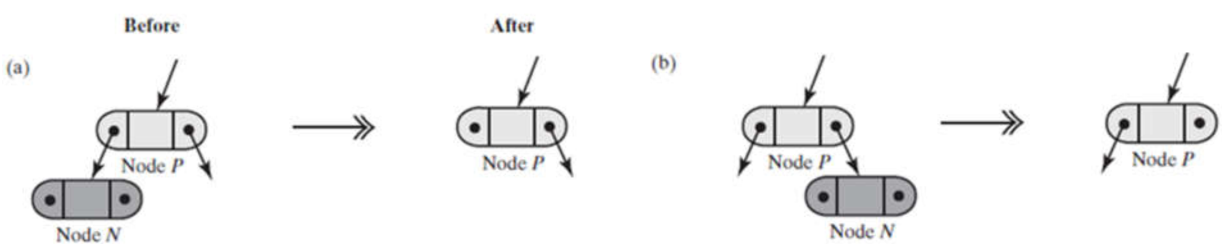
Example: insert(63)



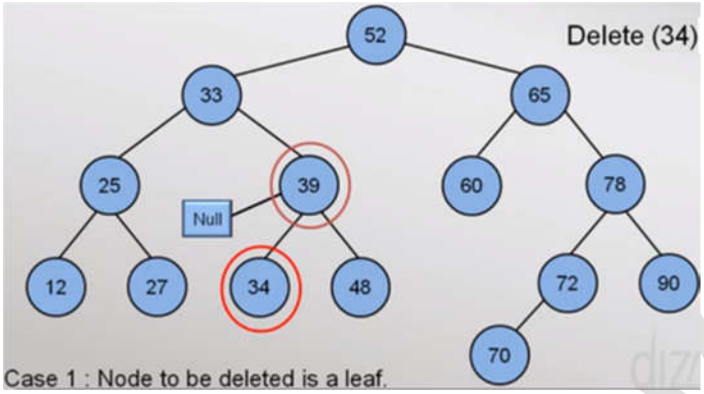

```
public void insert(T data) {
   if (isEmpty())
      root = new TNode(data);
   else
      insert(data, root);
}
public void insert(T data, TNode node) {
   if (data.compareTo((T) node.data) >= 0) { // insert into right subtree
      if (!node.hasRight())
         node.right = new TNode(data);
      else
         insert(data, node.right);
   } else {      // insert into left subtree
      if (!node.hasLeft())
         node.left = new TNode(data);
      else
         insert(data, node.left);
   }
}
```

Deleting a Node:

Case 1: Node to be deleted is a leaf. Two possible configurations of a leaf node N:
Being a **left** child or a **right** child:

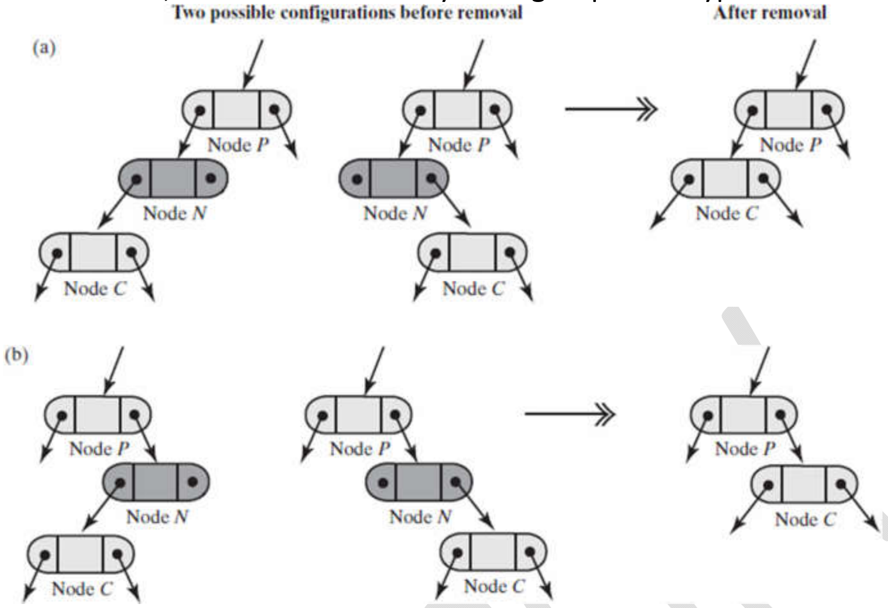


Example: delete(34)

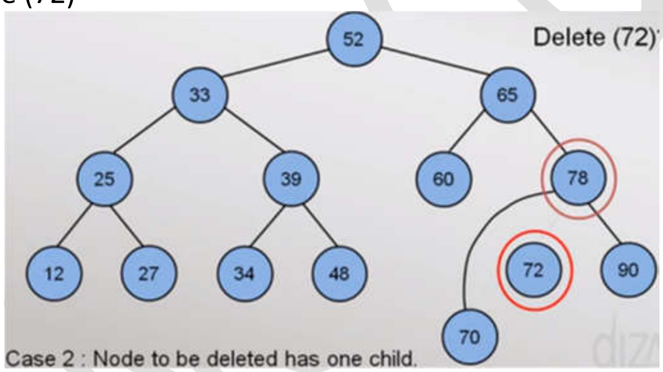



```java
public TNode delete(T data) {
   TNode current = root;
   TNode parent =  root;
   boolean isLeftChild = false;

   if (isEmpty()) return null;// tree is empty
   while (current != null && !current.data.equals(data)) {
      parent = current;
      if (data.compareTo((T)current.data) < 0) {
         current = current.left;
         isLeftChild = true;
      } else {
         current = current.right;
         isLeftChild = false;
      }
   }
   if (current == null) return null; // node to be deleted not found

   // case 1: node is a leaf
   if (!current.hasLeft() && !current.hasRight()) {
      if (current == root)  // tree has one node
         root = null;
      else {
         if (isLeftChild)           parent.left = null;
         else                       parent.right = null;
      }
   }

   // other cases
   return current;
}
```

Case 2: If a node has one child, it can be removed by having its parent bypass it.



Example: delete (72)

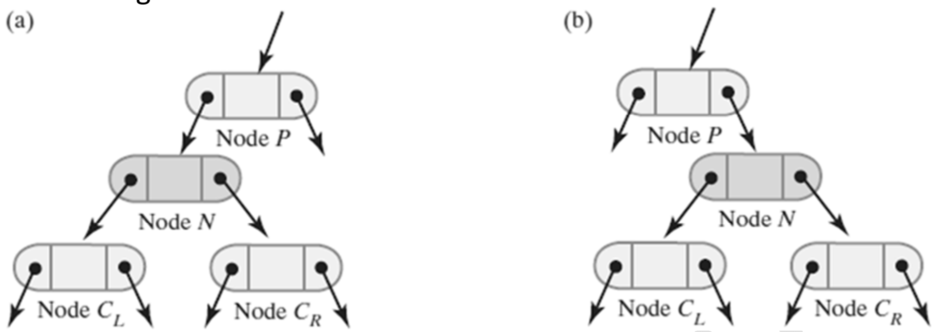


Note: The **root** is a special case because it does not have a parent.

```
// Case 2 broken down further into 2 separate cases
else if (current.hasLeft()) { // current has left child only
   if (current == root) {
      root = current.left;
   } else if (isLeftChild) {
      parent.left = current.left;
   } else {
      parent.right = current.left;
   }
} else if (current.hasRight()) { // current has right child only
   if (current == root) {
      root = current.right;
   } else if (isLeftChild) {
      parent.left = current.right;
   } else {
      parent.right = current.right;
   }
}
```

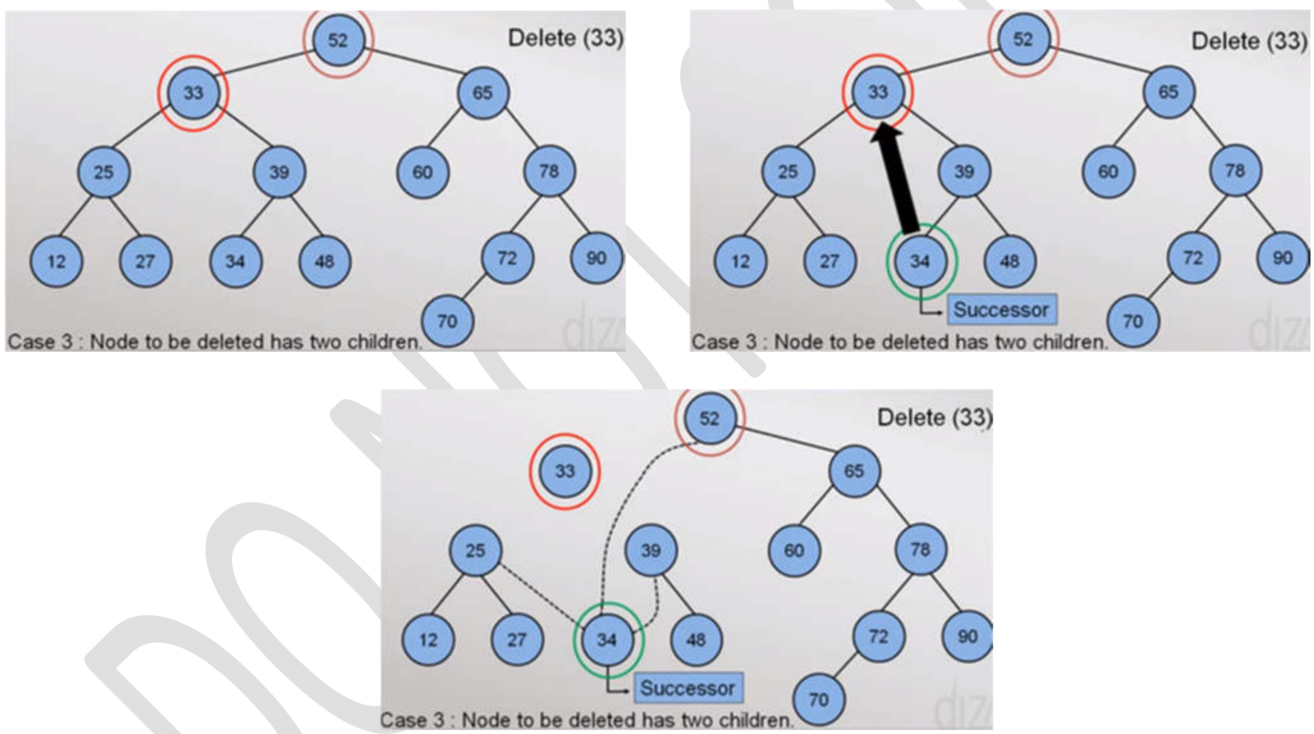
Case 3:

- Two possible configurations of a node N that has two children:

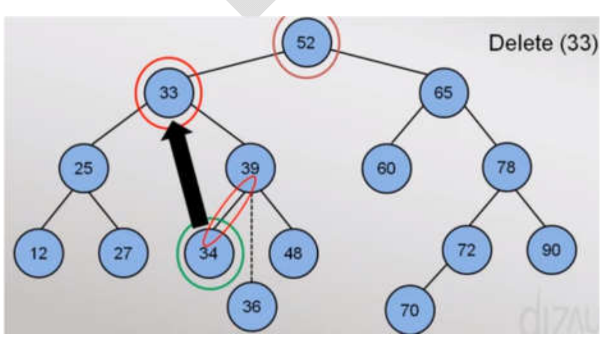


- A node with two children is replaced by using the **smallest** item in the right subtree (**Successor**).

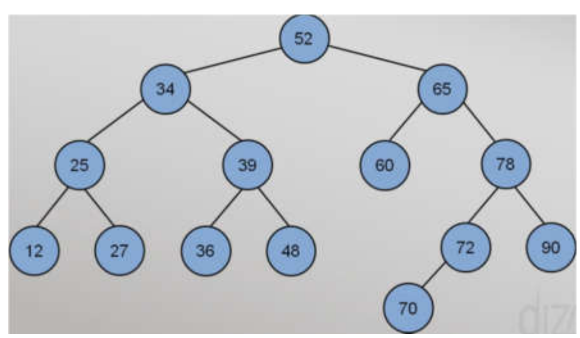
Example: delete(33)



What if node **34** has a right child (e.g. **36**)?



➔



```java
// case 3: node to be deleted has 2 children
else {
    Node successor = getSuccessor(current);
    if (current == root)
        root = successor;
    else if (isLeftChild) {
            parent.left= successor;
        } else {
            parent.right = successor;
        }
    successor.left = current.left;
}
```

```java
private Node getSuccessor(Node node) {
    Node parentOfSuccessor = node;
    Node successor = node;
    Node current = node.right;
    while (current != null) {
        parentOfSuccessor = successor;
        successor = current;
        current = current.left;
    }
    if (successor != node.right) {  // fix successor connections
        parentOfSuccessor.left = successor.right;
        successor.right = node.right;
    }
    return successor;
}
```

Soft Delete (lazy deletion):

When an element is to be deleted, it is left in the tree and simply **marked** as being deleted.

- If a deleted item is reinserted, the overhead of allocating a new cell is avoided.

Tree Height:

```java
public int height() {  return height(root); }
public int height(TNode node) {
    if (node == null) return 0;
    if (node.isLeaf()) return 1;
    int left = 0;
    int right = 0;
    if (node.hasLeft())       left = height(node.left);
    if (node.hasRight())      right = height(node.right);
    return (left > right) ? (left + 1) : (right + 1);
}
```

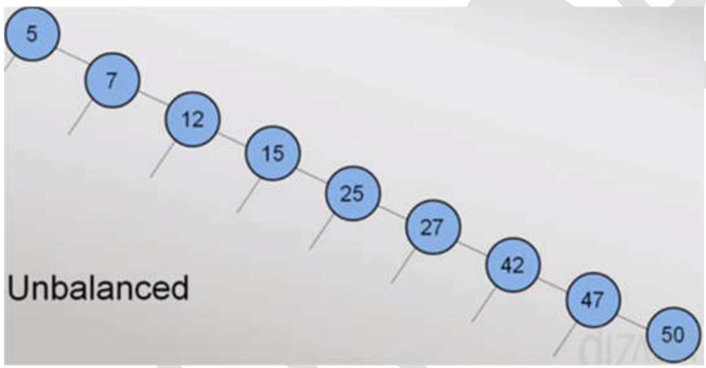
Efficiency of Operations:

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

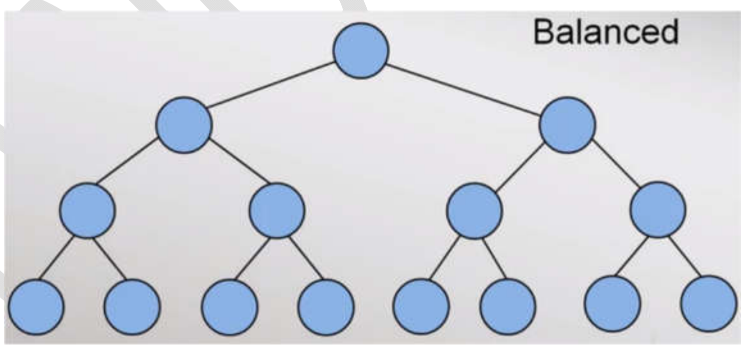
Unbalanced Tree:

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

Example: add 5, 7, 12, 15, 25, 27, 42, 47, 50



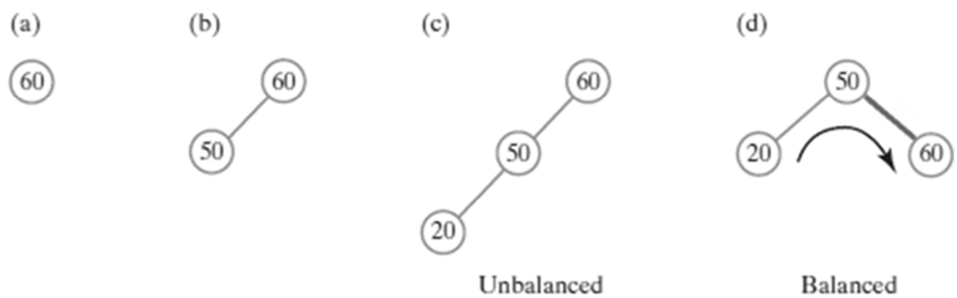
- If you add entries into an initially empty binary search tree, do not add them in sorted order.



