جامعة بيرزيت
BIRZEIT UNIVERSITY

COMP232

Data Structure

Lectures Note

Prepared by: **Dr. Mamoun Nawahdah**

2015

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,

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

Table of Contents

(Lecture 3) What is an Algorithm?

Definition:

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

Euclid's Algorithm (300BC)

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

Euclid's Algorithm:

- 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

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.

Analysis of Algorithms

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

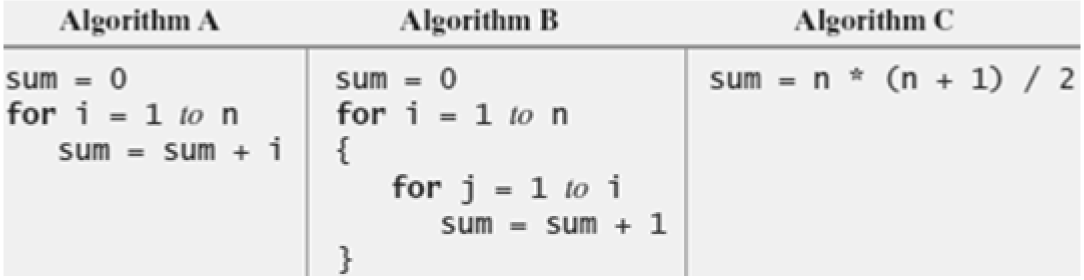
- **Space Complexity** ➔ memory and storage is very cheap nowadays. ✗
- **Time Complexity** ✓ Different platforms ➔ different time. Absolute time is hard to measure as it depends on many factors.

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

Example:

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

Come up with an algorithm to solve this problem.



Algorithm A

```
sum = 0
for i = 1 to n
    sum = sum + i
```

Algorithm B

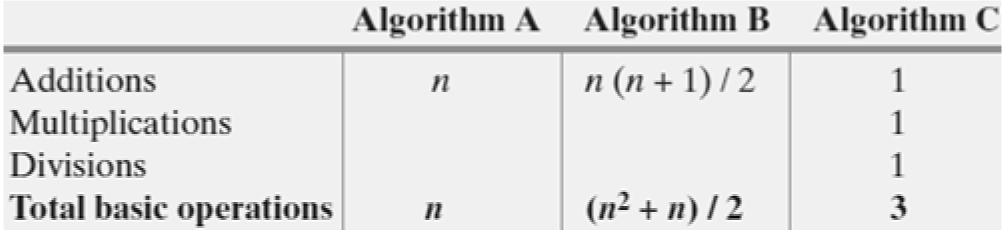
```
sum = 0
for i = 1 to n
{
    for j = 1 to i
        sum = sum + 1
}
```

Algorithm C

```
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**

- **Space Complexity** ×
- **Time Complexity** ✓

How to calculate the time complexity?

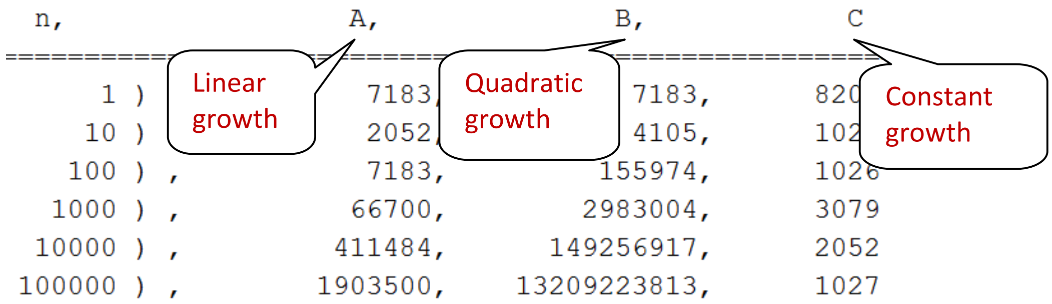
- Measure execution time. × Algorithm for small data size will take small time comparing to a large data.
- Calculate time required for an algorithm in terms of the size of input data. × Does not work as the same algorithm over the same data will not take the same time.

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

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

RAM model of computation

We assume that:

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

Bubble sort

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

Make a demo using the following data set

12	8	7	5	2

Worst case analysis

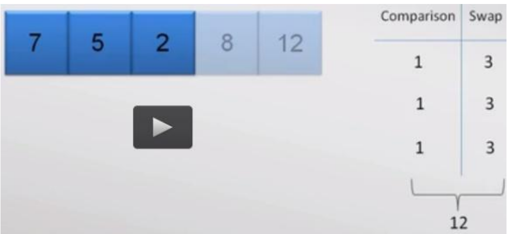
After 1st round:

8	7	5	2	12

Comparison	Swap
1	3
1	3
1	3
1	3

$\underbrace{\qquad}_{16}$

After 2nd round:



7	5	2	8	12

Comparison	Swap
1	3
1	3
1	3

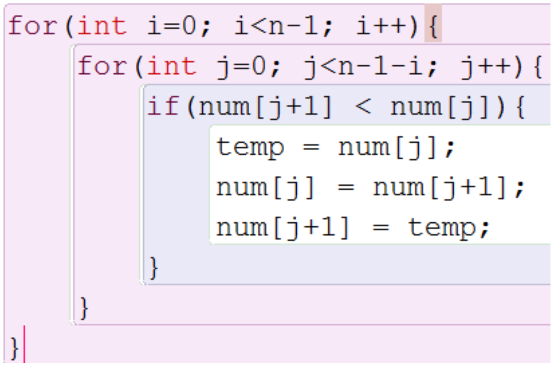
$\underbrace{\qquad}_{12}$

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

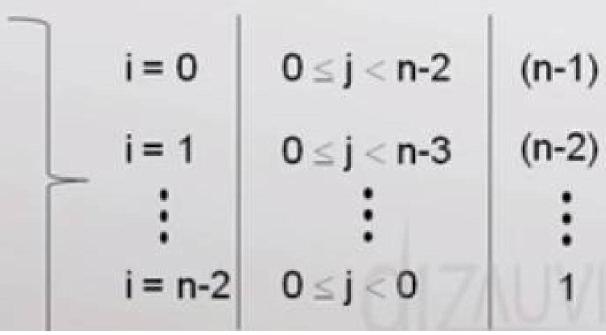
$= 4\,(4 + 3 + 2 + 1) = 4\,(n-1 + n-2 + \dots + 2 + 1) = 4\,(n-1 * n/2) =$

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

Implement and test effectiveness of bubble sort algorithm



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



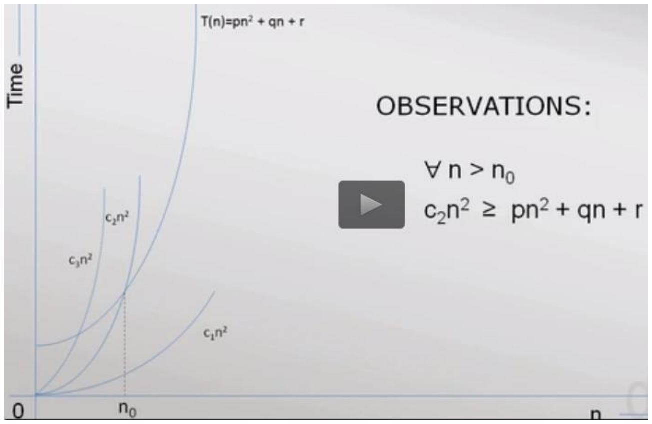
$i = 0$	$0 \le j < n-2$	$(n-1)$
$i = 1$	$0 \le j < n-3$	$(n-2)$
⋮	⋮	⋮
$i = n-2$	$0 \le j < 0$	1

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.



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

In general

$O(g(n))$: f(n): there exist positive constants **c** and n_0 such that $0 <= f(n) <= cg(n)$ for all $n >= n_0$

Example 1:

$5n^2 + 6 \in O(n^2)$??? ✓

Find cn^2 ➔ c=6 and $n_0=3$

➔ c=5.1 n0=8

Example 2:

$5n + 6 \in O(n^2)$??? ✓

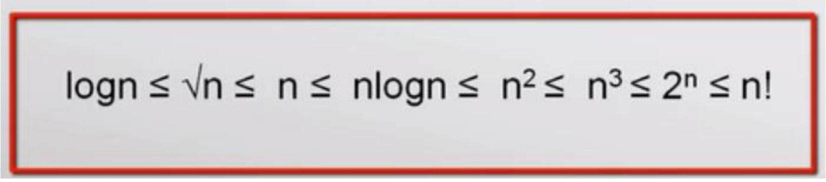
Find cn^2 ➔ c=11 and $n_0=1$

Example 3:

$n^3 + 2n^2 + 4n + 8 \in O(n^2)$??? ×

Find cn^2 >= $n^3 + 2n^2 + 4n + 8$??? ×

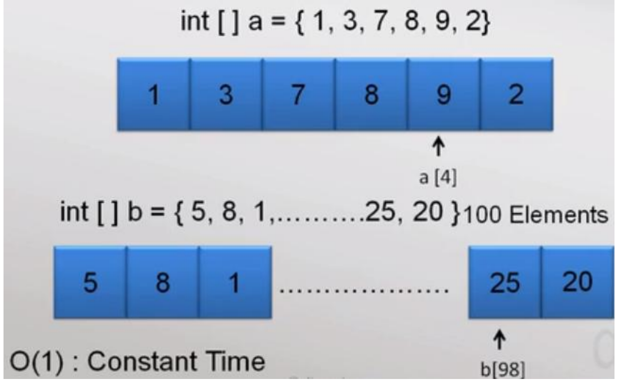
$a_m n^m + a_{m-1} n^{m-1}$ - - - - - - - - - - - -+ $a_0 \in O(n^m)$



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

What does it mean?

Array element access:

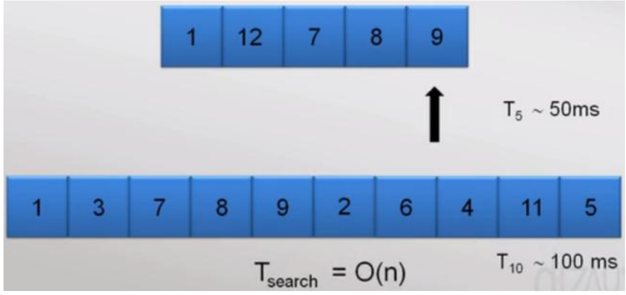


int [ ] a = { 1, 3, 7, 8, 9, 2}

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

O(1) : Constant Time

Array element search:

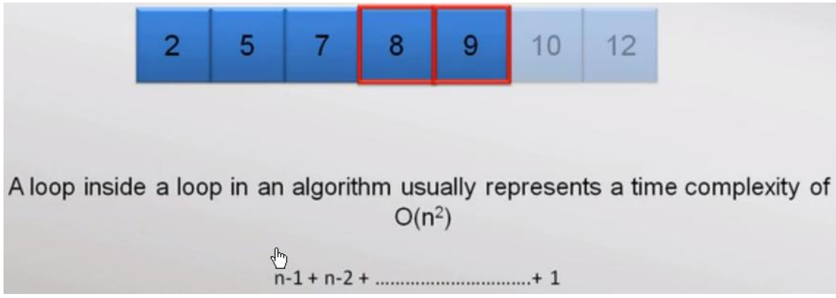


T_5 ~ 50ms

T_{search} = O(n)

T_{10} ~ 100 ms

Bubble sort algorithm:



A loop inside a loop in an algorithm usually represents a time complexity of $O(n^2)$

n-1 + n-2 + ……………………………+ 1

(Lecture 5) Asymptotic Analysis

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

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

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

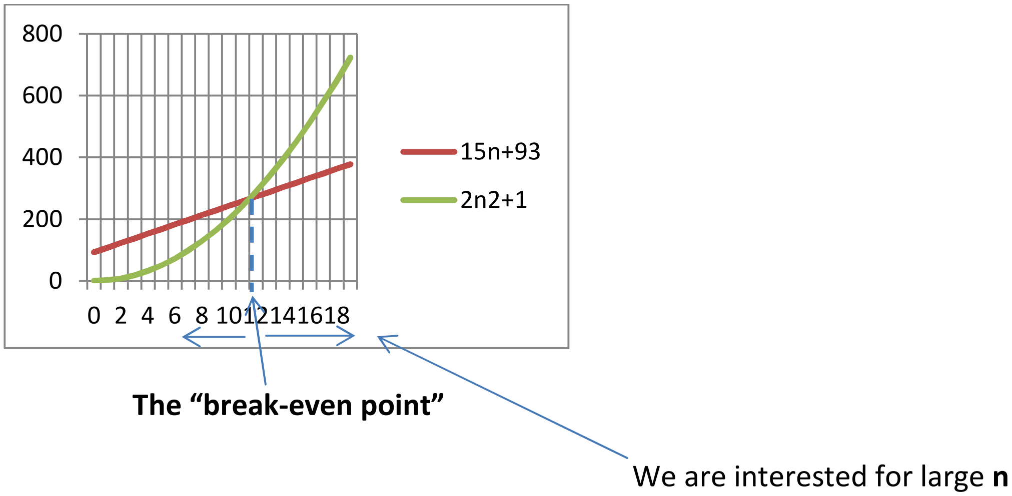
Worst case? The advantage to analyzing the worst case is that you know for certain that the algorithm must perform at least that well.

Example:

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

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

Graph using Excel



*** 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 $T(n) \le cf(n)$ for all $n > n_0$.

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

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

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

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

Graph using Excel

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

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

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

$2n^2+ n^2$ ➔ $3n^2$ <provided c=3>

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

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

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, $c_1n^2 + c_2n \leq c_1n^2 + c_2n^2 \leq (c_1 + c_2)n^2$ for all $n > 1$. So, $\mathbf{T}(n) \leq cn^2$ for $c = c_1 + c_2$, and $n_0 = 1$. Therefore, $\mathbf{T}(n)$ is in $\mathrm{O}(n^2)$ by the second definition.

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

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

* prove that **15n+93** is **Ω(n)**

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

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

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

Graph using Excel

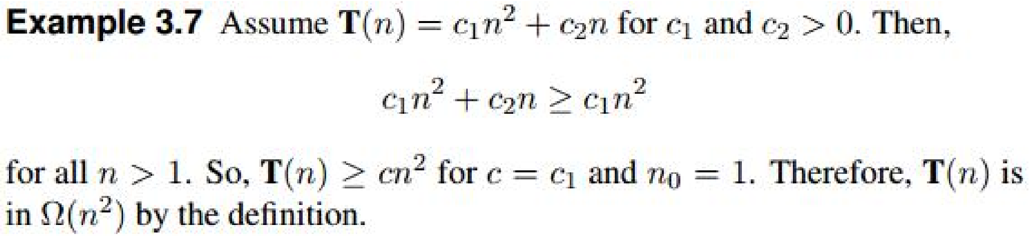
* prove that **$2n^2+1$** is **$\Omega(n^2)$**

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

<because 1 is +ve>

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

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,

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

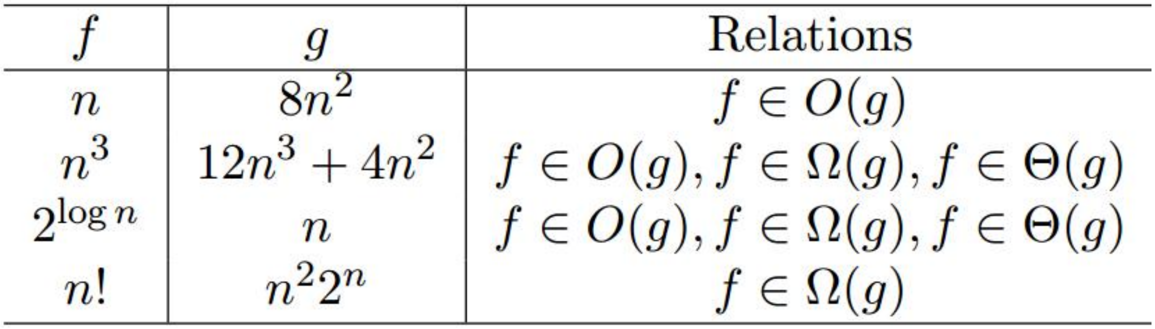
for all $n > 1$. So, $\mathbf{T}(n) \geq cn^2$ for $c = c_1$ and $n_0 = 1$. Therefore, $\mathbf{T}(n)$ is in $\Omega(n^2)$ by the definition.

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))**

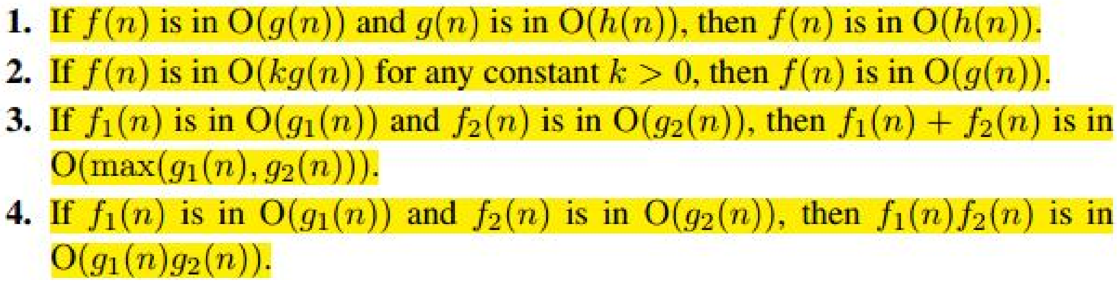
Example: Because the **sequential search algorithm** is both in **O(*n*)** and in **Ω(*n*)** in the average case, we say it is **Θ(*n*)** in the average case.

Examples:



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

Simplifying Rules



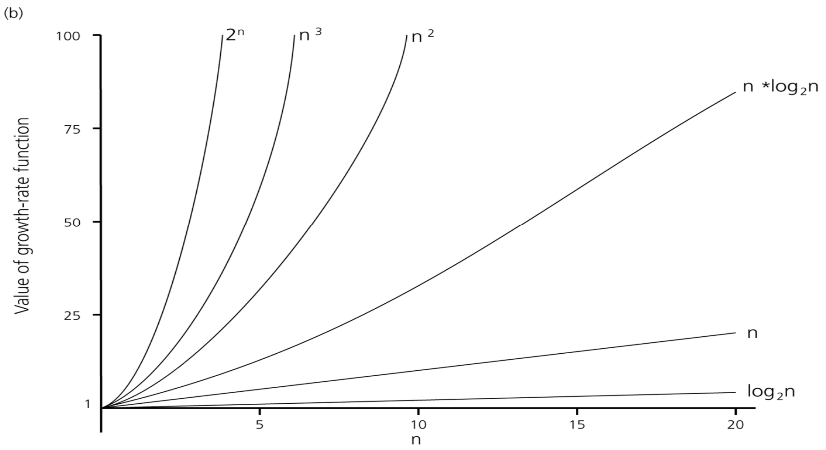
1. If $f(n)$ is in $\mathrm{O}(g(n))$ and $g(n)$ is in $\mathrm{O}(h(n))$, then $f(n)$ is in $\mathrm{O}(h(n))$.
2. If $f(n)$ is in $\mathrm{O}(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $\mathrm{O}(g(n))$.
3. If $f_1(n)$ is in $\mathrm{O}(g_1(n))$ and $f_2(n)$ is in $\mathrm{O}(g_2(n))$, then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

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

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

Order of growth of some common functions

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



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

Limitations of big-oh analysis

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

(Lecture 6) Analyzing algorithm examples

General Rules of analyzing algorithm code:

Rule 1—*for* loops.

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

Rule 2 — Nested loops.

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

Rule 3—Consecutive Statements.

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

Rule 4 —*if/else*.

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

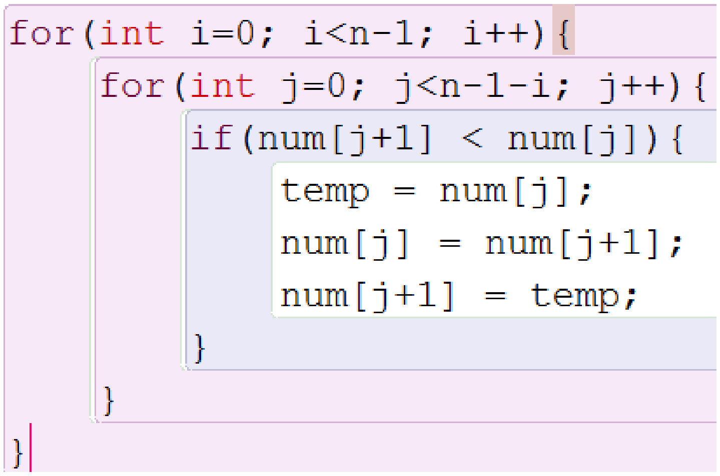
The running time of an **if/else** statement is never more than the running time of the **test** plus the larger of the running times of **S1** and **S2**.

Rule 5 —*methods call*.

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

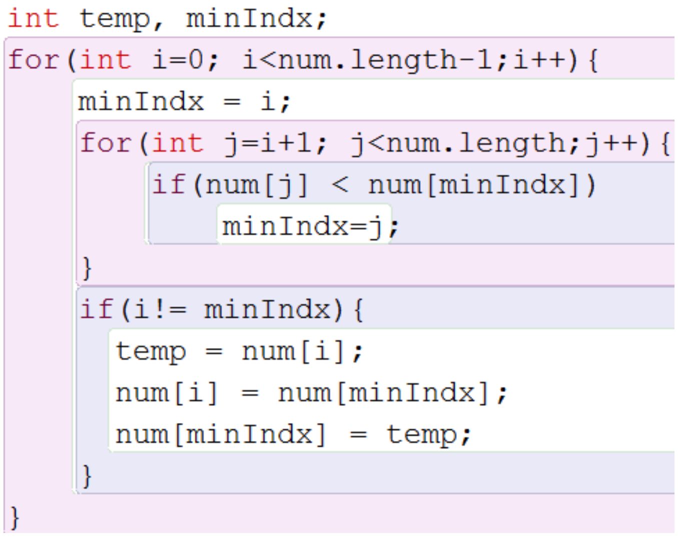
Sorting Algorithm

1- Bubble Sort (revision) ➔ O(n^2)



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

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

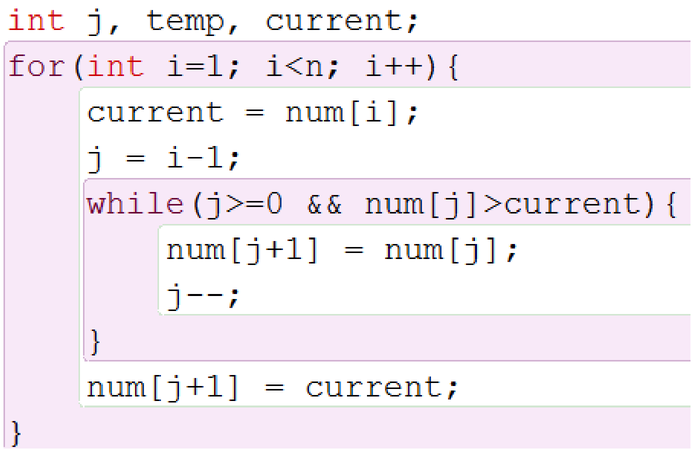


```
int temp, minIndx;
for(int i=0; i<num.length-1;i++){
    minIndx = i;
    for(int j=i+1; j<num.length;j++){
        if(num[j] < num[minIndx])
            minIndx=j;
    }
    if(i!= minIndx){
        temp = num[i];
        num[i] = num[minIndx];
        num[minIndx] = temp;
    }
}
```

3- **Insertion sort:**

Example:

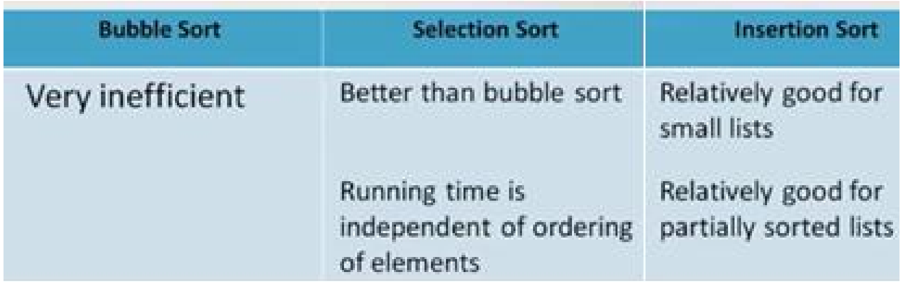
Pseudo code:



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

O(n^2) sorting algorithms comparison :

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

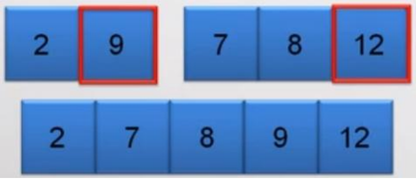


Bubble Sort	Selection Sort	Insertion Sort
Very inefficient	Better than bubble sort	Relatively good for small lists
	Running time is independent of ordering of elements	Relatively good for partially sorted lists

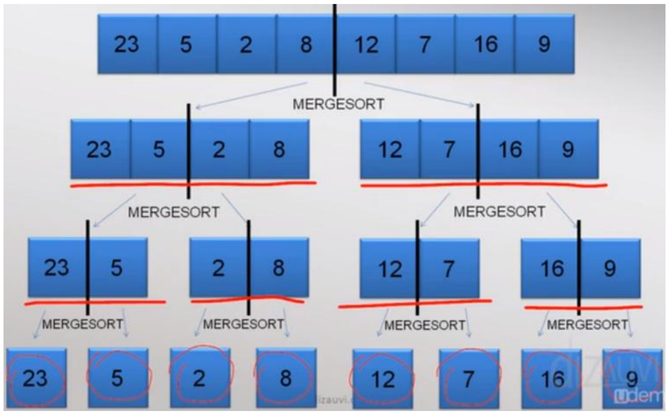
Merge sort : recursive algorithm

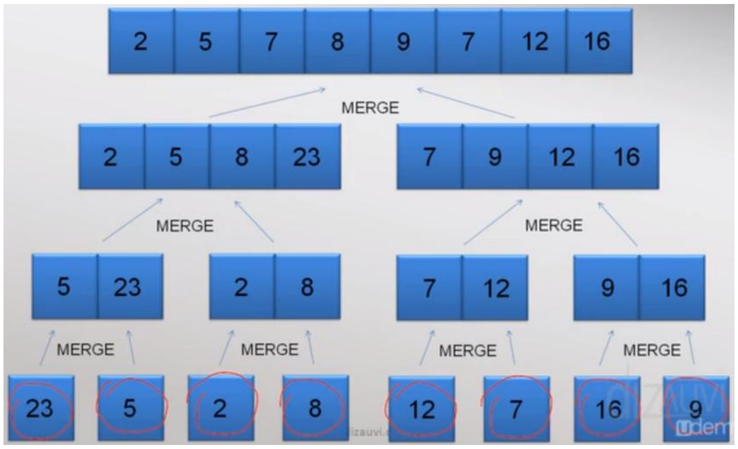
Merge: take 2 sorted arrays and merge them together into one.

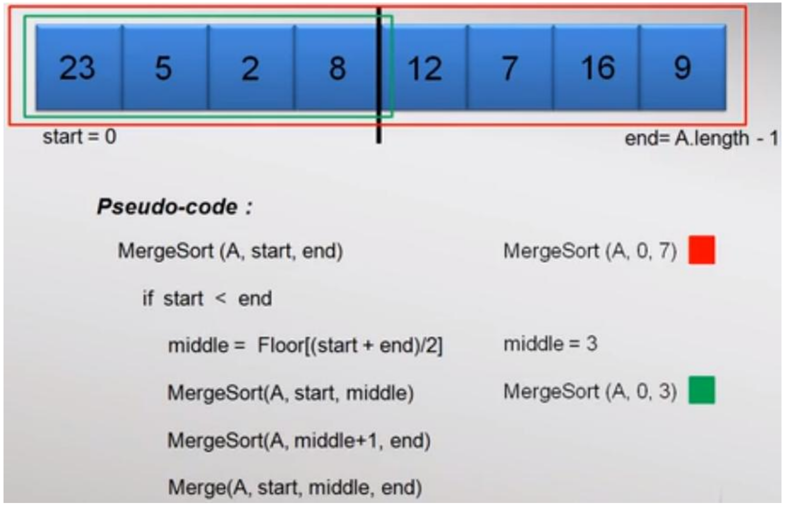
Example: merge method



Example: merge sort



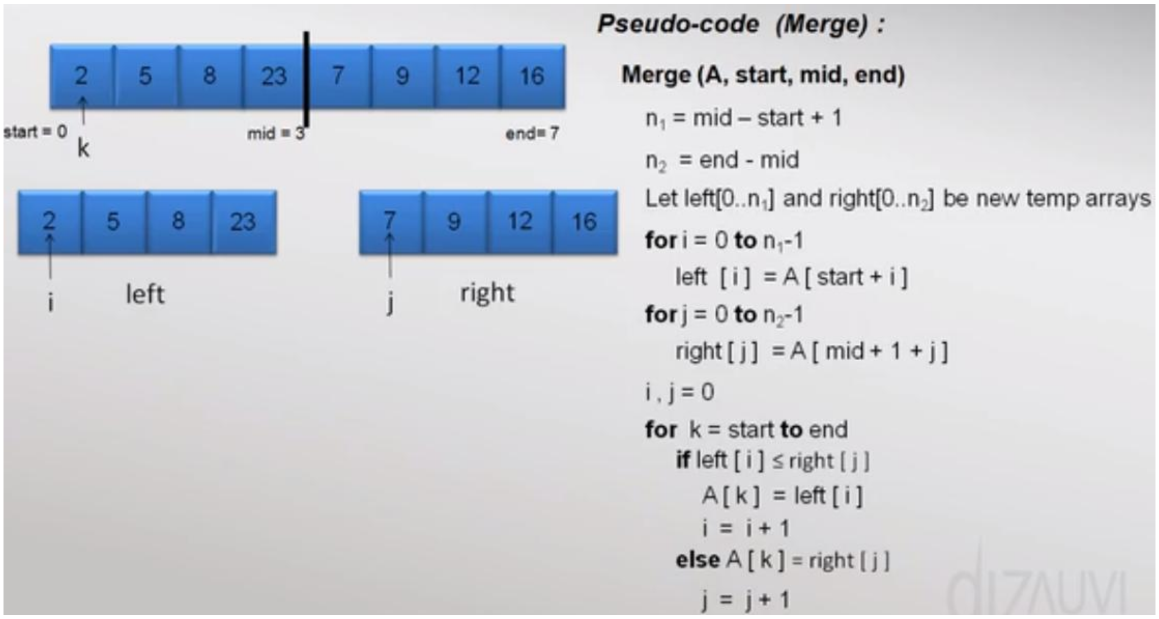




Pseudo-code :

```
MergeSort (A, start, end)                    MergeSort (A, 0, 7) 🟥
    if start < end
        middle = Floor[(start + end)/2]      middle = 3
        MergeSort(A, start, middle)          MergeSort (A, 0, 3) 🟩
        MergeSort(A, middle+1, end)
        Merge(A, start, middle, end)
```

Pseudo code:



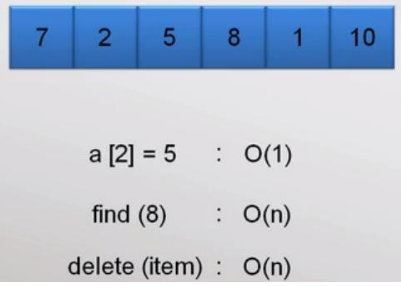
Pseudo-code (Merge) :

```
Merge (A, start, mid, end)
    $n_1$ = mid – start + 1
    $n_2$ = end - mid
    Let left[0..$n_1$] and right[0..$n_2$] be new temp arrays
    for i = 0 to $n_1$-1
        left [ i ] = A [ start + i ]
    for j = 0 to $n_2$-1
        right [ j ] = A [ mid + 1 + j ]
    i , j = 0
    for k = start to end
        if left [ i ] ≤ right [ j ]
            A [ k ] = left [ i ]
            i = i + 1
        else A [ k ] = right [ j ]
            j = j + 1
```

Make sure of array boundaries

H.W: implement merge sort your own

Searching elements in an array:

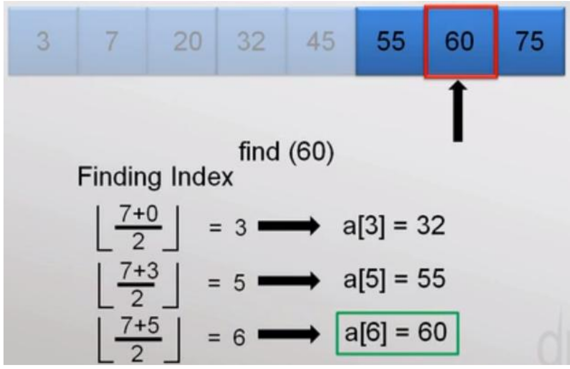


a [2] = 5 : O(1)

find (8) : O(n)

delete (item) : O(n)

Case 1: unordered array:



3	7	20	32	45	55	60	75

find (60)

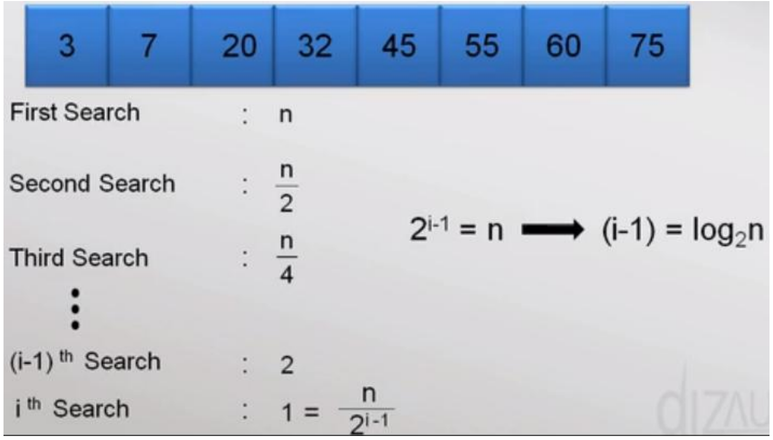
Finding Index

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

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

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

Case 2: ordered array: -Binary search-



3	7	20	32	45	55	60	75

First Search : n

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

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

⋮

(i-1) th Search : 2

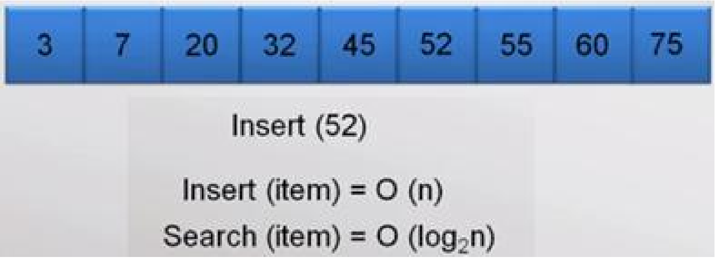
i th Search : $1 = \frac{n}{2^{i-1}}$

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

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

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

Inserting and deleting items from ordered array



3	7	20	32	45	52	55	60	75

Insert (52)

Insert (item) = O (n)

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

3	7	20	32	45	52	60	75	

Delete (55)

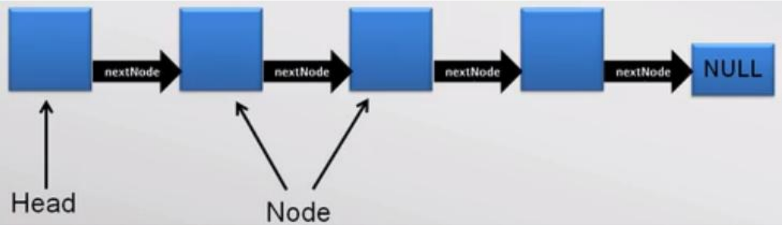
Delete (item) = O (n)

(Lecture 7) Linked List

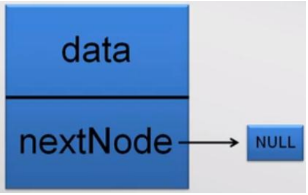
Algorithm - abstract way to perform computation tasks

Data Structure - abstract way to organize information

Linked List:



Node:



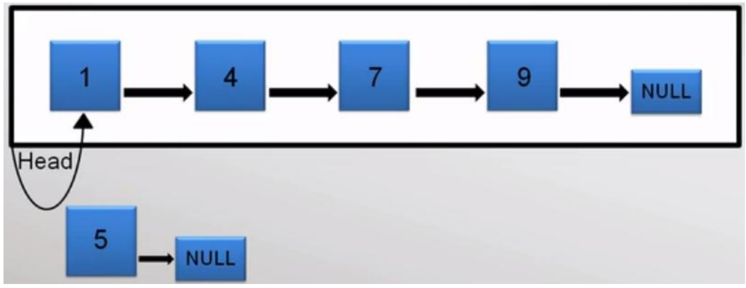
Node code:

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

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

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

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

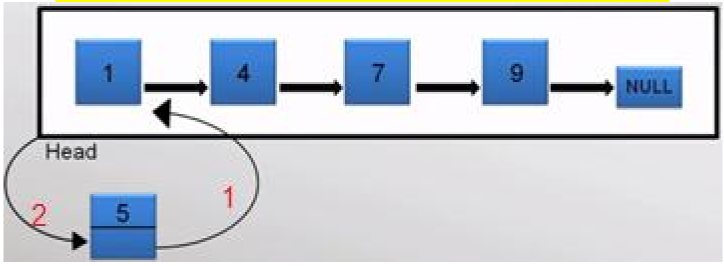
Linked List Code:

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

Inserting a new node:



Connect Head → new node ?? we lose pointer to linked list
Order of connecting the node is very important



Insert code:

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

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

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

Node toString:

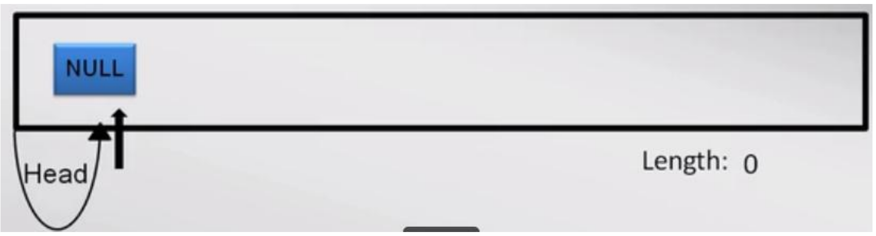
```
@Override
        public String toString() {  return this.data.toString();  }
```

LinkedList toString:

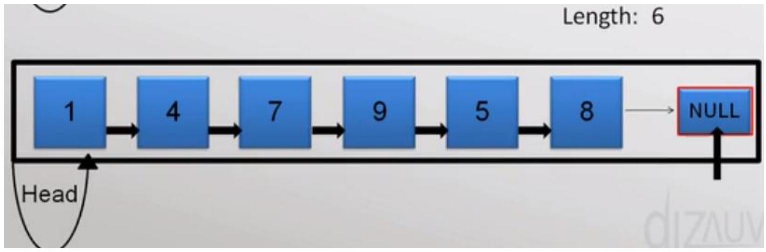
```
@Override
        public String toString() {
                String res = "→";
                Node<T> curr = this.head;
                while (curr != null) {
                        res += curr + "→ ";
                        curr = curr.getNextNode();
                }
                return res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

Length of Linked List?