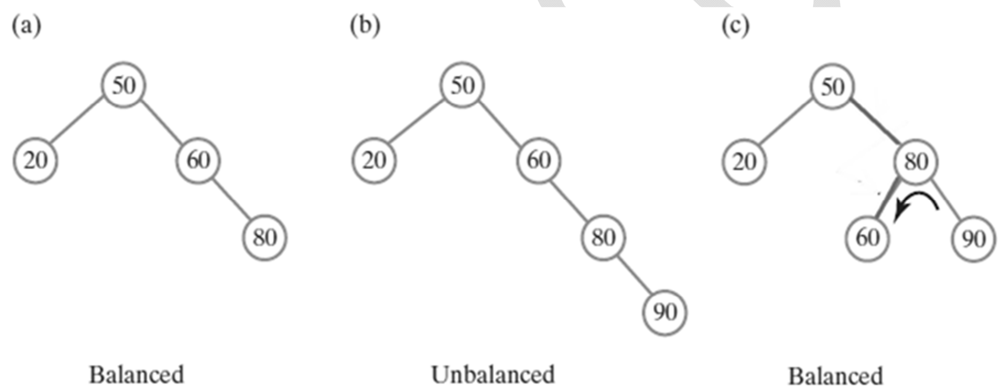
AVL Trees

- An **AVL tree (**Georgy **A**delson-**V**elsky and Evgenii **L**andis' 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 Rotation

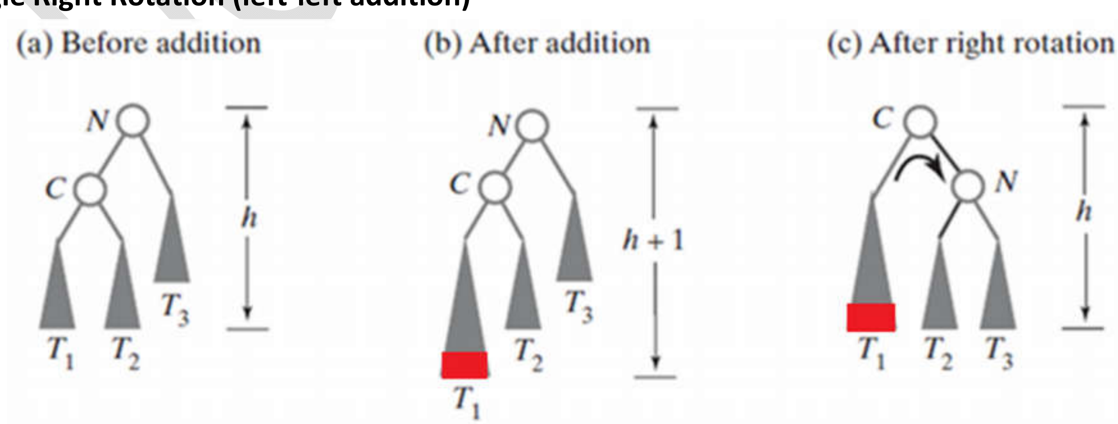


Example: After inserting (a) 60; (b) 50; and (c) 20 into an initially empty **BST**, the tree is **not balanced**; (d) a corresponding **AVL** tree rotates its nodes to restore balance



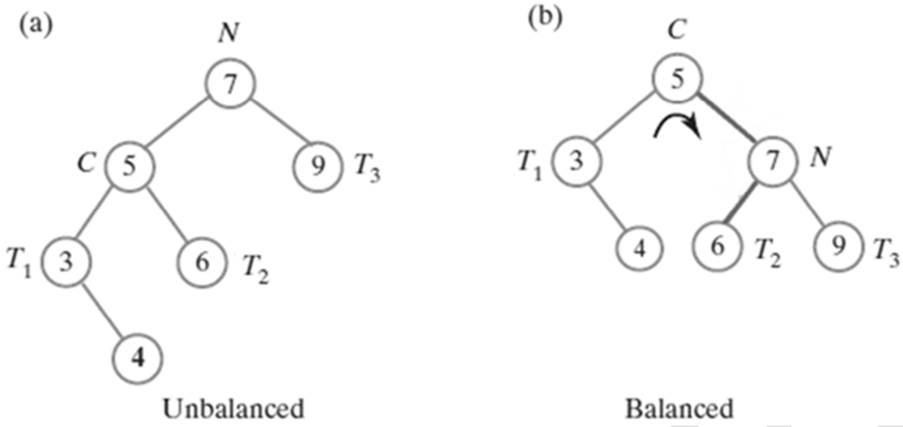
Example: (a) Adding 80 to the tree does not change the balance of the tree;
(b) a subsequent addition of 90 makes the tree **unbalanced** ;
(c) a left rotation restores its balance

Case 1: Single Right Rotation (left-left addition)



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

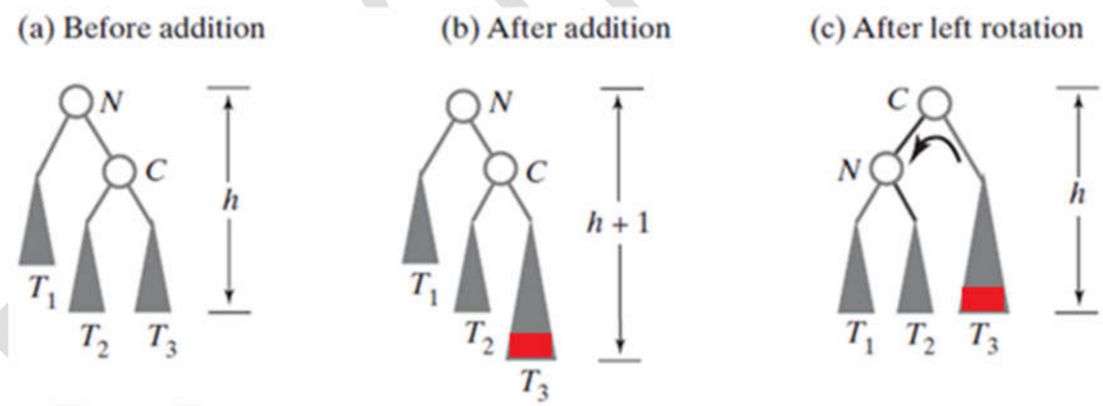
Example: a) before and b) after a **right rotation** restores balance to an **AVL** tree



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 (right-right addition)



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.*

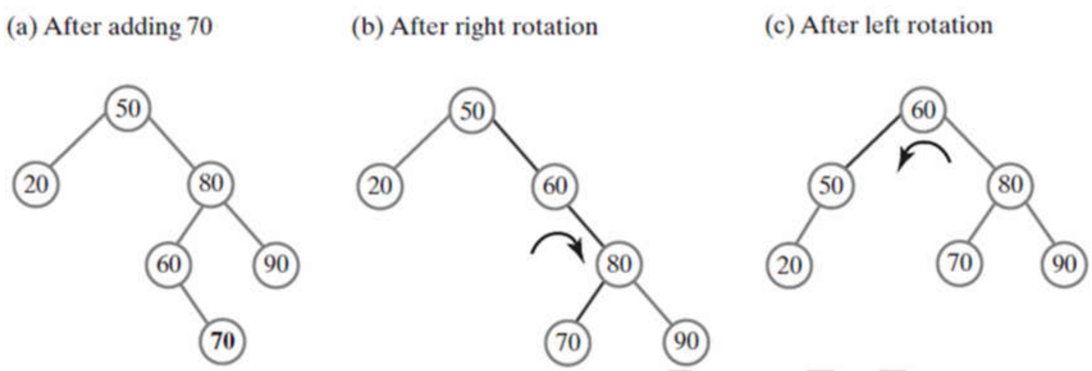
nodeC = *right child of* nodeN
Set nodeN*'s right child to* nodeC*'s left child*
Set nodeC*'s left child to* nodeN
return nodeC

Double Rotations

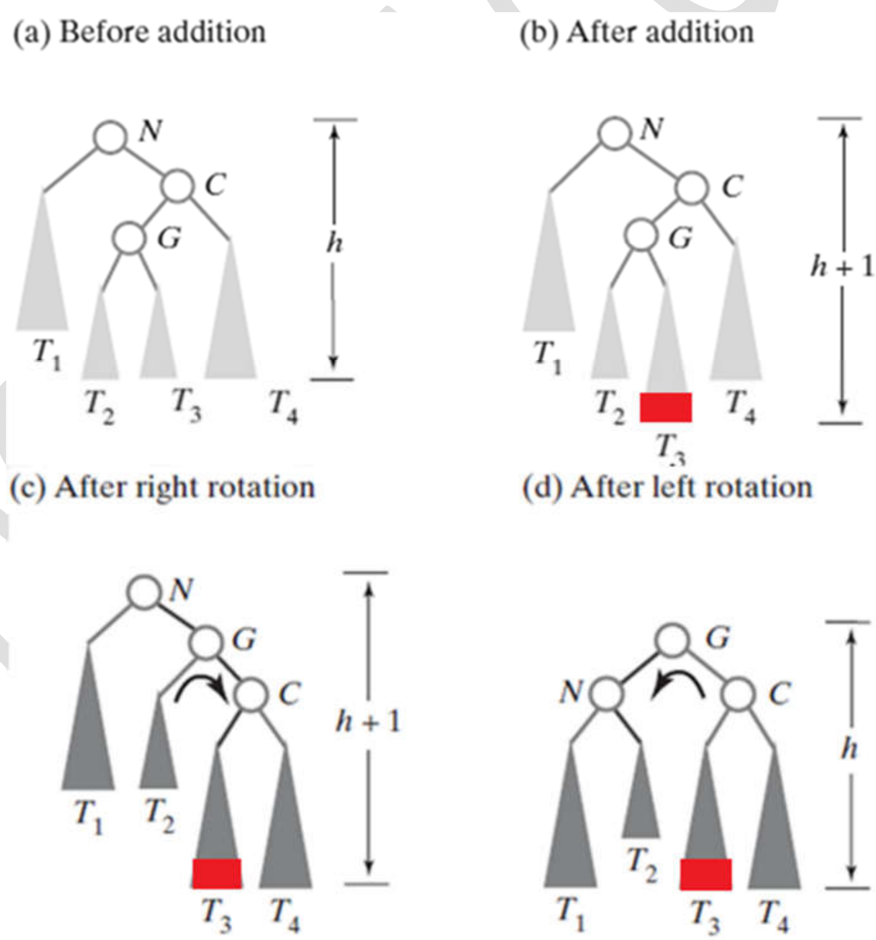
A **double rotation** is accomplished by performing two single rotations:

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 (right-left addition)



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

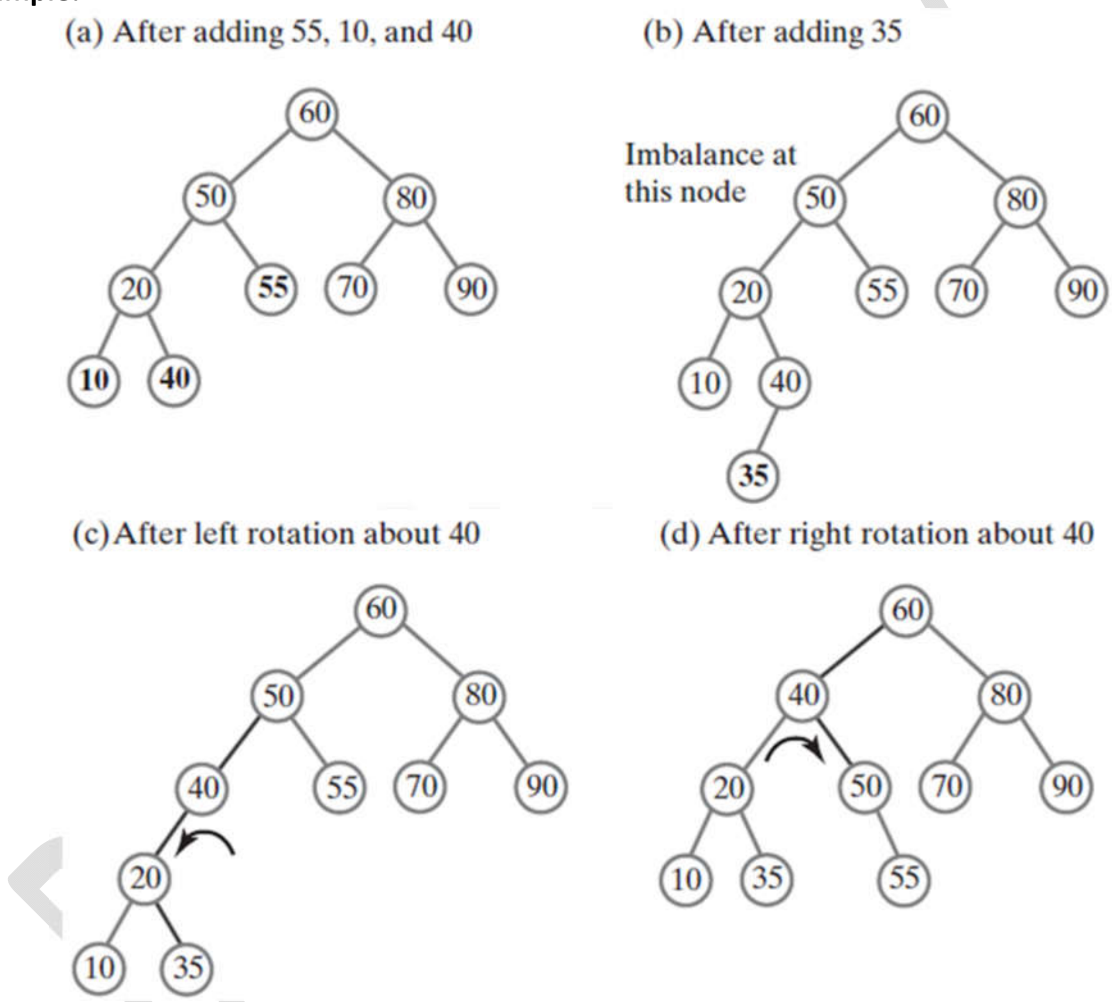


Before and after an addition to an **AVL** subtree that requires both a **right rotation** and a **left rotation** to maintain its balance

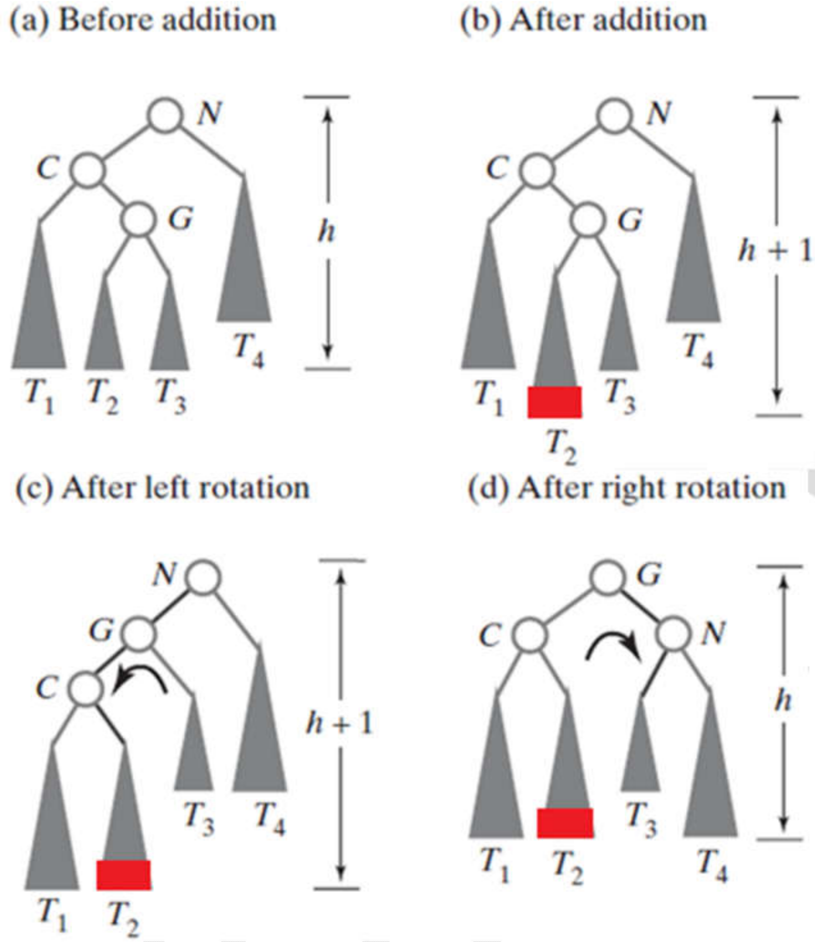
```
Algorithm rotateRightLeft(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the left subtree of nodeN's right child.

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 (left-right addition)

Example:



(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**



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)

Rebalance Code Implementation

- Pseudo-code to rebalance the tree:

```
Algorithm rebalance(nodeN)
if (nodeN's left subtree is taller than its right subtree by more than 1)
{   // Addition was in nodeN's left subtree
    if (the left child of nodeN has a left subtree that is taller than its right subtree)
        rotateRight(nodeN)        // Addition was in left subtree of left child
    else
        rotateLeftRight(nodeN)  // Addition was in right subtree of left child
}
else if (nodeN's right subtree is taller than its left subtree by more than 1)
{   // Addition was in nodeN's right subtree
    if (the right child of nodeN has a right subtree that is taller than its left subtree)
        rotateLeft(nodeN)         // Addition was in right subtree of right child
    else
        rotateRightLeft(nodeN)  // Addition was in left subtree of right child
}
```

```java
private TNode rebalance(TNode nodeN){
   int diff = getHeightDifference(nodeN);
   if ( diff > 1) { // addition was in node's left subtree
      if(getHeightDifference(nodeN.left)>0)
         nodeN = rotateRight(nodeN);
      else
         nodeN = rotateLeftRight(nodeN);
   }
   else if ( diff < -1){  // addition was in node's right subtree
      if(getHeightDifference(nodeN.right)<0)
         nodeN = rotateLeft(nodeN);
      else
         nodeN = rotateRightLeft(nodeN);
   }
   return nodeN;
}
```

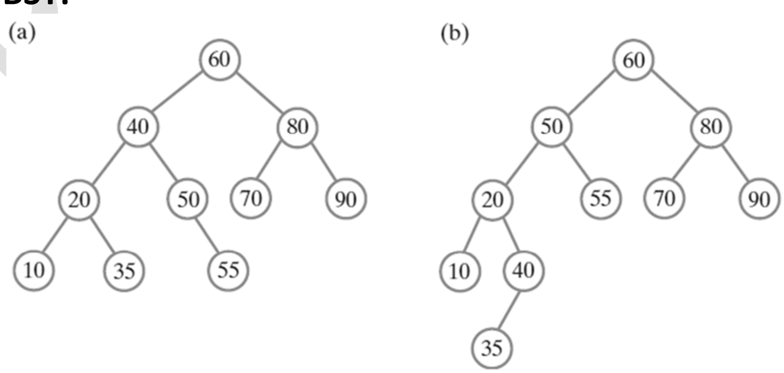
Insert Code Implementation:

```
public void insert(T data) {
   if(isEmpty())       root = new TNode<>(data);
   else {
      TNode rootNode = root;
      addEntry(data, rootNode);
      root = rebalance(rootNode);
   }
}

public void addEntry(T data, TNode rootNode){
   assert rootNode != null;
   if(data.compareTo((T)rootNode.data) < 0){ // right into left subtree
      if(rootNode.hasLeft()){
         TNode leftChild = rootNode.left;
         addEntry(data, leftChild);
         rootNode.left=rebalance(leftChild);
      }
      else         rootNode.left = new TNode(data);
   }
   else {  // right into right subtree
      if(rootNode.hasRight()){
         TNode rightChild = rootNode.right;
         addEntry(data, rightChild);
         rootNode.right=rebalance(rightChild);
      }
      else         rootNode.right = new TNode(data);
   }
}
```

Delete Code Implementation:

```
public TNode delete(T data) {
   TNode temp = super.delete(data);
   if(temp!= null){
      TNode rootNode = root;
      root = rebalance(rootNode);
   }
   return temp;
}
```

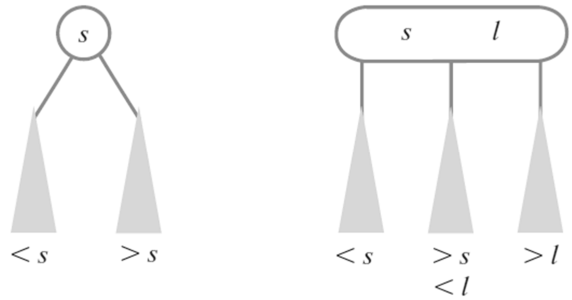
An **AVL** Tree versus a **BST**:



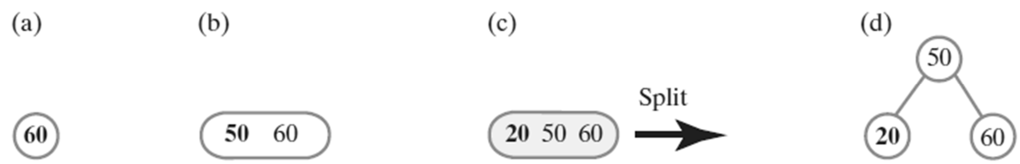
Example: The result of adding 60, 50, 20, 80, 90, 70, 55, 10, 40, and 35 to an initially empty (a) **AVL** tree; (b) **BST**

2-3 Trees

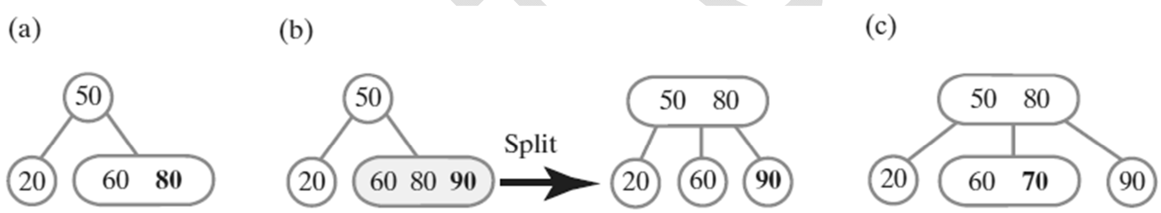
- **Definition**: general search tree whose interior nodes must have either **2** or **3** children.
  - A **2-node** contains one data item ***s*** and has two children.
  - A **3-node** contains two data items, ***s*** and ***l***, and has three children.



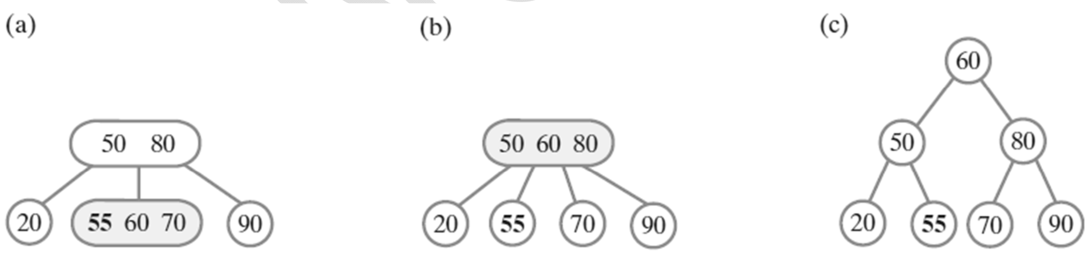
Adding Entries to a 2-3 Tree:



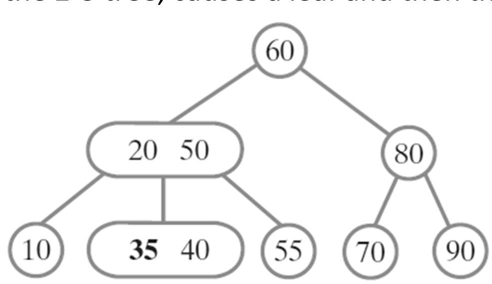
Adding (a) **60** and (b) **50**; (c), (d) adding **20** causes the 3-node to split



The 2-3 tree after adding (a) **80**; (b) **90**; (c) **70**



Adding **55** to the 2-3 tree, causes a leaf and then the root to split

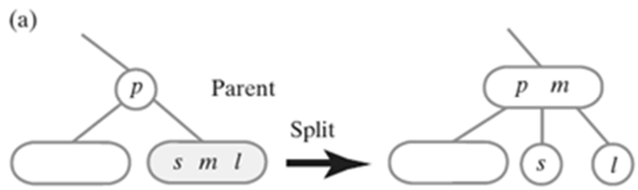


The 2-3 tree, after adding **10, 40, 35**

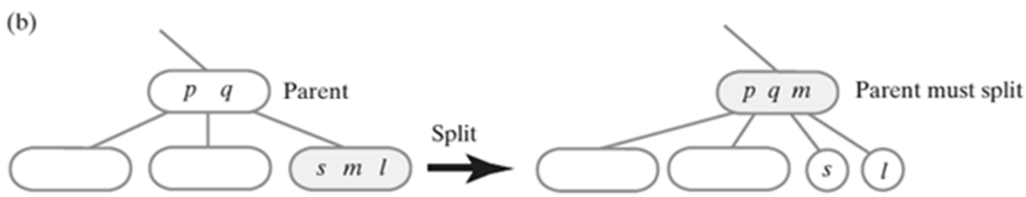
Splitting Nodes during Addition:

- Splitting a **leaf** to accommodate a new entry when the leaf's **parent** contains:

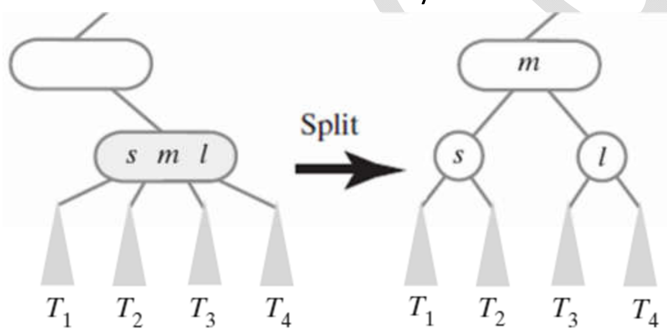
(a) one entry:



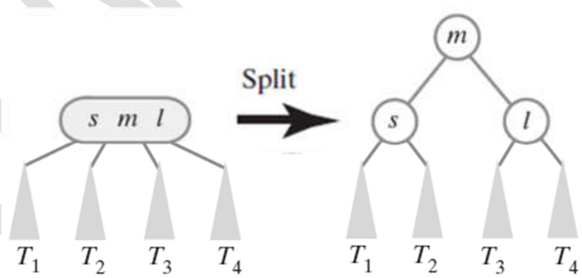
(b) two entries:



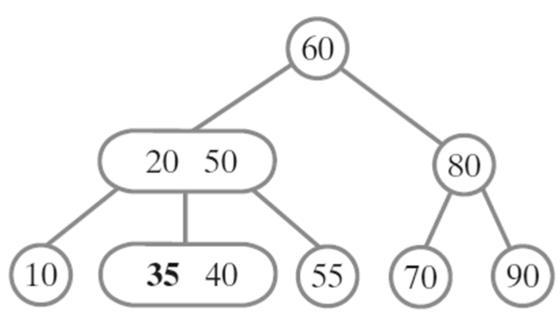
- Splitting an internal node to accommodate a new entry:



- Splitting the root to accommodate a new entry:



Searching a 2-3 Tree:



2-3 tree: performance:

2-3 tree is a perfect balanced tree: Every path from **root** to a **leaf** has same length.

Tree height:

- Worst case: **log N**. [all 2-nodes]
- Best case: **$\log_3 N \approx .631 \log N$**. [all 3-nodes]
  - Between 12 and 20 for a million nodes.
  - Between 18 and 30 for a billion nodes.

2-3 tree: implementation?

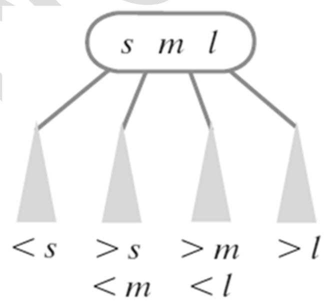
Direct implementation is complicated, because:

- Maintaining multiple node types is cumbersome.
- Need multiple compares to move down tree.
- Need to move back up the tree to split 4-nodes.
- Large number of cases for splitting.

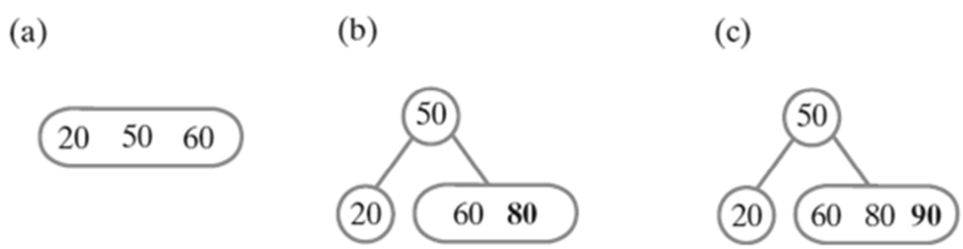
exercise: 50 60 70 40 30 20 10 80 90 100

2-4 Trees

- Sometimes called a 2-3-4 tree.
  - General search tree
  - Interior nodes must have either two, three, or four children
  - Leaves occur on the same level
  - A 4-node contains three data items ***s, m***, and ***l*** and has four children.

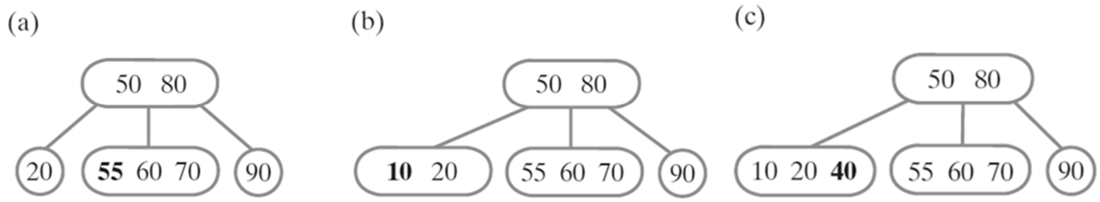


Adding Entries to a 2-4 Tree



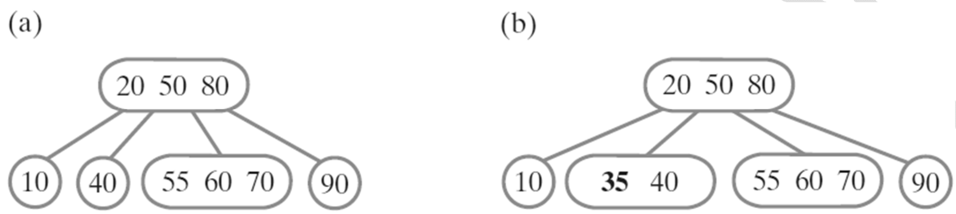
The 2-4 tree, after (a) adding **20**, **50**, and **60** (b) adding **80** and splitting the root; (c) adding **90**

Adding 70



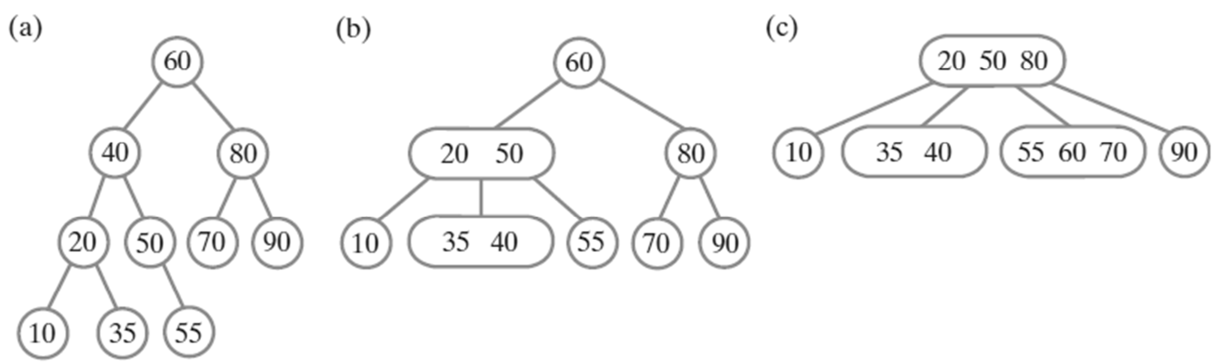
The 2-4 tree after adding (a) **55**; (b) **10**; (c) **40**

Adding 5



The 2-4 tree after (a) splitting the leftmost 4-node; (b) adding **35**

Comparing AVL, 2-3, and 2-4 Trees:



Three balanced search trees obtained by adding 60, 50, 20, 80, 90, 70, 55, 10, 40, and 35:
(a) AVL tree; (b) 2-3 tree; (c) 2-4 tree

B-Trees

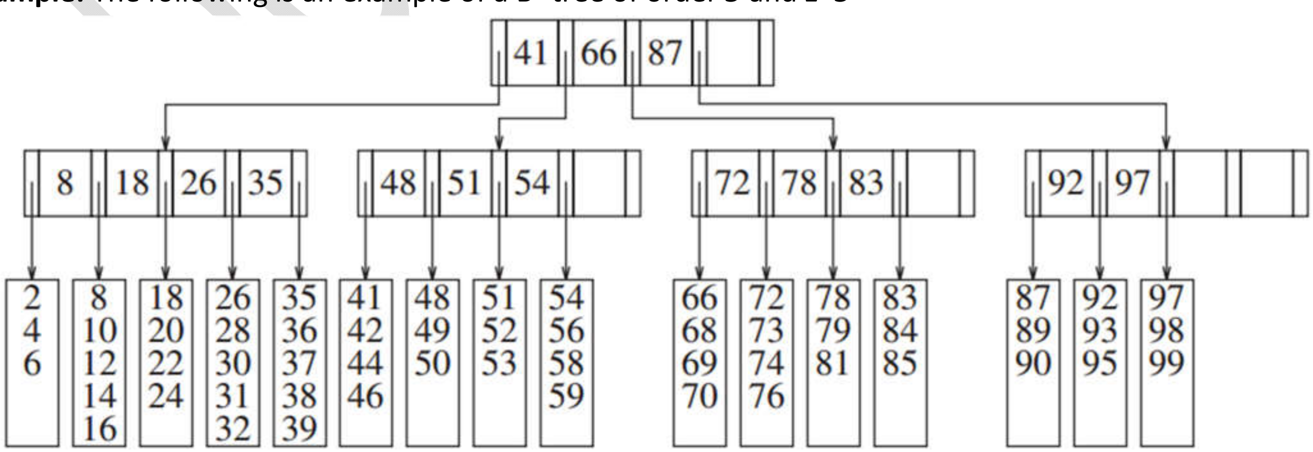
B-trees (Bayer-McCreight, 1972)

- **Definition**: multiway search tree of order ***m***
  - A general tree whose nodes have up to ***m*** children each
- A binary search tree is a multiway search tree of order 2. In a binary search tree, we need one key to decide which of two branches to take. In an M-ary search tree, we need M 1 keys to decide which branch to take.
- 2-3 trees and 2-4 trees are balanced multiway search trees of orders 3 and 4, respectively.
- As branching increases, the depth decreases. Whereas a complete binary tree has height that is roughly $\log_2 N$, a complete M-ary tree has height that is roughly $\log_M N$.
- The B-tree is the most popular data structure for disk bound searching.
- To make this scheme efficient in the worst case, we need to ensure that the M-ary search tree is balanced in some way.
- Additional properties to maintain balance:
  - The **root** has either no children or between **2** and ***m*** children.
  - Other interior nodes (non-leaves) have between $\lceil m/2 \rceil$ and ***m*** children each.
  - All leaves are on the same level.

A B-tree of order ***M*** is an ***M-ary*** tree with the following properties: (**B^+ tree**)

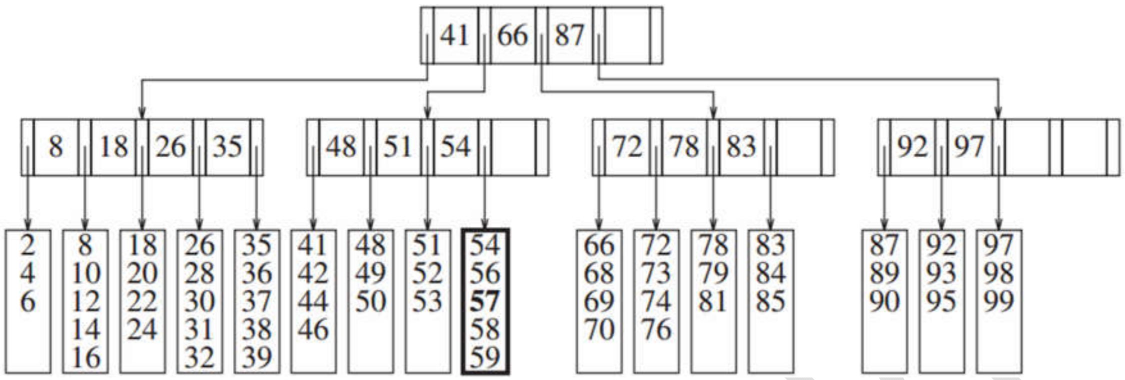
1. The data items are stored at leaves.
2. The non-leaf nodes store up to ***M 1*** keys to guide the searching; key ***i*** represents the smallest key in subtree ***i+1***.
3. The **root** is either a leaf or has between two and ***M*** children.
4. All non-leaf nodes (except the **root**) have between ***M/2*** and ***M*** children.
5. All leaves are at the same depth and have between ***L/2*** and ***L*** data items, for some ***L*** (the determination of L is described shortly).

Example: The following is an example of a B^+ tree of order ***5*** and ***L=5***

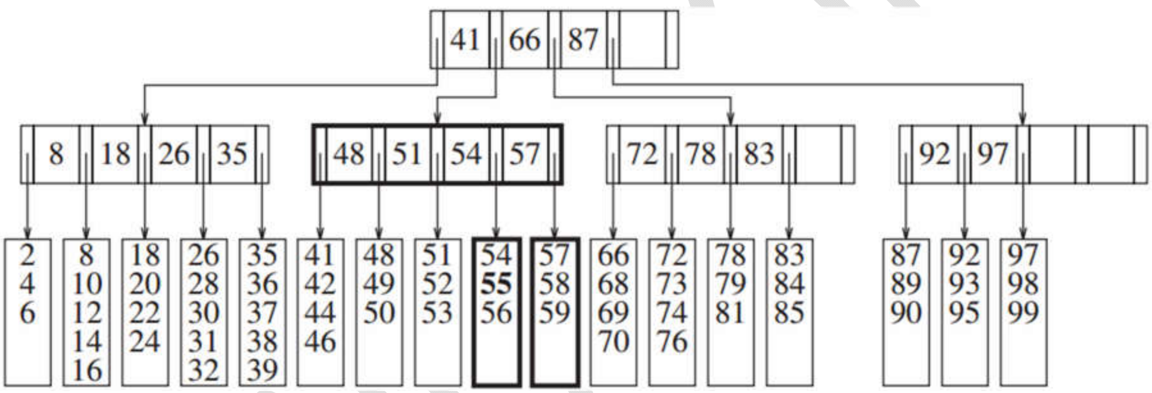


Add items from the B+ tree:

- **Insert 57**: A search down the tree reveals that it is not already in the tree. We can then add it to the leaf as a fifth item:

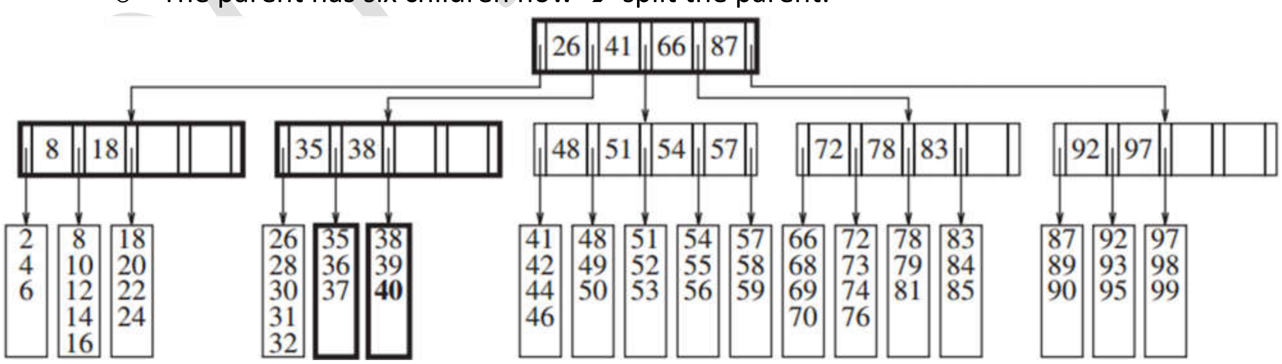


- **Insert 55**: The leaf where 55 wants to go is already full. Solution: split them into two leaves:



Note: The node splitting in the previous example worked because the parent did not have its full complement of children.

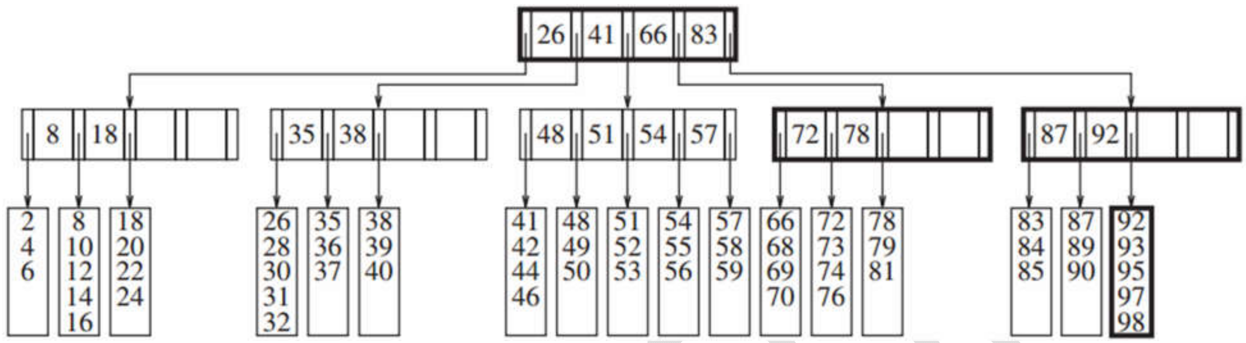
- **Insert 40**: We have to split the leaf containing the keys 35 through 39, and now 40, into two leaves.
  - The parent has six children now → split the parent.



Note:
- When the parent is split, we must update the values of the keys and also the parent's parent.
- if the parent already has reached its limit of children? In that case, we continue splitting nodes up the tree until either we find a parent that does not need to be split or we reach the root. Then we split the root and this will generate a new level.

Remove items from the B+ tree:

- We can perform deletion by finding the item that needs to be removed and then removing it.
    - The problem is that if the leaf it was in had the minimum number of data items, then it is now below the minimum.
- **Remove 99**: Since the leaf has only two items, and its neighbor is already at its minimum of three, we combine the items into a new leaf of five items.



Splay Trees

Recall: **Asymptotic analysis** examines how an algorithm will perform in worst case.

Amortized analysis examines how an algorithm will perform in practice or on average.

The **90–10 rule** states that **90%** of the accesses are to **10%** of the data items.
However, balanced search trees do not take advantage of this rule.

- The **90–10** rule has been used for many years in **disk I/O systems**.
- A **cache** stores in main memory the contents of some of the disk blocks. The hope is that when a disk access is requested, the block can be found in the main memory cache and thus save the cost of an expensive disk access.
- **Browsers** make use of the same idea: A cache stores locally the previously visited Web pages.

Splay Trees:

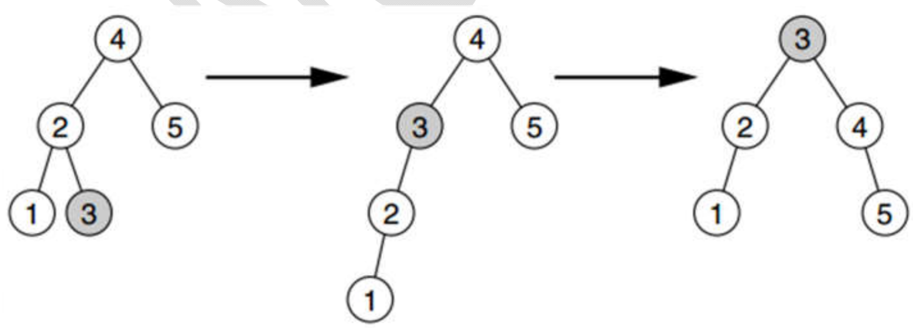
- Like **AVL** trees, use the standard binary search tree property.
- After any operation on a node, make that node the new root of the tree.

A simple self-adjusting strategy (that does not work)

The easiest way to move a frequently accessed item toward the root is to rotate it continually with its parent. Moving the item closer to the root, a process called the **rotate-to-root strategy**.

- If the item is accessed a second time, the second access is cheap.

Example: Rotate-to-root strategy applied when node **3** is accessed



- As a result of the rotation:
  - future accesses of node **3** are cheap
  - Unfortunately, in the process of moving node **3** up two levels, nodes **4** and **5** each move down a level.
- Thus, if access patterns do not follow the **90–10 rule**, a long sequence of bad accesses can occur.

The basic bottom-up splay tree

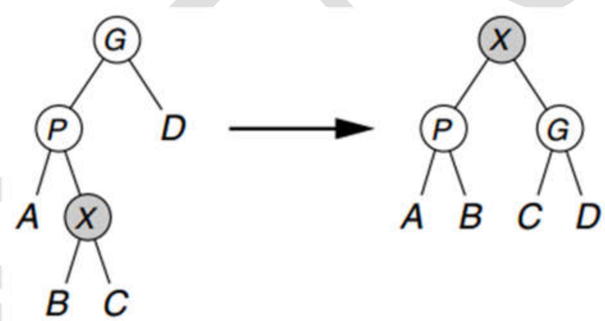
Splaying cases:

- **The zig case** (normal single rotation)

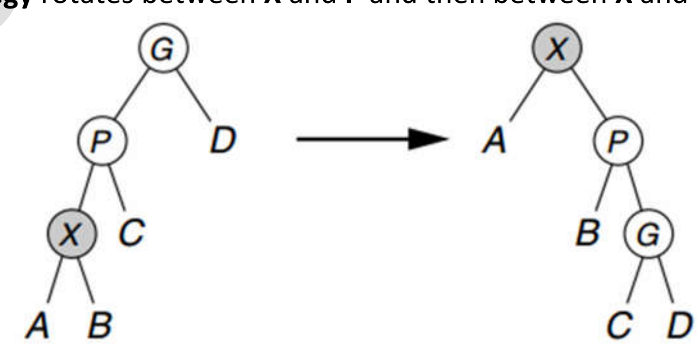
  If **X** is a non-root node on the access path on which we are rotating and the parent of **X** is the root of the tree, we merely rotate **X** and the root, as shown:

Otherwise, **X** has both a parent **P** and a grandparent **G**, and we must consider two cases and symmetries.

- **zig-zag case**:
  - This corresponds to the inside case for **AVL** trees.
  - Here **X** is a right child and **P** is a left child (or vice versa: **X** is a left child and **P** is a right child).
  - We perform a **double rotation** exactly like an **AVL** double rotation, as shown:

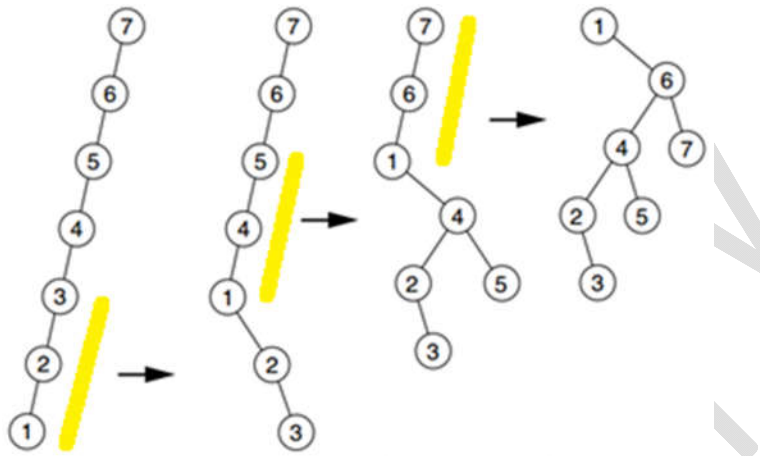


- **zig-zig case:**
  - The outside case for **AVL** trees.
  - Here, **X** and **P** are either both left children or both right children.
  - In this case, we transform the left-hand tree to the right-hand tree (or vice versa).
  - Note that this method differs from the **rotate-to-root strategy**.
    - The **zig-zig** splay rotates between **P** and **G** and then **X** and **P**, whereas the **rotate-to-root strategy** rotates between **X** and **P** and then between **X** and **G**.



Splaying has the effect of roughly **halving** the depth of most nodes on the access path and increasing by at most **two levels** the depth of a few other nodes.

Example: Result of splaying at node **1** (three zig-zigs)



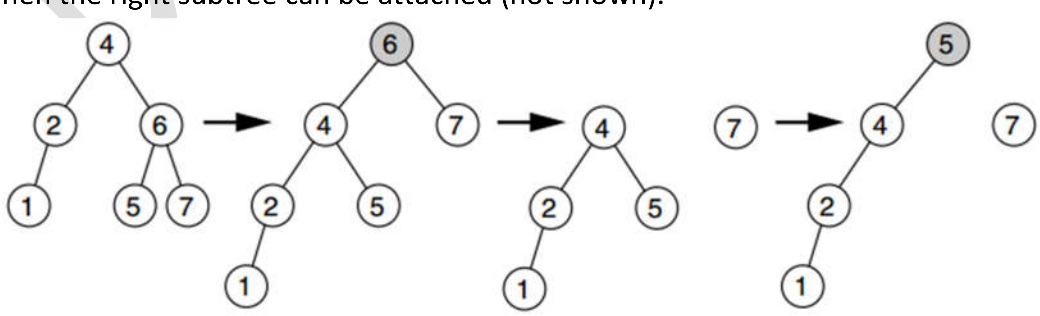
Exercise: perform rotate-to-root strategy

Basic splay tree operations

A splay operation is performed after each access:

- After an item has been inserted as a leaf, it is **splayed** to the root.
- All searching operations incorporate a **splay**. (**find, findMin** and **findMax**)
- To perform deletion, we access the node to be deleted, which puts the node at the root. If it is deleted, we get two subtrees, **L** and **R** (left and right). If we find the largest element in **L**, using a **findMax** operation, its largest element is rotated to **L**'s root and **L**'s root has no right child. We finish the remove operation by making **R** the right child of **L**'s root. An example of the remove operation is shown below:

  Example: The remove operation applied to node **6**:
  - First, **6** is splayed to the root, leaving two subtrees;
  - A **findMax** is performed on the left subtree, raising **5** to the root of the left subtree;
  - Then the right subtree can be attached (not shown).



- The cost of the remove operation is **two splays**.

Recursion (Time Analysis Revision)

Example 1: Write a recursive method to calculate the sum of squares of the first ***n*** natural numbers. ***n*** is to be given as an input.