Case 1: If it's empty:



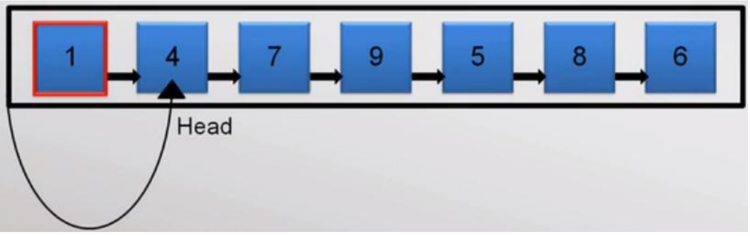
Case 2: If not: Make a pointer and move over all the nodes and maintain a counter



Length code: Time Complexity ➔ O(n)

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

Deleting the head node:

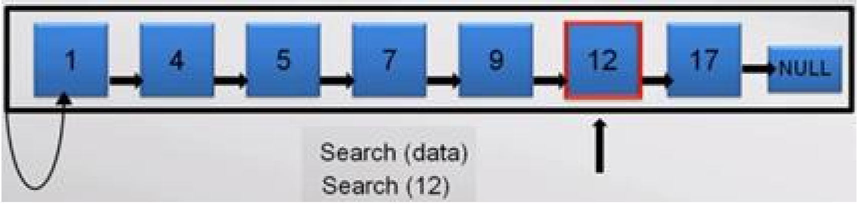


Simply move the **head** to the **head.nextNode**
Now first Node has no reference to it ➔ Garbage
Time Complexity ➔ **O(1)**

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

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

Searching for an Item in a Linked List:



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

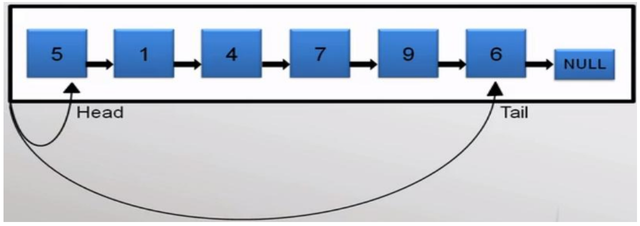
Find code:

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

How to use Java generics?? (Optional)

Provided by java, to be able to parameterize the Node and Linked List objects.

Doubly Ended Linked List:



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

Doubly Ended list code:

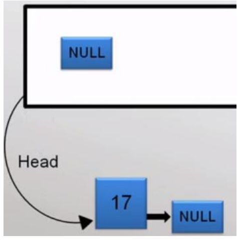
```
public class DoubleEndedList<T> extends LinkedList<T> {
        private Node<T> tail;

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

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

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

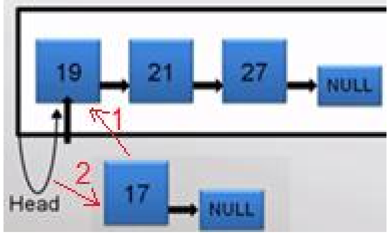
```
  @Override
  public void addAtStart(T data) {
    Node<T> newNode = new Node<T>(data);
    if (this.head == null) {  // empty
      this.head = newNode;
      this.tail = newNode;
    }
    else{
      newNode.setNextNode(this.head);
      this.head = newNode;
    }
  }
```

Inserting new Node to a sorted linked list:

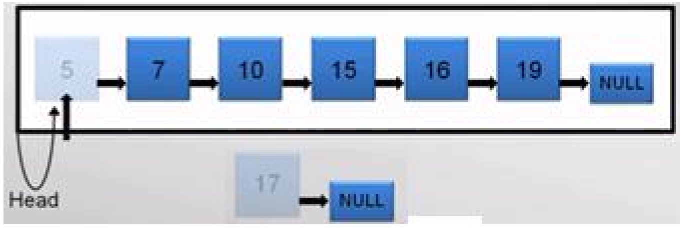
Case 1: empty linked list: in this case we added as first element.

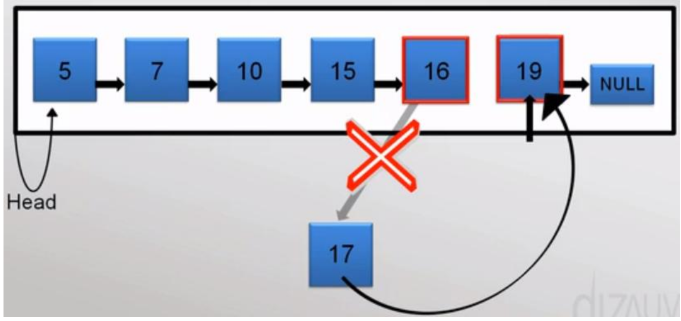


Case 2: adding first to a sorted linked list:

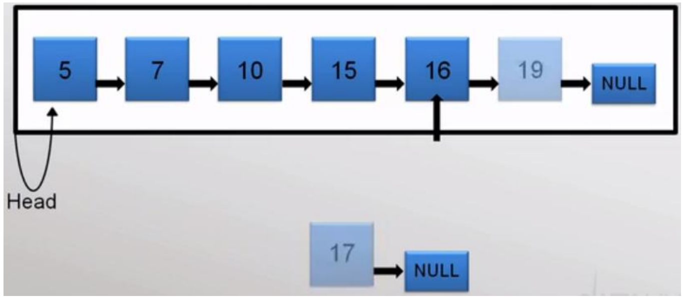


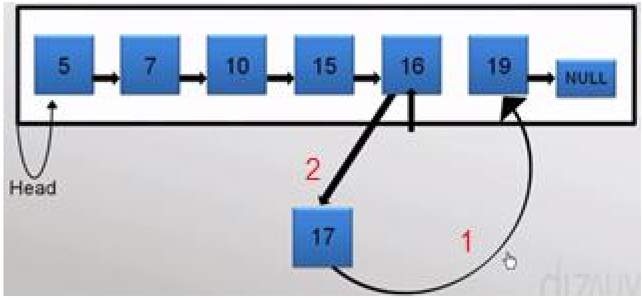
Case 3: adding in the middle in a sorted linked list:





However we can access the next node from the current node.





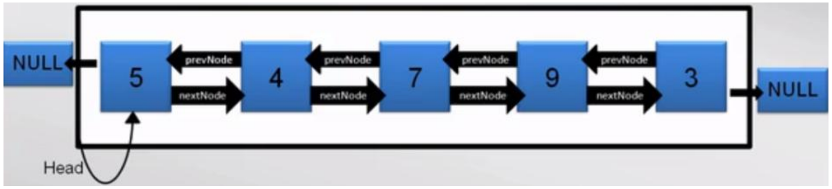
Time Complexity ➔ **O(n)**

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

(Lecture 8) Doubly Linked List

Node:

Doubly Linked List:

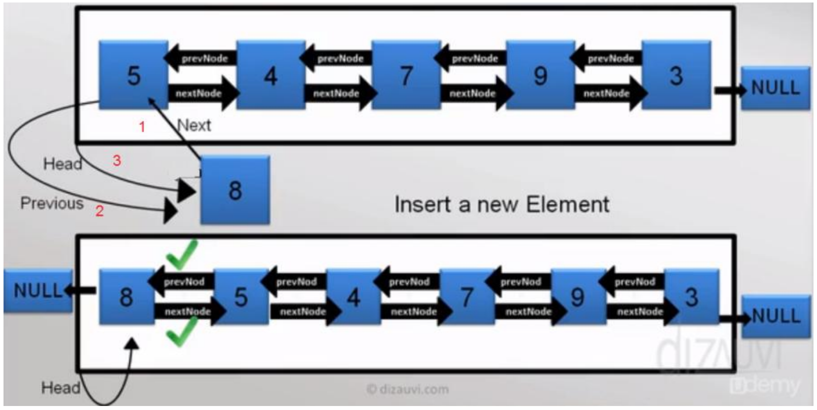


Doubly Node Code:

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

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

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

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

Doubly Linked List code:

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



Insert a new node at head:

Insert at head code:

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

Length of a doubly linked list code:

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

Override toString method code:

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

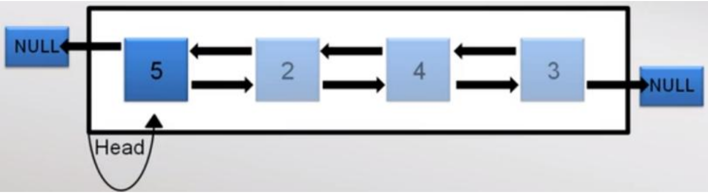
Student Activity: insert at last

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

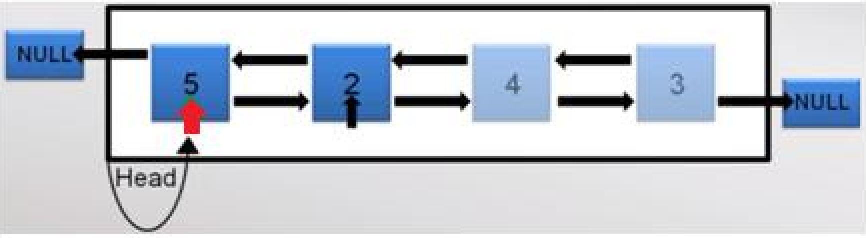
Insertion Sort using doubly linked list:

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.

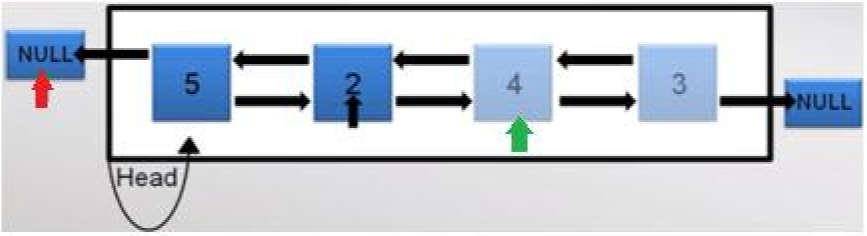
Step 1: The **red** pointer keeps move backward until it reaches a node which has a value **smaller** than the **current** node **or** reach **NULL**.

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

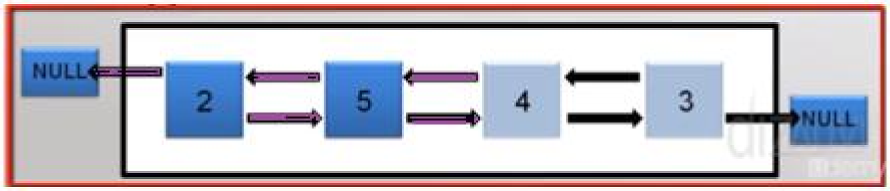
Make sure you maintain references correctly.

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

Initial state:



Final state:

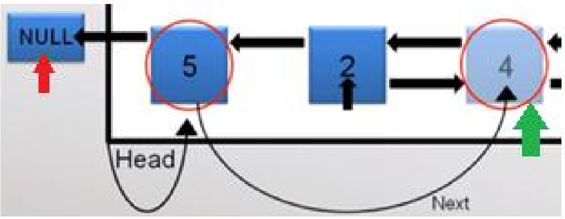


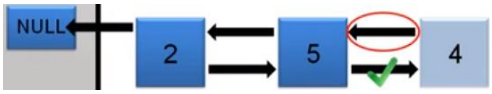
Case 1: insert to head

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

Step 2.1: 

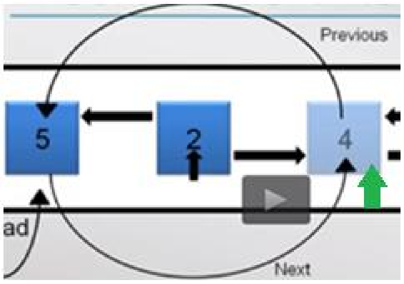
➔ **black.prevNode.nextNode = green**

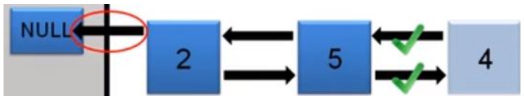


Step 2.2: 

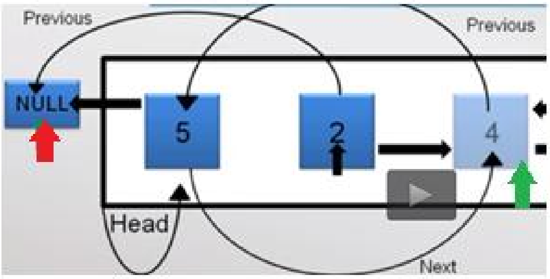
➔

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



Step 2.3: 

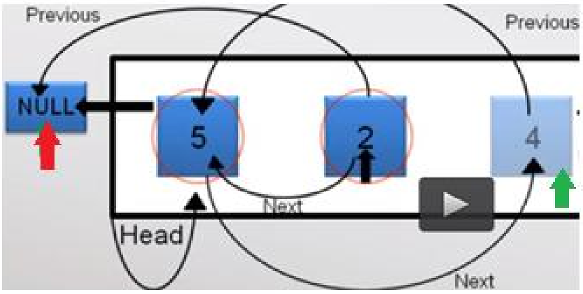
➔ **black.prevNode = red**



Step 2.4: 

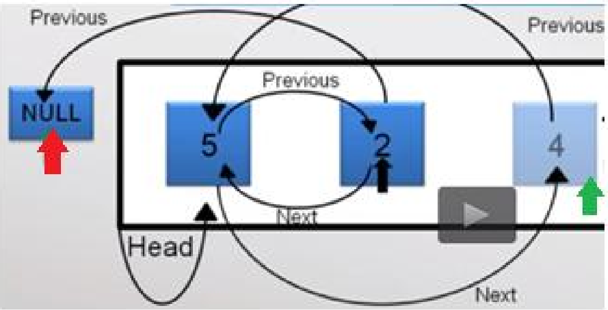
➔

if(red==null) black.nextNode = black.nextNode.prevNode
else black.nextNode = red.nextNode



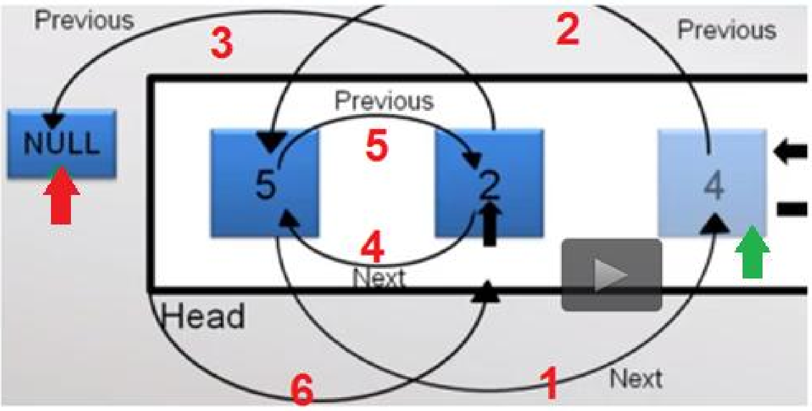
Step 2.5: ➔

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



Step 2.6:

if (red == NULL) head = black
else red.setNextNode = black;

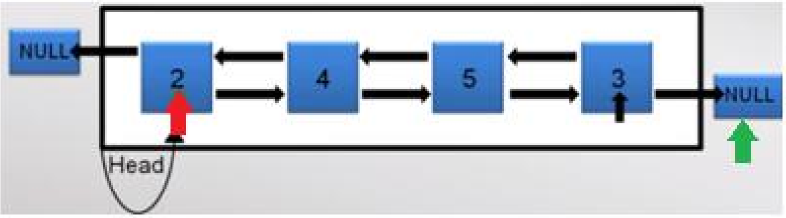


Step 2.7: black = green

Case 2: insert 4 in the middle

Practice yourself

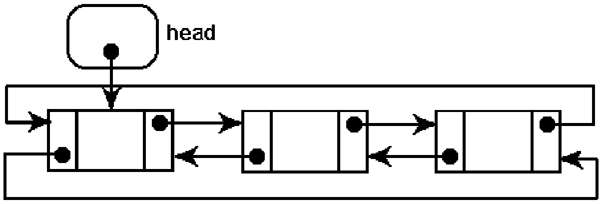
Case 3: insert last element



Insertion Sort Code:

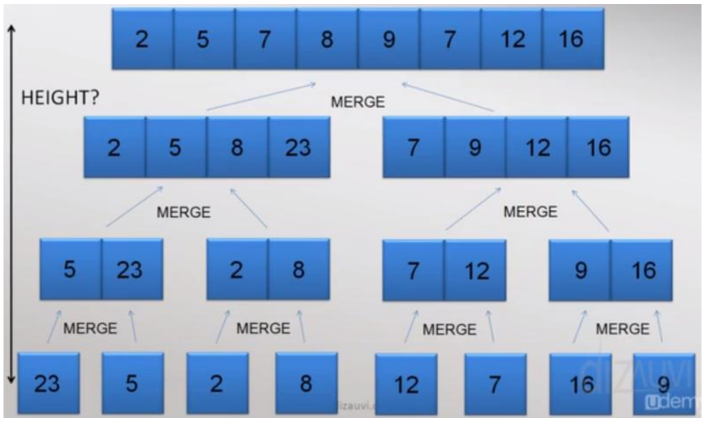
```
    public void sort() {
        DNode black = this.head;
        while (black != null) {
            DNode red = black.getPrevNode();
            while (red != null && (red.getData() > black.getData())) {
                red = red.getPrevNode();
            }
            DNode green = black.getNextNode();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

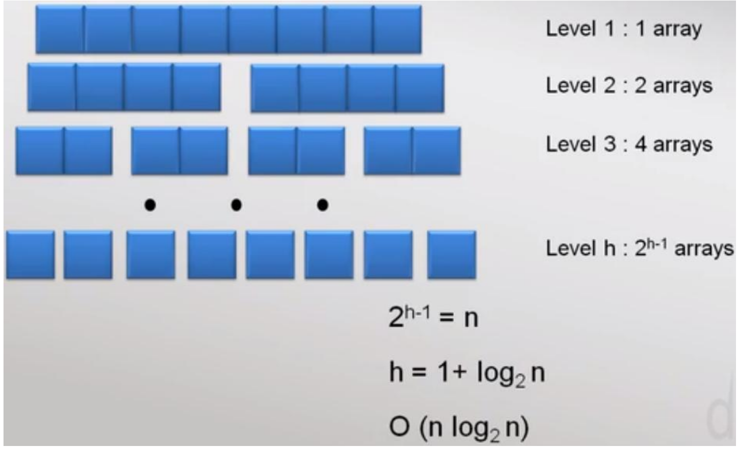
Circular Double Linked List:



Doubly Linked Circular list

(Lecture 9) Analyzing the Complexity of Merge Sort





In Place vs. Not in Place Sorting

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

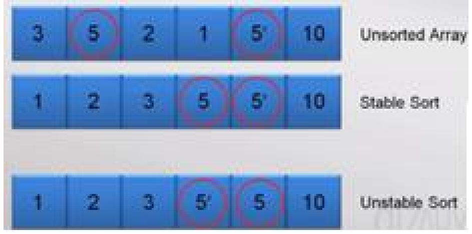
What about selection, bubble, insertion algorithms?

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

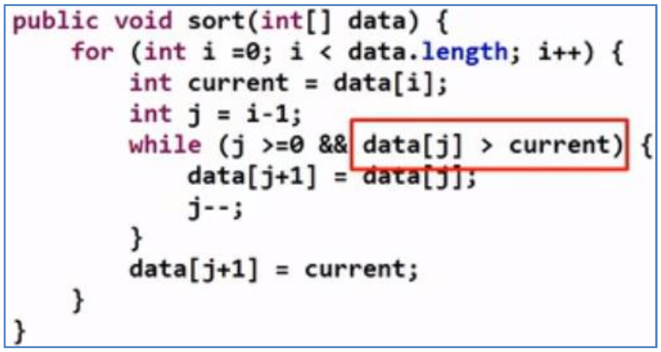
But in case extra memory (merging sort), which is **proportional** to the input data size, is used, then it is **NOT IN PLACE sorting**.

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

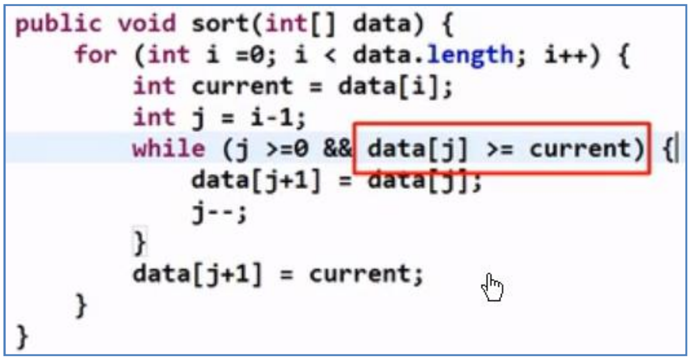
Stable vs. Unstable Sort



Example: Insertion Sort Code:

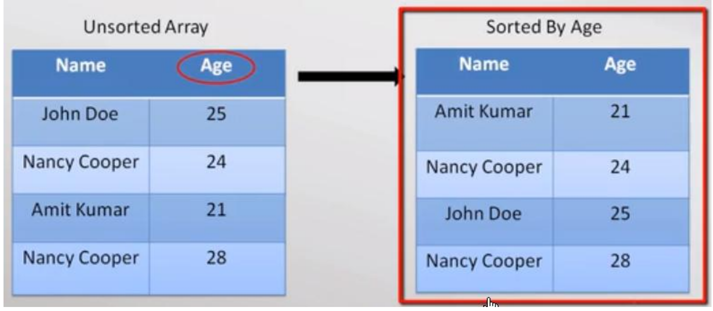


```
public void sort(int[] data) {
    for (int i =0; i < data.length; i++) {
        int current = data[i];
        int j = i-1;
        while (j >=0 && data[j] > current) {
            data[j+1] = data[j];
            j--;
        }
        data[j+1] = current;
    }
}
```



```
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:

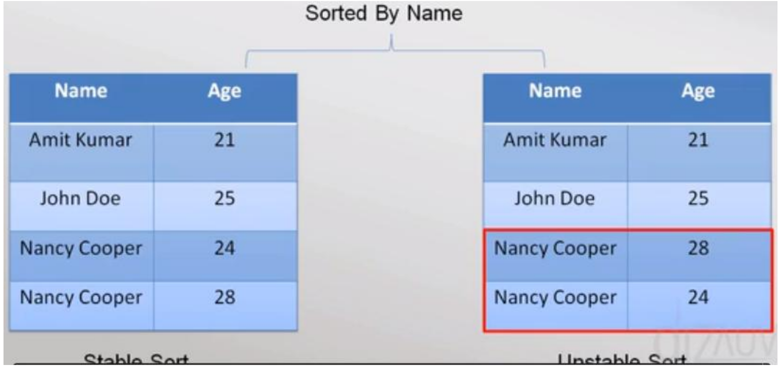


Unsorted Array

Name	Age
John Doe	25
Nancy Cooper	24
Amit Kumar	21
Nancy Cooper	28

Sorted By Age

Name	Age
Amit Kumar	21
Nancy Cooper	24
John Doe	25
Nancy Cooper	28



Sorted By Name

Name	Age
Amit Kumar	21
John Doe	25
Nancy Cooper	24
Nancy Cooper	28

Stable Sort

Name	Age
Amit Kumar	21
John Doe	25
Nancy Cooper	28
Nancy Cooper	24

Unstable Sort

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

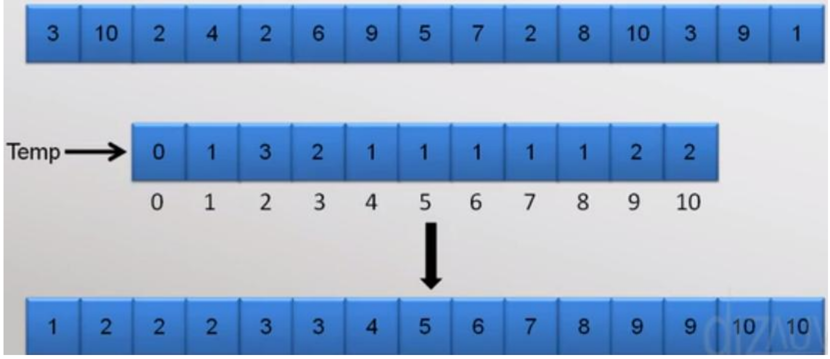
O(n log n) ➔ merge sort

O(n) ➔ (Sorting in linear time) ??

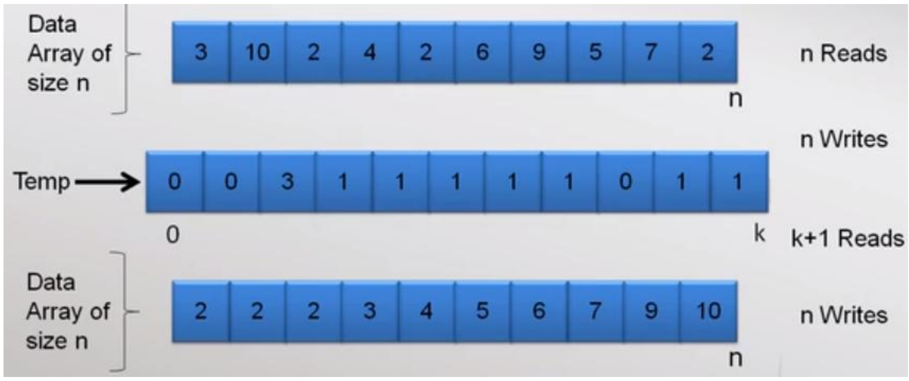
If we know some information about data to be sorted (e.g. students' marks -Range 50 to 99 –), we can achieve linear time sorting

Counting Sort:

Example: assume data range from 1 to 10

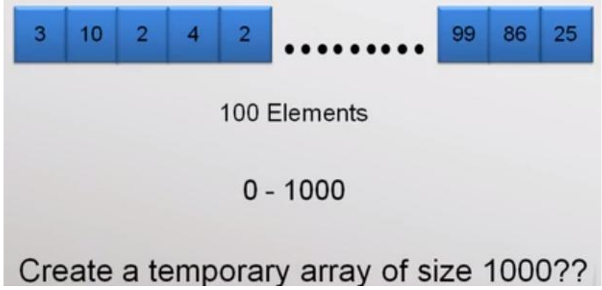


Time analysis:



Note: **K** is typically small comparing to **n**

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



Create a temporary array of size 1000??

Is counting sort is In-Place or Not-In-Place ?? why?

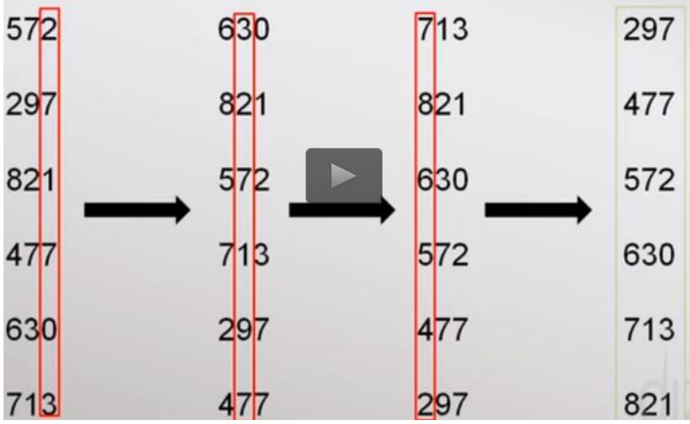
Radix Sort:

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

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

Example Radix Sort:

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



How to implement Radix Sort:

Radix Sort Algorithm using linked list:

Consider the following array

9	179	139	38	10	5	36

Create an array of linked lists as follow:

0
1
2
3
4
5
6
7
8
9

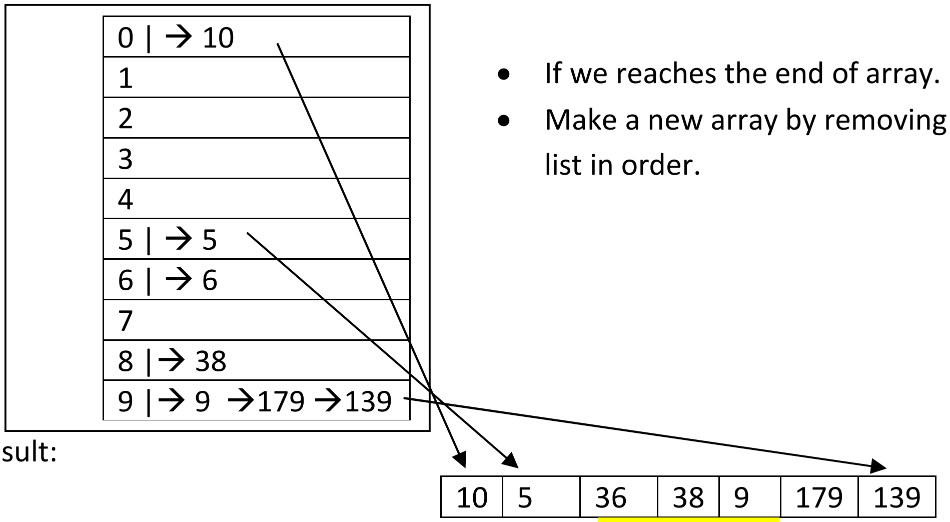
- Total of 10 linked lists
- 0 to 9 refer to actual numbers
- With input numbers, we will start with mod 10 then divide the resulted number by 1

Code:

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

e.g. Arr[0] = 9
9 % m = 9 ↘
9 / n = 9

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



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

Result:

10	5	36	38	9	179	139

Is this sorted?

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

Code:

m = m * 10 = 100
n = n * 10 = 10

e.g. Arr[0] = 10
10 % m = 10
↘ 10 / n = 1

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

Result:

5	9	10	36	38	139	179

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

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

Code:

m = m * 10 = 1000
n = n * 10 = 100

e.g. Arr[0] = 5
5 % m = 5
↘ 5 / n = 0

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

Result:

5	9	10	36	38	139	179

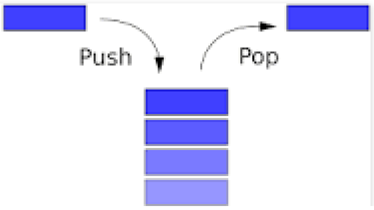
Is this sorted? What is the time complexity

HW: implement Radix sort using Doubly Linked List

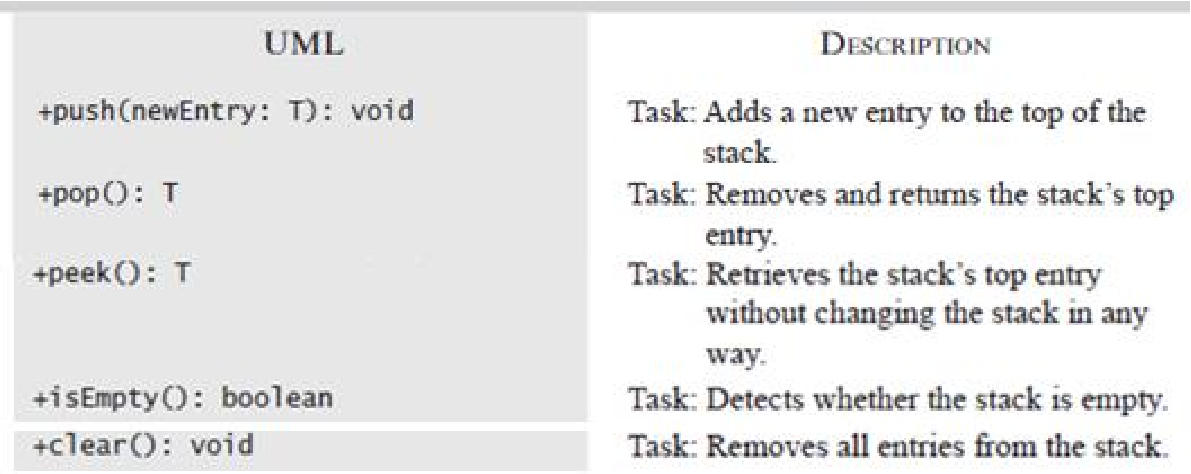
(Lecture 10) Stacks 1

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

- **push** adds an element to the collection;
- **pop** removes the last element that was added.



- **Last In, First Out ➔ LIFO**



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

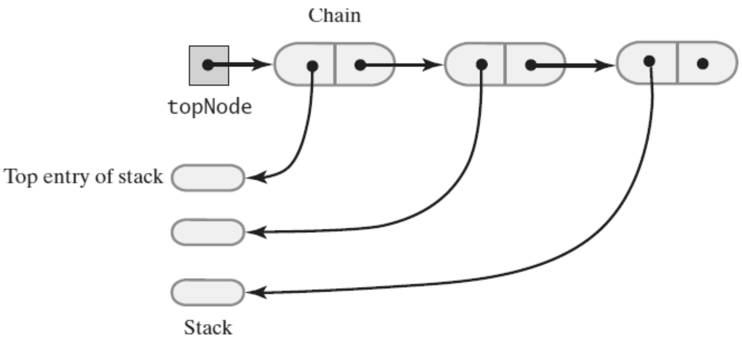
Linked Implementation:

Each of the following operation involves top of stack

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

Head or Tail for topNode??

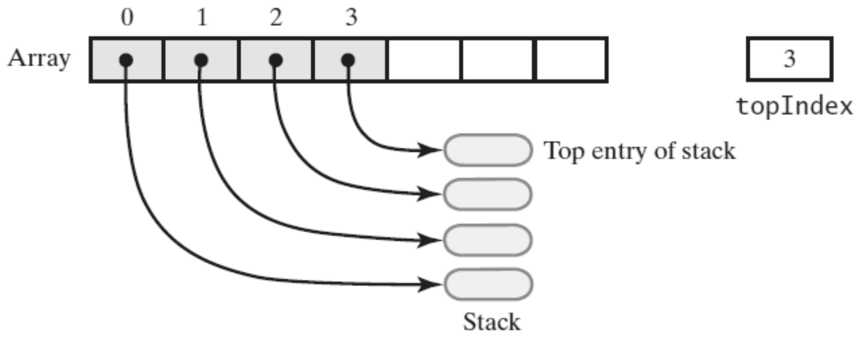
Head of linked list easiest, fastest to access ➔ Let this be the top of the stack



```
public class LinkedStack<T> {

        private Node<T> topNode;
        public void push(T data) {
            Node<T> newNode = new Node<T>(data);
            newNode.setNextNode(topNode);
            topNode = newNode;
        }
        public Node<T> pop() {
            Node<T> toDel = topNode;
            assert topNode!=null : "Empty Stack" ;
            topNode = topNode.getNextNode();
            return toDel;
        }

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

Array-Based Implementation

- End of the array easiest to access
    - Let this be top of stack
    - Let first entry be bottom of stack

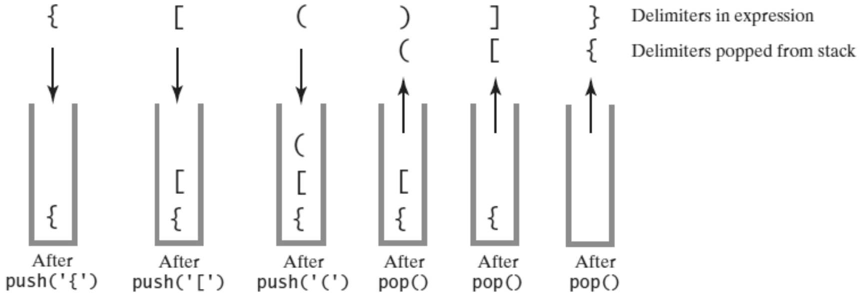


H.W. implement array based stack

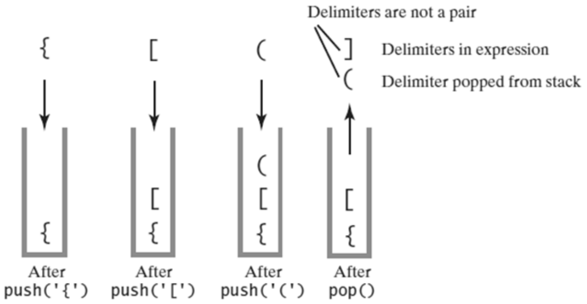
Balanced Expressions

Delimiters paired correctly ➔ compilers

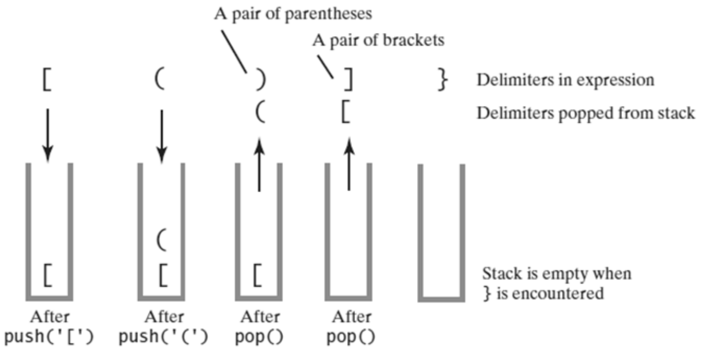
Example 1: The contents of a stack during the scan of an expression that contains the **balanced delimiters**
{ [( )] }



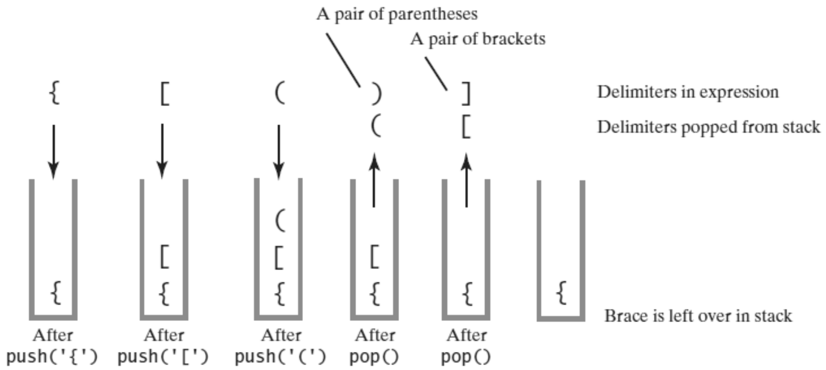
Example 2: The contents of a stack during the scan of an expression that contains the **unbalanced delimiters**
{ [( ]) }



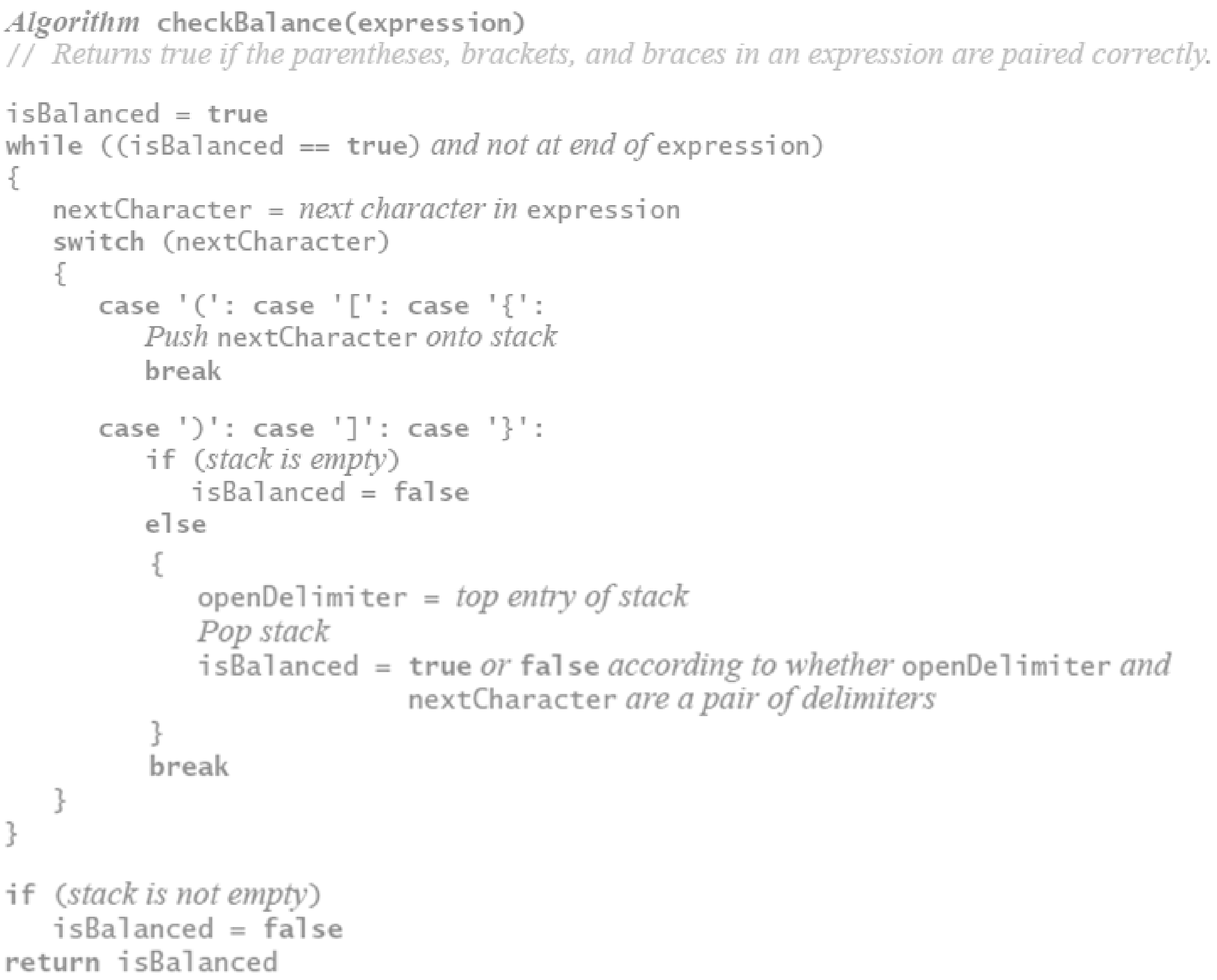
Example 3: The contents of a stack during the scan of an expression that contains the **unbalanced delimiters**
[( )] }



Example 4: The contents of a stack during the scan of an expression that contains the **unbalanced delimiters**
{ [( )]



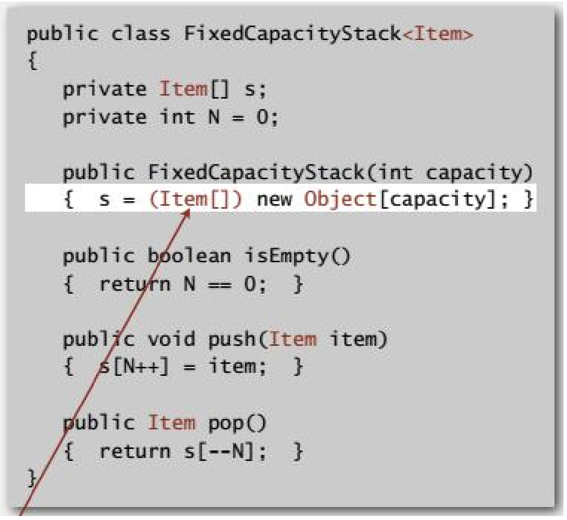
Algorithm to process for balanced expression:



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

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

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

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

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

Generic stack: array implementation



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

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

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

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

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

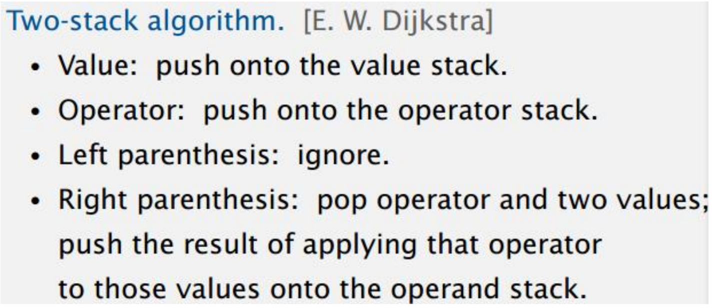
(Lecture 11) Stacks 2

Processing Algebraic Expressions

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

Arithmetic expression evaluation

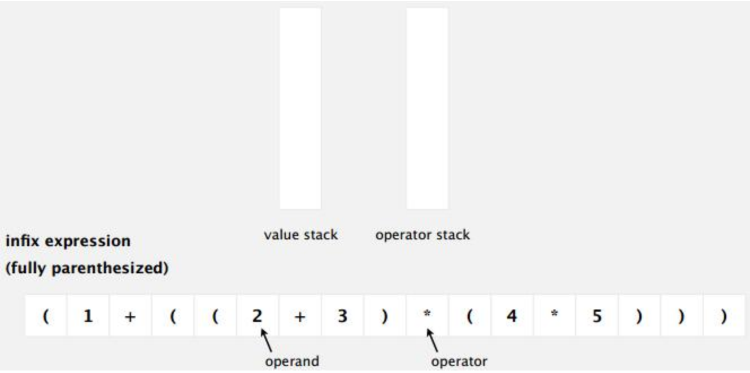
Evaluate infix expressions.

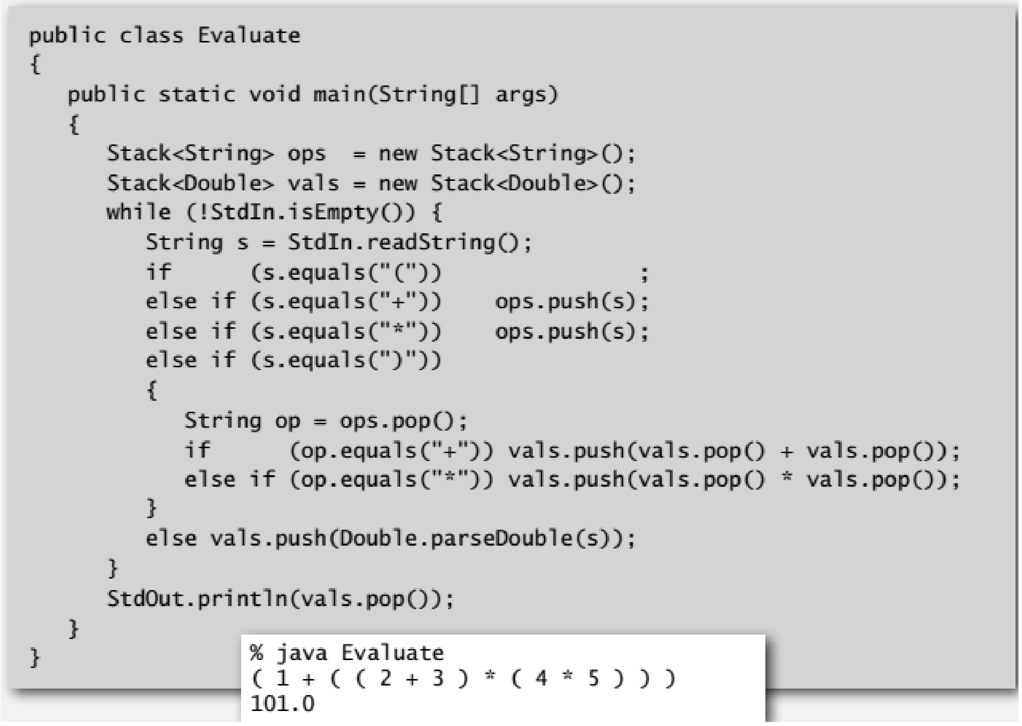


Two-stack algorithm. [E. W. Dijkstra]

- Value: push onto the value stack.
- Operator: push onto the operator stack.
- Left parenthesis: ignore.
- Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

Example:



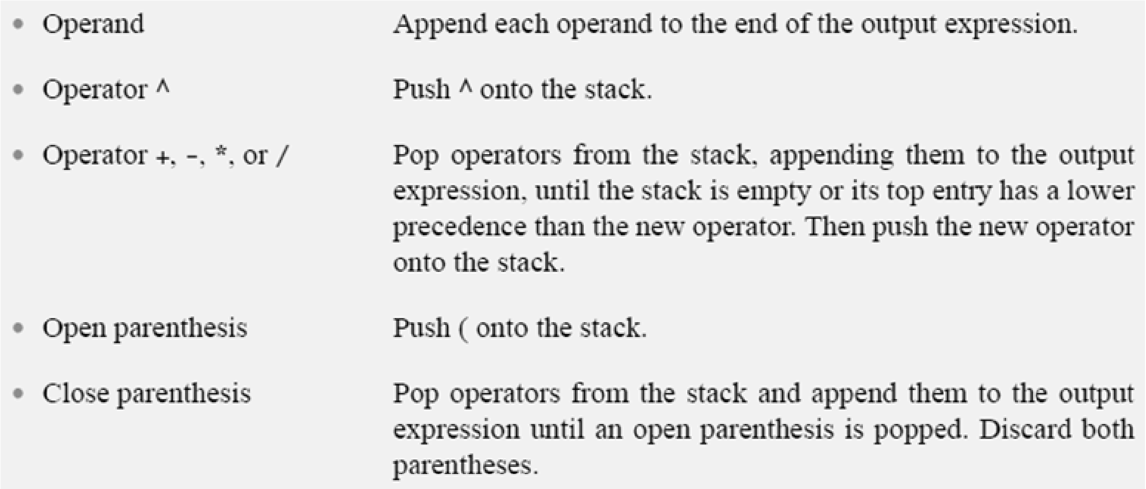


```
public class Evaluate
{
   public static void main(String[] args)
   {
      Stack<String> ops  = new Stack<String>();
      Stack<Double> vals = new Stack<Double>();
      while (!StdIn.isEmpty()) {
         String s = StdIn.readString();
         if      (s.equals("("))               ;
       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

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

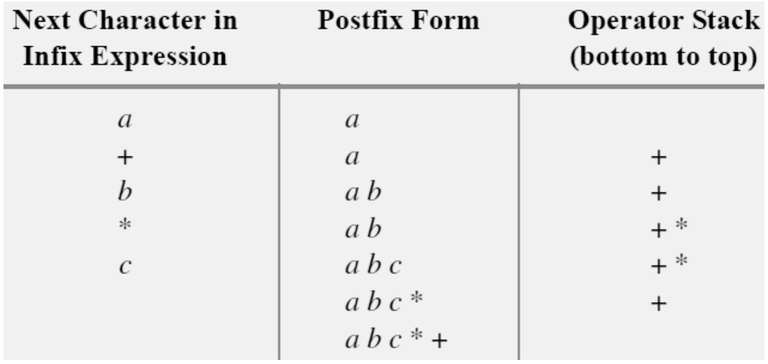
Infix to Postfix

Infix-to-postfix Conversion:



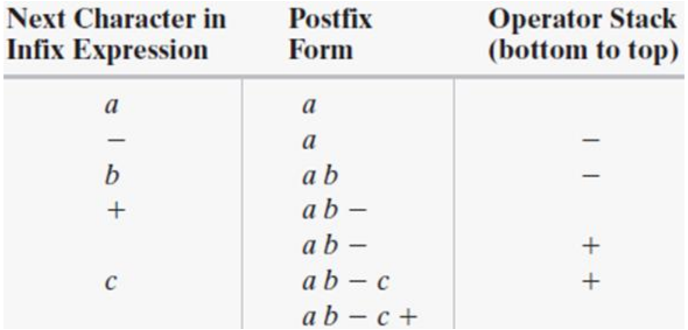
• Operand	Append each operand to the end of the output expression.
• Operator ^	Push ^ onto the stack.
• Operator +, -, *, or /	Pop operators from the stack, appending them to the output expression, until the stack is empty or its top entry has a lower precedence than the new operator. Then push the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

Example 1: Converting the **infix** expression **a + b * c** to **postfix** form



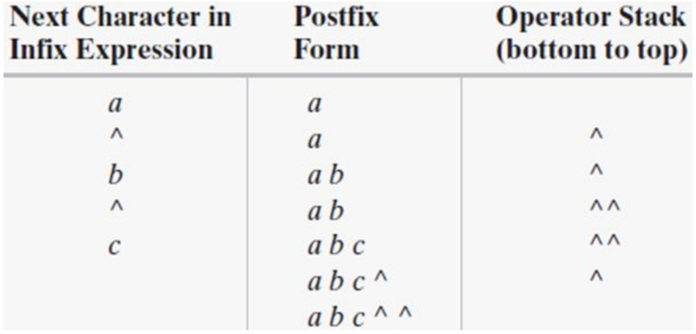
Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
+	*a*	+
b	*a b*	+
*	*a b*	+ *
c	*a b c*	+ *
	a b c *	+
	a b c * +	

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



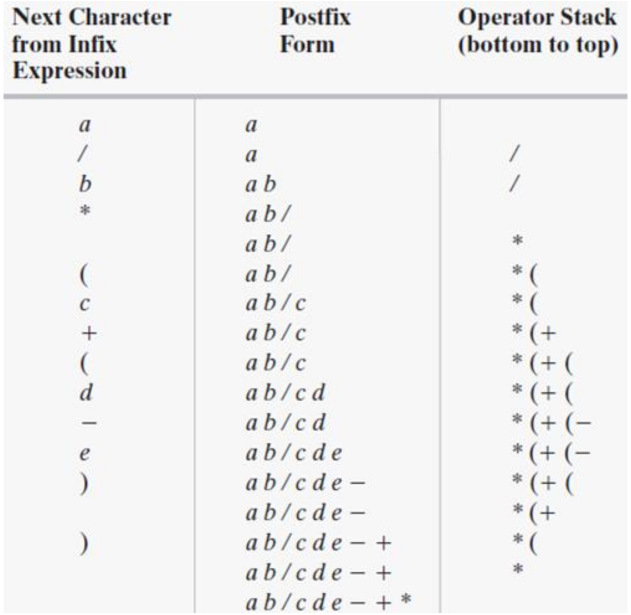
Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	a	
−	a	−
b	a b	−
+	a b −	
	a b −	+
c	a b − c	+
	a b − c +	

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



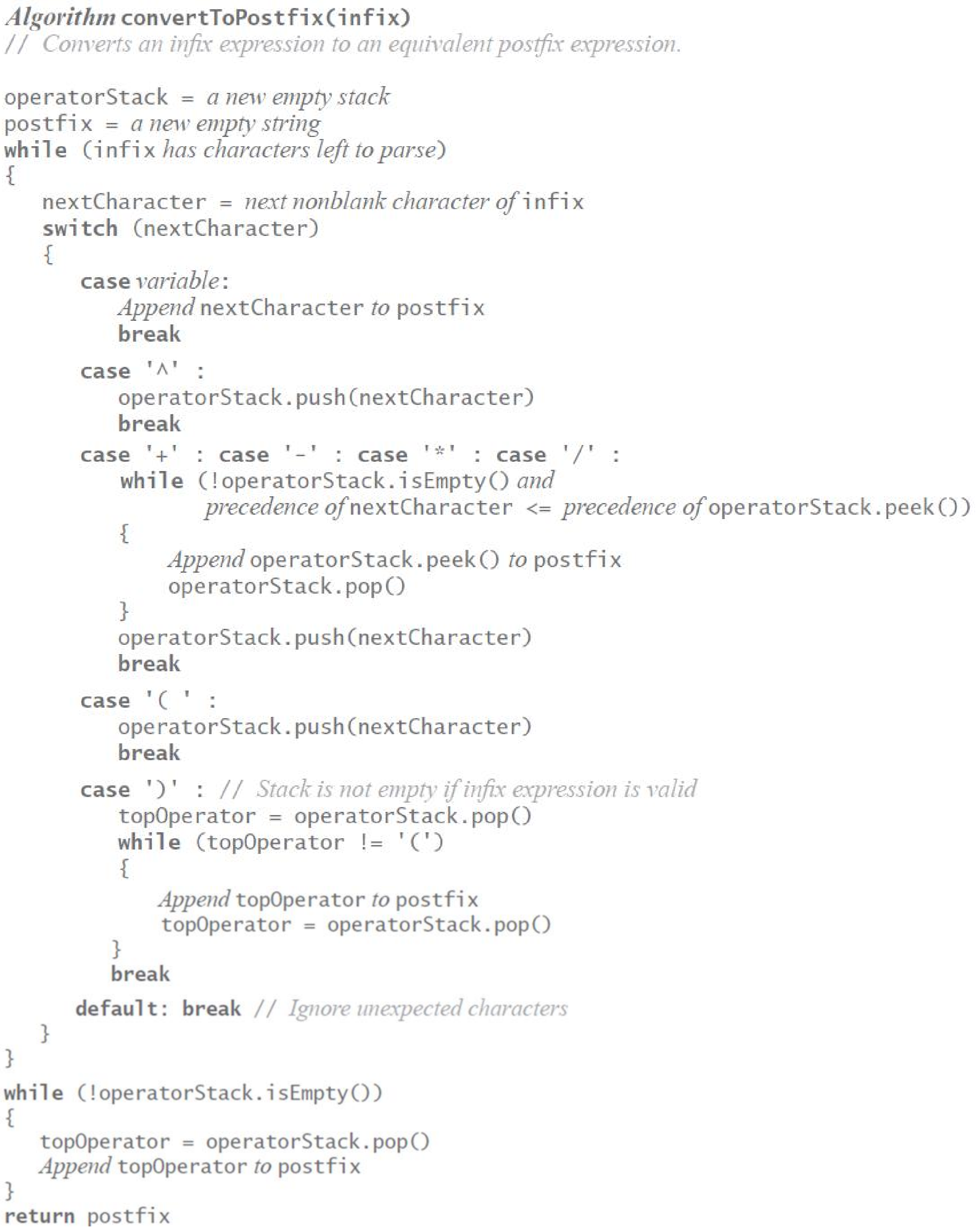
Next Character in Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	a	
^	a	^
b	a b	^
^	a b	^^
c	a b c	^^
	a b c ^	^
	a b c ^ ^	

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



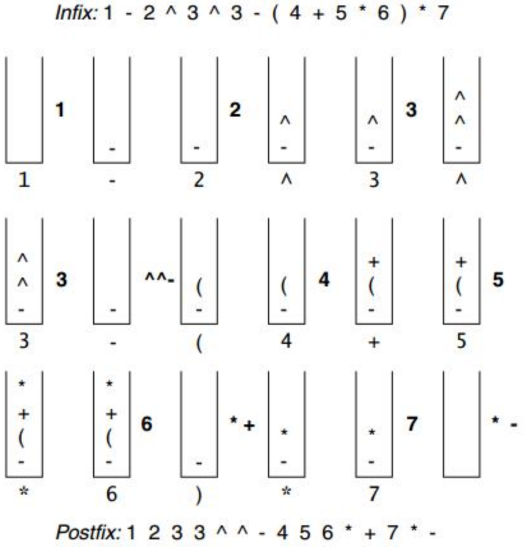
Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	a	
/	a	/
b	a b	/
*	a b /	
	a b /	*
(	a b /	* (
c	a b / c	* (
+	a b / c	* (+
(	a b / c	* (+ (
d	a b / c d	* (+ (
−	a b / c d	* (+ (−
e	a b / c d e	* (+ (−
)	a b / c d e −	* (+ (
	a b / c d e −	* (+
)	a b / c d e − +	* (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

Infix-to-postfix Algorithm



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

operatorStack = a new empty stack
postfix = a new empty string
while (infix has characters left to parse)
{
   nextCharacter = next nonblank character of infix
   switch (nextCharacter)
   {
      case variable:
         Append nextCharacter to postfix
         break
      case '^' :
         operatorStack.push(nextCharacter)
         break
      case '+' : case '-' : case '*' : case '/' :
         while (!operatorStack.isEmpty() and
                  precedence of nextCharacter <= precedence of operatorStack.peek())
         {
             Append operatorStack.peek() to postfix
             operatorStack.pop()
         }
         operatorStack.push(nextCharacter)
         break
      case '( ' :
         operatorStack.push(nextCharacter)
         break
      case ')' : // Stack is not empty if infix expression is valid
         topOperator = operatorStack.pop()
         while (topOperator != '(')
         {
            Append topOperator to postfix
            topOperator = operatorStack.pop()
         }
         break
      default: break // Ignore unexpected characters
   }
}
while (!operatorStack.isEmpty())
{
   topOperator = operatorStack.pop()
   Append topOperator to postfix
}
return postfix
```

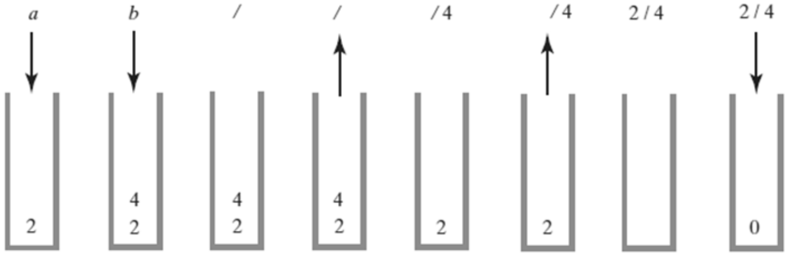
H.W. example:



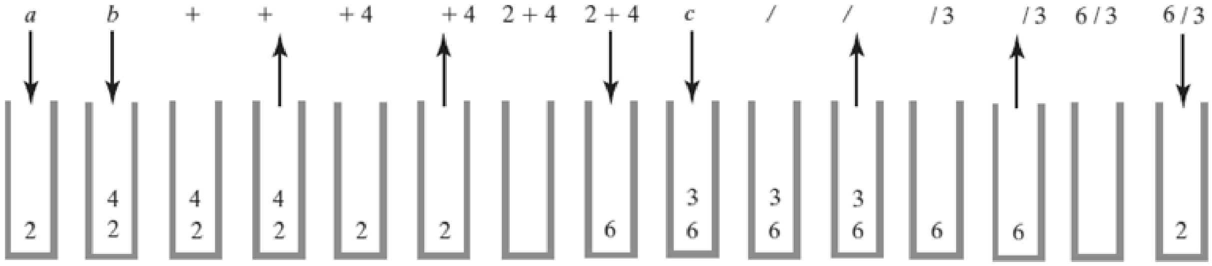
Evaluating Postfix Expressions

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

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



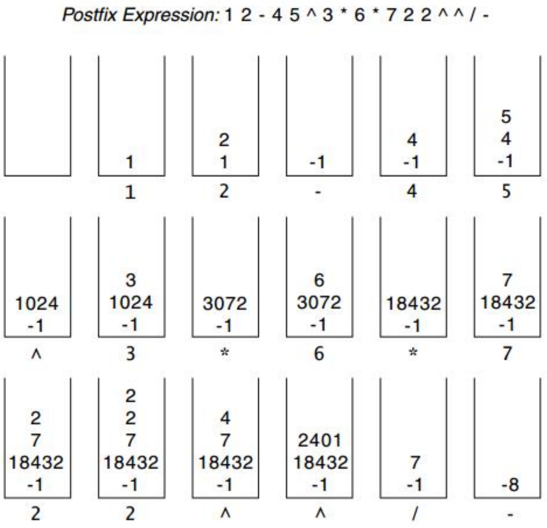
Example 2: The stack during the evaluation of the postfix expression **a b + c /** when ***a*** is ***2***, ***b*** is ***4***, and ***c*** is ***3***



Algorithm for evaluating postfix expressions.

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

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

H.W. Example:



Iteration (optional)

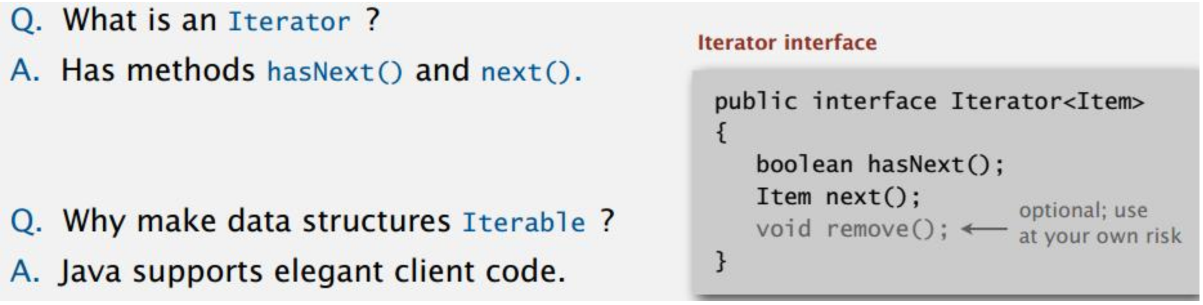
- **Design challenge**. Support iteration over stack items by client, without revealing the internal representation of the stack.
- **Java solution**. Make stack implement the **java.lang.Iterable** interface.

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()`.

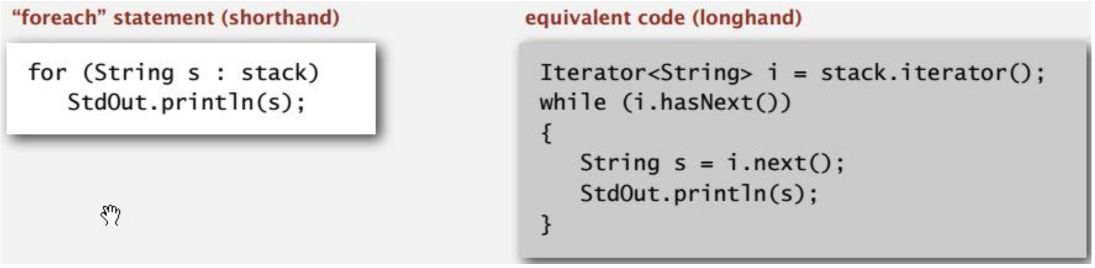
Q. Why make data structures `Iterable` ?