```
public int sumOfSquares(int n) {
   if (n==1)
      return 1;
   return   (n*n) + sumOfSquares(n-1);
}
```

Recursion may sometimes be very intuitive and simple, but it may not be the best thing to do.

Example 2: **Fibonacci sequence**:

F(n) = n if n=0, 1 ; F(n) = F(n-1) + F(n-2) if n > 1

0	1	1	2	3	5	8	13	..
F(0)	F(1)	F(2)	F(3)	F(4)	F(5)	F(6)	F(7)	..

Solution 1: **Iterative**

```
public static int fib1(int n){
    if(n<=1) return n;
    int f1 = 0,   f2 = 1,   res=0;
    for(int i=2; i<=n; i++){
       res =f1+f2;
       f1=f2;
       f2=res;
    }
    return res;
}
```

Solution 2: **Recursion**

```
public static int fib2(int n){
    if(n<=1) return n;
    return (fib2(n-1)+fib2(n-2));
 }
```

Test for n=6 and n=40

Why recursive solution is taking much time?

Do analyze the 2 algorithms in term of calculating F(n)

In **Solution 1**:

We have **F(0)** and **F(1)** given

Then we calculate F(2) using F(1) and F(0)

F(3) using F(2) and F(1)

F(4) using F(3) and F(2)

:

F(n) using F(n-1) and F(n-2)

In **Solution 2**:

```
                              F(5)
            F(4)                                  F(3)
      F(3)          F(2)                  F(2)            F(1)
  F(2)    F(1)   F(1)   F(0)           F(1)   F(0)
F(1)  F(0)
```

Note: we are calculating the same value multiple times!!

n	F(2)	F(3)	..
5	3	2	
6	5		
8	13		
:			
40	**63245986**		

Exponential growth

Time and Space complexity Analysis of recursion

Example: recursive factorial

```
fact(n){
        If (n==0) return 1;
        Return n *  fact(n-1);
}
```

- Calculate operation costs:
  - If statement takes 1 unit of time
  - Multiplication (*) takes 1 unit of time
  - Subtraction (-) takes 1 unit of time
  - Function call
- So $T(0) = 1$

  $T(n) = 3 + T(n-1)$ for $n > 0$

To solve this equation, reduce T(n) in term of its base conditions.

$$
\begin{aligned}
T(n) &= T(n-1) + 3 \\
&= T(n-2) + 6 \\
&= T(n-3) + 9 \\
&\vdots \\
&= T(n-k) + 3k
\end{aligned}
$$

For $T(0) \rightarrow n-k = 0 \rightarrow n = k$

Therefore $T(n) = T(0) + 3n$

$= 1 + 3n \rightarrow$ **O(n)**

Space analysis:

Recursive Tree

Fact(5) → Fact(4) → Fact(3) → Fact(2) → Fact(1) → Fact(0)

Each function call will cause to save current function state into memory (call stack, push):

Fact(1)
Fact(2)
Fact(3)
Fact(4)
Fact(5)

Each return statement will retrieve previous saved function state from memory (pop):

So needed space is proportional to n ➔ **O(n)**

Fibonacci sequence time complexity analysis

```
public static int fib2(int n){
    if(n<=1) return n;
    return (fib2(n-1)+fib2(n-2));
}
```

- Calculate operation costs:
  - If statement takes 1 unit of time
  - 2 subtractions (-) takes 2 unit of time
  - 1 addition (+) takes 1 unit of time
  - 2 function calls
- So $T(0) = T(1) = 1$

  $T(n) = T(n-1) + T(n-2) + 4$ for $n > 1$

To solve this equation, reduce T(n) in term of its base conditions.

For approximation assume $T(n-1) \approx T(n-2)$ ➔ in reality $T(n-1) > T(n-2)$

$T(n) = 2\,T(n-2) + 4$ → $c = 4$

$= 2\,T(n-2) + c$ → $T(n-2) = 2\,T(n-4) + c$

$= 2\,\{ 2\,T(n-4) + c \} + c$

$= 4\,T(n-4) + 3c$

$= 8\,T(n-6) + 7c$

$= 16\,T(n-8) + 15c$

:

$= 2^k\,T(n-2k) + (2^k-1)c$

For T(0) ➔ $n-2k = 0$ ➔ $k = n/2$

Therefore $T(n) = 2^{n/2}\,T(0) + (2^{n/2} - 1)\,c$ ➔ $2^{n/2}\,(1+c) - c$

T(n) is proportional to $2^{n/2}$ ➔ **$O(2^{n/2})$** 🡐 lower bound analysis

Similarly, for approximation assume $T(n-2) \approx T(n-1)$ ➔ in reality $T(n-2) < T(n-1)$

$T(n) = 2\,T(n-1) + c \rightarrow T(n-1) = 2\,T(n-2) + c$

$= 2\{ 2\,T(n-2) + c \} + c$

$= 4\,T(n-2) + 3c$

$= 8\,T(n-3) + 7c$

$= 16\,T(n-4) + 15c$

:

$= 2^k\,T(n-k) + (2^k-1)c$

For $T(0) \rightarrow n-k = 0 \rightarrow k = n$

Therefore $T(n) = 2^n\,T(0) + (2^n - 1)\,c \rightarrow 2^n\,(1+c) - c$

$T(n)$ is proportional to 2^n → **$O(2^n)$** ← upper bound analysis → worst case analysis

While for iterative solution → **O(n)**

Recursion with memorization

Solution: don't calculate something already has been calculated.

Algorithm:

```
fib(n){
        If (n<=1)   return n
        If(F[n] is in memory)   return F[n]
        F[n] =  fib(n-1) + fib(n-2)
        Return F[n]
}
```

Time complexity → **O(n)**

Calculate **X^n** using recursion

Iterative solution: **O(n)** $X^n = X * X * X * X * \dots * X$ n-1 multiplication	Recursive solution 1: **O(n)** $X^n = X * X^{n-1}$ if $n > 0$ $X^0 = 1$ if $n > 0$	Recursive solution 2: **O(log n)** $X^n = X^{n/2} * X^{n/2}$ if n is even $X^n = X * X^{n-1}$ if n is odd $X^0 = 1$ if $n > 0$

res = 1
for i←1 to n
  res ← res * x
	pow(x, n){
	  if n==0 return 1
	  return x * pow(x, n-1)
	}
		pow(x, n){
		  if n==0 return 1
		  if n%2 == 0 {
		    y ← pow(x, n/2)
		    return y * y
		  }
		  return x * pow(x, n-1)
		}

Recursive solution 1: Time analysis

T(1) = 1
T(n) = T(n-1) + c
= (T(n-2) + c) + c ➔ T(n-2) + 2c
= T(n-3) + 3c
:
= T(n-k) + kc
For T(0) ➔ n-k = 0 ➔ n = k
T(n) = T(0) + nc ➔ 1 + nc ➔ **O(n)**

Recursive solution 2: Time analysis

- $X^n = X^{n/2} * X^{n/2}$ if n is even
- $X^n = X * X^{n-1}$ if n is odd
- $X^n = 1$ if n == 0
- $X^n = X * 1$ if n == 1

If even ➔ T(n) = T(n/2) + c1
If odd ➔ T(n) = T(n-1) + c2
If 0 ➔ T(0) = 1
If 1 ➔ T(1) = c3

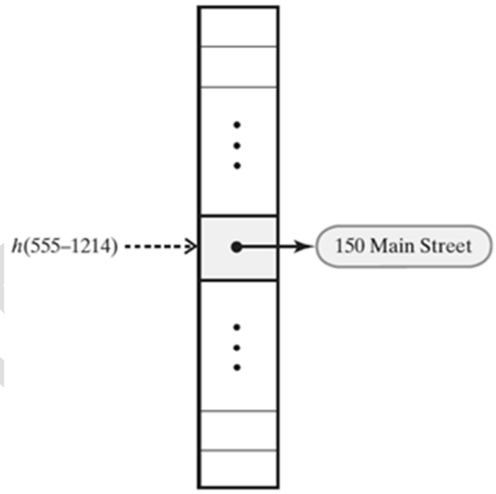
If odd, next call will become even:
T(n) = T((n-1)/2) + c1 + c2
If even
T(n) = T(n/2) + c
= T(n/4) + 2c
= T(n/8) + 3c
:
= $T(n/2^k)$ + k c
For T(1) ➔ T(0) + c ➔ 1
$n/2^k = 1$ ➔ $n = 2^k$ ➔ k = log n
= c3 + c log n ➔ **O(log n)**

Hash Tables

- **Hashing**: is a technique that determines element **index** using only element's distinct **search key**.
- **Hash function**:
  - Takes a **search key** and produces the integer **index** of an element in the **hash table.**
  - Search key-maps, or hashes, to the index.

Example 1: Phone numbers (xxx-xxxx).
- **Bad**: first three digits. // identical for same area
- **Better**: last four digits. // distinct

Example 2: Social Security numbers (ID number).
- **Bad**: first three digits. // identical for same period
- **Better**: last three digits. // distinct



Hash table

Practical challenge: Need different approaches for each key type.

Simple algorithms for the hash operations that **add** and **retrieve**:

```
Algorithm add(key, value)
index = h(key)
hashTable[index] = value

Algorithm getValue(key)
index = h(key)
return hashTable[index]
```

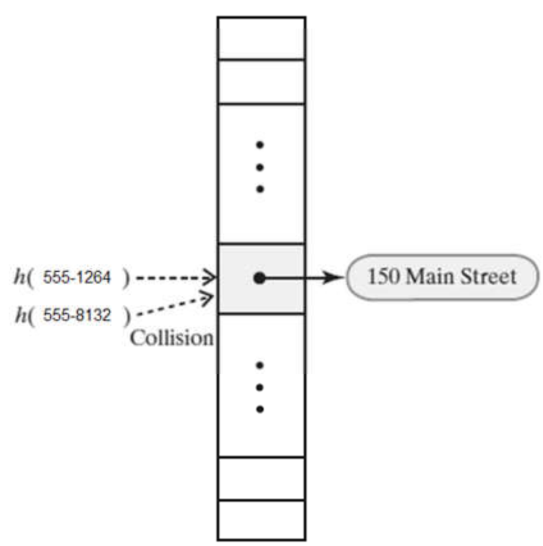
Typical Hashing

Typical hash functions perform two steps:
1. **Convert search key** to an integer called the **hash code**.
2. **Compress hash code** into the range of indices for hash table.