A. Java supports elegant client code.

Iterator interface

```
public interface Iterator<Item>
{
   boolean hasNext();
   Item next();
   void remove();  ←
}
```

optional; use at your own risk

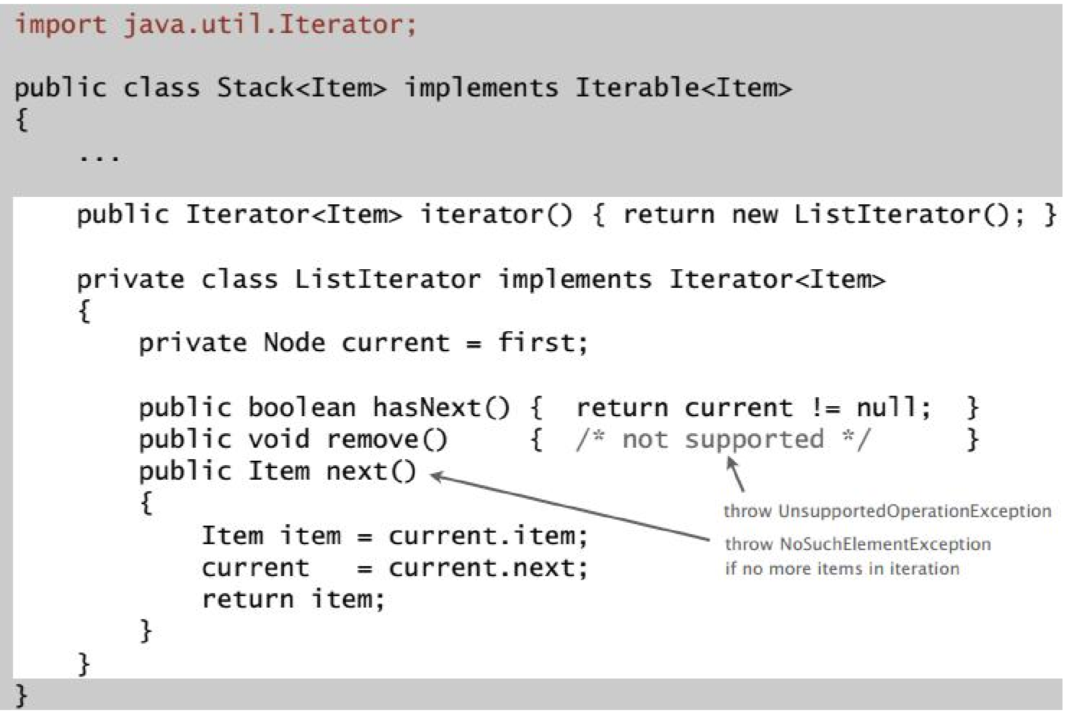


"foreach" statement (shorthand)

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

equivalent code (longhand)

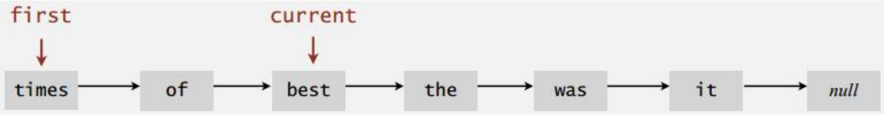
```
Iterator<String> i = stack.iterator();
while (i.hasNext())
{
   String s = i.next();
   StdOut.println(s);
}
```



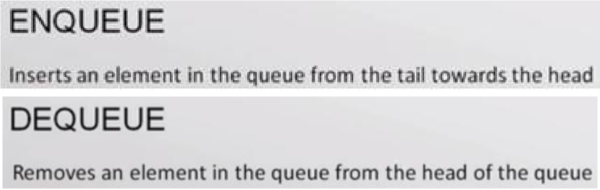
```
import java.util.Iterator;

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

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

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

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

throw UnsupportedOperationException

throw NoSuchElementException if no more items in iteration



(Lecture 12) Queues

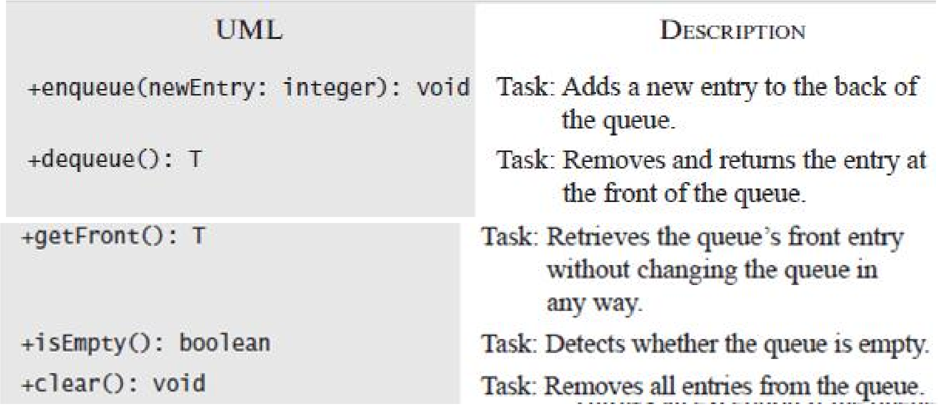


ENQUEUE

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

DEQUEUE

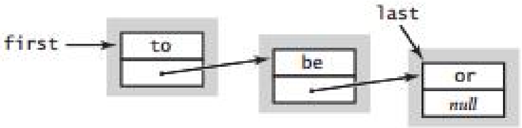
Removes an element in the queue from the head of the queue



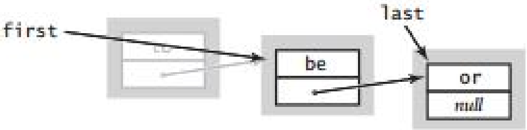
UML	DESCRIPTION
+enqueue(newEntry: integer): void	Task: Adds a new entry to the back of the queue.
+dequeue(): T	Task: Removes and returns the entry at the front of the queue.
+getFront(): T	Task: Retrieves the queue's front entry without changing the queue in any way.
+isEmpty(): boolean	Task: Detects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

Linked-list Representation of a Queue

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



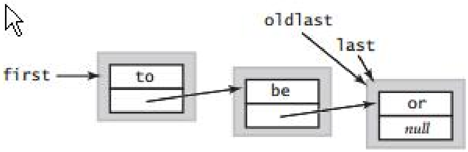
Delete dequeue:



Add enqueue:

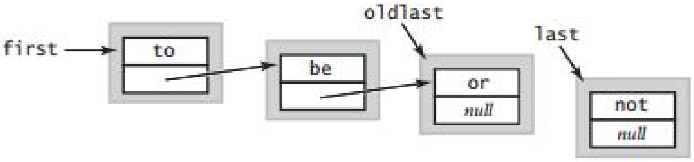
save a link to the last node

```
Node oldlast = last;
```



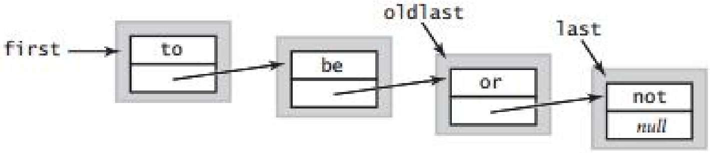
create a new node for the end

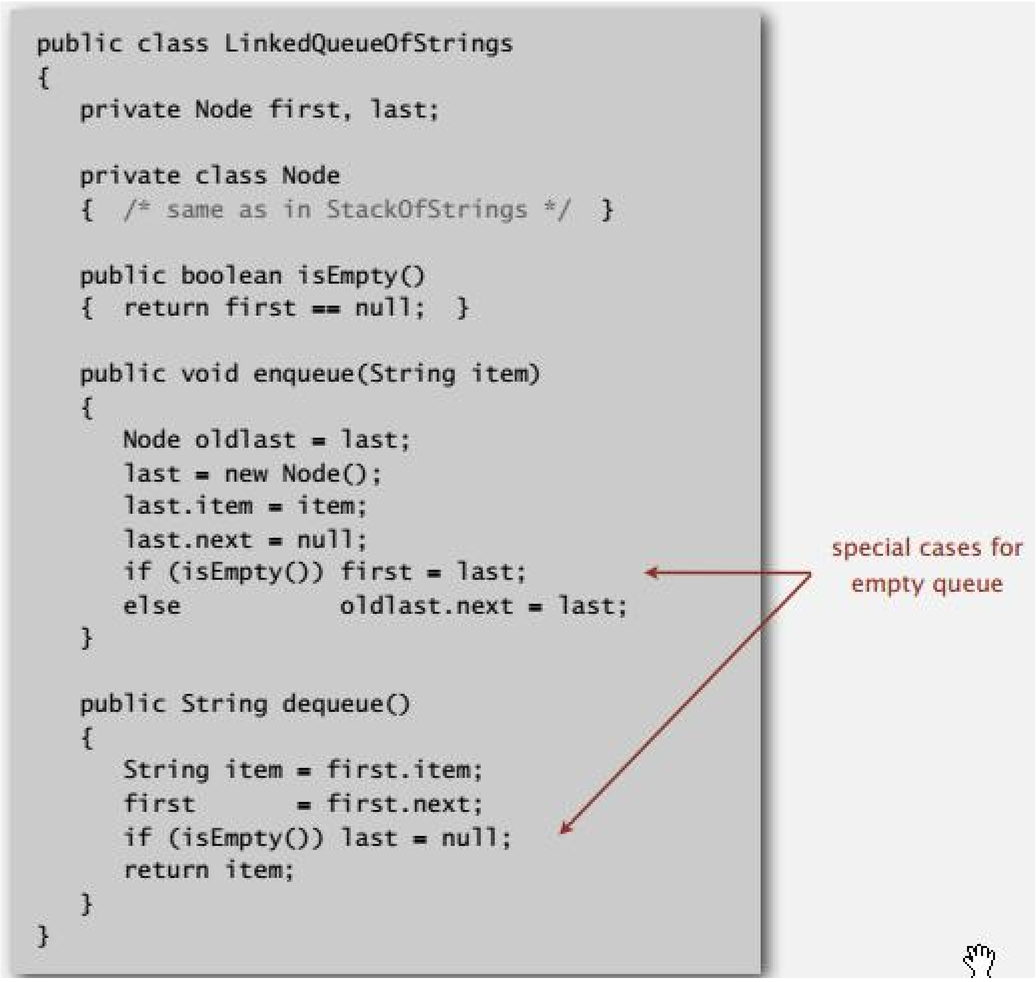
```
last = new Node();
last.item = "not";
```



link the new node to the end of the list

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

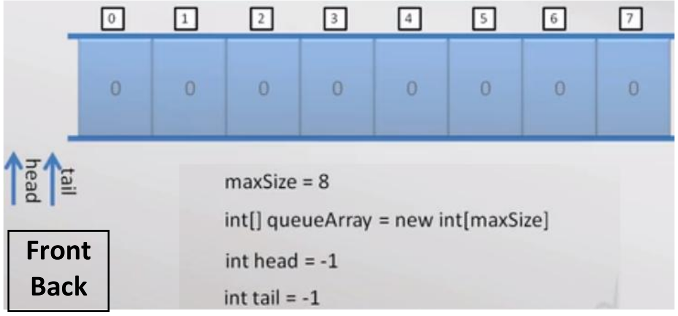




```
public class LinkedQueueOfStrings
{
   private Node first, last;

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

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

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

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

special cases for empty queue

Array implementation of a Queue.

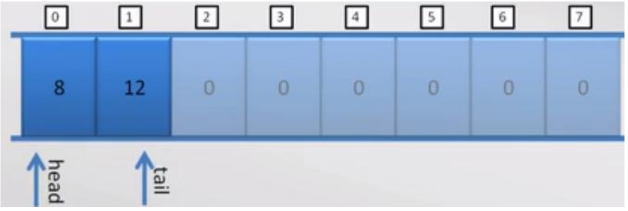


```
maxSize = 8
int[] queueArray = new int[maxSize]
int head = -1
int tail = -1
```

- **enqueue**(): add new item at q[tail] .
- **dequeue**(): remove item from q[head] .

enqueue(8)

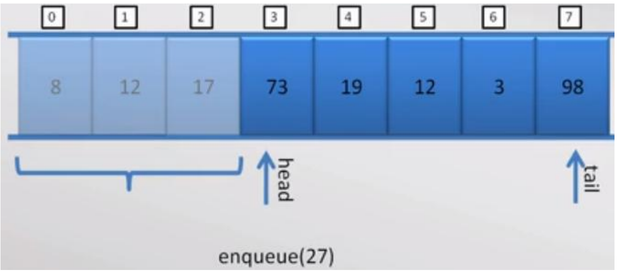
enqueue (12)



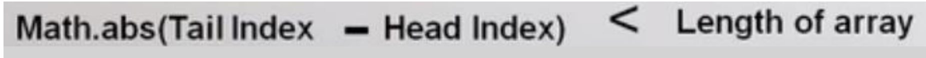
After a number of enqueues:

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

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

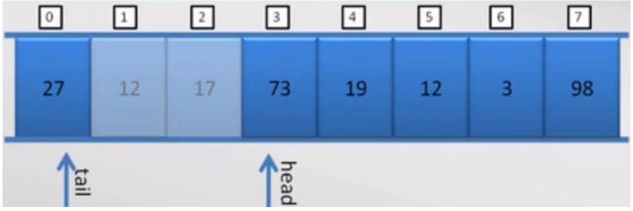


How to find free spaces??

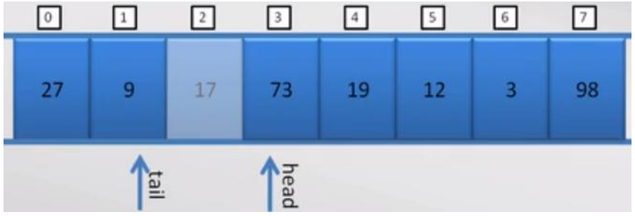


Math.abs(Tail Index − Head Index) < Length of array -1

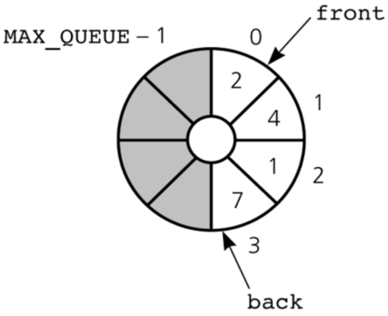
So, if tail at max index and we have free spaces, we move tail to 1st index. ➔ **Circular Queue**

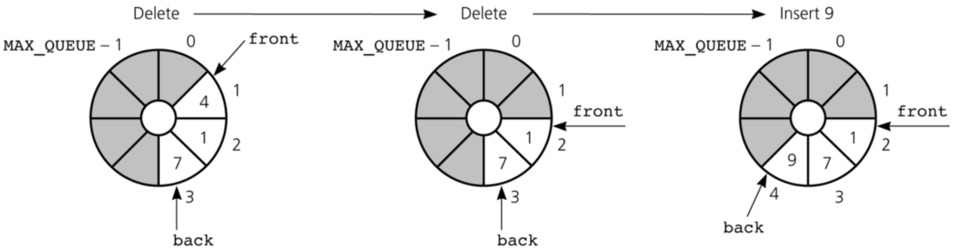


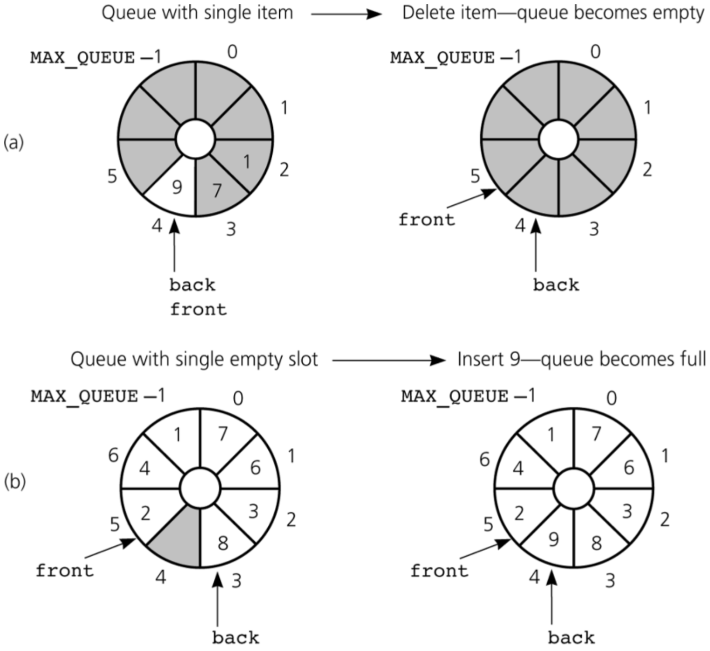
enqueue (9) ??



Circular Queue







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

Inserting into a queue

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

Deleting from a queue

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

DE Queue (Double Ended Queue)

Allows add / remove elements from both head/tail.

HW This of implementations using linked List and Arrays.

(Lecture 13) Cursor Implementation of Linked Lists

Many Languages do not support pointers.

If data max length is known, using Array is faster

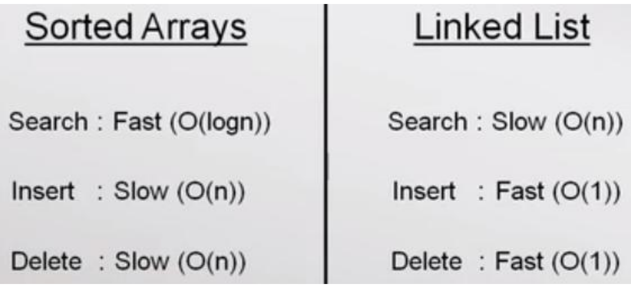
Solution ➔ Cursor Implementation

2 features present in a pointer implementation of linked lists:

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

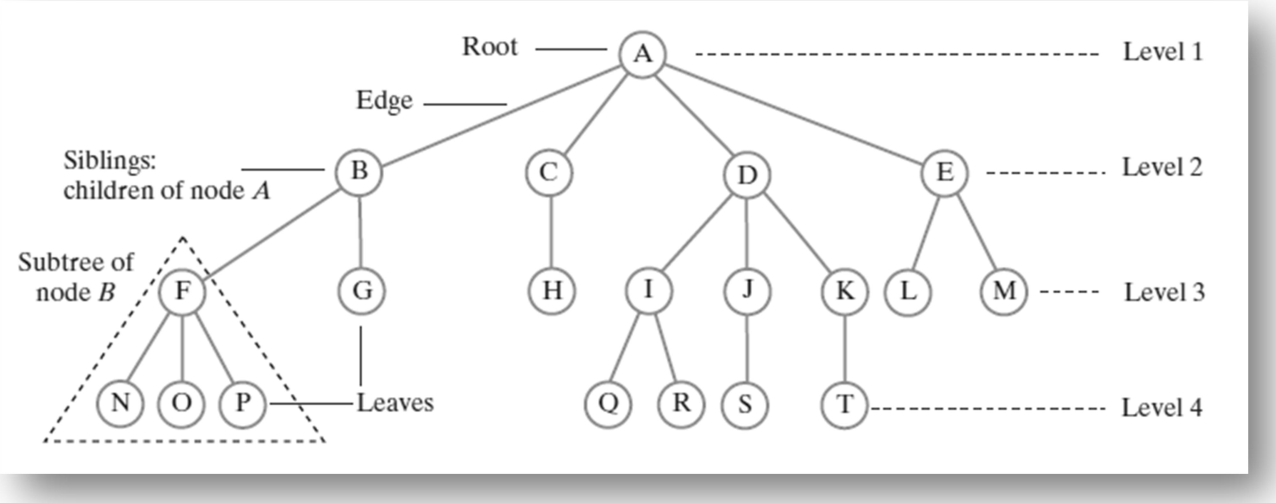
To Be Completed

(Lecture 14) Trees



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

Tree



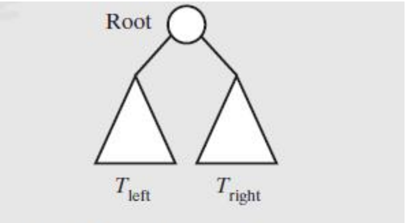
- A **tree** is a collection of *N* **nodes**, one of which is the **root**, and *N* − 1 **edges**.
- Every node except the **root** has one **parent**.
- Nodes with no children are known as **leaves**.
- An **internal node (parent)** is any node that has at least one non-empty child.
- Nodes with the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

Example: Family Trees (one parent)

Example: file system tree

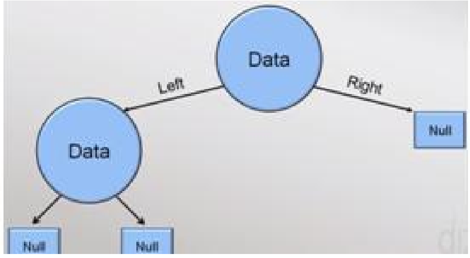
Binary Trees

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

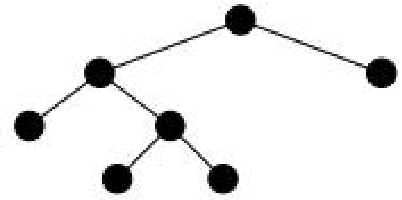


where T_{left} and T_{right} are binary trees.

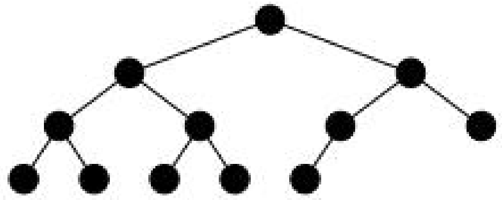
Binary Tree Node:



- Each node in a **full binary tree** is either:
  (1) an internal node with exactly two non-empty children or
  (2) a leaf.
- A **complete binary tree** has a restricted shape obtained by starting at the root and filling the tree by levels from **left** to **right**.

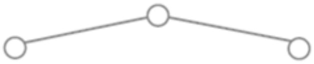


(a) This tree is full (but not complete).



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

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

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

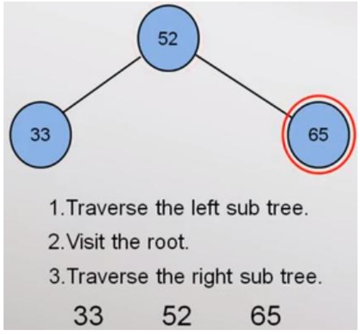
Implementation:

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

Tree Traversal

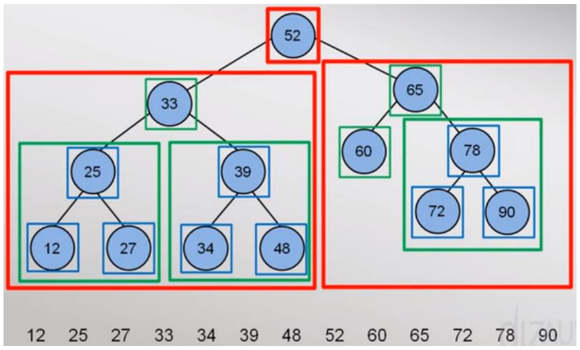
Definition: visit, or process, each data item exactly once.

In-Order Traversal:



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

33 52 65



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

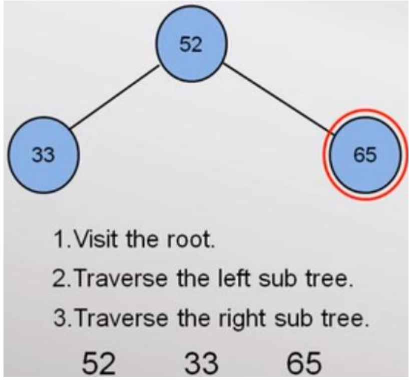
@ TreeNode

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

@BinarySerachTree

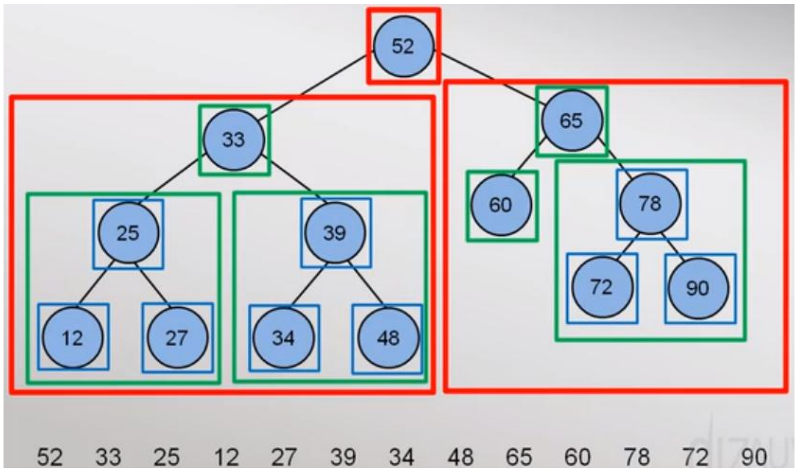
```
public void traverseInOrder() {
        if (this.root != null)
                this.root.traverseInOrder();
        System.out.println();
}
```

Pre-Order Traversal



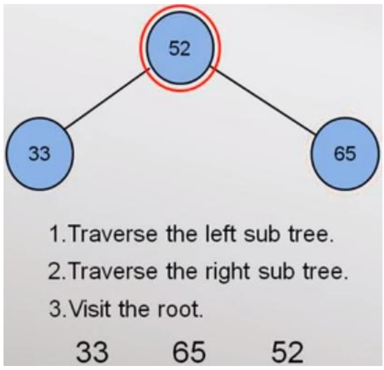
1. Visit the root.
2. Traverse the left sub tree.
3. Traverse the right sub tree.

52 33 65



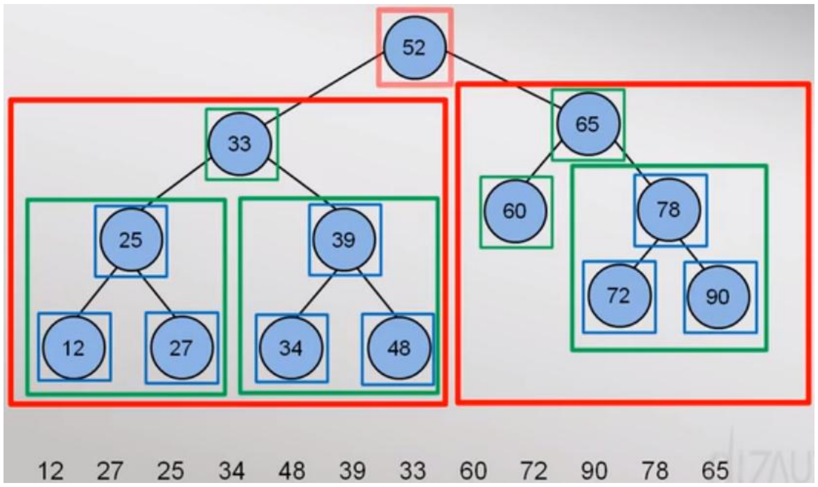
52 33 25 12 27 39 34 48 65 60 78 72 90

Post-Order Traversal



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

33 65 52



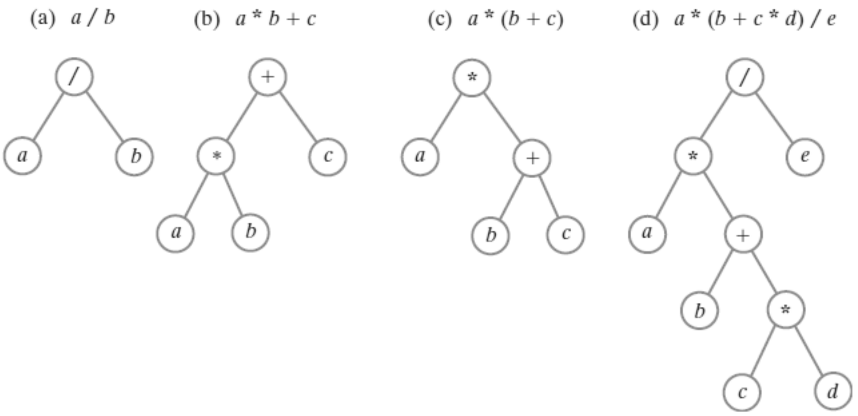
12 27 25 34 48 39 33 60 72 90 78 65

Level-Order Traversal (Optional)

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

HW: implement level-order traversal

(Lecture 15) Expression Trees

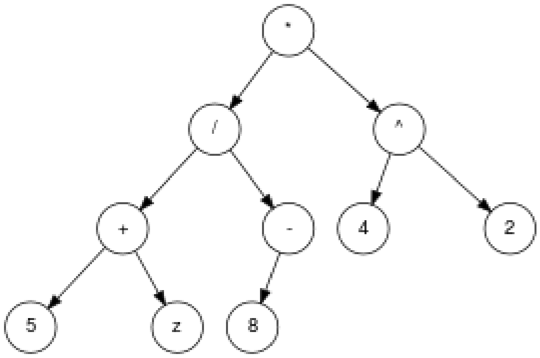

(a) a / b (b) $a * b + c$ (c) $a * (b + c)$ (d) $a * (b + c * d) / e$

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

Algebraic expressions:

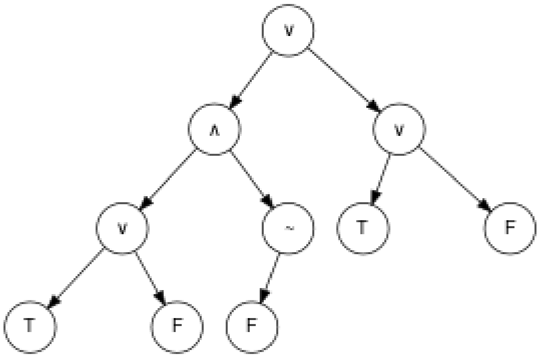
- Algebraic expression trees represent expressions that contain numbers, variables, and unary and binary operators.
- Some of the common operators are **×** (multiplication), **÷** (division), **+** (addition), **−** (subtraction), **^** (exponentiation), and **-** (negation).

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

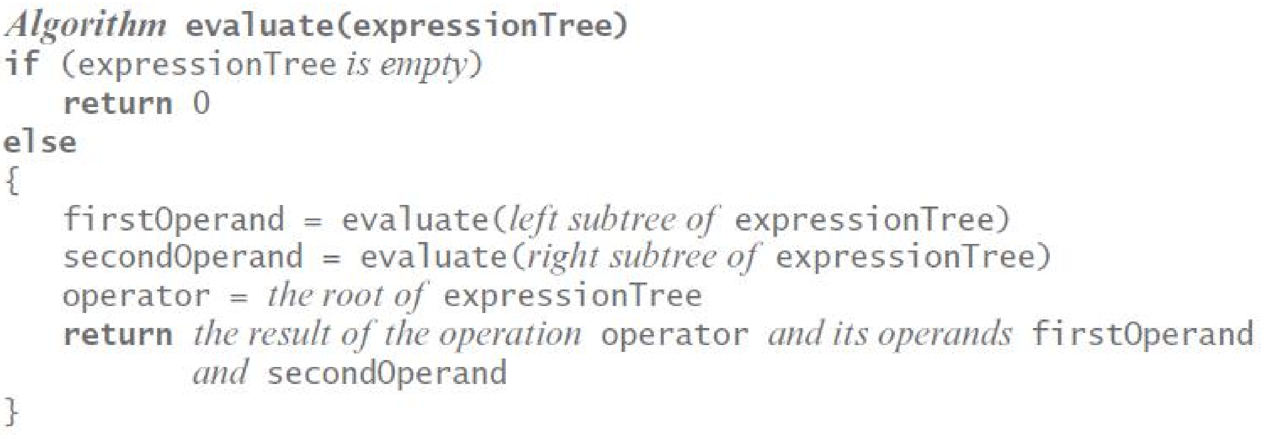


Boolean expressions:

- Boolean expressions are represented very similarly to algebraic expressions, the only difference being the specific values and operators used.
- Boolean expressions use **true** and **false** as constant values, and the operators include **∧ (AND), V (OR), ~ (NOT).**



Algorithm for evaluation of an expression tree:



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

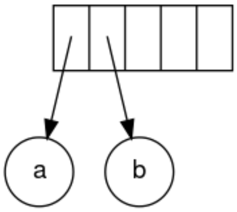
Constructing an expression tree:

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

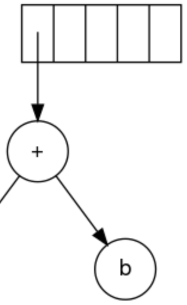
- If the symbol is an **operand**, one-node tree is created and a pointer is pushed onto a **stack**.
- If the symbol is an **operator**,
  - Two pointers trees T1 and T2 are popped from the stack
  - A new tree whose root is the **operator** and whose left and right children point to T2 and T1 respectively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

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

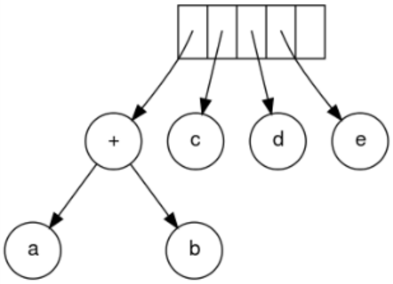
- Since the first two symbols are operands, one-node trees are created and pointers are pushed to them onto a stack.



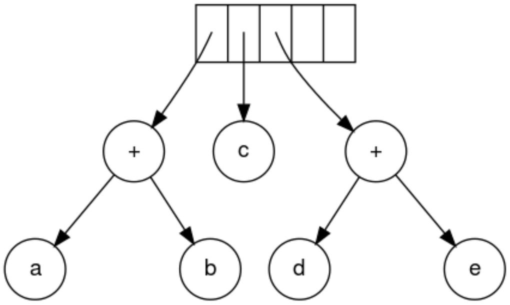
- The next symbol is a '**+**'. It pops two pointers, a new tree is formed, and a pointer to it is pushed onto to the stack.



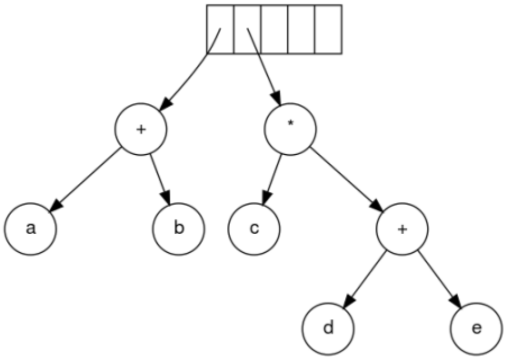
- Next, **c**, **d**, and **e** are read. A one-node tree is created for each and a pointer to the corresponding tree is pushed onto the stack.