```
Algorithm getHashIndex(phoneNumber)
// Returns an index to an array of tableSize locations.

i = last four digits of phoneNumber
return i % tableSize
```

- Typical hash functions are **not perfect**:
  - Can allow more than one **search key** to map into a **single index.**
  - Causes a **collision** in the hash table.

Example: Consider table (array) size = **101**
- **getHashIndex(555-1264)** = 52
- **getHashIndex(555-8132**) = 52 also!!!



Hash table

Hash Functions

- A good hash function should:
  - **Minimize collisions**
  - **Be fast to compute**
- To reduce the chance of a collision
  - Choose a hash function that distributes entries **uniformly** throughout hash table.

Java's hash code conventions

All Java classes inherit a method ***hashCode()***, which returns a **32-bit** int.

Default implementation: **Memory address**.

Customized implementations: Integer, Double, String, File, URL, Date, …

User-defined types: Users are on their own.

Java library implementations:

Integer

```
public final class Integer
{
   private final int value;
   ...

   public int hashCode()
   {  return value;  }
}
```

Boolean

```
public final class Boolean
{
   private final boolean value;
   ...

   public int hashCode()
   {
      if (value) return 1231;
      else       return 1237;
   }
}
```

Double

```
public final class Double
{
   private final double value;
   ...

   public int hashCode()
   {
      long bits = doubleToLongBits(value);
      return (int) (bits ^ (bits >>> 32));
   }
}
```

convert to IEEE 64-bit representation;
xor most significant 32-bits
with least significant 32-bits

String

```
public final class String
{
   private final char[] s;
   ...

   public int hashCode()
   {
      int hash = 0;
      for (int i = 0; i < length(); i++)
         hash = s[i] + (31 * hash);
      return hash;
   }
}
```

i^{th} character of s

char	Unicode
…	…
'a'	97
'b'	98
'c'	99
…	…

Horner's method to hash a String of length ***L***:

$$h = s[0] \cdot 31^{L-1} + \ldots + s[L-3] \cdot 31^2 + s[L-2] \cdot 31^1 + s[L-1] \cdot 31^0.$$

Example:

```
String s = "call";
int code = s.hashCode();
```

$$\begin{array}{l} \quad\quad\quad\quad\quad\quad\ \ \text{c} \quad\quad\quad \text{a} \quad\quad\quad\ \ \text{l} \quad\quad\quad\quad \text{l} \\ 3045982 = 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0 \\ \quad\quad\quad\ \ = 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99))) \end{array}$$

Implementing hash code: user-defined types

Hash code design

"**Standard**" recipe for user-defined types:

- Combine each significant field using the ***31x + y*** rule.
- If field is a primitive type, use **wrapper** type **hashCode()**.
- If field is **null**, return **0**.
- If field is a reference type, use **hashCode()**.
- If field is an array, apply to each entry. ← or use **Arrays.deepHashCode()**

Example:

```
public final class Transaction {
   private final String  who;
   private final Date    when;
   private final double  amount;

   public int hashCode()
   {
      int hash = 17;
      hash = 31*hash + who.hashCode();
      hash = 31*hash + when.hashCode();
      hash = 31*hash + ((Double) amount).hashCode();
      return hash;
   }
}
```

nonzero constant

for reference types, use hashCode()

for primitive types, use hashCode() of wrapper type

typically a small prime

Compressing a Hash Code

Hash code: An **int** between -2^{31} and $2^{31} - 1$.

Hash function: returns an **int** between **0** and **M-1** (for use as array index).

- Common way to scale an integer
  - Use Java **%** operator ➔ **hash code % m**
- Avoid **m** as power of **2** or **10**
- Best to use an **odd** number for **m**
- **Prime numbers** often give good distribution of hash values

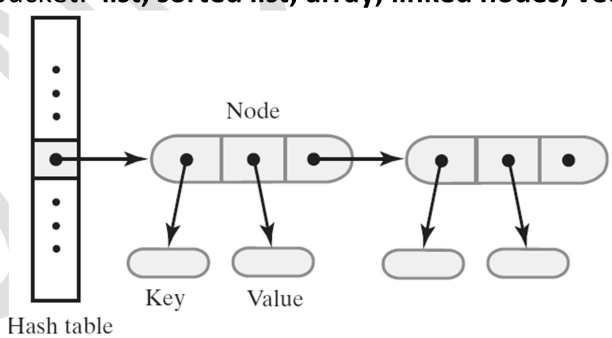
```
private int hash(Key key)
{  return (key.hashCode() & 0x7fffffff) % M;  }
```

Resolving Collisions

- **Collisions**: Two distinct **keys** hashing to same **index**.
- Two choices:
  - Change the structure of the hash table so that each array location can represent more than one value. (**Separate Chaining**)
  - Use another empty location in the hash table. (**Open Addressing**)

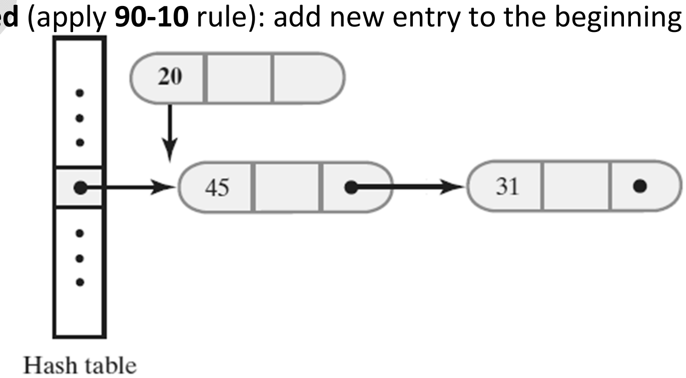
Separate Chaining

- Alter the structure of the hash table:
  - Each location can represent more than one value.
  - Such a location is called a ***bucket***
- Decide how to represent a bucket: **list, sorted list; array; linked nodes; vector; etc.**

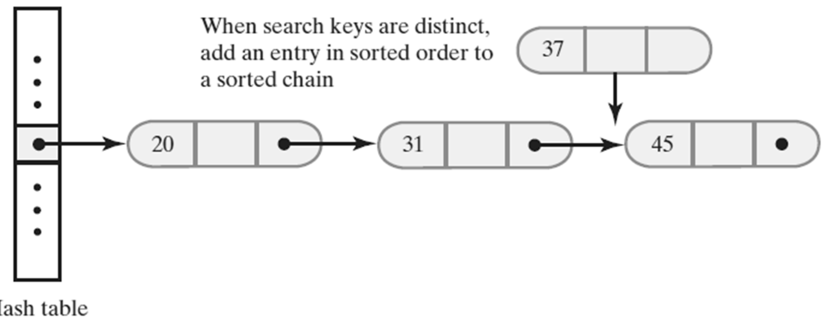


Where to insert a new entry into a linked bucket?

(a) If **unsorted** (apply **90-10** rule): add new entry to the beginning of chain



(b) If **sorted**:



Time Complexity

Worst case: all keys mapped to the same location ➔ one long list of size **N**

Find(key) ➔ **O(n)** ☹

Best case: hashing uniformly distribute records over the hash table ➔ each list long = **N/M** = **α** **(α** is load factor**)**

Find(key) ➔ **O(1 + α)** ☺

Design Consequences

- **M** too large ➔ too many empty chains.
- **M** too small ➔ chains too long.
- Typical choice: **M ≈ N / 5** ➔ constant-time ops.

Open Addressing

➢ Linear Probing

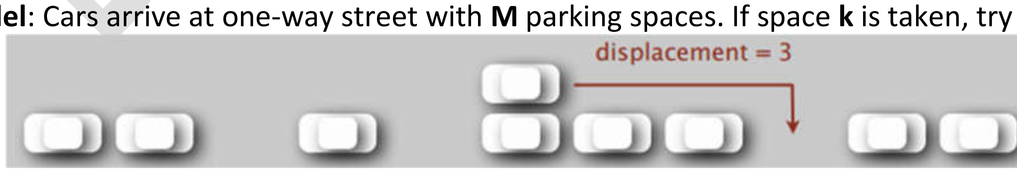
- When a new key collides, find **next** empty slot, and put it there.
- **Hash**: Map key to integer **k** between **0** and **M-1**.
- **Insert**: Put at table index **k** if free; **if not** try **k+1**, **k+2**, etc.
  - If reaches end of table, go to beginning of table (**Circular hash table**)
- Hash function: $h(k, i) = (h(k, 0)+i) \% m$
- Array size **M** must be greater than number of key-value pairs **N**.

Example: Linear hash table demo: **take last 2 digits of student's ID and run a demo**

Clustering problem: A contiguous block of items will be easily formed which in turn will affect performance.

Knuth's Parking Problem

- **Model**: Cars arrive at one-way street with **M** parking spaces. If space **k** is taken, try **k+1**, **k+2**, etc.

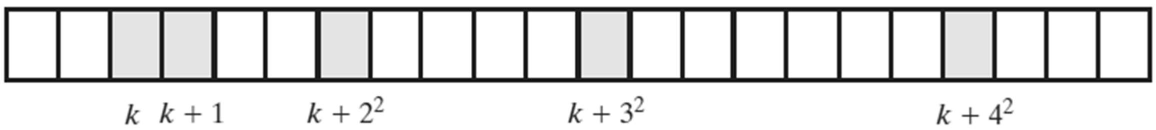


Parameters.

- M too large ⇒ too many empty array entries.
- M too small ⇒ search time blows up.
- Typical choice: $\alpha = N / M \sim ½$. ⟵ # probes for search hit is about 3/2; # probes for search miss is about 5/2

➢ Quadratic Probing

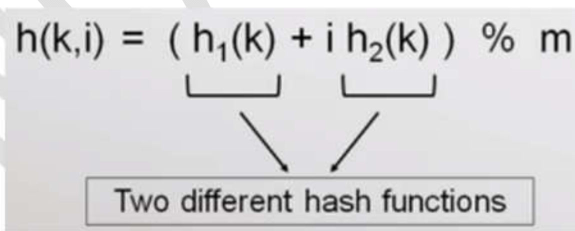
- Linear probing looks at **consecutive** locations beginning at index ***k***
- Quadratic probing, considers the locations at indices $\mathbf{k + j^2}$
    - Uses the indices ***k, k*+1, *k* + 4, *k* + 9**, …

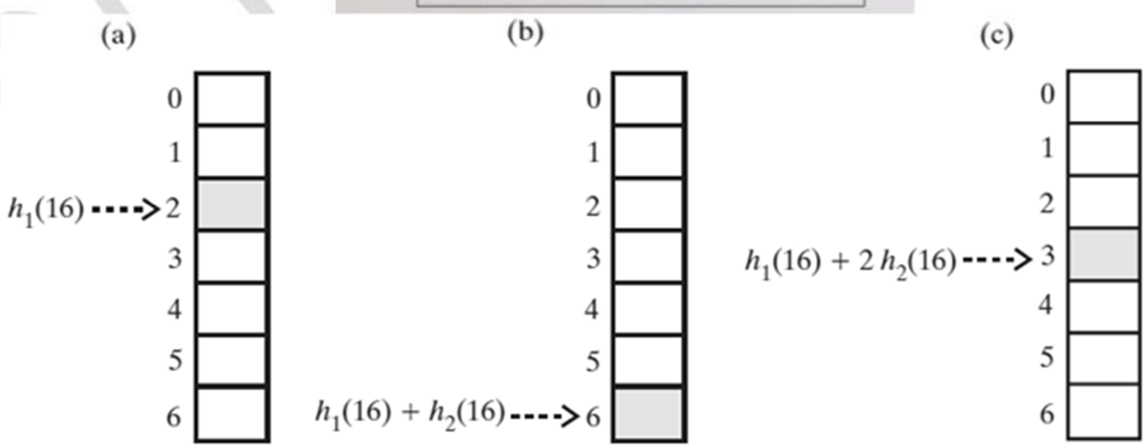


- Hash function: $\mathbf{h(k\,,i) = (h(k\,,0)+i^2)\ \%\ m}$
- For linear probing it is a bad idea to let the hash table get nearly full, because performance degrades.
- For quadratic probing, the situation is even worse: There is no guarantee of finding an empty cell once the table gets more than half full, or even before the table gets half full if the table size is not **prime**.
- Standard **deletion** cannot be performed in a probing hash table, because the cell might have caused a collision to go past it. (instead **soft deletion** is used)

Double Hashing

- **Linear probing** and **quadratic probing** add increments to **k** to define a probe sequence
    - Both are ***independent*** of the search key
- **Double hashing** uses a **second hash function** to compute these increments
    - This is a key-***dependent*** method.
    - The 2nd hash function must never evaluate to **zero**.





The 1st three locations in a probe sequence generated by double hashing for the search key 16

Potential Problem with Open Addressing

- Note that each location is either **occupied**, **empty (null)**, or **available (removed)**
  - Frequent additions and removals can result in *no* locations that are **null**
- Thus searching a probe sequence will not work
- Consider separate chaining as a solution

Time Complexity

Worst case: O(n)

Average case:

$$\text{Number of probes} \leq \frac{1}{1-\alpha} \qquad \alpha = n/m$$

if, $\alpha < 1$ (i.e. $n < m$)

If the table is 50% full, $\alpha = 0.5$

Number of probes ≤ 2

If the table is 80% full, $\alpha = 0.8$

Number of probes ≤ 5

$\alpha \rightarrow 1$ (near full space utilization), Performance $\downarrow$

Rehashing

- If the table gets **too full**, the running time for the operations will start taking too long and insertions might fail for open addressing hashing with quadratic resolution.
- A solution, then, is to build another table that is about **twice as big** (with an associated new hash function) and scan down the entire original hash table, computing the new hash value for each (non-deleted) element and inserting it in the new table.
- This entire operation is called **rehashing**.
  - This is obviously a very expensive operation; the running time is **O(N)**, since there are **N** elements to rehash and the table size is roughly **2N**, but it is actually not all that bad, because it happens very infrequently.

Priority Queues (Heaps)

A ***priority queue*** is a data structure that allows **at least** the following two operations:

- **Insert**: which does the obvious thing;
- **deleteMin (or deleteMax)**: which finds, returns, and removes the minimum (or maximum) element in the priority queue.

Simple Implementations:

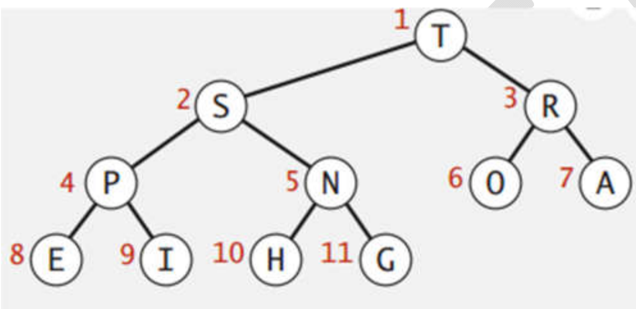
- **Unsorted Linked list**, performing insertions at the front in **O(1)** and traversing the list, which requires **O(N)** time, to delete the minimum/maximum.
- **Sorted Linked list**, performing insertions in **O(N)** and **O(1)** to delete the minimum/maximum.
- **Binary search tree**: this gives an **O(log N)** average running time for both operations.

Binary Heap

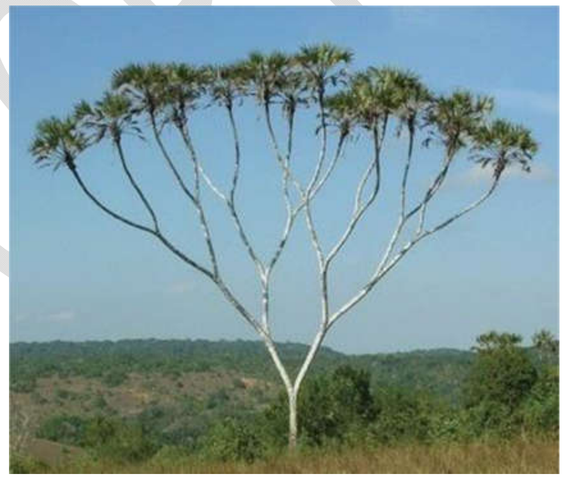
A heap is a binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right.

Such a tree is known as a **complete binary tree**.

A complete binary tree of height h has between 2^h and $2^{h+1}-1$ nodes.



Heap representations



As complete binary tree is so **regular**, therefore, it can be represented as an array:

i	0	1	2	3	4	5	6	7	8	9	10	11
a[i]	-	T	S	R	P	N	O	A	E	I	H	G

- Parent of node at i is at $i/2$.
- Children of node at i are at $2i$ (left child) and $2i+1$ (right child).

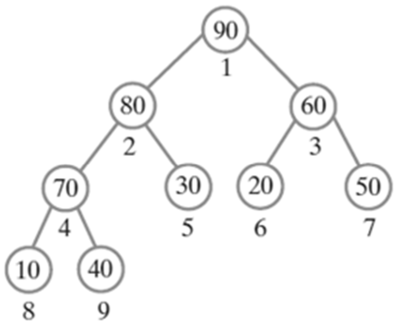
Heap-order property:

- In a **min heap**, for every node **X**, the key in the parent of **X** is smaller than (*or equal to*) the key in **X**, with the exception of the root (which has no parent). Therefore, the minimum element can always be found at the root.
- In a **max heap**, for every node **X**, the key in the parent of **X** is larger than (*or equal to*) the key in **X**, with the exception of the root (which has no parent). Therefore, the maximum element can always be found at the root.

Interface for the max-heap

```java
public interface MaxHeapInterface<T extends Comparable<? super T>>
{
    public void add(T newEntry);
    public T removeMax();
    public T getMax();
    public boolean isEmpty();
    public int getSize();
    public void clear();
} // end MaxHeapInterface
```

An Array to Represent a Heap



	90	80	60	70	30	20	50	10	40			
0	1	2	3	4	5	6	7	8	9	10	11	12

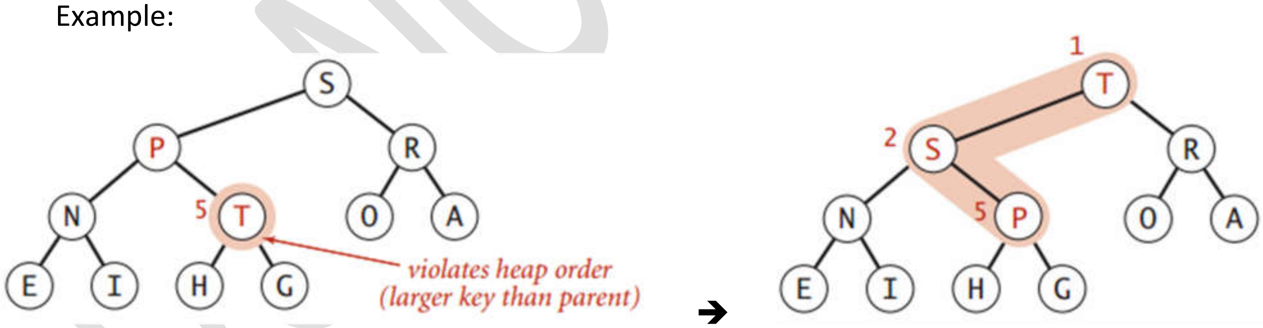
Promotion (ترفيع) in a max heap

Scenario: Child's key becomes larger than its parent's key.

To eliminate the violation:

- Exchange key in **child** with key in **parent**.
- Repeat until heap order restored.

Example:




```java
private void swim(int k)
{
    while (k > 1 && less(k/2, k))
    {
        exch(k, k/2);
        k = k/2;
    }
}
```

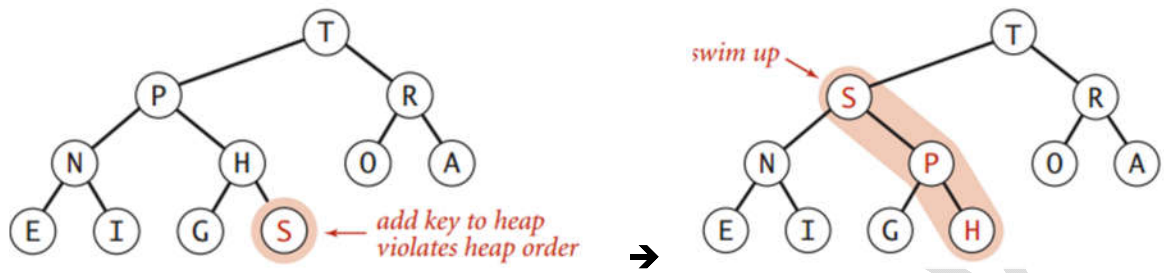
parent of node at k is at k/2

Insertion in a max heap

Insert: Add node at end, then swim it up.

Cost: At most ***1 + log N*** compares.

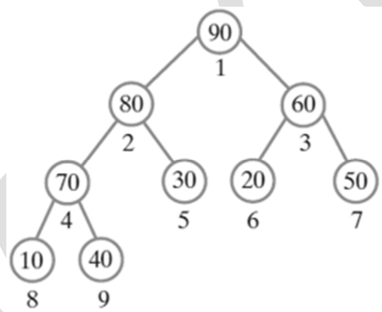
Example 1: insert **S**



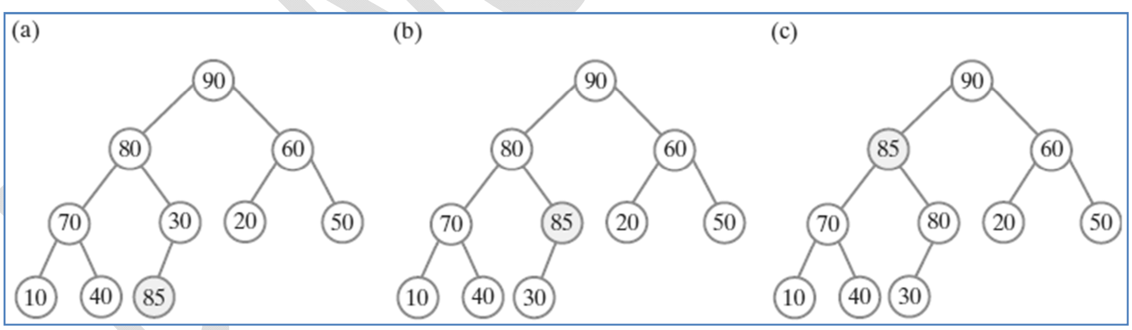
➔

```
public void insert(Key x)
{
   pq[++N] = x;
   swim(N);
}
```

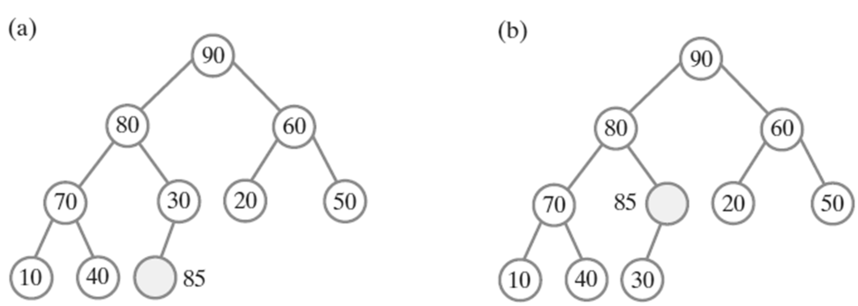
Example 2: insert **85**

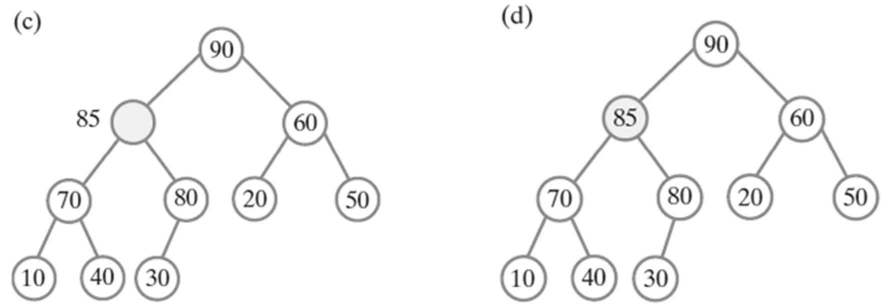


Method 1: The steps in adding 85 to the previous max-heap



Method 2: A revision of the steps shown in the previous figure, **to avoid swaps**:





The following figures shows array representation of the steps in the previous figures:

(a)

0	1	2	3	4	5 (10/2)	6	7	8	9	10	11	12	
	90	80	60	70	**30**	20	50	10	40				85 > 30

(b)

0	1	2	3	4	5	6	7	8	9	10	11	12	
	90	80	60	70		20	50	10	40	**30**			Move 30

(c)

0	1	2 (5/2)	3	4	5	6	7	8	9	10	11	12	
	90	**80**	60	70		20	50	10	40	30			85 > 80

(d)

0	1	2	3	4	5	6	7	8	9	10	11	12	
	90		60	70	**80**	20	50	10	40	30			Move 80

(e)

0	1 (2/2)	2	3	4	5	6	7	8	9	10	11	12	
	90		60	70	80	20	50	10	40	30			85 < 90

(f)

0	1	2	3	4	5	6	7	8	9	10	11	12	
	90	**85**	60	70	80	20	50	10	40	30			Insert 85

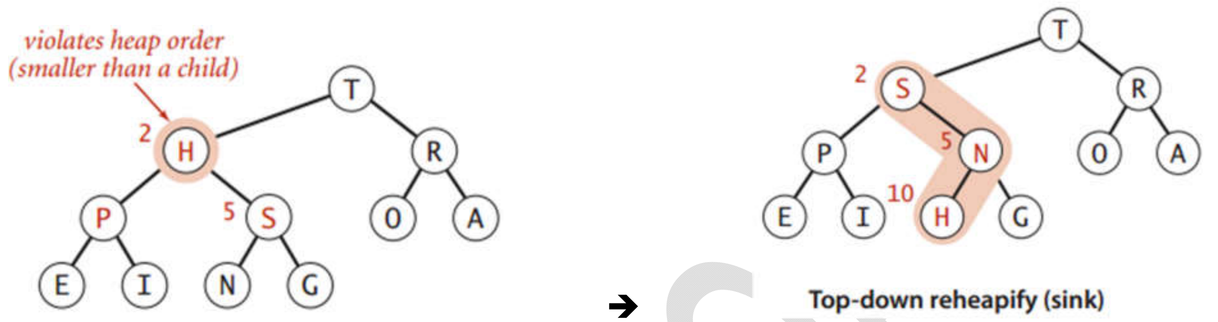
Demotion (إنزال رتبة) in a max heap

Scenario: Parent's key becomes smaller than one (or both) of its children's.

To eliminate the violation:

- Exchange key in parent with key in larger child.
- Repeat until heap order restored.

Example 1:



Top-down reheapify (sink)

```
private void sink(int k)
{
   while (2*k <= N)
   {
      int j = 2*k;
      if (j < N && less(j, j+1)) j++;
      if (!less(k, j)) break;
      exch(k, j);
      k = j;
   }
}
```

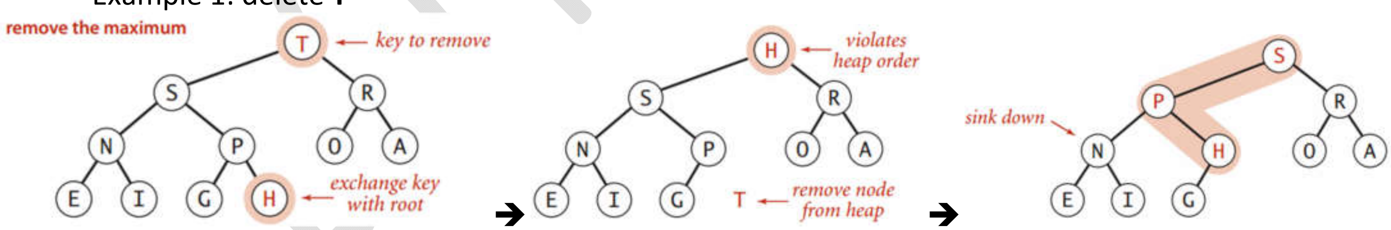
children of node at k are 2k and 2k+1

Delete the maximum in a max heap (Removing the root)

Delete max: Exchange root with node at end, and then sink it down.

Cost: At most ***2 log N*** compares.

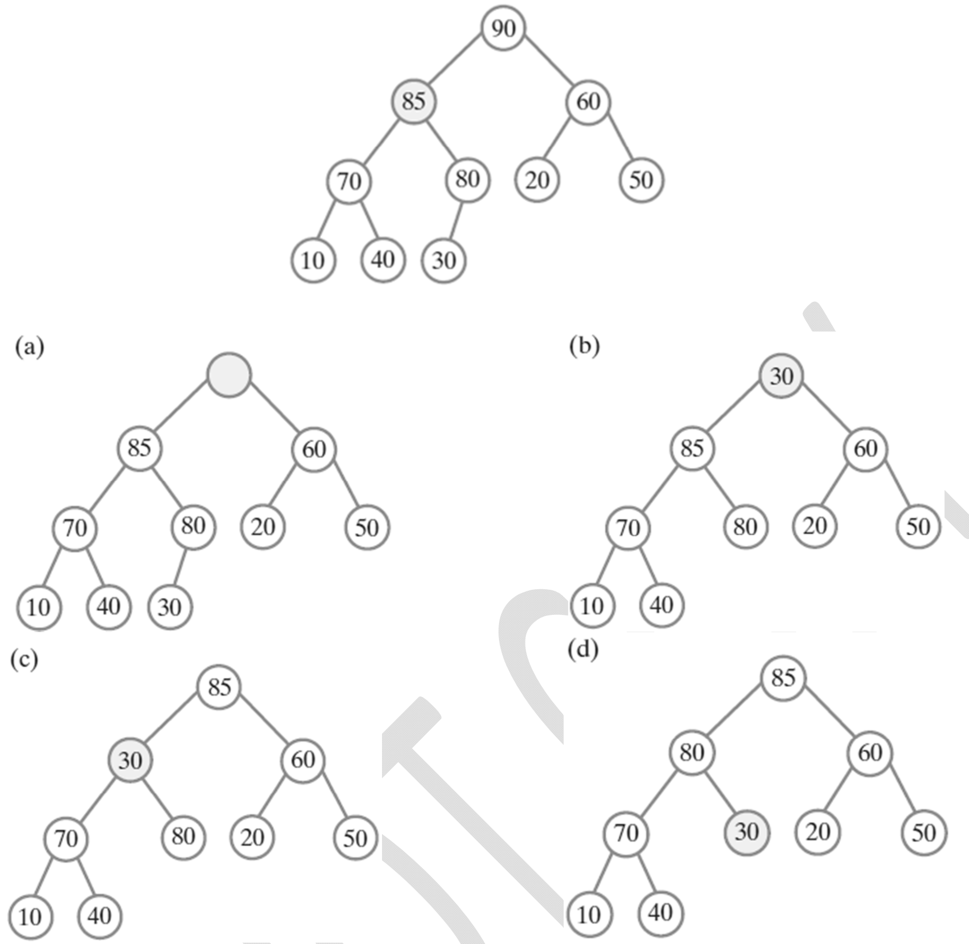
Example 1: delete **T**




```
public Key delMax()
{
   Key max = pq[1];
   exch(1, N--);
   sink(1);
   pq[N+1] = null;
   return max;
}
```

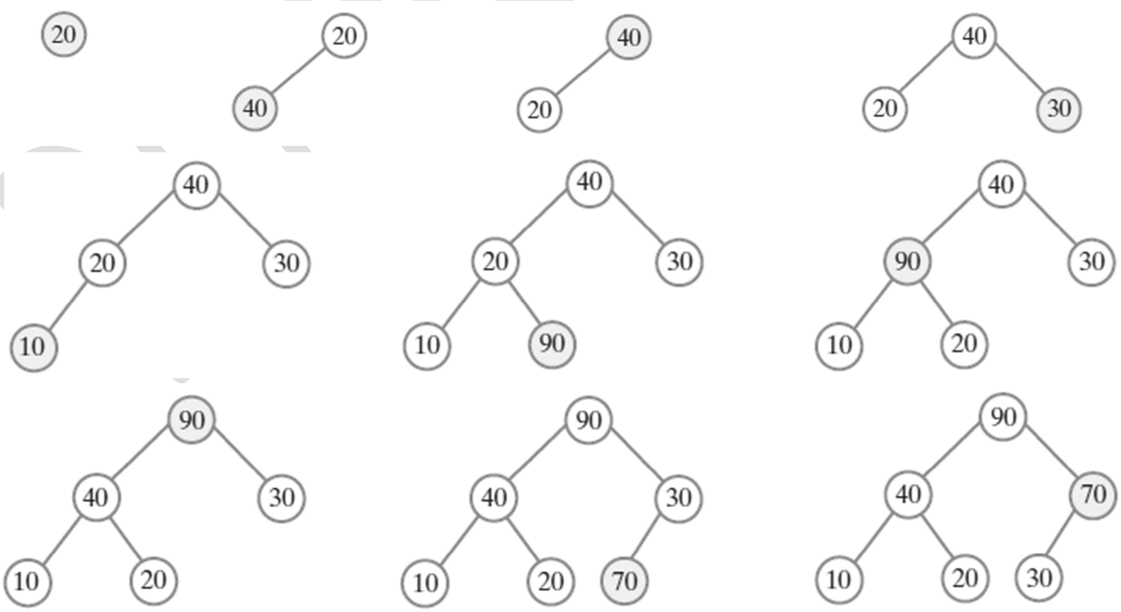
prevent loitering

Example 2: delete **root (max)**



Creating a Heap

The steps in adding 20, 40, 30, 10, 90, and 70 to an initially empty heap



	20	40	30	10	90	70
0	1	2	3	4	5	6

Binary heap: Java implementation