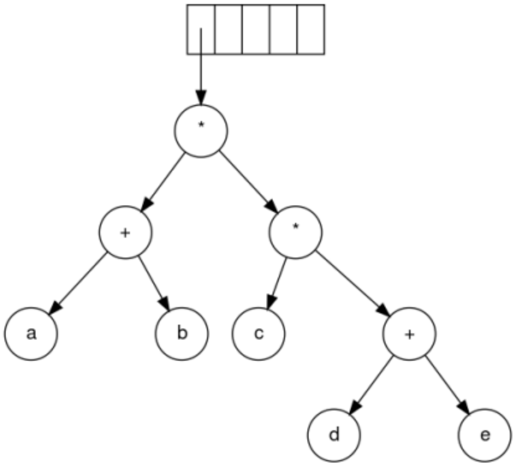
- Continuing, a '**+**' is read, and it merges the last two trees.



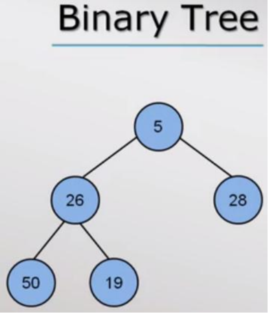
- Now, a '*****' is read. The last two tree pointers are popped and a new tree is formed with a '*****' as the root.

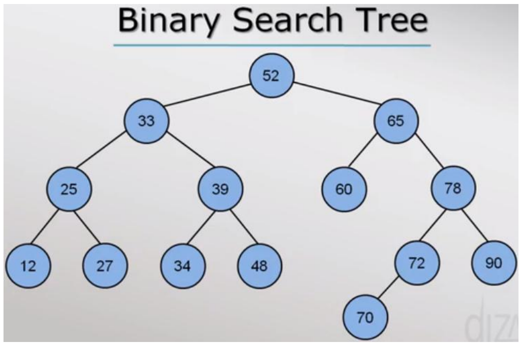


- Finally, the last symbol is read. The two trees are merged and a pointer to the final tree remains on the stack.

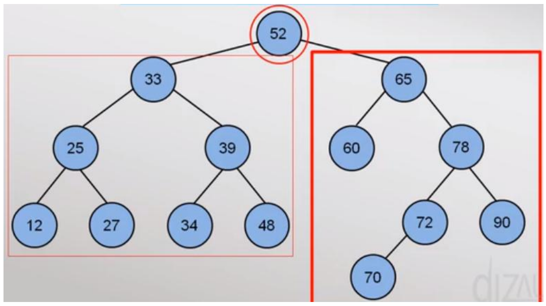


(Lecture 16) Binary Search Trees BST





- In a **binary search tree** for every node , **X**, in the tree, the values of all the items in its **left subtree** are smaller than the item in **X**, and the values of all the items in its **right subtree** are larger (**or equal**) than the item in **X**.



Search for an item: **Find(52) , Find(39) , Find(35)**

@ TreeNode

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

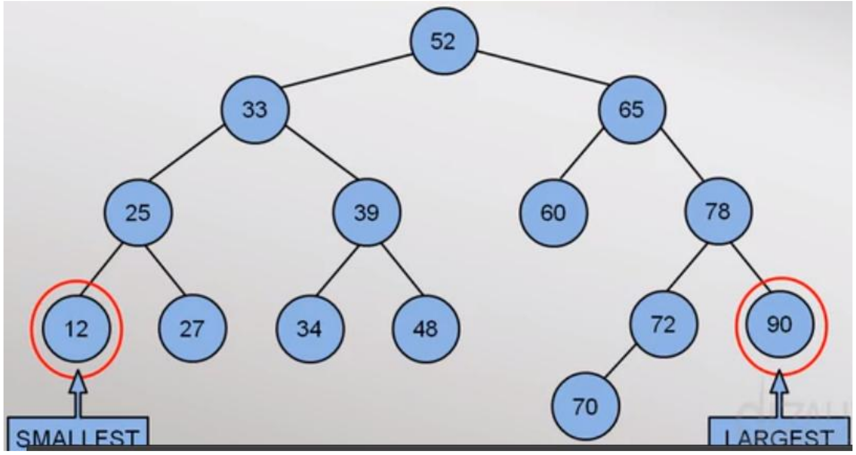
@BinarySerachTree

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

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

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

Finding Max and Min Values



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

@TreeNode

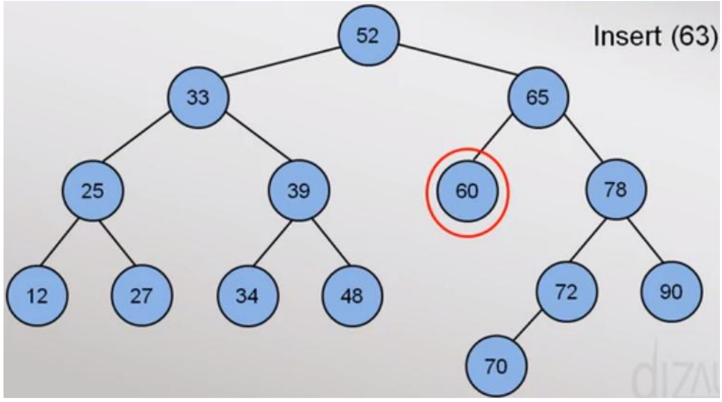
```
	public Integer largest() {
		if (this.rightChild == null)
			return this.data;
		return this.rightChild.largest();
	}

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

@BinarySerachTree

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

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

Insert in Binary Search Tree

Insert(63)



@TreeNode

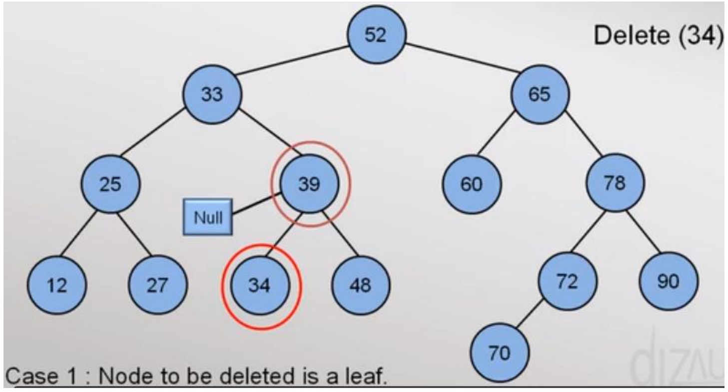
```
	public void insert(Integer data) {
		if (data >= this.data) {		// insert in right subtree
			if (this.rightChild == null)
				this.rightChild = new TreeNode(data);
			else
				this.rightChild.insert(data);
		} else {	// insert in left subtree
			if (this.leftChild == null)
				this.leftChild = new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

@BinarySerachTree

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

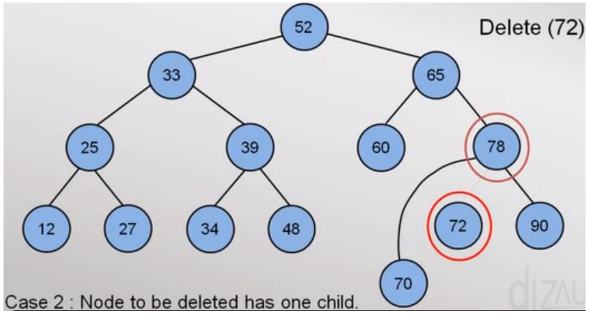
Deleting a Node

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



@BinarySerachTree

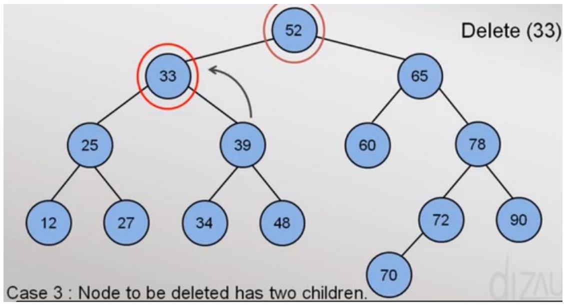
```
public void delete(Integer data) {
        TreeNode current = this.root;
        TreeNode parent = this.root;
        boolean isLeftChild = false;

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

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

        // this is case 1
        if (current.getLeftChild() == null && current.getRightChild() == null) {
                if (current == root) {   root = null;  // no elements in tree now
                } else {
                        if (isLeftChild)
                                parent.setLeftChild(null);
                        else
                                parent.setRightChild(null);
                }
        }
}
```

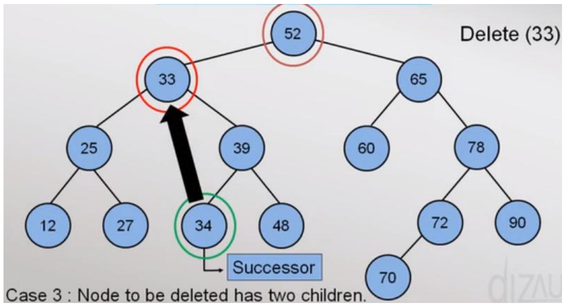


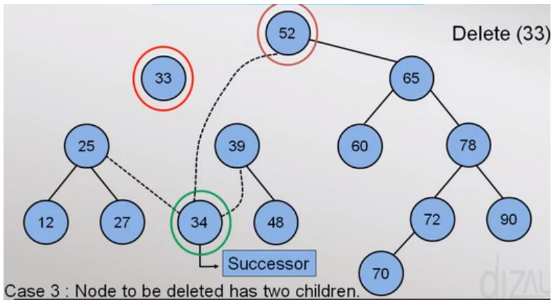
If a node has one child, it can be removed by having its parent bypass it.
Note: The **root** is a special case because it does not have a parent.

@BinarySerachTree

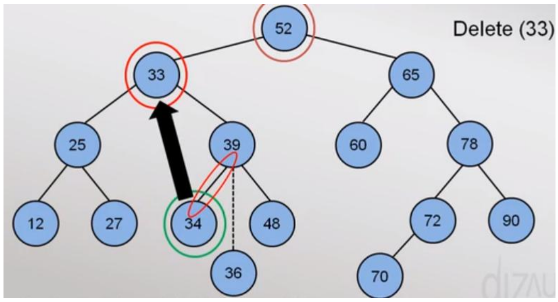
```
        // This is case 2 broken down further into 2 separate cases
        else if (current.getRightChild() == null) {      // current has left child
                if (current == root) {
                        root = current.getLeftChild();
                } else if (isLeftChild) {
                        parent.setLeftChild(current.getLeftChild());
                } else {
                        parent.setRightChild(current.getLeftChild());
                }
        } else if (current.getLeftChild() == null) {      // current has right child
                if (current == root) {
                        root = current.getRightChild();
                } else if (isLeftChild) {
                        parent.setLeftChild(current.getRightChild());
                } else {
                        parent.setRightChild(current.getRightChild());
                }
        }
```







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



What if **34** has a right child?

@BinarySerachTree

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


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

Soft Delete (lazy deletion): When an element is to be deleted, it is left in the tree and merely **marked** as being deleted.

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

Tree Height

@BinarySerachTree

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

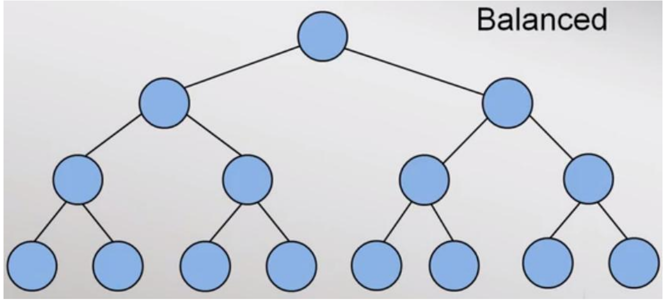
@TreeNode

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

Efficiency of Operations

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

Unbalanced Tree



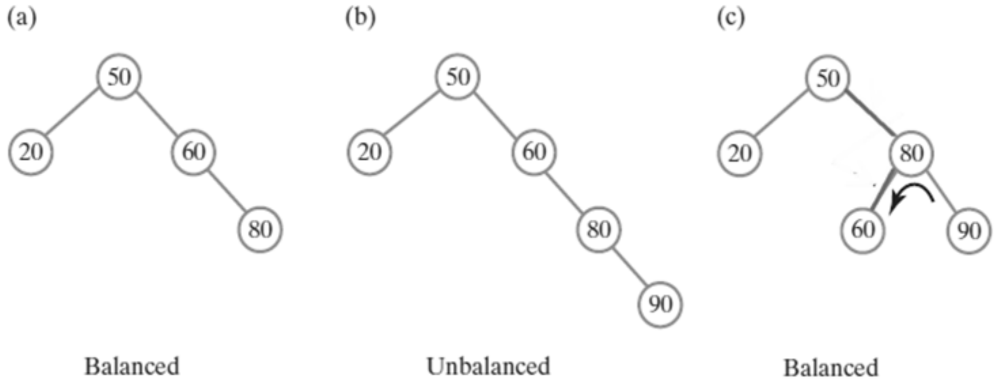
- The order in which you add entries to a binary search tree affects the shape of the tree.

(Lecture 17, 18) AVL Trees

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

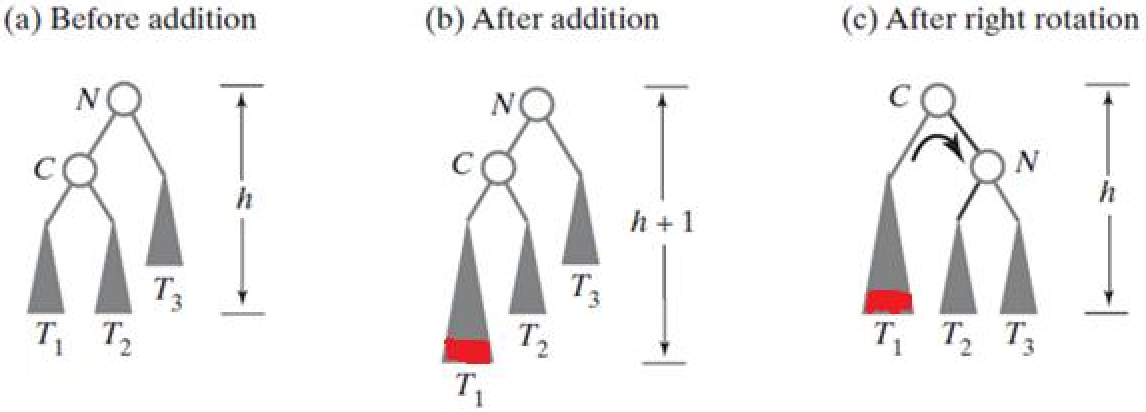
Single Rotations

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

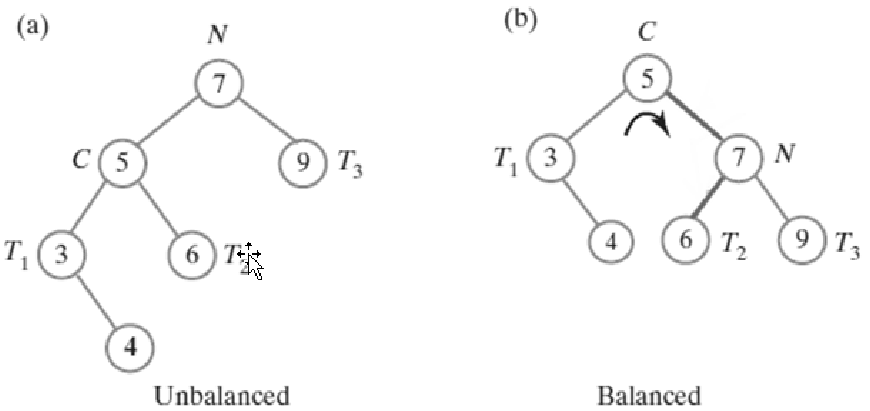


Example: (a) Adding 80 to the tree does not change the balance of the tree;
(b) a subsequent addition of 90 makes the tree **unbalanced** ;
(c) a left rotation restores its balance

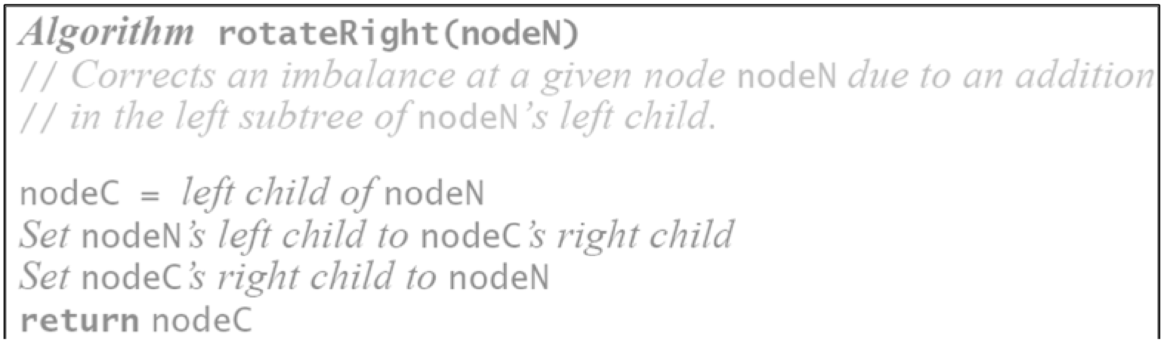
Case 1: Single Right Rotation



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

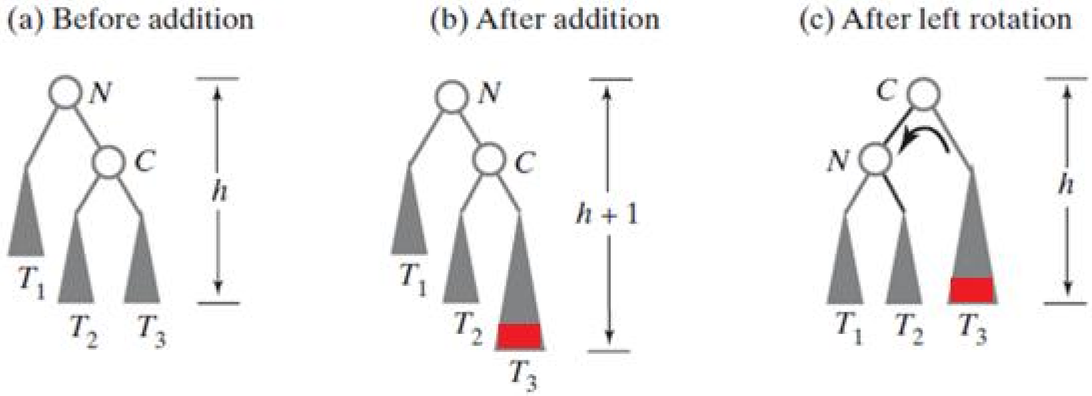


Example: Before and after a **right rotation** restores balance to an **AVL** tree

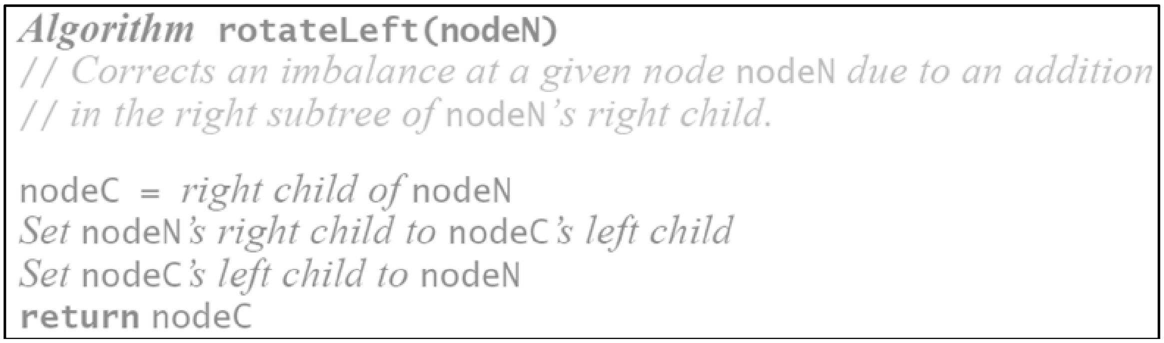


```
Algorithm rotateRight(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the left subtree of nodeN's left child.

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

Case 2: Single Left Rotation



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



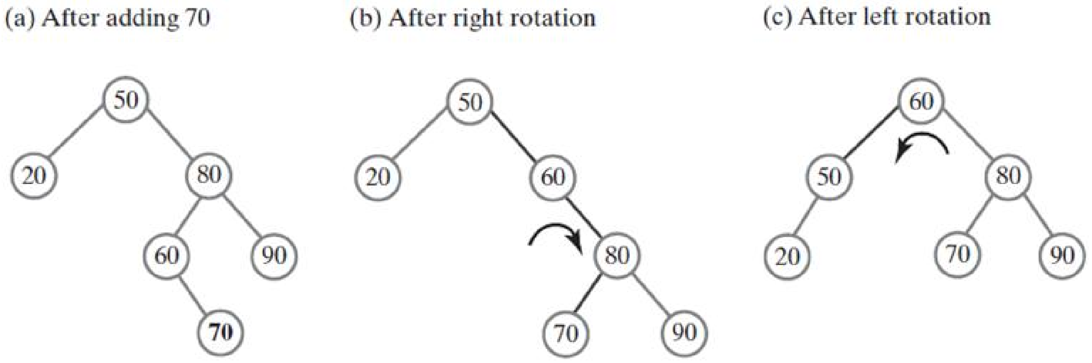
```
Algorithm rotateLeft(nodeN)
// Corrects an imbalance at a given node nodeN due to an addition
// in the right subtree of nodeN's right child.

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

Double Rotations

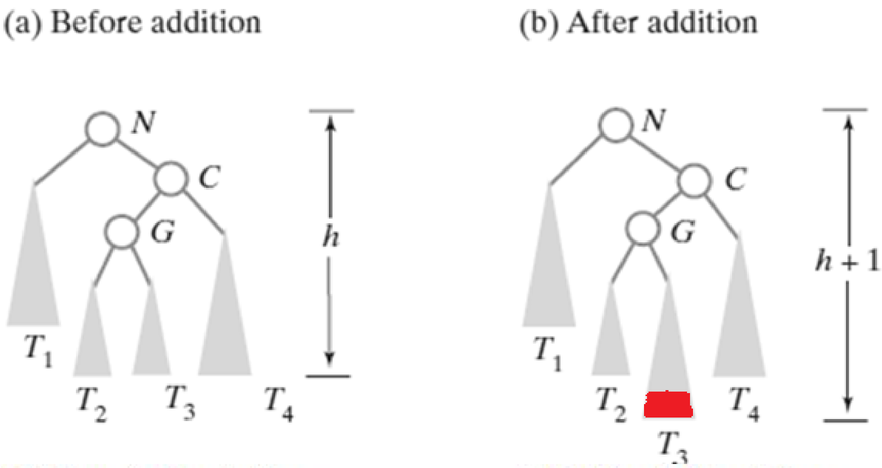
A **double rotation** is accomplished by performing two single rotations:

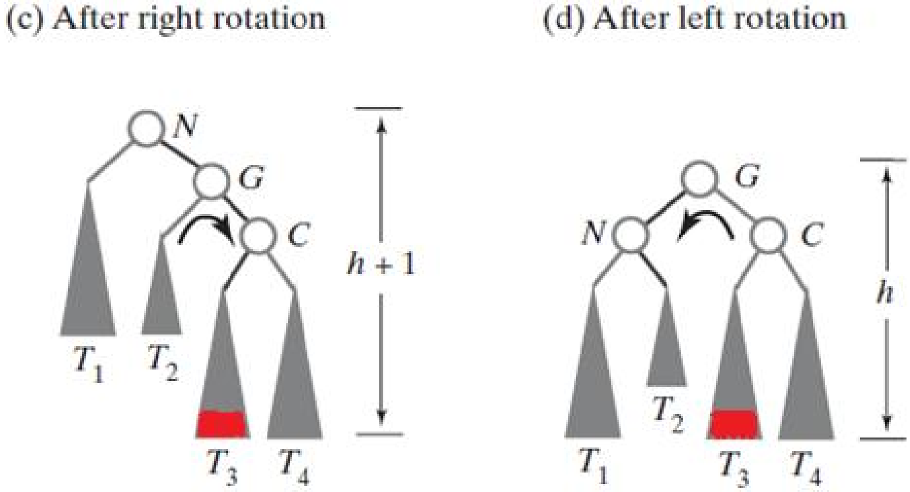
1. A rotation about node ***N*'s grandchild *G*** (its child's child)
2. A rotation about node ***N*'s new child**

Case 3: Right-Left Double Rotations

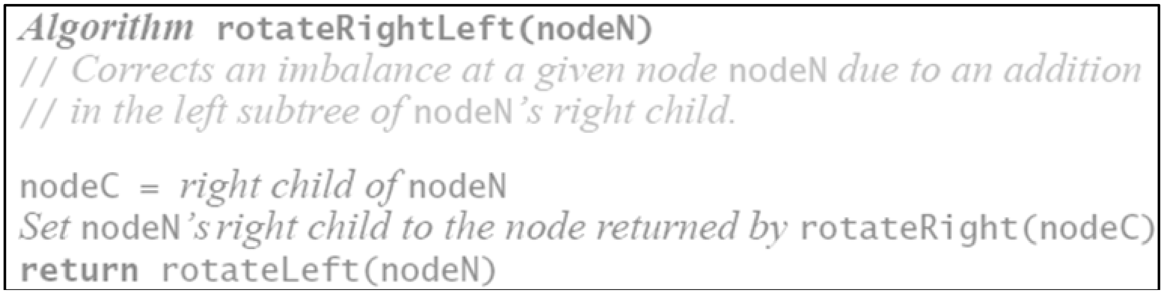


Example: (a) Adding 70 destroys tree's balance; to restore the balance, perform both (b) a **right rotation** and (c) a **left rotation**





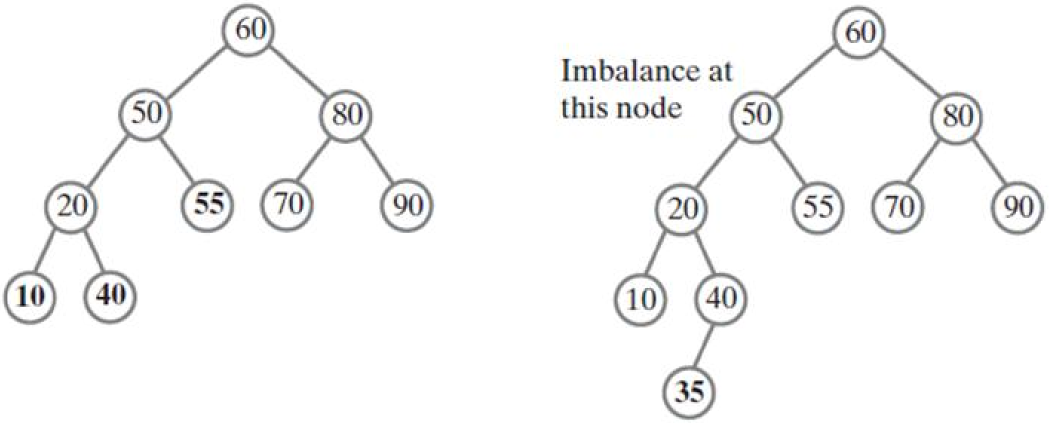
Before and after an addition to an **AVL** subtree that requires both a **right rotation** and a **left rotation** to maintain its balance



```
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)
```

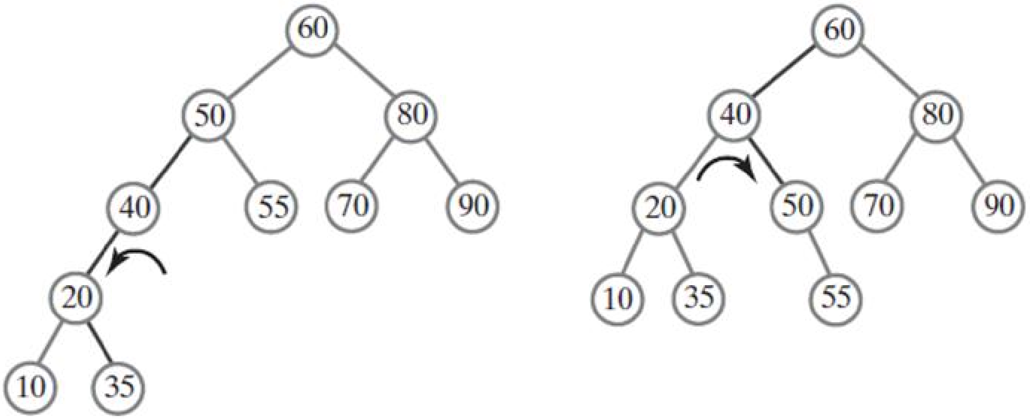
Case 4: Left-Right Double Rotations

Example:



(a) After adding 55, 10, and 40

(b) After adding 35

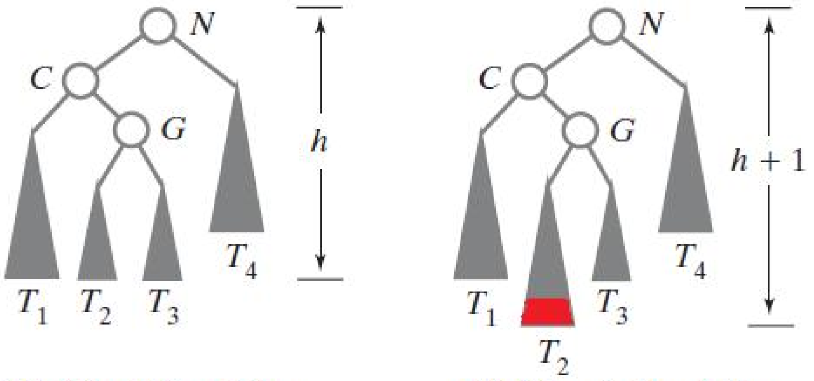


(c) After left rotation about 40

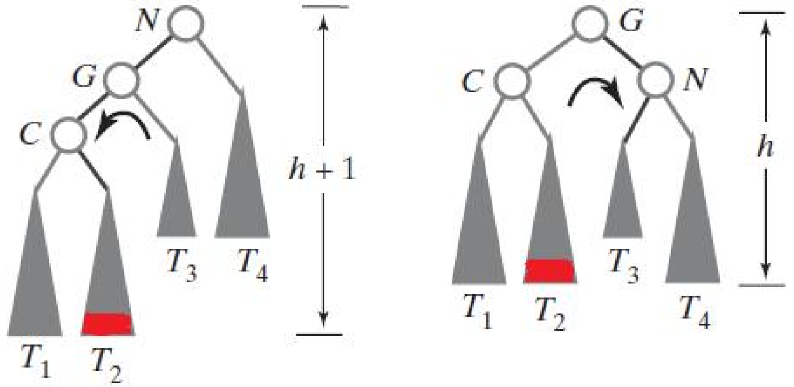
(d) After right rotation about 40

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

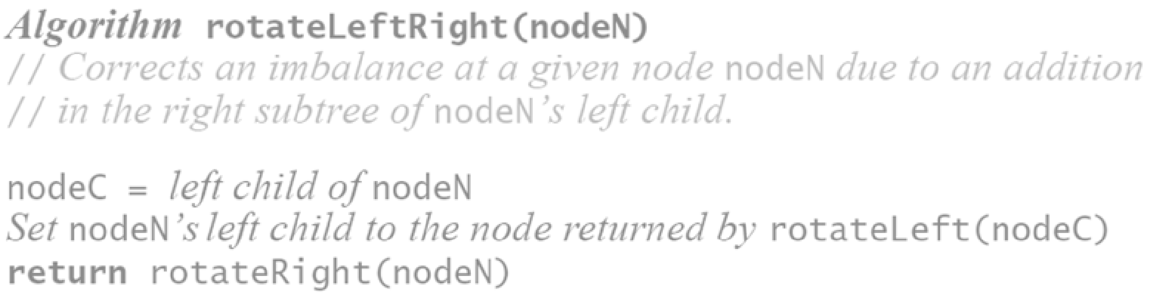
(a) Before addition (b) After addition



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



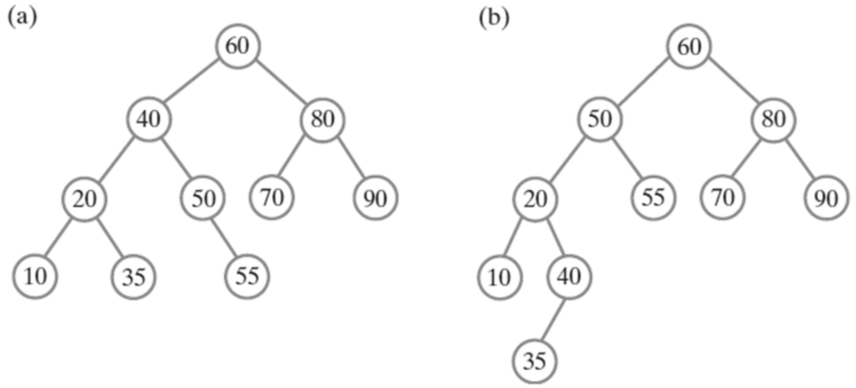
Before and after an **addition** to an **AVL** subtree that requires both a **left rotation** and a **right rotation** to maintain its balance



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

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

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

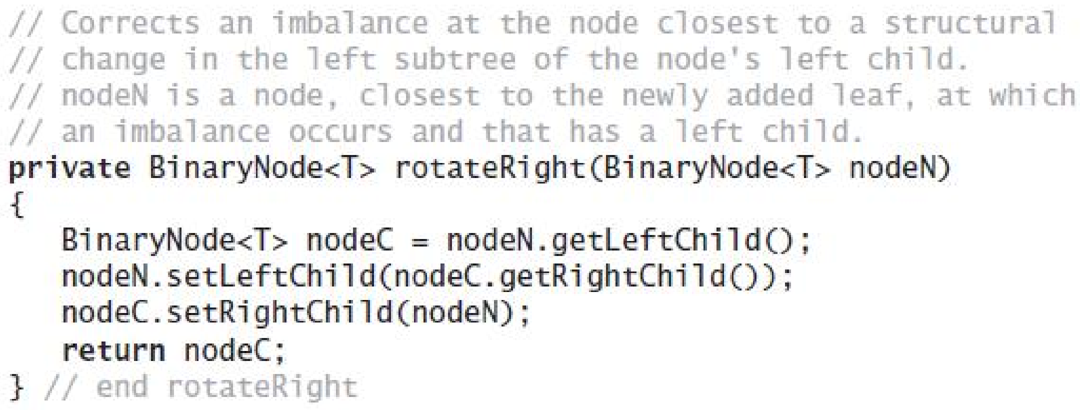
An **AVL** Tree Versus a **BST**



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**

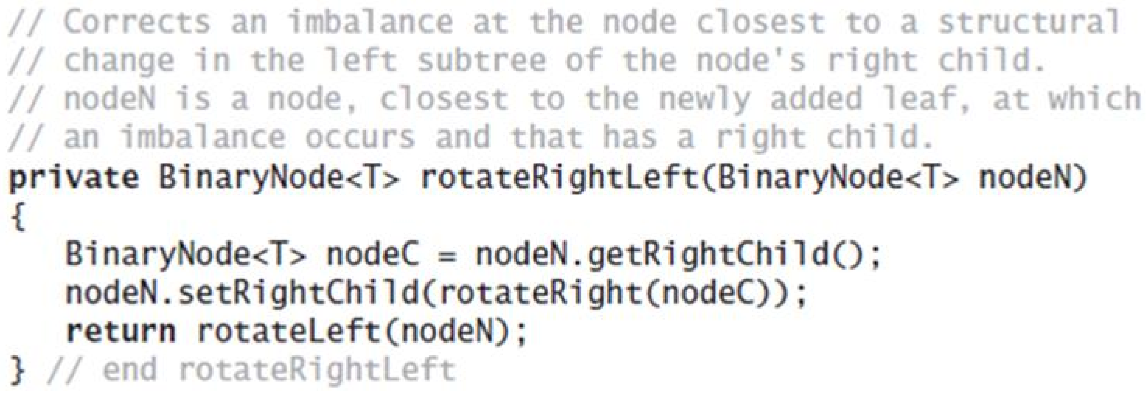
Code Implementation (Optional)

- The implementation of the method for a **single right rotation**:



```
// Corrects an imbalance at the node closest to a structural
// change in the left subtree of the node's left child.
// nodeN is a node, closest to the newly added leaf, at which
// an imbalance occurs and that has a left child.
private BinaryNode<T> rotateRight(BinaryNode<T> nodeN)
{
   BinaryNode<T> nodeC = nodeN.getLeftChild();
   nodeN.setLeftChild(nodeC.getRightChild());
   nodeC.setRightChild(nodeN);
   return nodeC;
} // end rotateRight
```

- The implementation for a **right-left double rotation:**



```
// Corrects an imbalance at the node closest to a structural
// change in the left subtree of the node's right child.
// nodeN is a node, closest to the newly added leaf, at which
// an imbalance occurs and that has a right child.
private BinaryNode<T> rotateRightLeft(BinaryNode<T> nodeN)
{
   BinaryNode<T> nodeC = nodeN.getRightChild();
   nodeN.setRightChild(rotateRight(nodeC));
   return rotateLeft(nodeN);
} // end rotateRightLeft
```

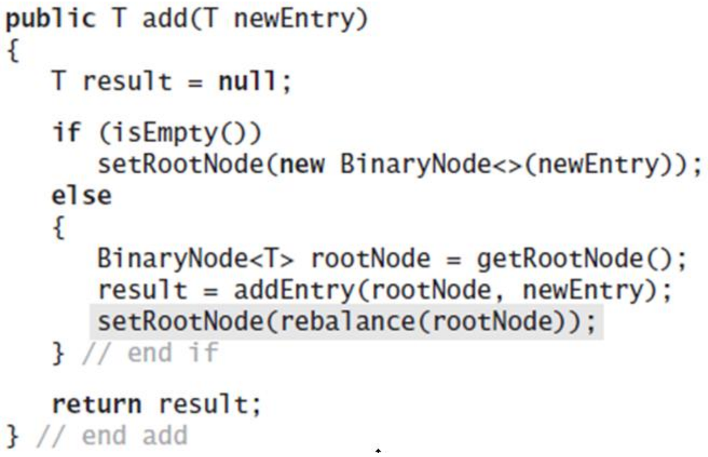
- Pseudo-code to rebalance the tree:

```
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
}
```

- Implementation for **rebalancing** within the class **AVLTree:**

```
private BinaryNode<T> rebalance(BinaryNode<T> nodeN)
{
   int heightDifference = getHeightDifference(nodeN);

   if (heightDifference > 1)
   {  // Left subtree is taller by more than 1,
      // so addition was in left subtree
      if (getHeightDifference(nodeN.getLeftChild()) > 0)
         // Addition was in left subtree of left child
         nodeN = rotateRight(nodeN);
      else
         // Addition was in right subtree of left child
         nodeN = rotateLeftRight(nodeN);
   }
   else if (heightDifference < -1)
   {  // Right subtree is taller by more than 1,
      // so addition was in right subtree
      if (getHeightDifference(nodeN.getRightChild()) < 0)
         // Addition was in right subtree of right child
         nodeN = rotateLeft(nodeN);
      else
         // Addition was in left subtree of right child
         nodeN = rotateRightLeft(nodeN);
   } // end if
   // Else nodeN is balanced

   return nodeN;
} // end rebalance
```

- Methods to **Add:**



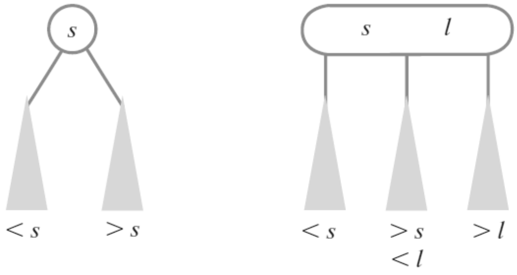
```
public T add(T newEntry)
{
   T result = null;

   if (isEmpty())
      setRootNode(new BinaryNode<>(newEntry));
   else
   {
      BinaryNode<T> rootNode = getRootNode();
      result = addEntry(rootNode, newEntry);
      setRootNode(rebalance(rootNode));
   } // end if

   return result;
} // end add
```

- **AddEntry** code:

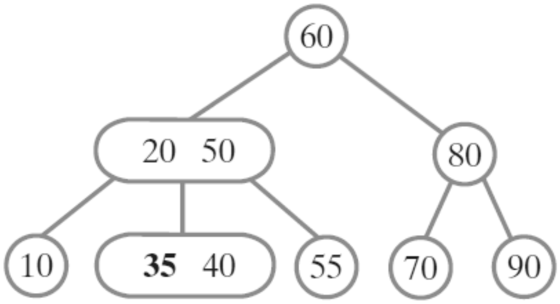
```
private T addEntry(BinaryNode<T> rootNode, T newEntry)
{
   assert rootNode != null;
   T result = null;
   int comparison = newEntry.compareTo(rootNode.getData());
   if (comparison == 0)
   {
      result = rootNode.getData();
      rootNode.setData(newEntry);
   }
   else if (comparison < 0)
   {
      if (rootNode.hasLeftChild())
      {
         BinaryNode<T> leftChild = rootNode.getLeftChild();
         result = addEntry(leftChild, newEntry);
         rootNode.setLeftChild(rebalance(leftChild));
      }
      else
         rootNode.setLeftChild(new BinaryNode<>(newEntry));
   }
   else
   {
      assert comparison > 0;

      if (rootNode.hasRightChild())
      {
         BinaryNode<T> rightChild = rootNode.getRightChild();
         result = addEntry(rightChild, newEntry);
         rootNode.setRightChild(rebalance(rightChild));
      }
      else
         rootNode.setRightChild(new BinaryNode<>(newEntry));
   } // end if

   return result;
} // end addEntry
```

(Lecture 19) 2-3 Trees

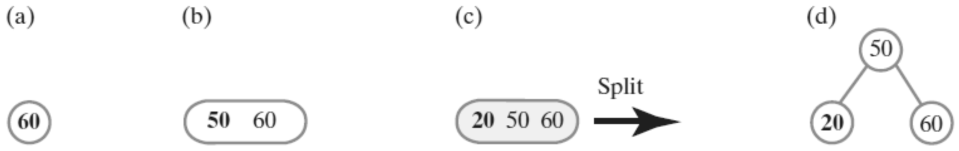
- Definition: general search tree whose interior nodes must have either **2** or **3** children.
    - A 2-node contains one data item ***s*** and has two children
    - A 3-node contains two data items, ***s*** and ***l***, and has three children



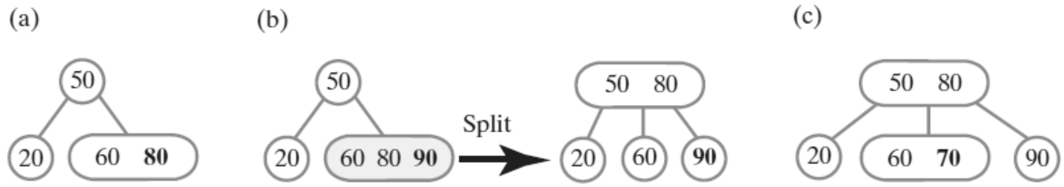
Searching a 2-3 Tree:



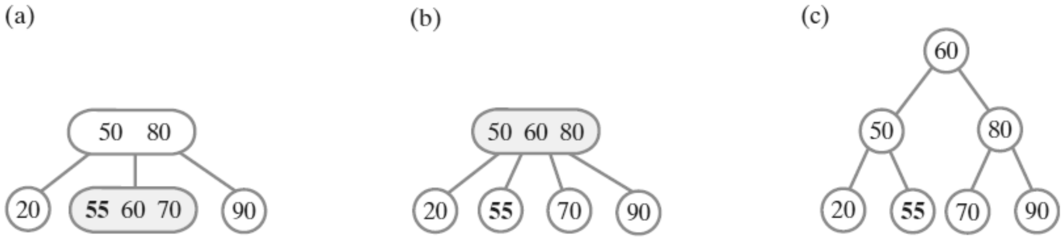
Adding Entries to a 2-3 Tree:



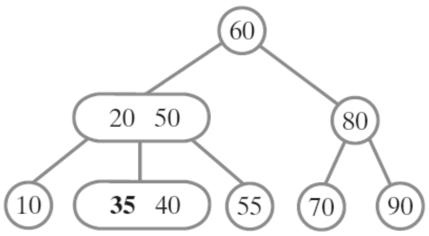
Adding (a) **60** and (b) **50**; (c), (d) adding **20** causes the 3-node to split



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



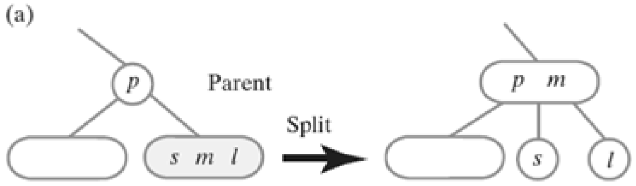
Adding **55** to the 2-3 tree, causes a leaf and then the root to split



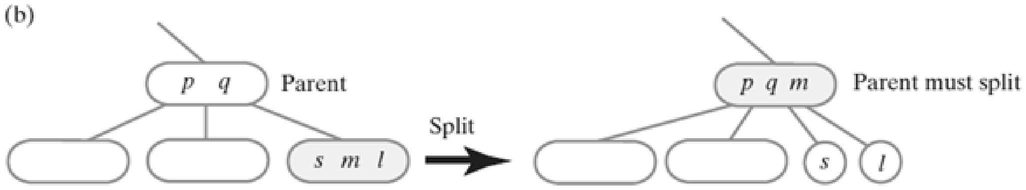
The 2-3 tree, after adding **10, 40, 35**

Splitting Nodes during Addition

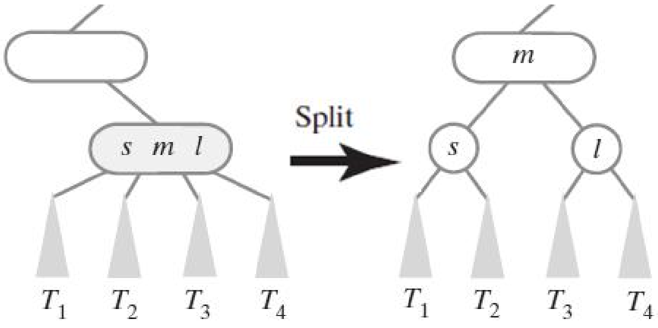
- Splitting a leaf to accommodate a new entry when the leaf's parent contains:
    - **(a) one entry;**



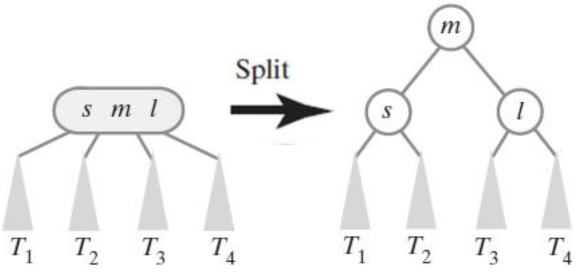
-
    - **(b) two entries**



- Splitting an internal node to accommodate a new entry:



- Splitting the root to accommodate a new entry:



2-3 tree: performance

Perfect balance. Every path from root to null link 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:

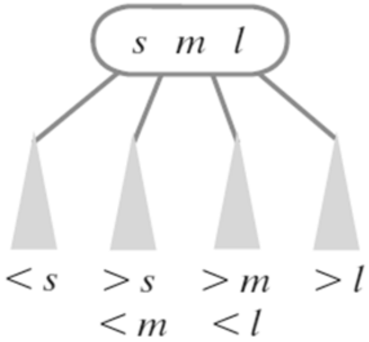
- Maintaining multiple node types is cumbersome.
- Need multiple compares to move down tree.
- Need to move back up the tree to split 4-nodes.
- Large number of cases for splitting.

Bottom line. Could do it, but there's a better way.

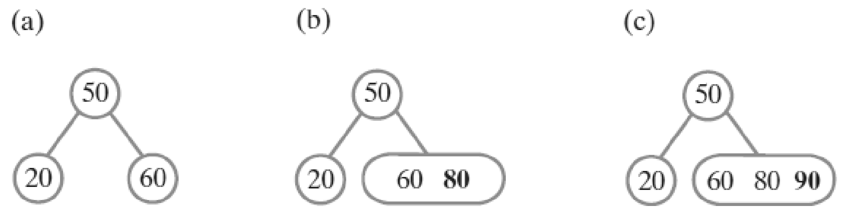
HW: 50 60 70 40 30 20 10 80 90 100

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.

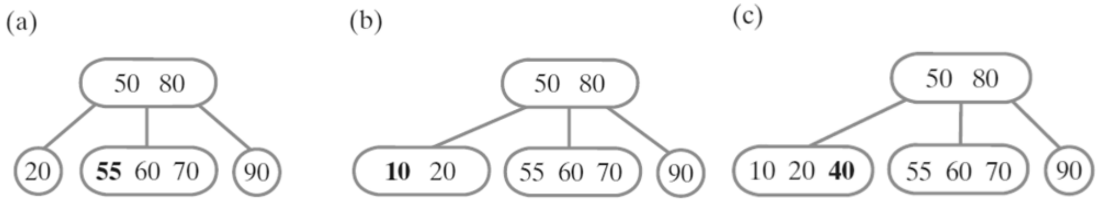


Adding Entries to a 2-4 Tree



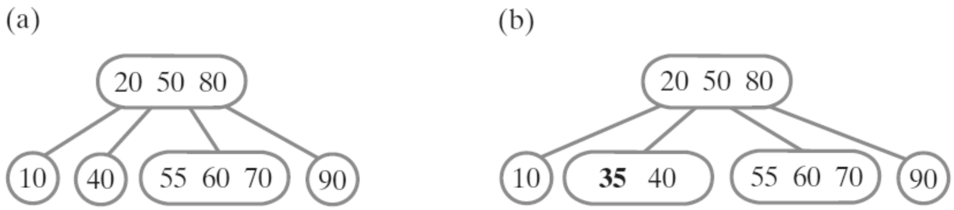
The 2-4 tree, after (a) splitting the root; (b) adding **80**; (c) adding **90**

Adding 70



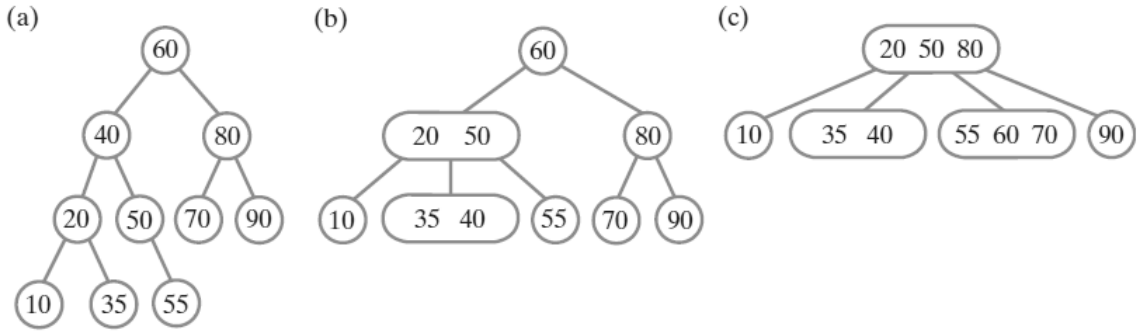
The 2-4 tree after adding (a) **55**; (b) **10**; (c) **40**

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

(Lecture 20) 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?

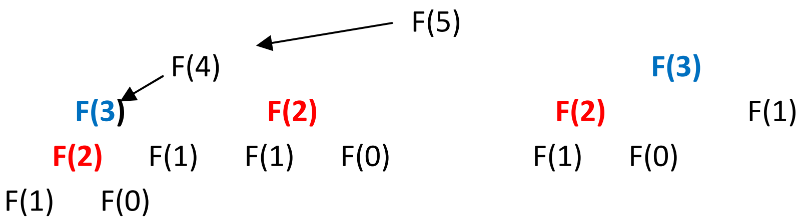
Do analyze the 2 algorithms in term of calculating F(n)

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**:



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 $T(n-2) < T(n-1)$

$$
\begin{aligned}
T(n) &= 2\,T(n-1) + c \quad \rightarrow \quad T(n-1) = 2\,T(n-2) + c \\
&= 2\,\{ 2\,T(n-2) + c \} + c \\
&= 4\,T(n-2) + 3c \\
&= 8\,T(n-3) + 7c \\
&= 16\,T(n-4) + 15c \\
&\;: \\
&= 2^k\,T(n-k) + (2^k-1)c
\end{aligned}
$$

For $T(0)$ ➔ $n-k = 0$ ➔ $k = n$

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

$T(n)$ 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 $\mathbf{X^n}$ using recursion

Iterative solution: **O(n)** $X^n = X * X * X * X * \ldots * X$ 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

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

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 $n == 1$

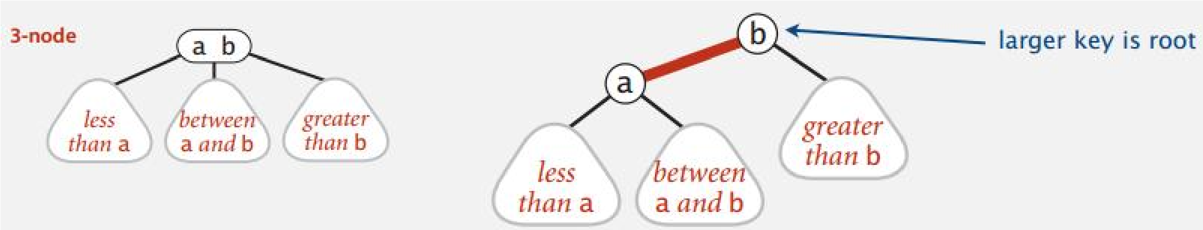
If even $\rightarrow T(n) = T(n/2) + c1$
If odd $\rightarrow T(n) = T(n-1) + c2$
If 0 $\rightarrow T(0) = 1$
If 1 $\rightarrow T(1) = c3$

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 $T(1) \rightarrow T(0) + c \rightarrow 1$
$n/2^k = 1 \rightarrow n = 2^k \rightarrow k = \log n$
$= c3 + c \log n \rightarrow$ **$O(\log n)$**

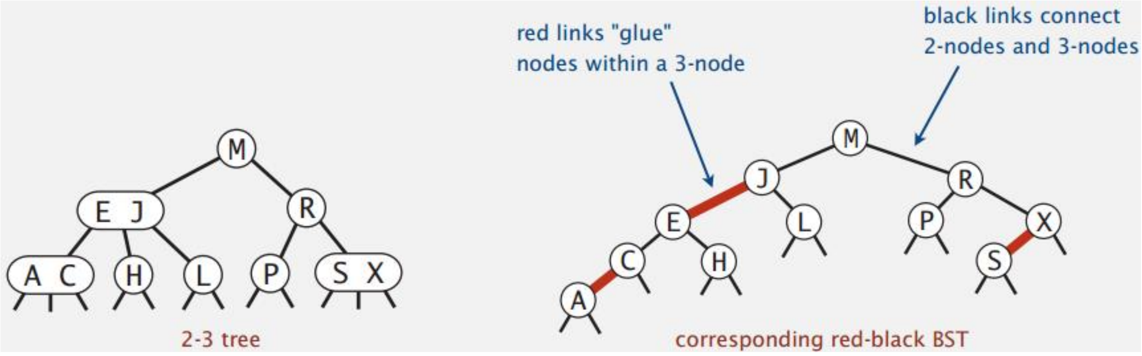
(Lecture xx) Red-Black Trees (Optional)

Left-leaning red-black BSTs (Guibas-Sedgewick 1979 and Sedgewick 2007): **LLRB**

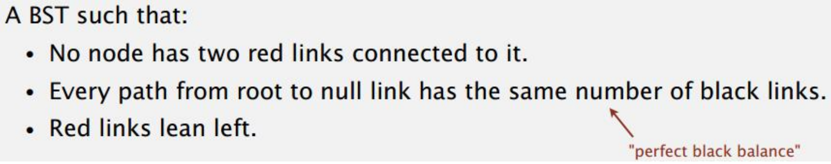
1. Represent **2–3** tree as a **BST**.
2. Use "internal" left-leaning links as "glue" for **3–nodes**.



Example:



An equivalent definition:

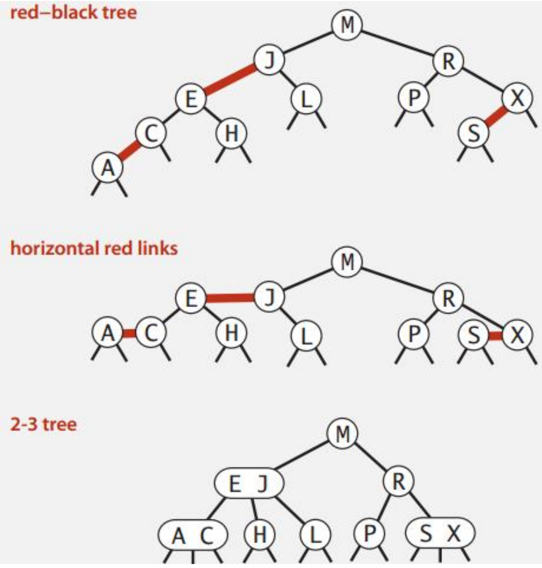


A BST such that:

- No node has two red links connected to it.
- Every path from root to null link has the same number of black links.
- Red links lean left.

"perfect black balance"

Key property. 1–1 correspondence between **2–3** and **LLRB**.



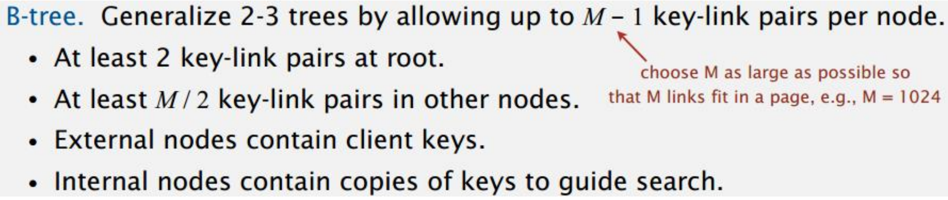
To be continue.

(Lecture 21) B-Trees

An **M**-ary search tree allows **M**-way branching.

As branching increases, the depth decreases.

B-trees (Bayer-McCreight, 1972)



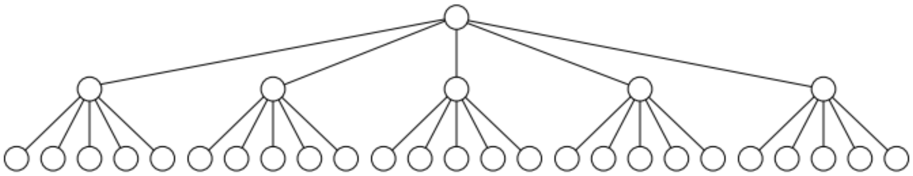
B-tree. Generalize 2-3 trees by allowing up to $M - 1$ key-link pairs per node.

- At least 2 key-link pairs at root.
- At least $M / 2$ key-link pairs in other nodes.
- External nodes contain client keys.
- Internal nodes contain copies of keys to guide search.

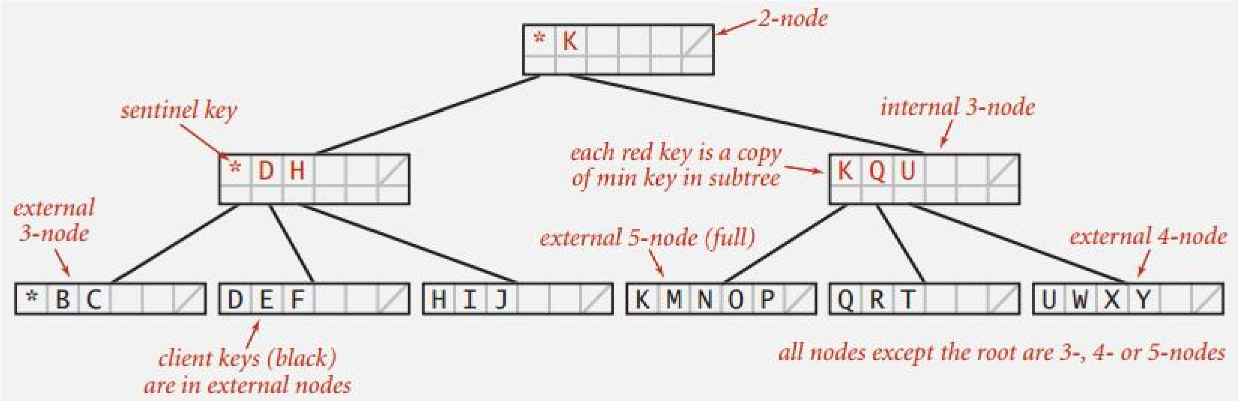
choose M as large as possible so that M links fit in a page, e.g., M = 1024

Nodes **must** be half full to guarantee that the tree does not degenerate into a simple binary tree.

Example: A 5-ary tree of 31 nodes has only three levels:



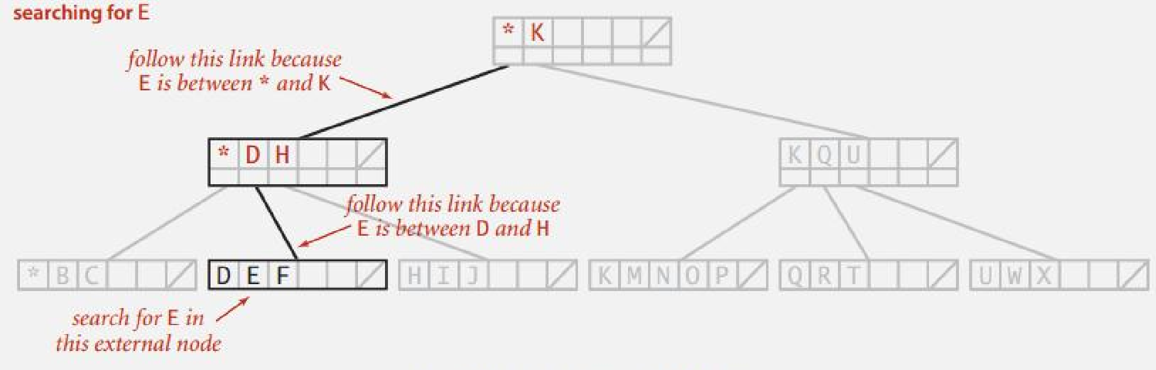
Example:



Anatomy of a B-tree set (M = 6)

Searching in a B-tree

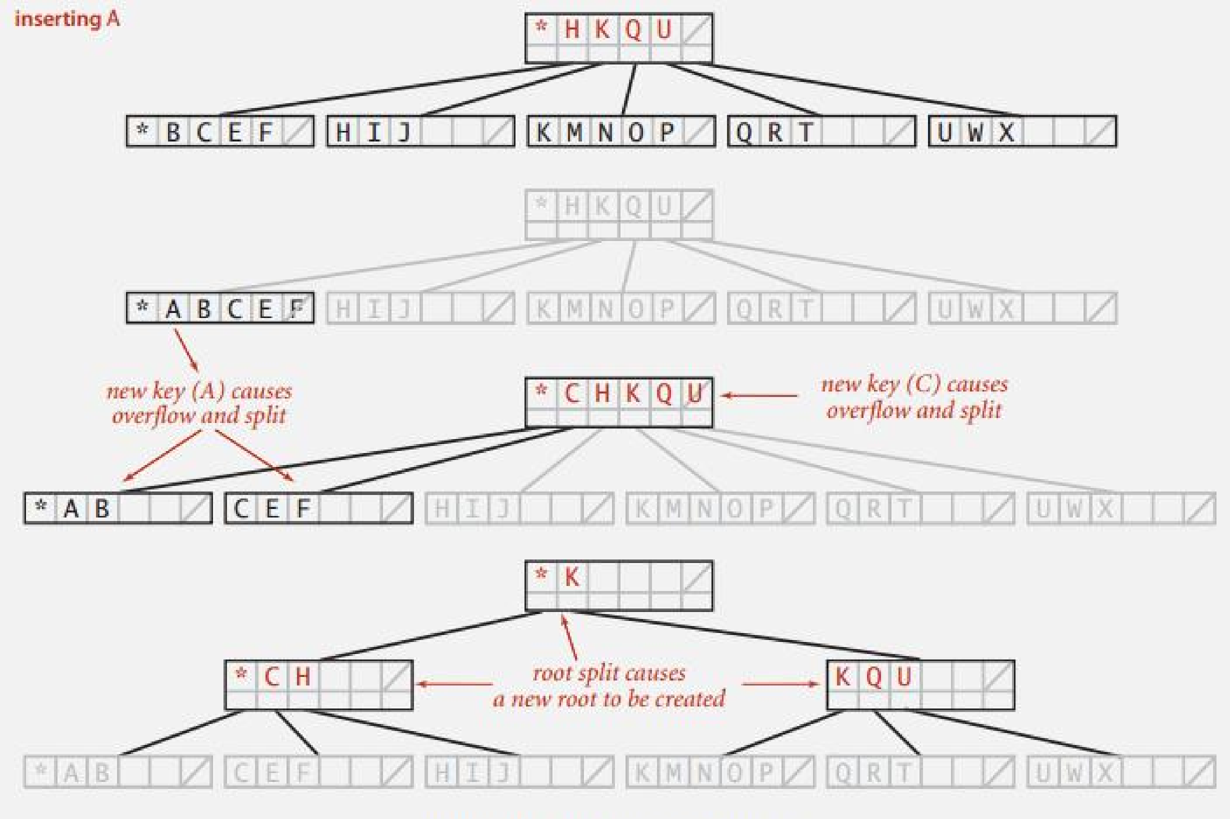
- Start at root.
- Find interval for search key and take corresponding link.
- Search terminates in external node.



Searching in a B-tree set (M = 6)

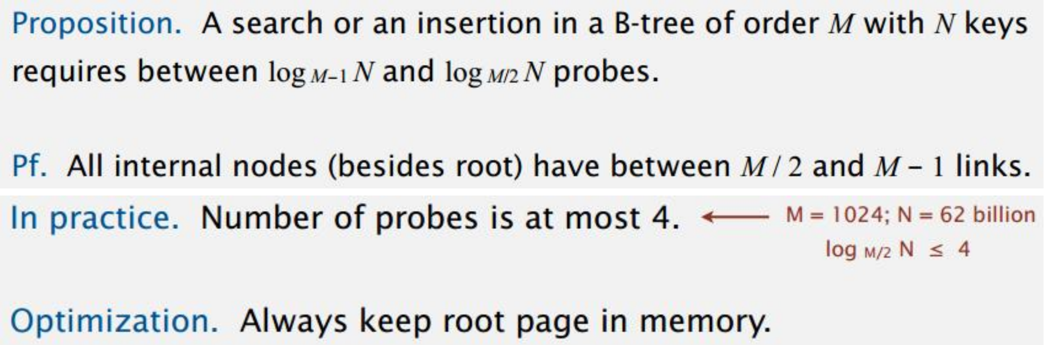
Insertion in a B-tree

- Search for new key.
- Insert at bottom.
- Split nodes with M key-link pairs on the way up the tree.



Inserting a new key into a B-tree set

Balance in B-tree



Proposition. A search or an insertion in a B-tree of order M with N keys requires between $\log_{M-1} N$ and $\log_{M/2} N$ probes.

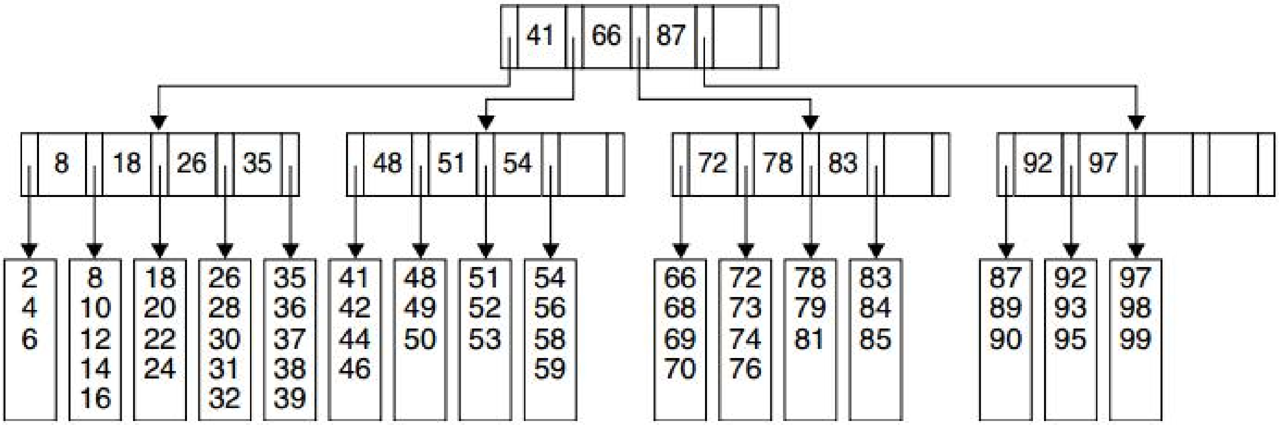
Pf. All internal nodes (besides root) have between $M / 2$ and $M - 1$ links.

In practice. Number of probes is at most 4. ← M = 1024; N = 62 billion $\log_{M/2} N \leq 4$

Optimization. Always keep root page in memory.