```
public class MaxPQ<Key extends Comparable<Key>>
{
   private Key[] pq;
   private int N;

   public MaxPQ(int capacity)
   {  pq = (Key[]) new Comparable[capacity+1];  }

   public boolean isEmpty()
   {   return N == 0;    }
   public void insert(Key key)
   public Key delMax()
   {   /* see previous code */   }

   private void swim(int k)
   private void sink(int k)
   {   /* see previous code */   }

   private boolean less(int i, int j)
   {   return pq[i].compareTo(pq[j]) < 0;  }
   private void exch(int i, int j)
   {   Key t = pq[i]; pq[i] = pq[j]; pq[j] = t;  }
}
```

fixed capacity (for simplicity)

PQ ops

heap helper functions

array helper functions

HeapSort

Basic plan:

- Create max heap with all ***N*** keys.
- Repeatedly remove the maximum key.

Heapsort demo:

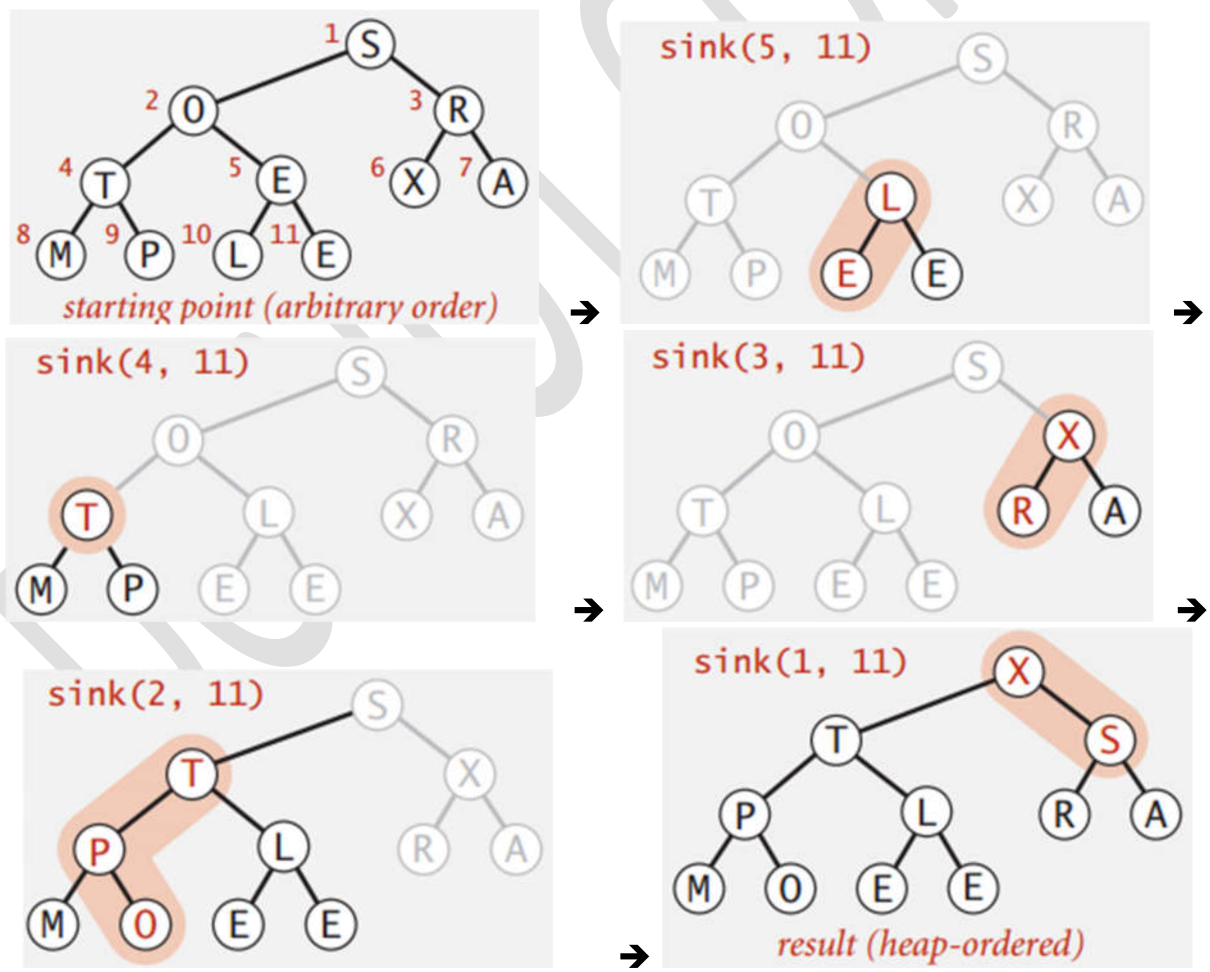
- **First pass**. Build heap using **bottom-up method**:

Array in arbitrary (random) order

S	O	R	T	E	X	A	M	P	L	E
1	2	3	4	5	6	7	8	9	10	11

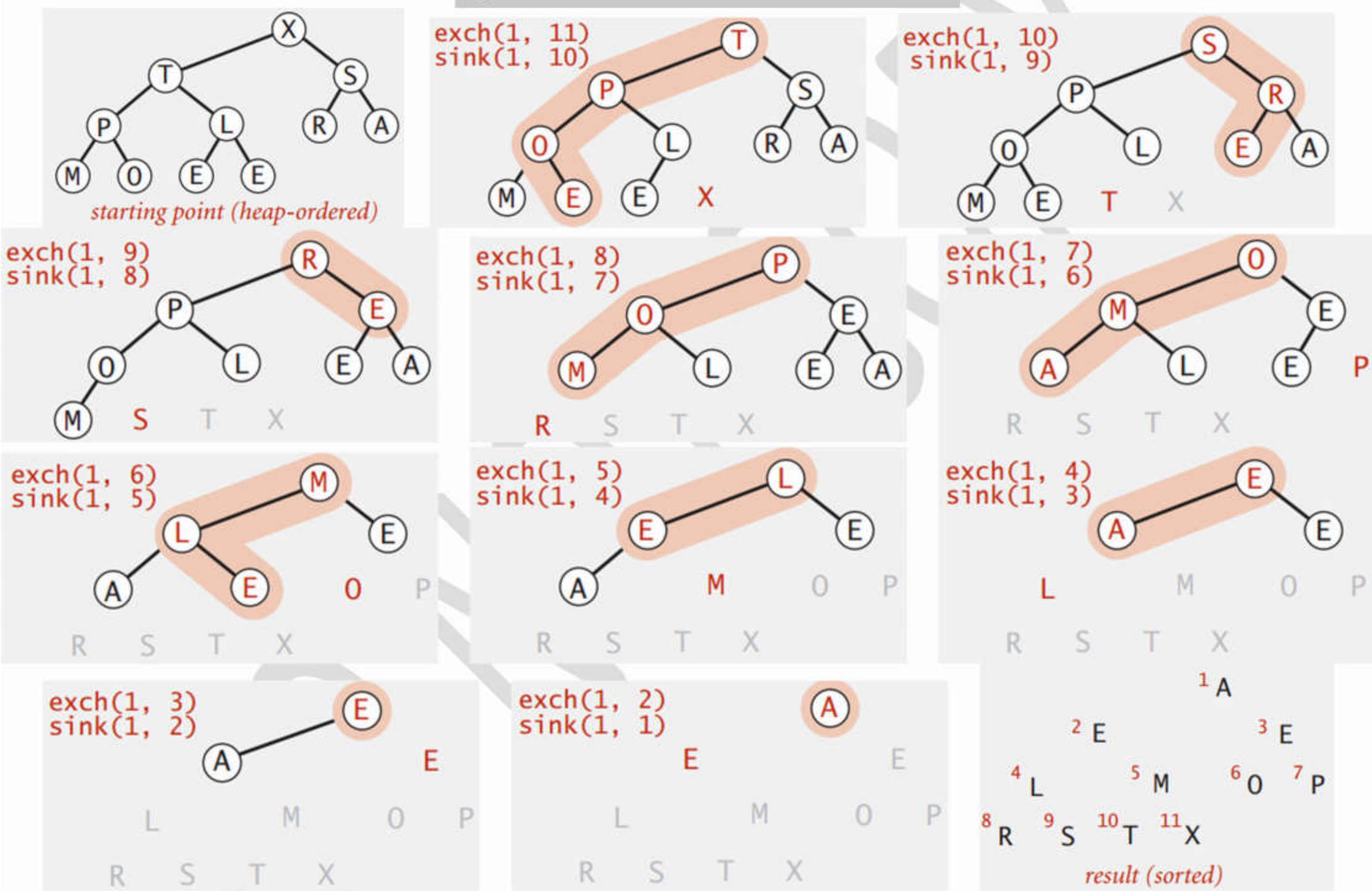
N=11

```
for (int k = N/2; k >= 1; k--)
    sink(a, k, N);
```



- **Second pass**:
  - Remove the maximum, one at a time.
  - Leave in array, instead of nulling out.

```
while (N > 1)
{
    exch(a, 1, N--);
    sink(a, 1, N);
}
```



Heapsort: trace

N	k	0	1	2	3	4	5	6	7	8	9	10	11
initial values			S	O	R	T	E	X	A	M	P	L	E
11	5		S	O	R	T	L	X	A	M	P	E	E
11	4		S	O	R	T	L	X	A	M	P	E	E
11	3		S	O	X	T	L	R	A	M	P	E	E
11	2		S	T	X	P	L	R	A	M	O	E	E
11	1		X	T	S	P	L	R	A	M	O	E	E
heap-ordered			X	T	S	P	L	R	A	M	O	E	E
10	1		T	P	S	O	L	R	A	M	E	E	X
9	1		S	P	R	O	L	E	A	M	E	T	X
8	1		R	P	E	O	L	E	A	M	S	T	X
7	1		P	O	E	M	L	E	A	R	S	T	X
6	1		O	M	E	A	L	E	P	R	S	T	X
5	1		M	L	E	A	E	O	P	R	S	T	X
4	1		L	E	E	A	M	O	P	R	S	T	X
3	1		E	A	E	L	M	O	P	R	S	T	X
2	1		E	A	E	L	M	O	P	R	S	T	X
1	1		A	E	E	L	M	O	P	R	S	T	X
sorted result			A	E	E	L	M	O	P	R	S	T	X

Heapsort trace (array contents just after each sink)

Heapsort: mathematical analysis

- Heap construction uses ***≤ 2 N*** compares and exchanges.
- Heapsort uses ***≤ 2 N lg N*** compares and exchanges.

Heapsort Significance: **In-place sorting** algorithm with ***N log N*** worst-case.
Heapsort is optimal for both time and space, but it makes poor use of cache memory and not stable.

Heapsort: Java implementation

```
public class Heap
{
   public static void sort(Comparable[] a)
   {
      int N = a.length-1;
      for (int k = N/2; k >= 1; k- )
         sink(a, k, N);
      while (N > 1)
      {
         exch(a, 1, N);
         sink(a, 1, --N);
      }
   }

   private static void sink(Comparable[] a, int k, int N)
   {  /* as before */  }

   private static boolean less(Comparable[] a, int i, int j)
   {  /* as before */  }

   private static void exch(Comparable[] a, int i, int j)
   {  /* as before */  }

}
```

Sorting

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** sort 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 algorithm), 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

3	**5**	2	1	**5’**	10	**Unsorted Array**
1	2	3	**5**	**5’**	10	**Stable sort**
1	2	3	**5’**	**5**	10	**Unstable Sort**

Example: Insertion Sort Code:

```
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;
    }
}
```

```
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:

Unsorted Array

Name	Age
Bob	25
Kevin	24
Stuart	21
Kevin	28

1) Sorted By Age

Name	Age
Stuart	21
Kevin	24
Bob	25
Kevin	28

2) Sorted By Name (Stable)

Name	Age
Bob	25
Kevin	24
Kevin	28
Stuart	21

3) Sorted By Name (Unstable)

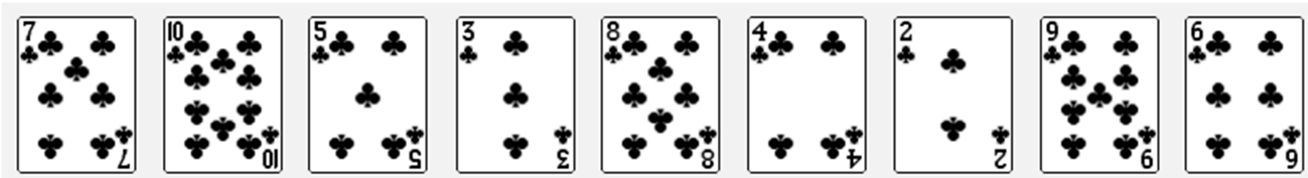
Name	Age
Bob	25
Kevin	28
Kevin	24
Stuart	21

http://www.sorting-algorithms.com/

Selection Sort

- In iteration ***i***, find index ***min*** of smallest remaining entry.
- Swap ***a[i]*** and ***a[min]***.

Demo:



Java implementation:

```
public class Selection
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int i = 0; i < N; i++)
      {
         int min = i;
         for (int j = i+1; j < N; j++)
            if (less(a[j], a[min]))
               min = j;
         exch(a, i, min);
      }
   }

   private static boolean less(Comparable v, Comparable w)
   {  /* as before */  }

   private static void exch(Comparable[] a, int i, int j)
   {  /* as before */  }
}
```

Mathematical analysis:

- Selection sort uses ***(N – 1) + (N – 2) + ... + 1 + 0*** ≈ $N^2/2$ compares and ***N*** exchanges.

Trace of selection sort:

- Running time insensitive to input: **Quadratic time**, **even if input is sorted**.
- Data movement is minimal: **Linear number of exchanges**.

i	min	a[] 0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R
5	7	A	E	E	L	M	X	S	O	P	T	R
6	8	A	E	E	L	M	O	S	X	P	T	R
7	10	A	E	E	L	M	O	P	X	S	T	R
8	8	A	E	E	L	M	O	P	R	S	T	X
9	9	A	E	E	L	M	O	P	R	S	T	X
10	10	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

entries in black are examined to find the minimum

entries in red are a[min]

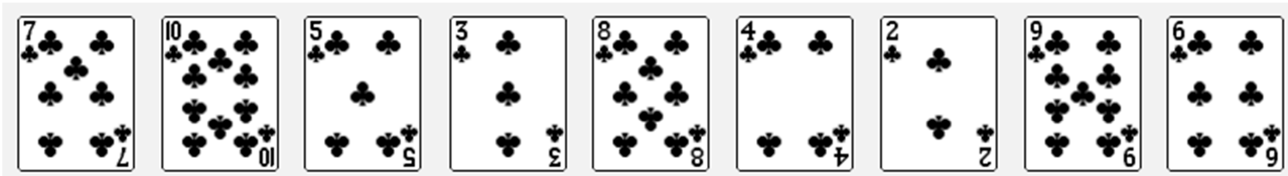
entries in gray are in final position

Trace of selection sort (array contents just after each exchange)

Insertion Sort

- In iteration ***i***, swap ***a[i]*** with each larger entry to its left.

Demo:



Java implementation:

```
public class Insertion
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int i = 0; i < N; i++)
         for (int j = i; j > 0; j--)
            if (less(a[j], a[j-1]))
               exch(a, j, j-1);
            else break;
   }

   private static boolean less(Comparable v, Comparable w)
   {  /* as before */  }

   private static void exch(Comparable[] a, int i, int j)
   {  /* as before */  }
}
```

Mathematical analysis:

- To sort a randomly-ordered array with distinct keys, insertion sort uses ≈ $\frac{1}{4}N^2$ compares and ≈ $\frac{1}{4}N^2$ exchanges on average.
- Expect each entry to move halfway back.

Trace of insertion sort:

- **Best case**: If the array is in ascending order, insertion sort makes ***N-1*** compares and ***0*** exchanges.
- **Worst case**: If the array is in descending order (and no duplicates), insertion sort makes ≈ $\frac{1}{2}N^2$ compares and ≈ $\frac{1}{2}N^2$ exchanges.
- For **partially-sorted** arrays, insertion sort runs in linear time.

		a[]										
i	j	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
1	0	O	S	R	T	E	X	A	M	P	L	E
2	1	O	R	S	T	E	X	A	M	P	L	E
3	3	O	R	S	T	E	X	A	M	P	L	E
4	0	E	O	R	S	T	X	A	M	P	L	E
5	5	E	O	R	S	T	X	A	M	P	L	E
6	0	A	E	O	R	S	T	X	M	P	L	E
7	2	A	E	M	O	R	S	T	X	P	L	E
8	4	A	E	M	O	P	R	S	T	X	L	E
9	2	A	E	L	M	O	P	R	S	T	X	E
10	2	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

entries in gray do not move

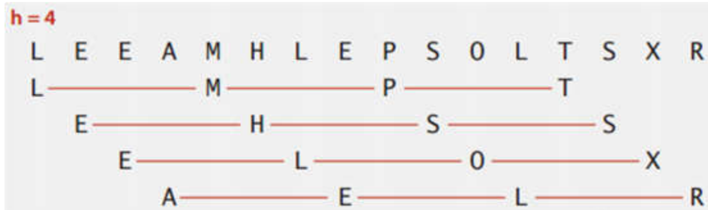
entry in red is a[j]

entries in black moved one position right for insertion

Trace of insertion sort (array contents just after each insertion)

Shell Sort

Idea: Move entries more than one position at a time by **h-sorting** the array.
an **h-sorted** array is ***h*** interleaved sorted subsequences:



Shell sort: [**Shell 1959**] **h-sort** array for decreasing sequence of values of ***h***.

input	S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E
13-sort	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
4-sort	L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R
1-sort	A	E	E	E	H	L	L	L	M	O	P	R	S	S	T	X

How to ***h-sort*** an array? Insertion sort, with stride length ***h***.

3-sorting an array

M	O	L	E	E	X	A	S	P	R	T
E	O	L	M	E	X	A	S	P	R	T
E	E	L	M	O	X	A	S	P	R	T
E	E	L	M	O	X	A	S	P	R	T
A	E	L	E	O	X	M	S	P	R	T
A	E	L	E	O	X	M	S	P	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T

Shell sort example: increments **7, 3, 1**

input

S	O	R	T	E	X	A	M	P	L	E

7-sort

S	O	R	T	E	X	A	M	P	L	E
M	O	R	T	E	X	A	S	P	L	E
M	O	R	T	E	X	A	S	P	L	E
M	O	L	T	E	X	A	S	P	R	E
M	O	L	E	E	X	A	S	P	R	T

3-sort

M	O	L	E	E	X	A	S	P	R	T
E	O	L	M	E	X	A	S	P	R	T
E	E	L	M	O	X	A	S	P	R	T
E	E	L	M	O	X	A	S	P	R	T
A	E	L	E	O	X	M	S	P	R	T
A	E	L	E	O	X	M	S	P	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T

1-sort

A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	L	E	O	P	M	S	X	R	T
A	E	E	L	O	P	M	S	X	R	T
A	E	E	L	O	P	M	S	X	R	T
A	E	E	L	O	P	M	S	X	R	T
A	E	E	L	M	O	P	S	X	R	T
A	E	E	L	M	O	P	S	X	R	T
A	E	E	L	M	O	P	S	X	R	T
A	E	E	L	M	O	P	R	S	X	T
A	E	E	L	M	O	P	R	S	T	X

result

A	E	E	L	M	O	P	R	S	T	X

Shell sort: which increment sequence to use?

- **Powers of two**: 1, 2, 4, 8, 16, 32, ... **No**
- **Powers of two minus one**: 1, 3, 7, 15, 31, 63, … **Maybe**
- **3x+1**: 1, 4, 13, 40, 121, 364, … **OK. Easy to compute**

Java implementation

```
public class Shell
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;

      int h = 1;
      while (h < N/3) h = 3*h + 1; // 1, 4, 13, 40, 121, 364, ...

      while (h >= 1)
      {  // h-sort the array.
         for (int i = h; i < N; i++)
         {
            for (int j = i; j >= h && less(a[j], a[j-h]); j -= h)
               exch(a, j, j-h);
         }

         h = h/3;
      }
   }

   private static boolean less(Comparable v, Comparable w)
   { /* as before */ }
   private static void exch(Comparable[] a, int i, int j)
   { /* as before */ }
}
```

3x+1 increment sequence

insertion sort

move to next increment

Analysis

- The **worst-case** number of compares used by shell sort with the ***3x+1*** increments is $O(N^{3/2})$.