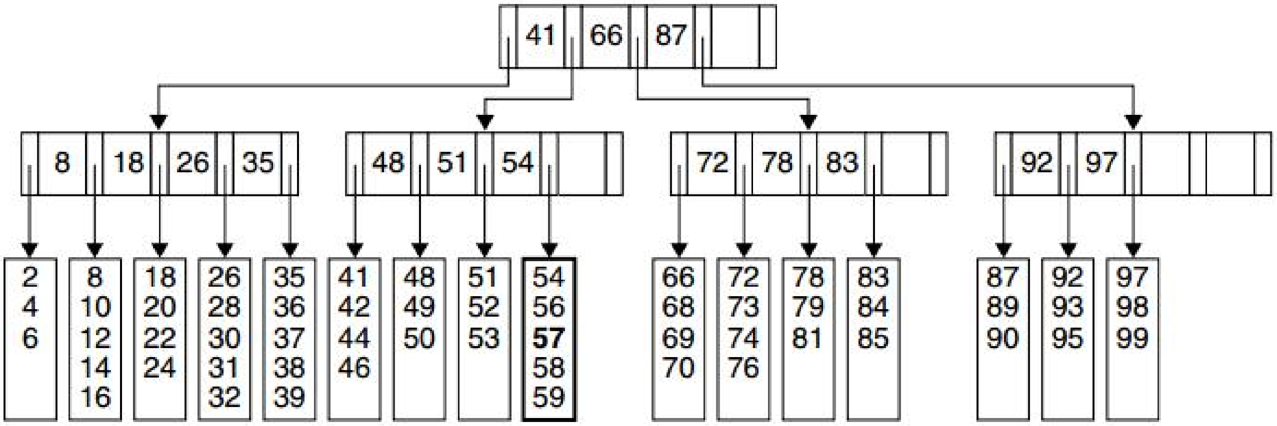
The B-tree is the most popular data structure for disk bound searching.

Example: A B-tree of order **5**



Insertion: insert 57

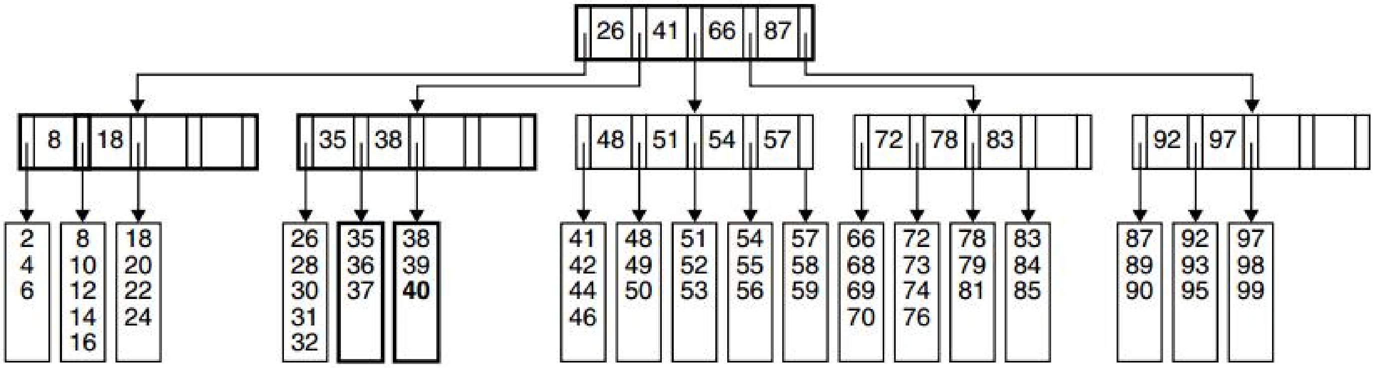
- If the leaf contains room for a new item, we insert it and are done.
- If the leaf is full, we can insert a new item by splitting the leaf and forming two half-empty nodes.



The B-tree after insertion of 57

Insertion: insert 40

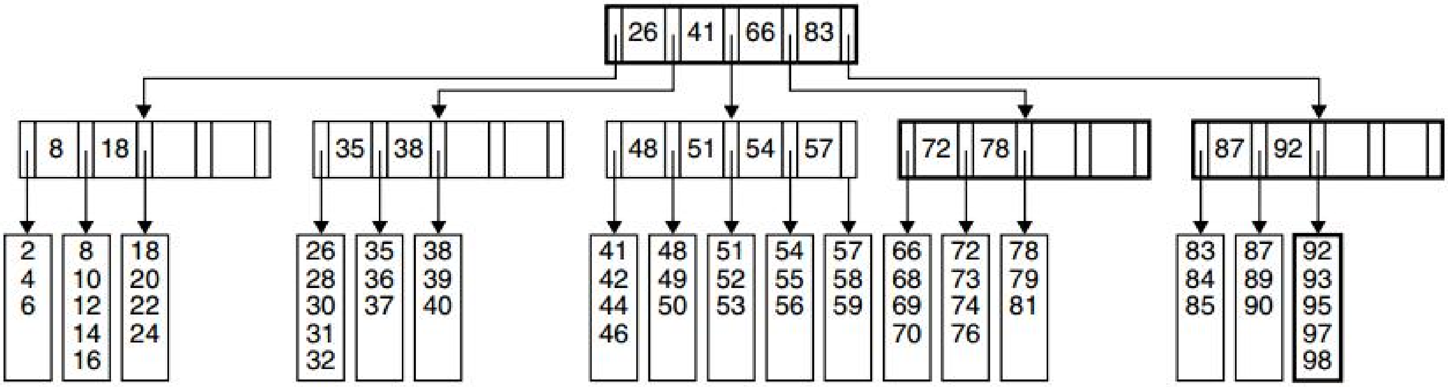
- Node splitting creates an extra child for the leaf's parent.
- If the parent already has a full number of children, we split the parent.
- We may have to continue splitting all the way up the tree (though this possibility is unlikely).
- In the worst case, we split the root, creating a new root with two children.



Insertion of **40** causes a split into two leaves and then a split of the parent node.

Deletion works in reverse: **remove 99:**

- If a leaf loses a child, it may need to combine with another leaf.
- Combining of nodes may continue all the way up the tree, though this possibility is unlikely.
- In the worst case, the root loses one of its two children. Then we delete the root and use the other child as the new root.



The B-tree after deletion of **99** from the tree

(Lecture 22) 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:

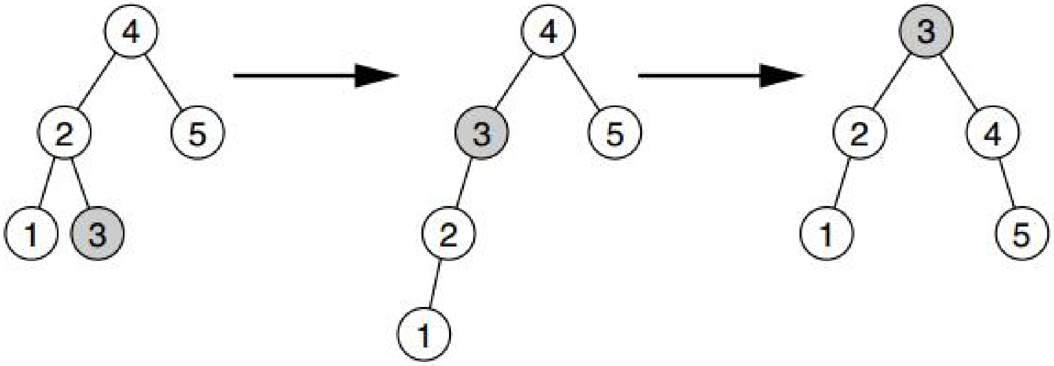
- Like **AVL** trees, use the standard binary search tree property.
- After any operation on a node, make that node the new root of the tree.

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

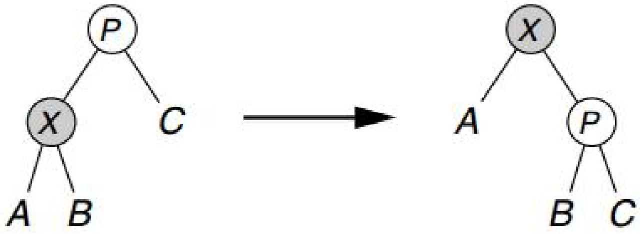


- As a result of the rotation:
    - future accesses of node **3** are cheap
    - Unfortunately, in the process of moving node **3** up two levels, nodes **4** and **5** each move down a level.
- Thus, if access patterns do not follow the **90–10 rule**, a long sequence of bad accesses can occur.

The basic bottom-up splay tree

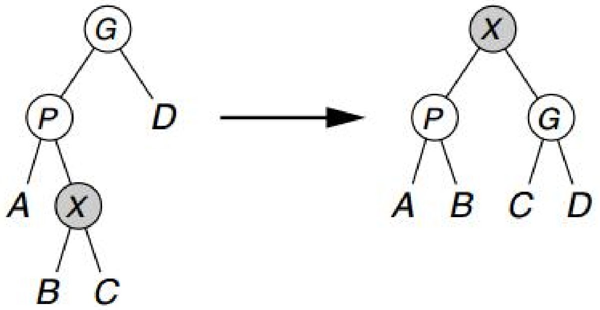
Splaying cases:

- **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:

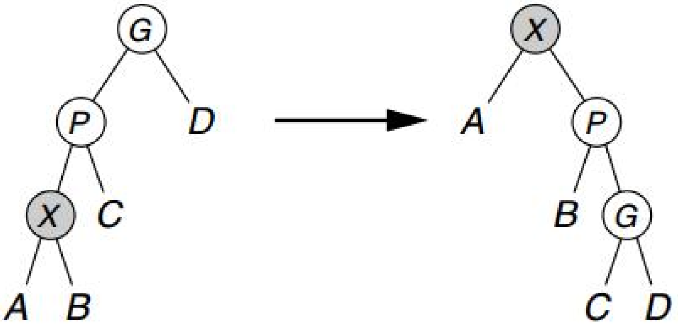


Otherwise, **X** has both a parent **P** and a grandparent **G**, and we must consider two cases and symmetries.

- **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:

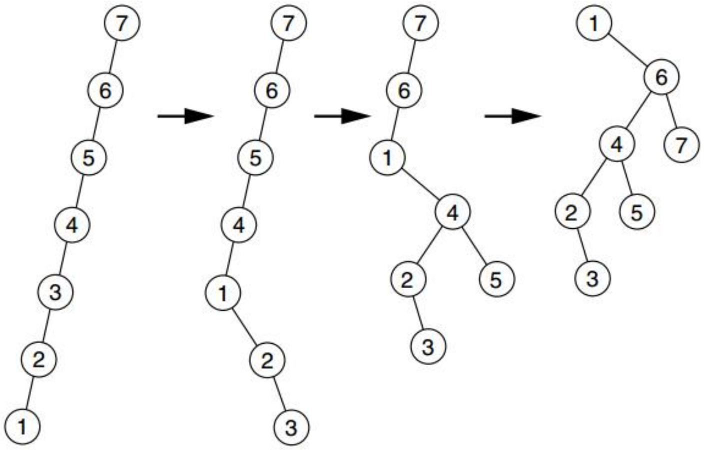


- **zig-zig case:**
  - The outside case for **AVL** trees.
  - Here, **X** and **P** are either both left children or both right children.
  - In this case, we transform the left-hand tree to the right-hand tree (or vice versa).
  - Note that this method differs from the **rotate-to-root strategy**.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



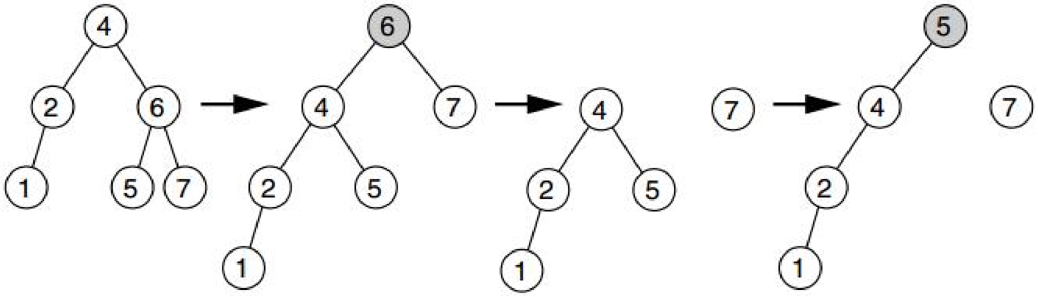
Exercise: perform rotate-to-root strategy

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:

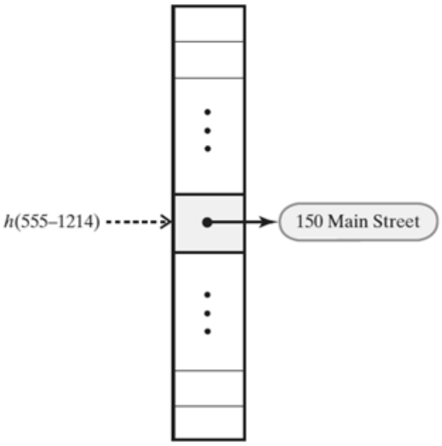
  Example: The remove operation applied to node **6**:
  - First, **6** is splayed to the root, leaving two subtrees;
  - A **findMax** is performed on the left subtree, raising **5** to the root of the left subtree;
  - Then the right subtree can be attached (not shown).



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

(Lecture 23 & 24) 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.



Hash table

Example 1: Phone numbers (xxx-xxxx).

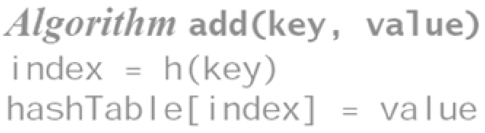
- **Bad**: first three digits. // identical for same area
- **Better**: last four digits. // distinct

Example 2: Social Security numbers (ID number).

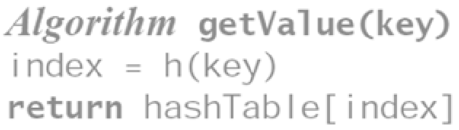
- **Bad**: first three digits. // identical for same period
- **Better**: last three digits. // distinct

Practical challenge: Need different approach for each key type.

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



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



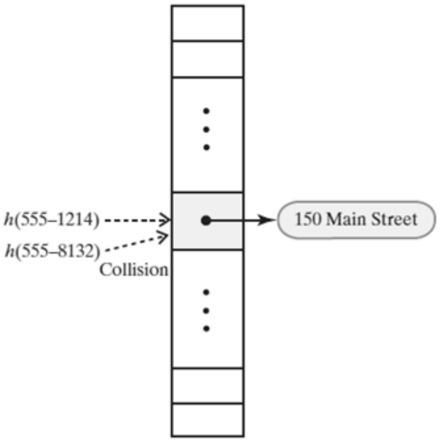
```
Algorithm getValue(key)
index = h(key)
return hashTable[index]
```

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
```



Hash table

- 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 tableSize = 101

- **getHashIndex(555-1264)** = 52
- **getHashIndex(555-8132**) = 52 also!!!

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.

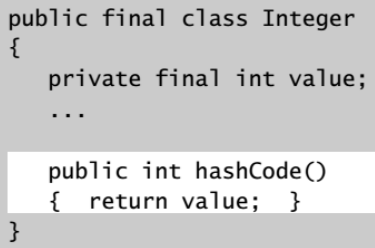
Default implementation: **Memory address**.

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

User-defined types: Users are on their own.

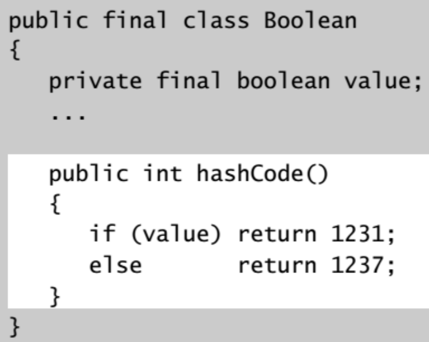
Java library implementations:

Integer



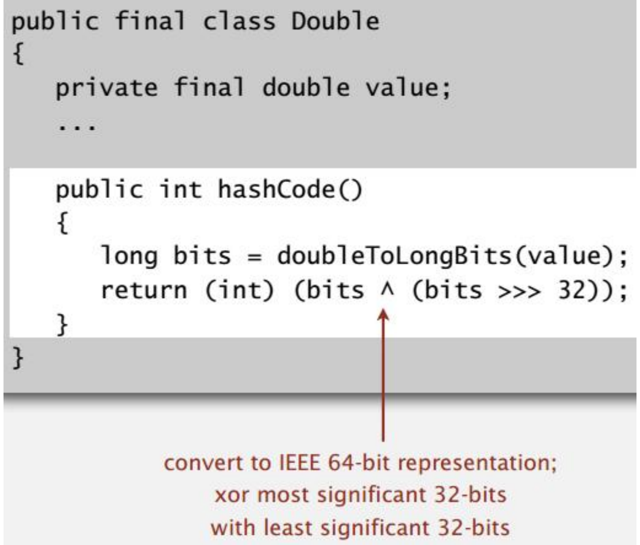
```
public final class Integer
{
   private final int value;
   ...

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

Boolean



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

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

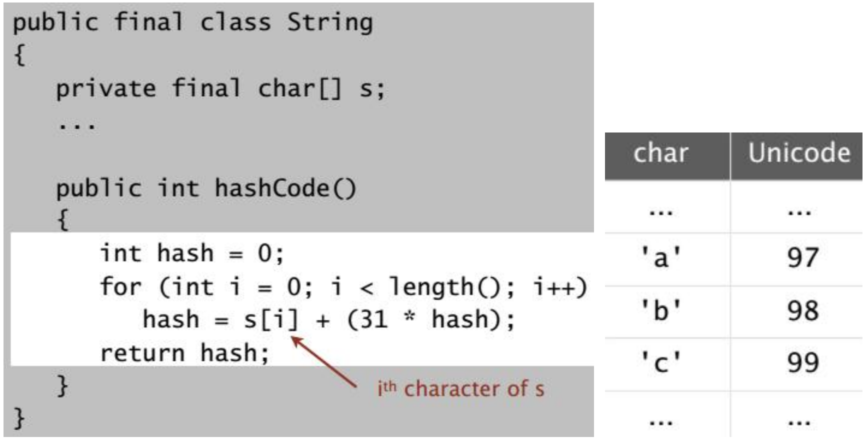
Double



```
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;
   }
}
```

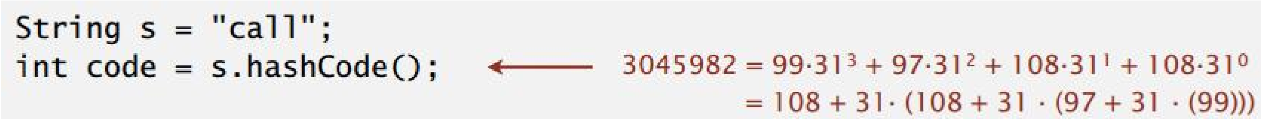
i^{th} character of s

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

Horner's method to hash 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();
```

$$\begin{aligned} 3045982 &= 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0 \\ &= 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99))) \end{aligned}$$

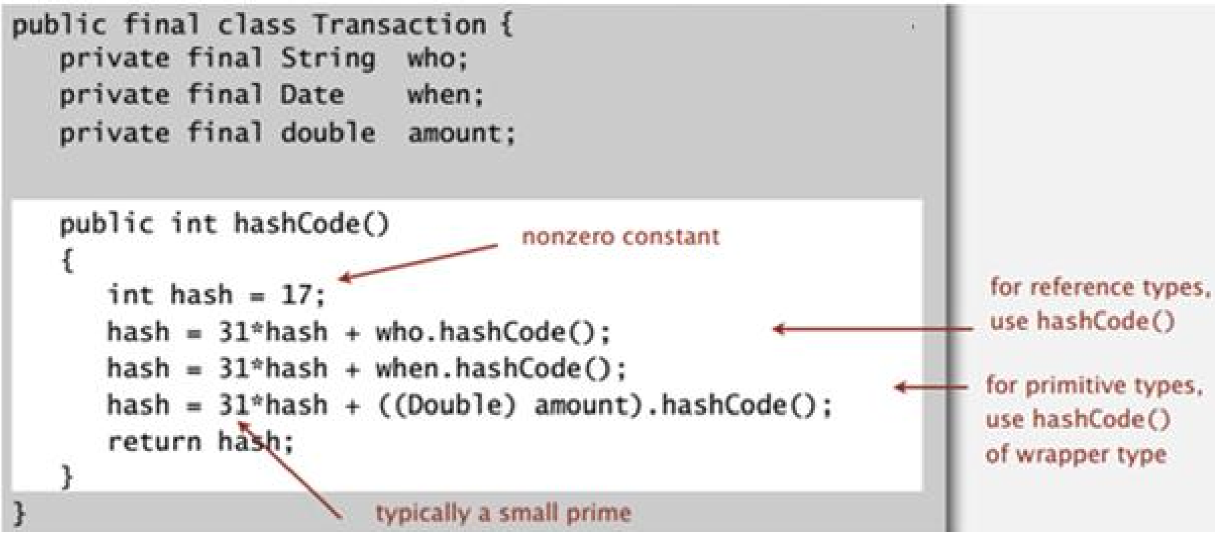
Implementing hash code: user-defined types

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: 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

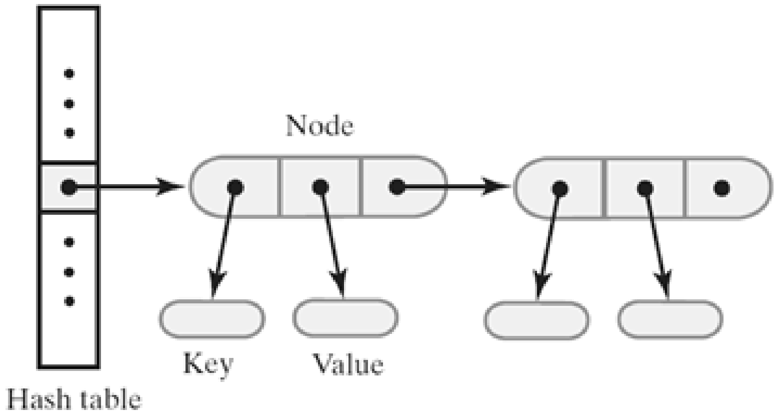
```
private int hash(Key key)
{  return (key.hashCode() & 0x7fffffff) % M;  }
```

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**)

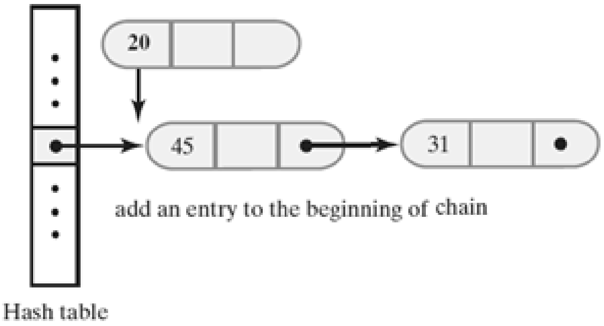
Separate Chaining

- 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.**

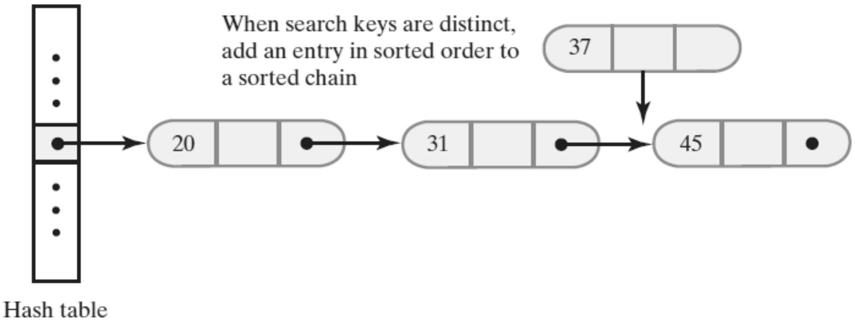


Where to insert a new entry into a linked bucket?

(a) If **unsorted** (apply 90-10 rule):



(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:

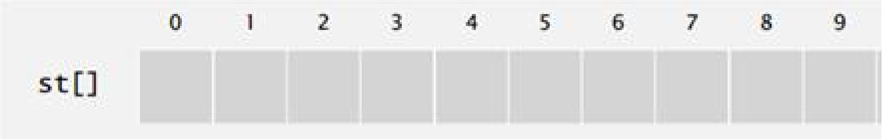
- **M** too large ➔ too many empty chains.
- **M** too small ➔ chains too long.
- Typical choice: **M** ≈ **N / 5** ➔ constant-time ops.

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**.

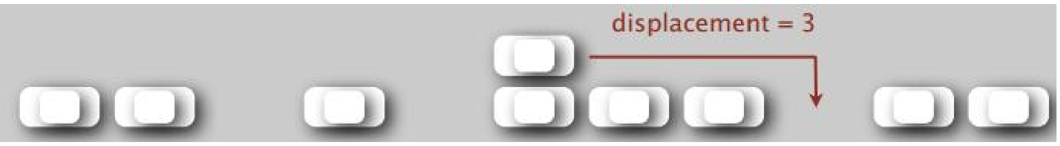
Example: Linear hash table demo: **take last 2 digits of student's ID and run a demo**

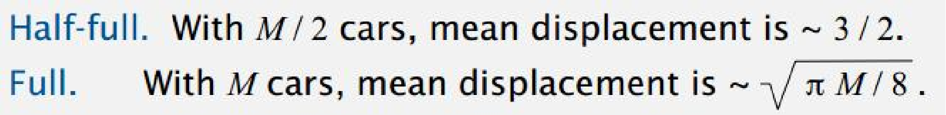


Clustering problem: A contiguous block of items will be easily formed which in turn will affect performance.

Q. What is mean **displacement** of items? (**Knuth's Parking Problem**)

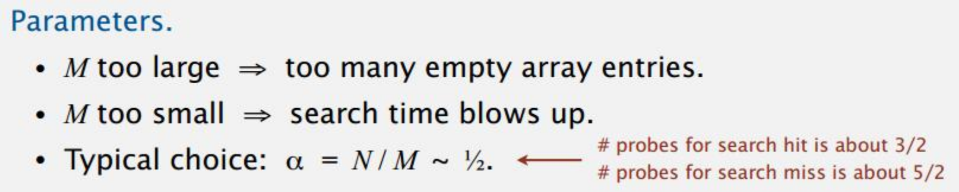
- **Model**: Cars arrive at one-way street with **M** parking spaces. If space **k** is taken, try **k+1**, **k+2**, etc.





Half-full. With $M/2$ cars, mean displacement is ~ $3/2$.

Full. With M cars, mean displacement is ~ $\sqrt{\pi M/8}$.

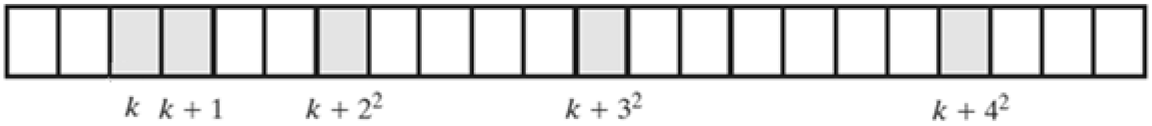


Parameters.

- 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 **$k + j^2$**
  - Uses the indices ***k*, *k*+1, *k* + 4, *k* + 9**, …



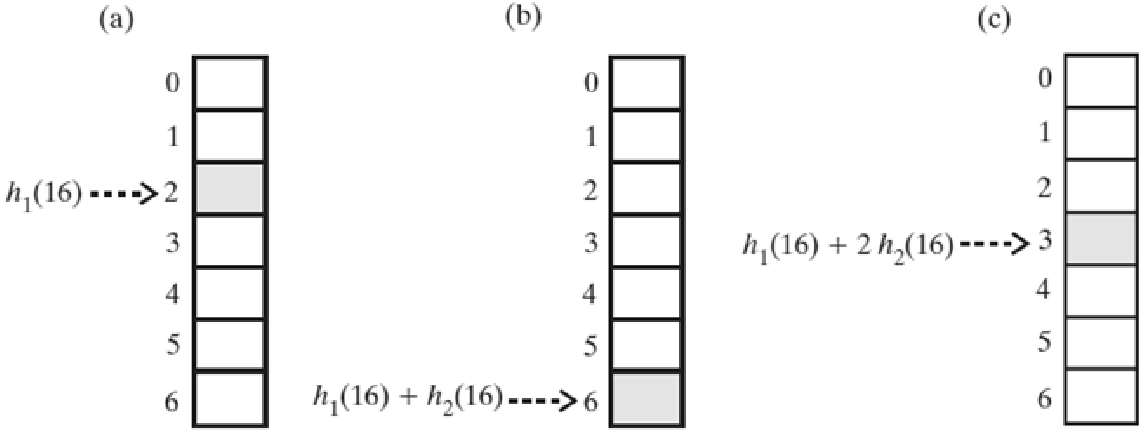
- Hash function: **$h(k,i) = (h(k,0)+i^2)\ \%\ m$**
- For linear probing it is a bad idea to let the hash table get nearly full, because performance degrades.
- For quadratic probing, the situation is even worse: There is no guarantee of finding an empty cell once the table gets more than half full, 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**.

$$h(k,i) = (\underbrace{h_1(k)} + i\,\underbrace{h_2(k)})\ \%\ m$$

Two different hash functions

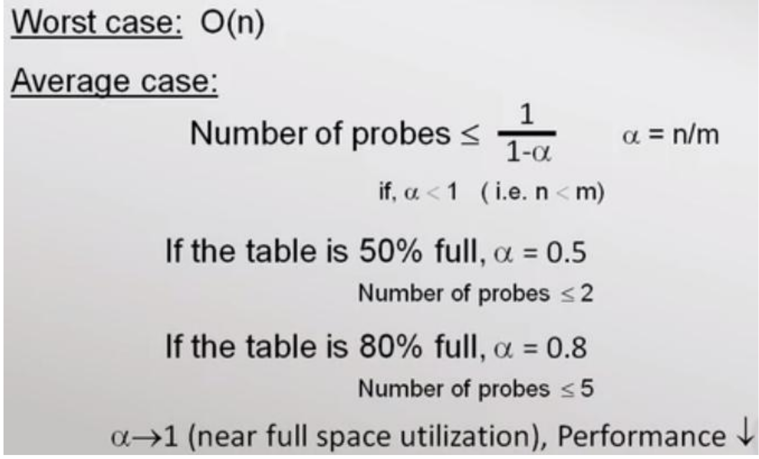


The 1st three locations in a probe sequence generated by double hashing for the search key 16

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:

Number of probes $\le \frac{1}{1-\alpha}$ $\alpha = n/m$

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

$\alpha \to 1$ (near full space utilization), Performance ↓

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.

(Lecture 25) 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:

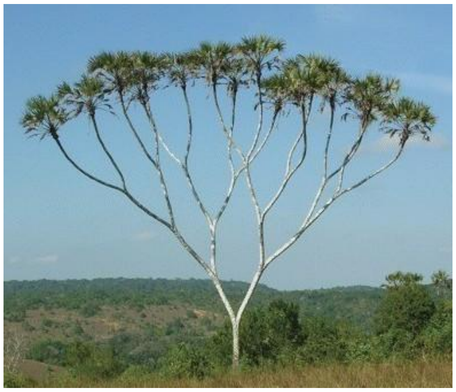
- **Unsorted Linked list**, performing insertions at the front in **O(1)** and traversing the list, which requires **O(N)** time, to delete the minimum/maximum.
- **Sorted Linked list**, performing insertions in **O(N)** and **O(1)** to delete the minimum/maximum.
- **Binary search tree**: this gives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

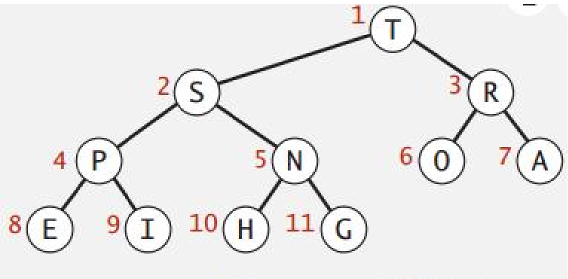
Binary Heap

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, 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.

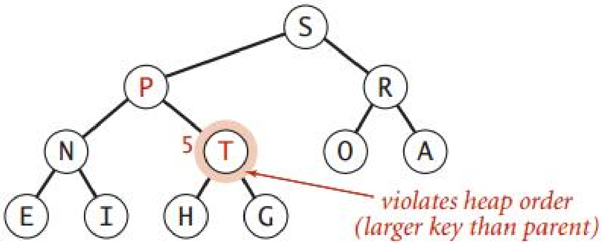
Promotion in a heap

Scenario 1: Child's key becomes larger than its parent's key.

To eliminate the violation:

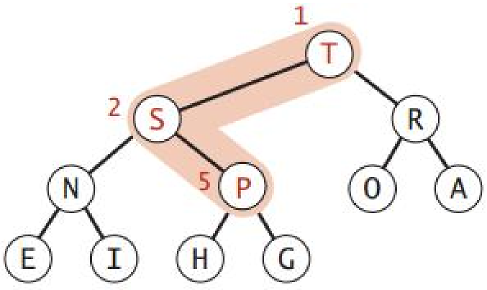
- Exchange key in child with key in parent.
- Repeat until heap order restored.

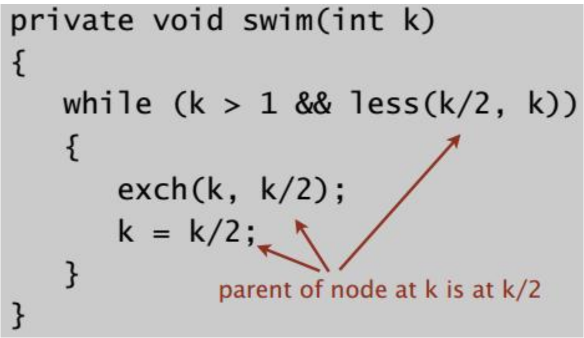
Example:



violates heap order (larger key than parent)

➔





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

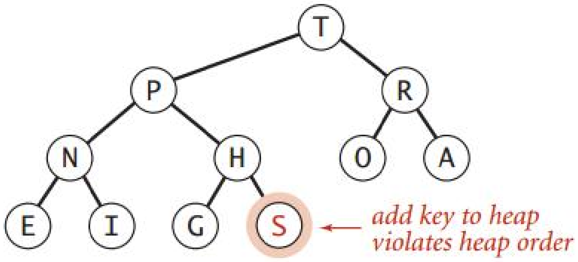
parent of node at k is at k/2

Insertion in a heap

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

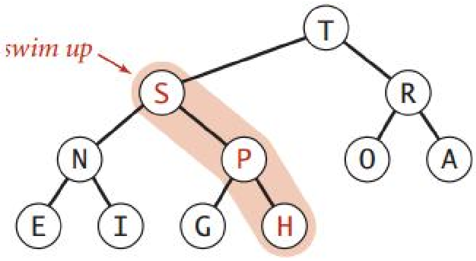
Cost: At most ***1 + lg N*** compares.

Example: Insert **S**

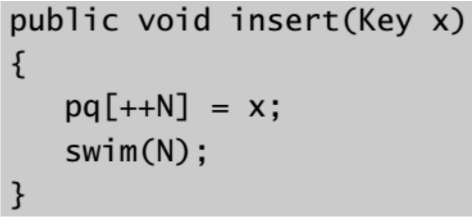


add key to heap violates heap order

➔



swim up



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

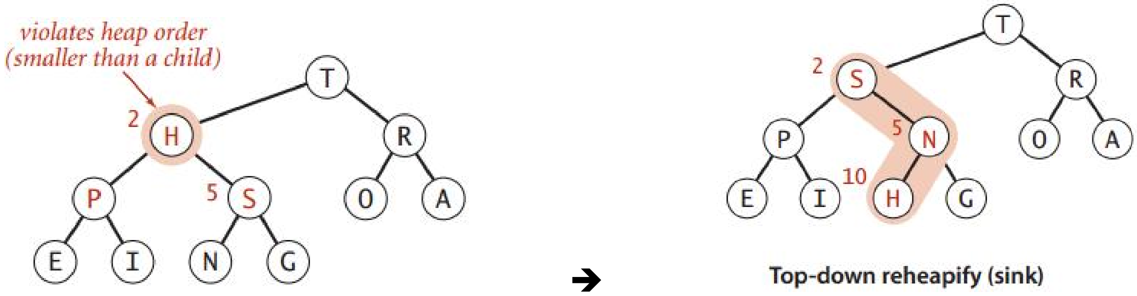
Demotion in a heap

Scenario 2: Parent's key becomes smaller than one (or both) of its children's.

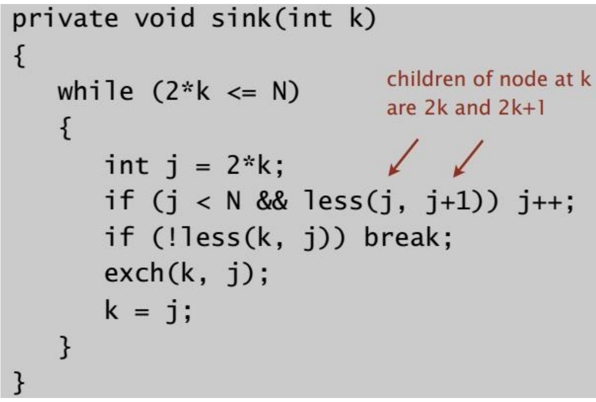
To eliminate the violation:

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

Example:



Top-down reheapify (sink)



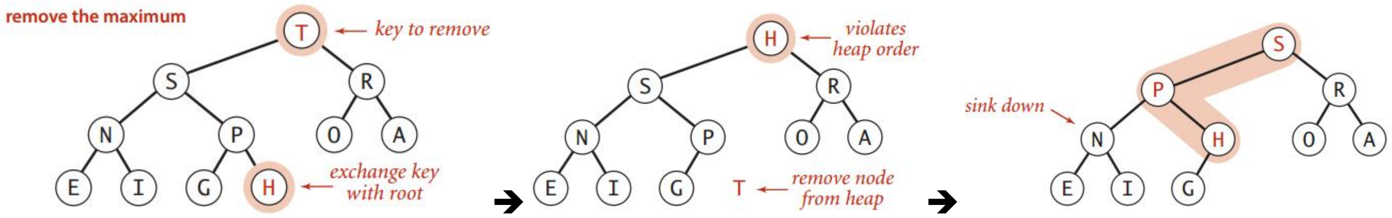
```
private void sink(int k)
{
   while (2*k <= N)
   {
      int j = 2*k;
      if (j < N && less(j, j+1)) j++;
      if (!less(k, j)) break;
      exch(k, j);
      k = j;
   }
}
```

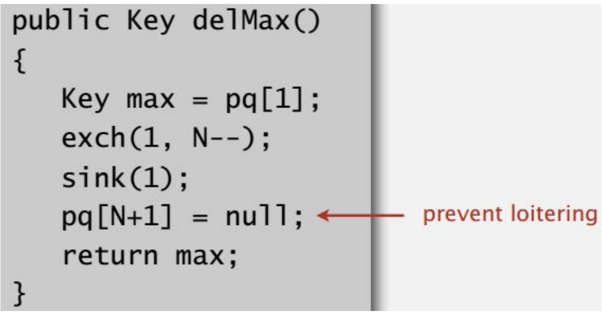
Delete the maximum in a heap

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

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

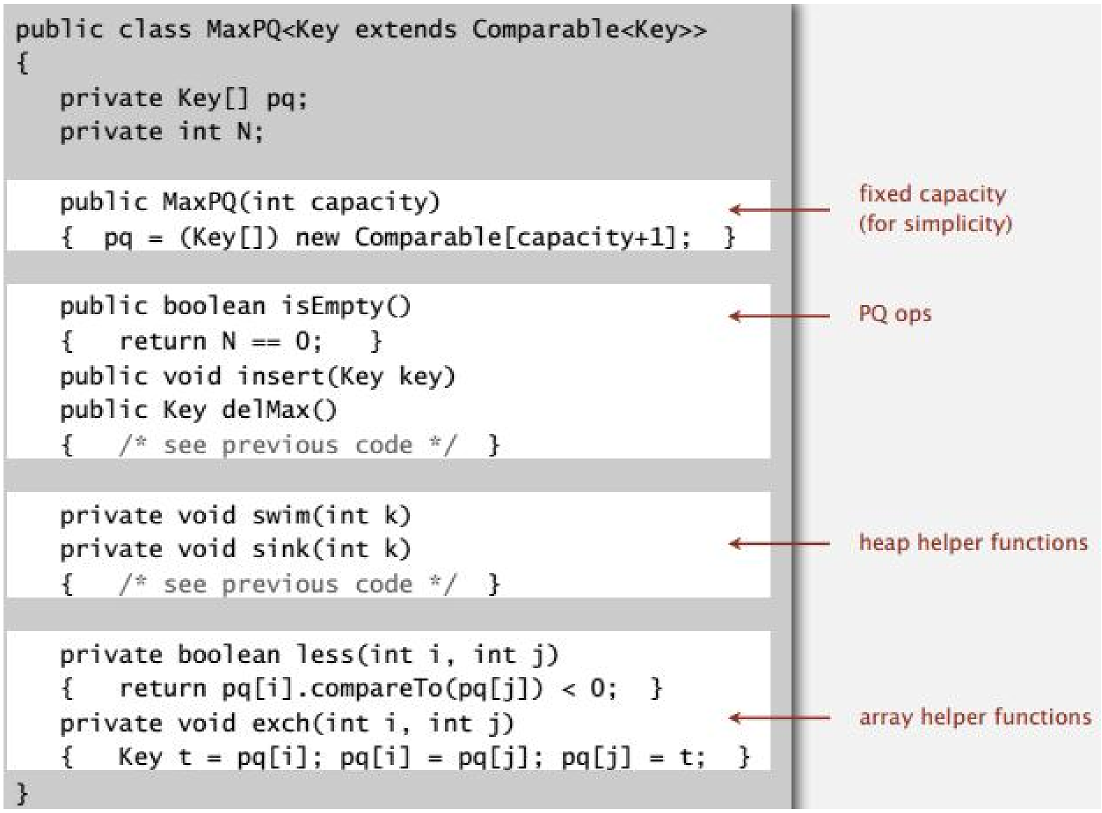
Example: delete **T**





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

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;  }
}
```