Merge Sort

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves.

input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Mergesort overview

Java implementation:

Merging:

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
   assert isSorted(a, lo, mid);     // precondition: a[lo..mid]   sorted
   assert isSorted(a, mid+1, hi);   // precondition: a[mid+1..hi] sorted

   for (int k = lo; k <= hi; k++)
      aux[k] = a[k];                                                  copy

   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++)                                     merge
   {
      if      (i > mid)              a[k] = aux[j++];
      else if (j > hi)               a[k] = aux[i++];
      else if (less(aux[j], aux[i])) a[k] = aux[j++];
      else                           a[k] = aux[i++];
   }

   assert isSorted(a, lo, hi);      // postcondition: a[lo..hi] sorted
}
```

Java implementation:

Merge Sort:

```
public class Merge
{
   private static void merge(...)
   {  /* as before */  }

   private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
   {
      if (hi <= lo) return;
      int mid = lo + (hi - lo) / 2;
      sort(a, aux, lo, mid);
      sort(a, aux, mid+1, hi);
      merge(a, aux, lo, mid, hi);
   }

   public static void sort(Comparable[] a)
   {
      aux = new Comparable[a.length];
      sort(a, aux, 0, a.length - 1);
   }
}
```

Merge Sort: trace

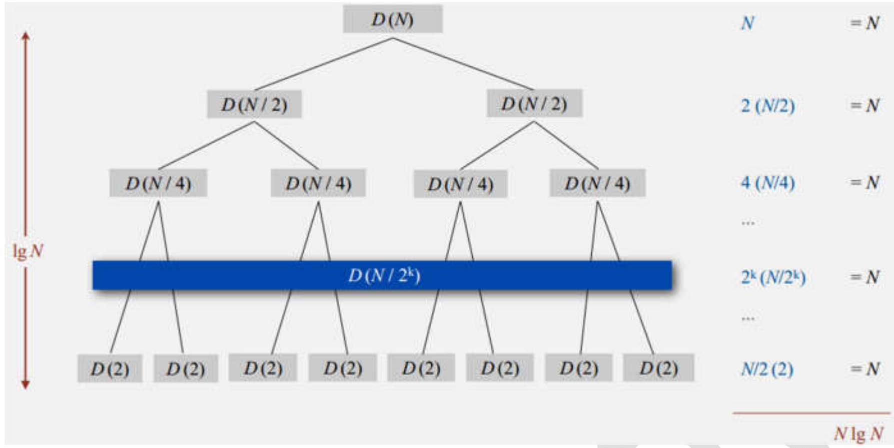
merge(a, aux, lo, mid, hi)	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]	a[12]	a[13]	a[14]	a[15]
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 0, 1, 3)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 4, 4, 5)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 6, 6, 7)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 4, 5, 7)	E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
merge(a, aux, 0, 3, 7)	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
merge(a, aux, 8, 8, 9)	E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
merge(a, aux, 10, 10, 11)	E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
merge(a, aux, 8, 9, 11)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, aux, 12, 12, 13)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, aux, 14, 14, 15)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
merge(a, aux, 12, 13, 15)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, aux, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge(a, aux, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Merge Sort: Empirical Analysis

	insertion sort (N^2)			mergesort (N log N)		
computer	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min
super	instant	1 second	1 week	instant	instant	instant

Good algorithms are better than supercomputers.

Divide-and-conquer recurrence: number of compares



Merge Sort analysis: memory (array accesses)

- Mergesort uses extra space proportional to ***N***.
- The array ***aux[]*** needs to be of size ***N*** for the last merge.

Practical Improvements:

- Use **insertion** sort for small subarrays:
  - Mergesort has too much overhead for tiny subarrays.
  - ***Cutoff*** to insertion sort for ≈ **7** items.

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
   if (hi <= lo + CUTOFF - 1)
   {
      Insertion.sort(a, lo, hi);
      return;
   }
   int mid = lo + (hi - lo) / 2;
   sort (a, aux, lo, mid);
   sort (a, aux, mid+1, hi);
   merge(a, aux, lo, mid, hi);
}
```

- Stop if already sorted:
  - Is biggest item in first half ≤ smallest item in second half?
  - Helps for partially-ordered arrays.

```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
   if (hi <= lo) return;
   int mid = lo + (hi - lo) / 2;
   sort (a, aux, lo, mid);
   sort (a, aux, mid+1, hi);
   if (!less(a[mid+1], a[mid])) return;
   merge(a, aux, lo, mid, hi);
}
```

- Eliminate the copy to the auxiliary array. Save time (but not space) by switching the role of the input and auxiliary array in each recursive call.

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++)
   {
      if      (i > mid)          aux[k] = a[j++];
      else if (j > hi)           aux[k] = a[i++];
      else if (less(a[j], a[i])) aux[k] = a[j++];      merge from a[] to aux[]
      else                       aux[k] = a[i++];
   }
}

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
   if (hi <= lo) return;
   int mid = lo + (hi - lo) / 2;
   sort (aux, a, lo, mid);
   sort (aux, a, mid+1, hi);          Note: sort(a) initializes aux[] and sets
   merge(a, aux, lo, mid, hi);        aux[i] = a[i] for each i.
}

   switch roles of aux[] and a[]
```

Complexity of sorting

- Compares? Mergesort is optimal with respect to number compares.
- Space? Mergesort is not optimal with respect to space usage.

Bottom-up Merge Sort

Basic plan:

- Pass through array, merging subarrays of size 1.
- Repeat for subarrays of size 2, 4, 8, 16, ...

	a[i] 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
sz = 1	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 0, 0, 1)	**E**	**M**	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 2, 2, 3)	E	M	**G**	**R**	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 4, 4, 5)	E	M	G	R	**E**	**S**	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 6, 6, 7)	E	M	G	R	E	S	**O**	**R**	T	E	X	A	M	P	L	E
merge(a, aux, 8, 8, 9)	E	M	G	R	E	S	O	R	**E**	**T**	X	A	M	P	L	E
merge(a, aux, 10, 10, 11)	E	M	G	R	E	S	O	R	E	T	**A**	**X**	M	P	L	E
merge(a, aux, 12, 12, 13)	E	M	G	R	E	S	O	R	E	T	A	X	**M**	**P**	L	E
merge(a, aux, 14, 14, 15)	E	M	G	R	E	S	O	R	E	T	A	X	M	P	**E**	**L**
sz = 2																
merge(a, aux, 0, 1, 3)	**E**	**G**	**M**	**R**	E	S	O	R	E	T	A	X	M	P	E	L
merge(a, aux, 4, 5, 7)	E	G	M	R	**E**	**O**	**R**	**S**	E	T	A	X	M	P	E	L
merge(a, aux, 8, 9, 11)	E	G	M	R	E	O	R	S	**A**	**E**	**T**	**X**	M	P	E	L
merge(a, aux, 12, 13, 15)	E	G	M	R	E	O	R	S	A	E	T	X	**E**	**L**	**M**	**P**
sz = 4																
merge(a, aux, 0, 3, 7)	**E**	**E**	**G**	**M**	**O**	**R**	**R**	**S**	A	E	T	X	E	L	M	P
merge(a, aux, 8, 11, 15)	E	E	G	M	O	R	R	S	**A**	**E**	**E**	**L**	**M**	**P**	**T**	**X**
sz = 8																
merge(a, aux, 0, 7, 15)	**A**	**E**	**E**	**E**	**E**	**G**	**L**	**M**	**M**	**O**	**P**	**R**	**R**	**S**	**T**	**X**

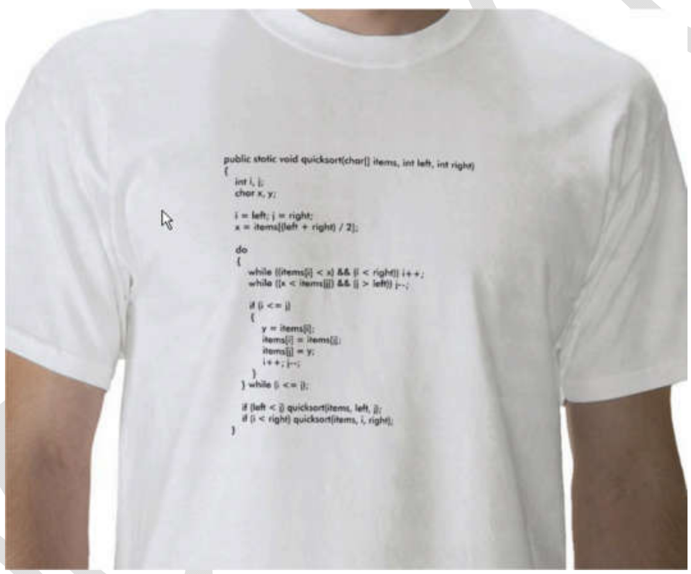
Java implementation

```
public class MergeBU
{
   private static void merge(...)
   {  /* as before */  }

   public static void sort(Comparable[] a)
   {
      int N = a.length;
      Comparable[] aux = new Comparable[N];
      for (int sz = 1; sz < N; sz = sz+sz)
         for (int lo = 0; lo < N-sz; lo += sz+sz)
            merge(a, aux, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
   }
}
```

Quick Sort

Basic plan:

- Shuffle the array. (*shuffle needed for performance guarantee*)
- Partition so that, for some ***j***
  - entry ***a[j]*** is in place
  - no larger entry to the left of ***j***
  - no smaller entry to the right of ***j***
- Sort each piece recursively.



Quicksort t-shirt

Quicksort partitioning demo

Repeat until ***i*** and ***j*** pointers cross.

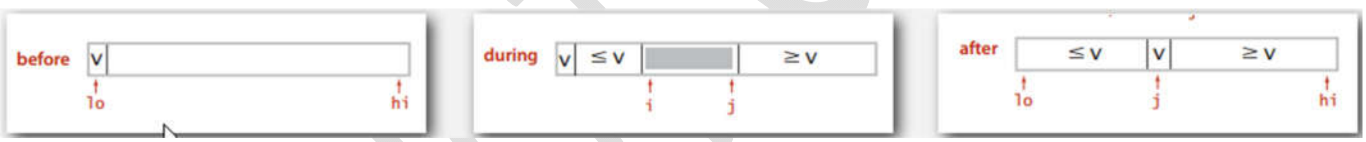
- Scan ***i*** from left to right so long as (***a[i] < a[lo]***).
- Scan ***j*** from right to left so long as (***a[j] > a[lo]***).
- Exchange ***a[i]*** with ***a[j]*** .

When pointers (***i*** and ***j***)cross.

- Exchange ***a[lo]*** with ***a[j]*** .

Quicksort: Java code for partitioning

```
private static int partition(Comparable[] a, int lo, int hi)
{
   int i = lo, j = hi+1;
   while (true)
   {
      while (less(a[++i], a[lo]))                find item on left to swap
         if (i == hi) break;

      while (less(a[lo], a[--j]))                find item on right to swap
         if (j == lo) break;

      if (i >= j) break;                         check if pointers cross
      exch(a, i, j);                                                    swap
   }

   exch(a, lo, j);                               swap with partitioning item
   return j;                       return index of item now known to be in place
}
```




```
public class Quick
{
   private static int partition(Comparable[] a, int lo, int hi)
   {  /* see previous slide */  }

   public static void sort(Comparable[] a)
   {
      StdRandom.shuffle(a);
      sort(a, 0, a.length - 1);
   }

   private static void sort(Comparable[] a, int lo, int hi)
   {
      if (hi <= lo) return;
      int j = partition(a, lo, hi);
      sort(a, lo, j-1);
      sort(a, j+1, hi);
   }
}
```

Quicksort trace

no partition for subarrays of size 1

	lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values				Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
random shuffle				K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
	0	5	15	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
	0	3	4	E	C	A	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	2	2	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	0	1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	1		1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	4		4	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	6	6	15	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	7	9	15	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	7	7	8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	8		8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	10	13	15	A	C	E	E	I	K	L	M	O	P	S	Q	R	T	U	X
	10	12	12	A	C	E	E	I	K	L	M	O	P	R	Q	S	T	U	X
	10	11	11	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	10		10	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	14	14	15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	15		15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
result				A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

Quicksort trace (array contents after each partition)

Quicksort: Empirical Analysis

	insertion sort (N^2)			mergesort (N log N)			quicksort (N log N)		
computer	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.6 sec	12 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

Quicksort: Compare analysis

Best case: Number of compares is ≈ ***N log N***

a[ ]

	lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values				H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
random shuffle				H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
	0	7	14	D	A	C	B	F	E	G	H	L	I	K	J	N	M	O
	0	3	6	B	A	C	D	F	E	G	H	L	I	K	J	N	M	O
	0	1	2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
	0		0	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
	2		2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
	4	5	6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
	4		4	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
	6		6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
	8	11	14	A	B	C	D	E	F	G	H	J	I	K	L	N	M	O
	8	9	10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
	8		8	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
	10		10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
	12	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
	12		12	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
	14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
				A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Worst case: Number of compares is ≈ ***½N²***

			a[ ]														
lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
random shuffle			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0	0	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	1	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
2	2	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
3	3	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
4	4	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	5	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
6	6	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
7	7	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
8	8	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
9	9	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
10	10	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
11	11	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12	12	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
13	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Average-case analysis: Complicated ➔ ***2N log N***

Quicksort: summary of performance characteristics

Worst case: Number of compares is quadratic.

- ***N + (N - 1) + (N - 2) + … + 1*** ≈ ***½ N²***
- but this **rarely** to happen.

Average case: Number of compares is ≈ ***1.39 N lg N***

- 39% more compares than Mergesort
- But faster than Mergesort in practice because of less data movement.

Random shuffle

- Probabilistic guarantee against worst case.

Quicksort is an **in-place** sorting algorithm.

Quicksort is **not stable**.

Quicksort: practical improvements

1- **Insertion sort small subarrays:**

- Even quicksort has too much overhead for tiny subarrays.
- **Cutoff** to insertion sort for ≈ 10 items.
- Note: could delay insertion sort until one pass at end.

```
private static void sort(Comparable[] a, int lo, int hi)
{
   if (hi <= lo + CUTOFF - 1)
   {
      Insertion.sort(a, lo, hi);
      return;
   }
   int j = partition(a, lo, hi);
   sort(a, lo, j-1);
   sort(a, j+1, hi);
}
```

2- **Median of sample:**

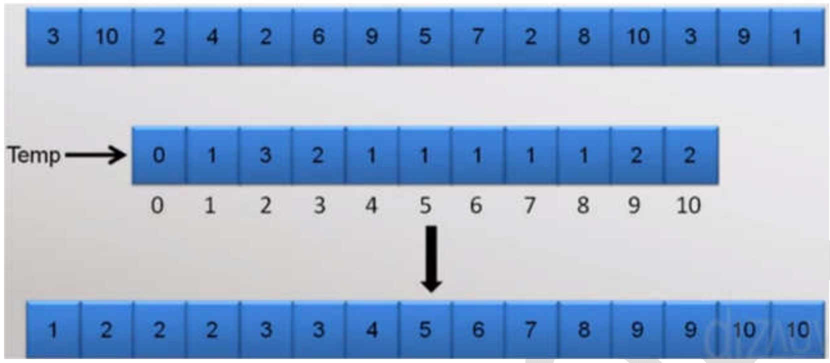
- Best choice of pivot item = median.
- Estimate true median by taking median of sample.

```
private static void sort(Comparable[] a, int lo, int hi)
{
   if (hi <= lo) return;

   int m = medianOf3(a, lo, lo + (hi - lo)/2, hi);
   swap(a, lo, m);

   int j = partition(a, lo, hi);
   sort(a, lo, j-1);
   sort(a, j+1, hi);
}
```

Counting Sort

If we know some information about data to be sorted (e.g. students' marks [Range 55 to 99]), we can achieve linear time sorting

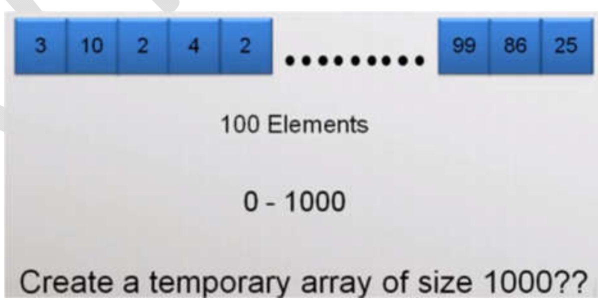
Example: assume data range from 1 to 10

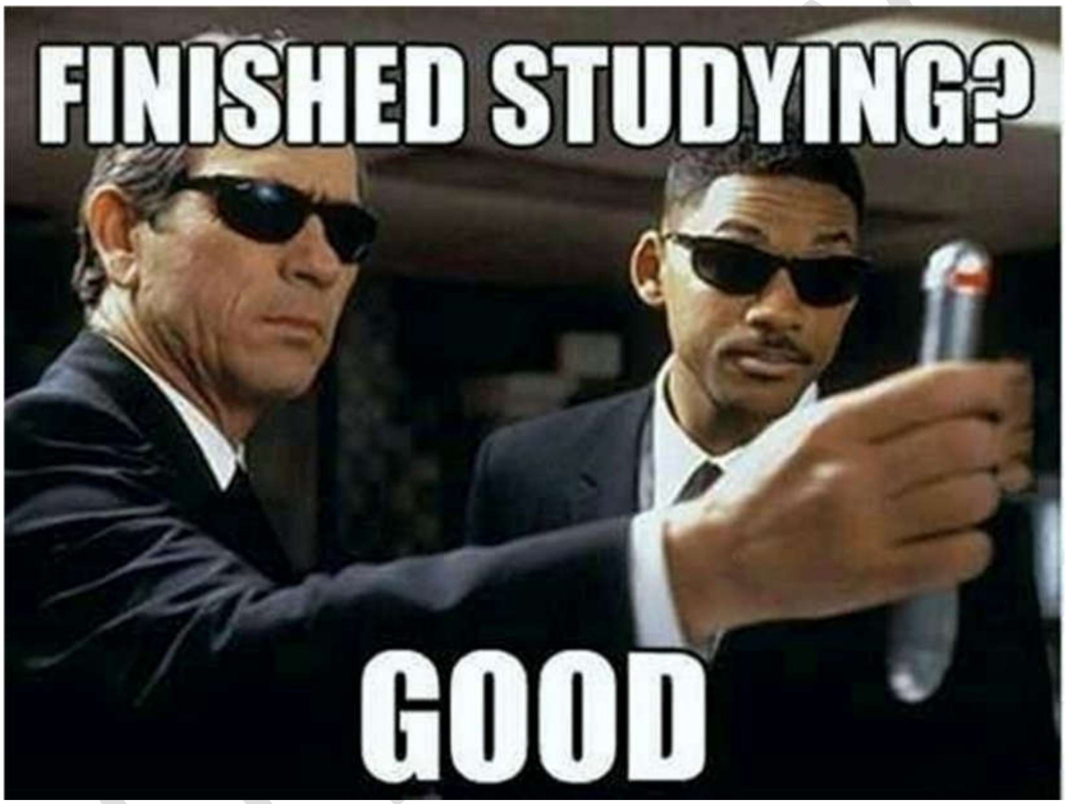


Time analysis:

Note: K is typically small comparing to n

Bad Situation: what if K is larger than n ??




FINISHED STUDYING?
GOOD