(Lecture 26) HeapSort

Basic plan for **in-place** sort:

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

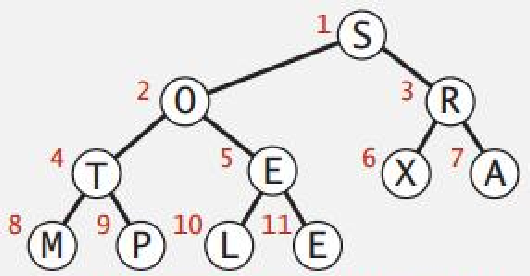
Heapsort demo:

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

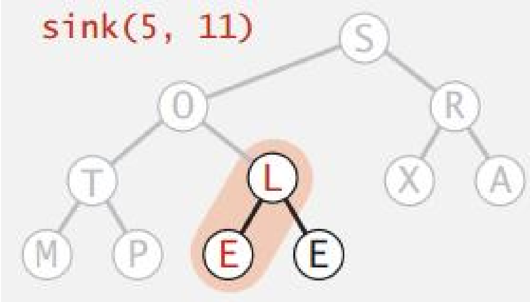
Array in arbitrary order

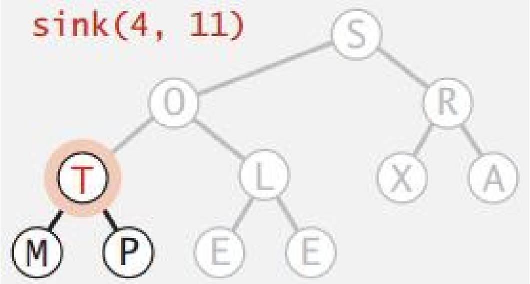
S	O	R	T	E	X	A	M	P	L	E
1	2	3	4	5	6	7	8	9	10	11

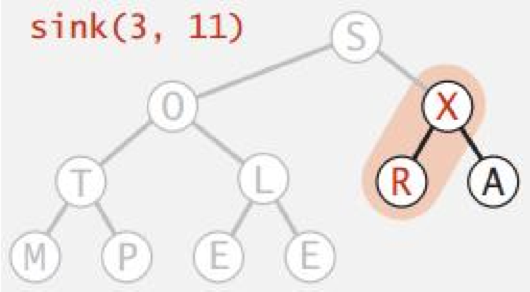
```
for (int k = N/2; k >= 1; k--)
   sink(a, k, N);
```

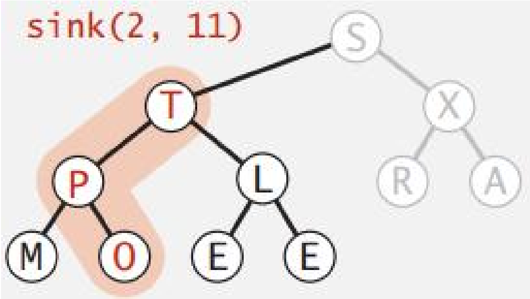


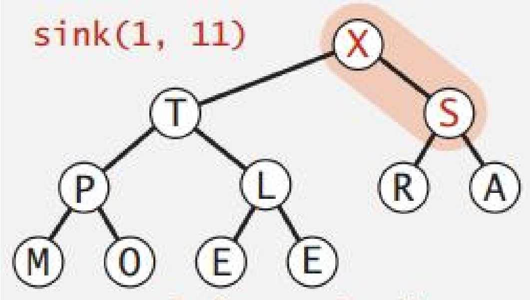
starting point (arbitrary order)







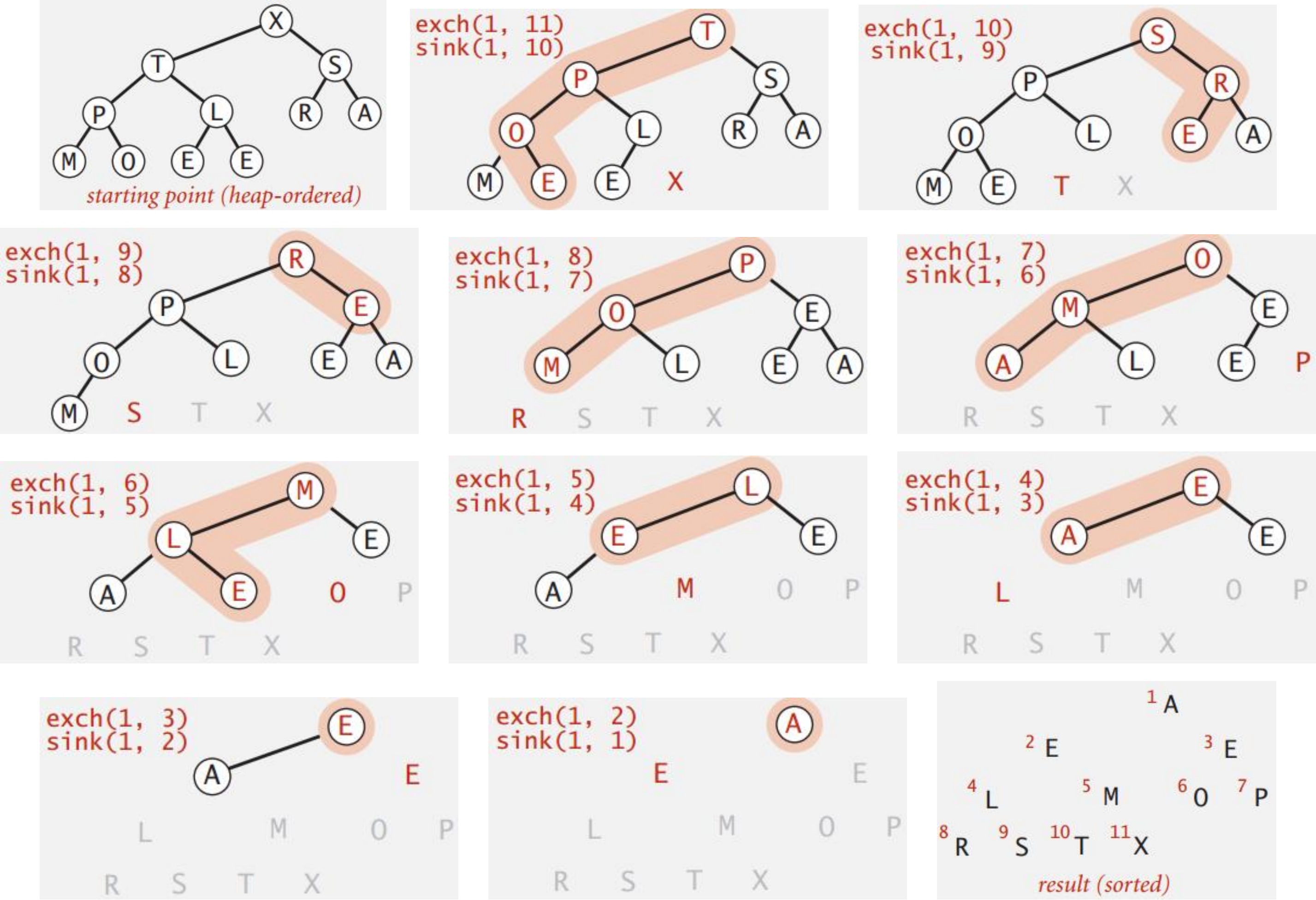




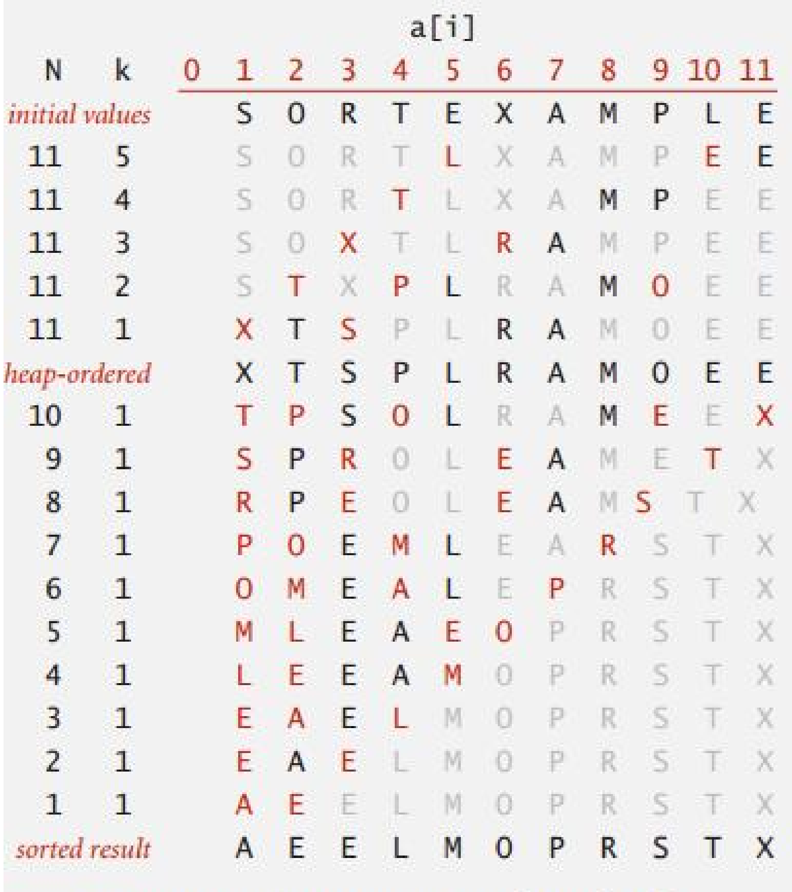
result (heap-ordered)

- **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



		a[i]											
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)

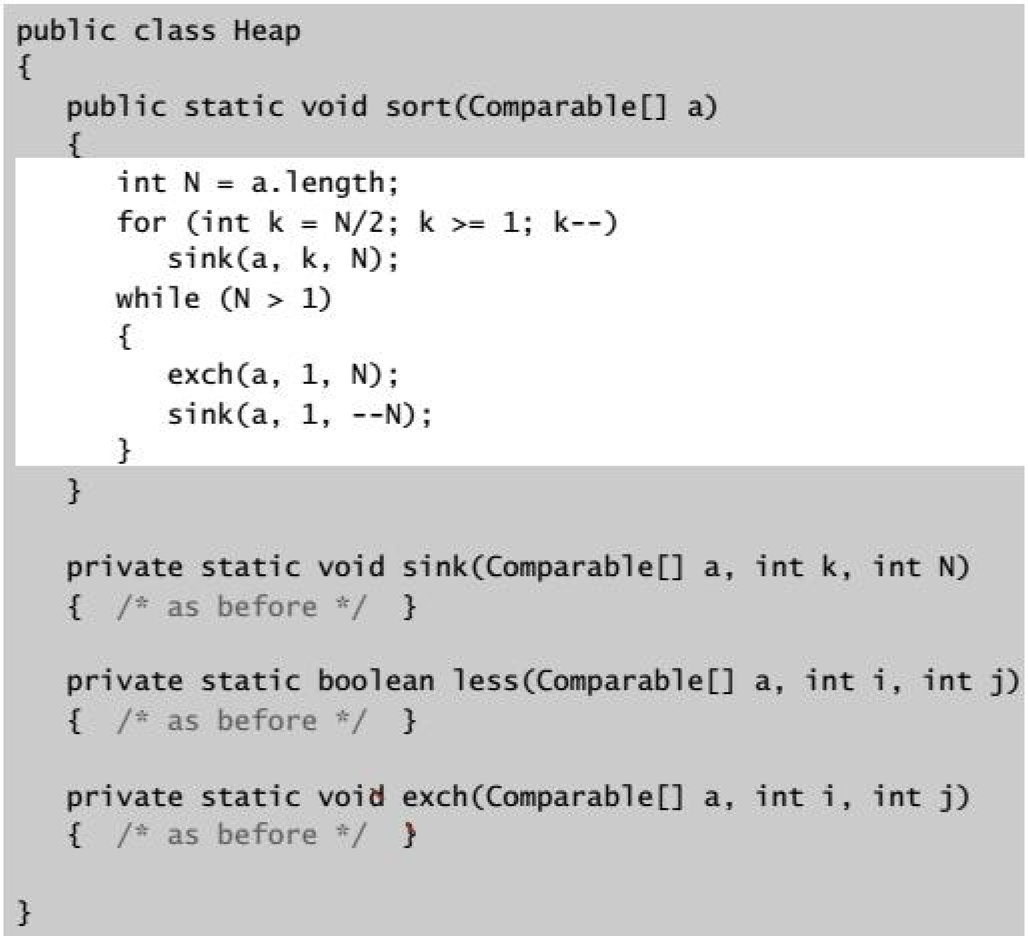
Heapsort: mathematical analysis

- 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



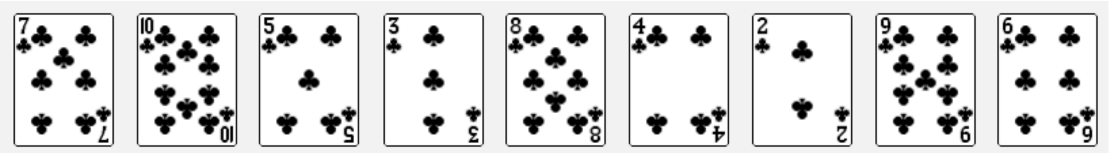
```
public class Heap
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int k = N/2; k >= 1; k--)
         sink(a, k, N);
      while (N > 1)
      {
         exch(a, 1, N);
         sink(a, 1, --N);
      }
   }

   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 */  }

}
```

(Lecture 27) Sorting I

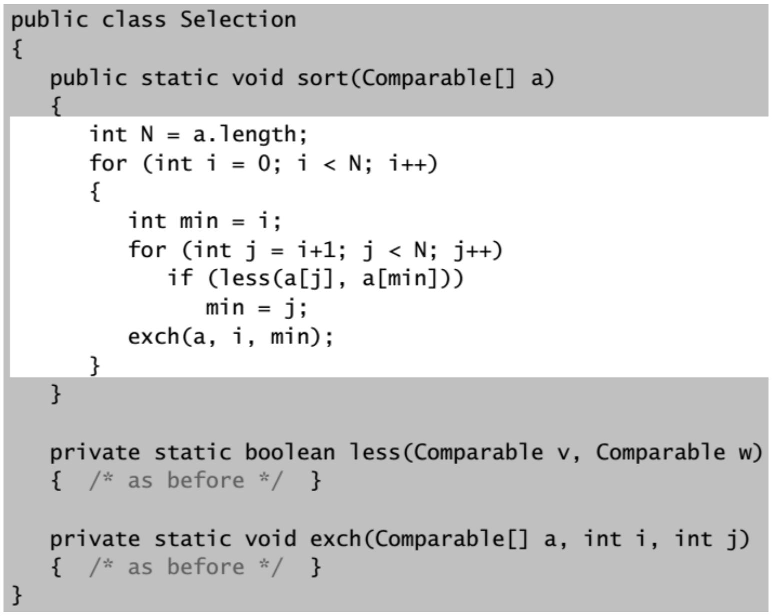
Selection Sort

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

Demo:



Java implementation:



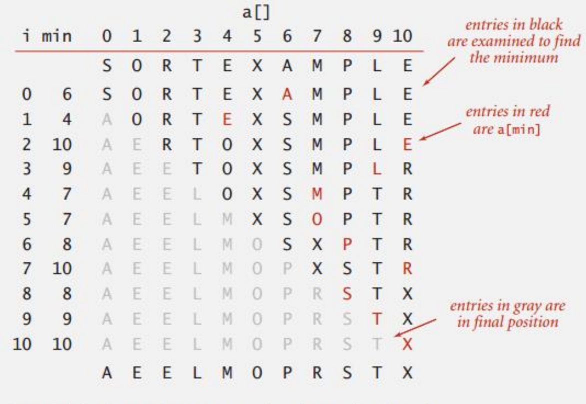
```
public class Selection
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int i = 0; i < N; i++)
      {
         int min = i;
         for (int j = i+1; j < N; j++)
            if (less(a[j], a[min]))
               min = j;
         exch(a, i, min);
      }
  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

   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 ~ N²/ 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.



		a[]										
i	min	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]

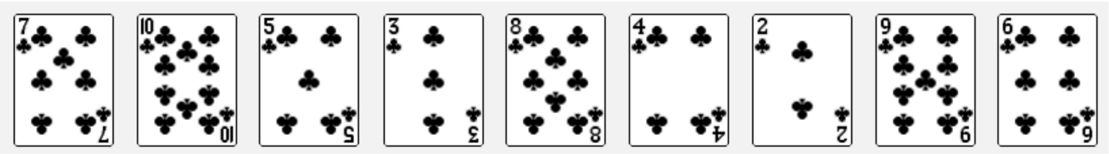
entries in gray are in final position

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

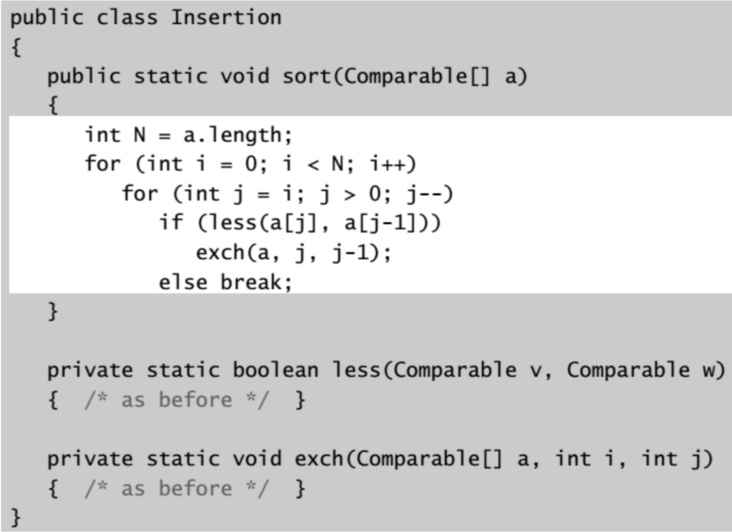
Insertion sort

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

Demo:



Java implementation:



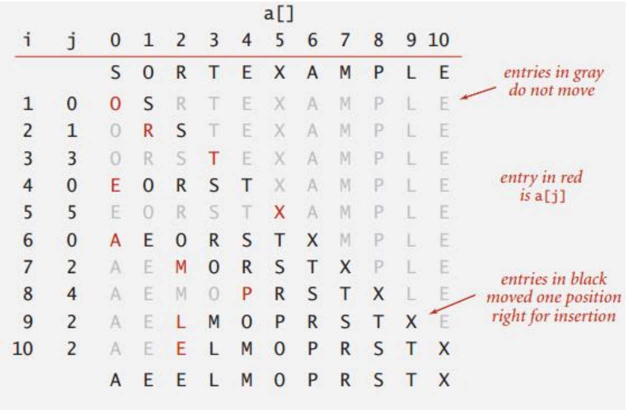
```
public class Insertion
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int i = 0; i < N; i++)
         for (int j = i; j > 0; j--)
            if (less(a[j], a[j-1]))
               exch(a, j, j-1);
            else break;
   }

   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 ~ ***¼ N^2*** compares and ~ ***¼ 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 ~ ***½ N^2*** compares and ~ ***½ 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***.

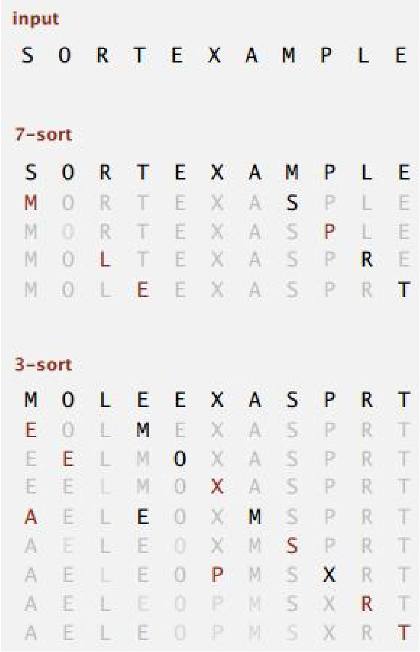
input	S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E
13-sort	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
4-sort	L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R
1-sort	A	E	E	E	H	L	L	L	M	O	P	R	S	S	T	X

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

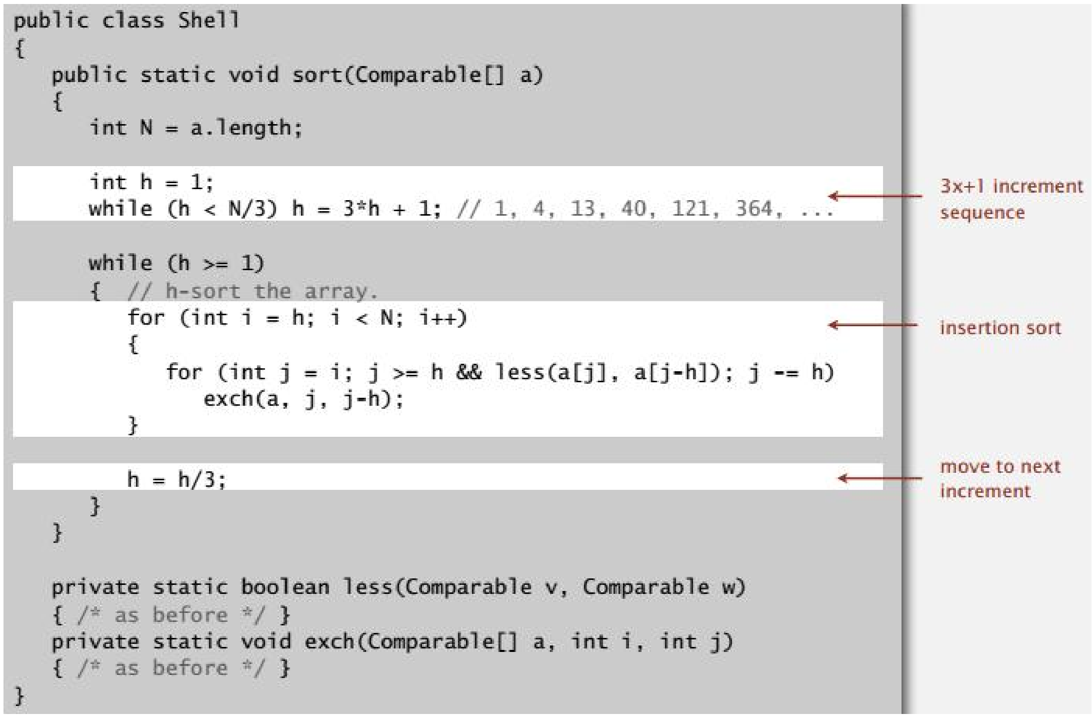
result

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($N^{3/2}$)***.

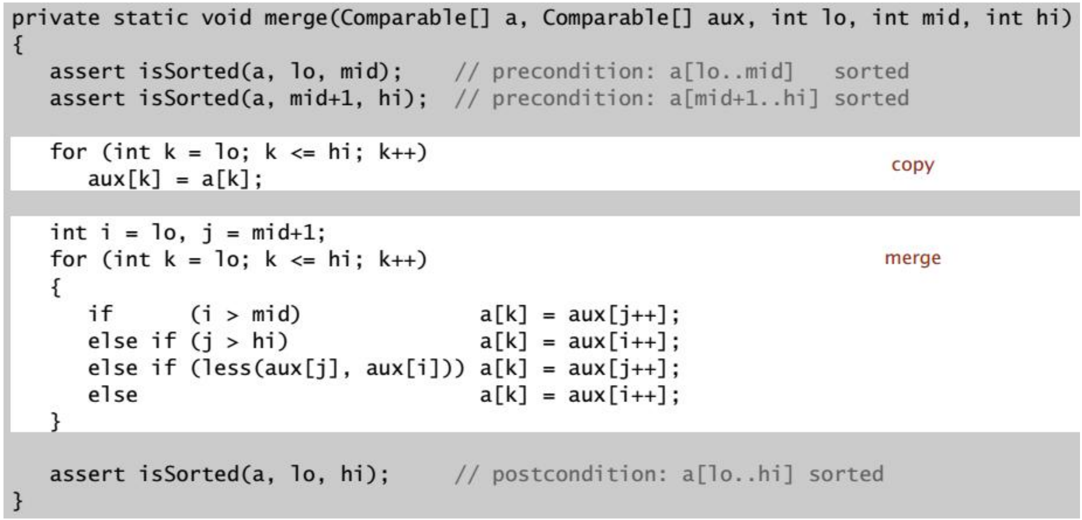
Mergesort

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

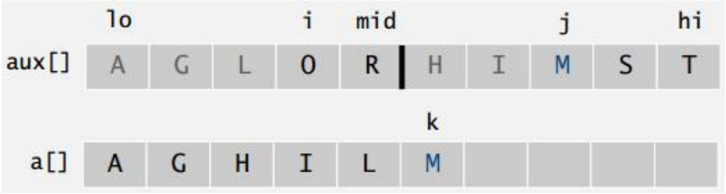
Mergesort overview

Java implementation:

Merging:

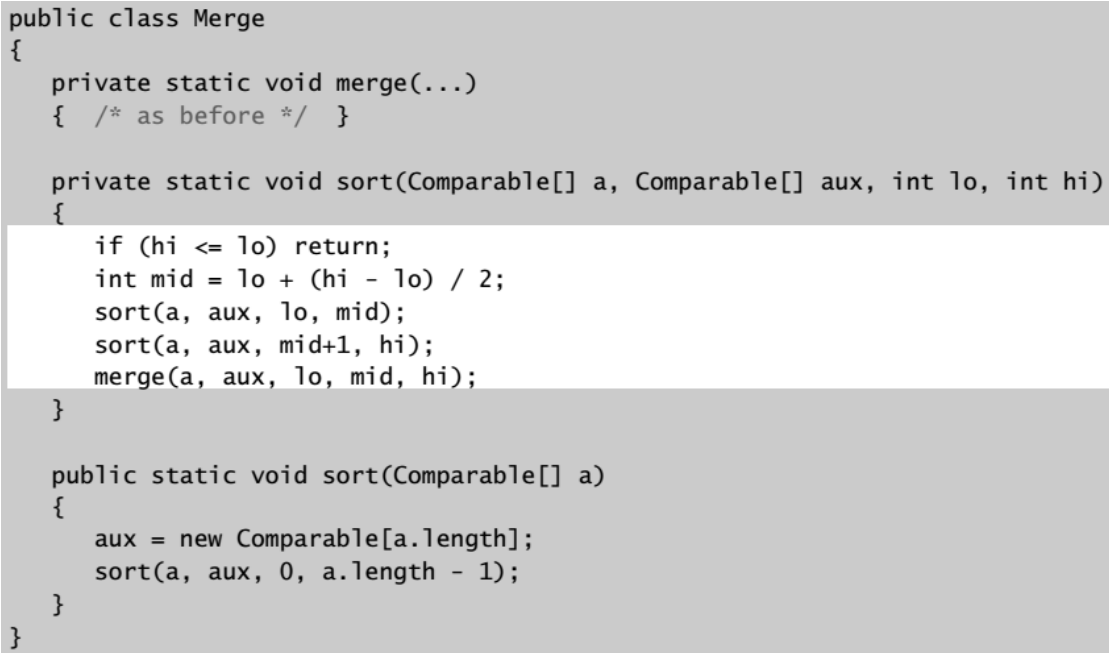


```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
   assert isSorted(a, lo, mid);    // precondition: a[lo..mid]   sorted
   assert isSorted(a, mid+1, hi);  // precondition: a[mid+1..hi] sorted

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

   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++)
   {
      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
}
```



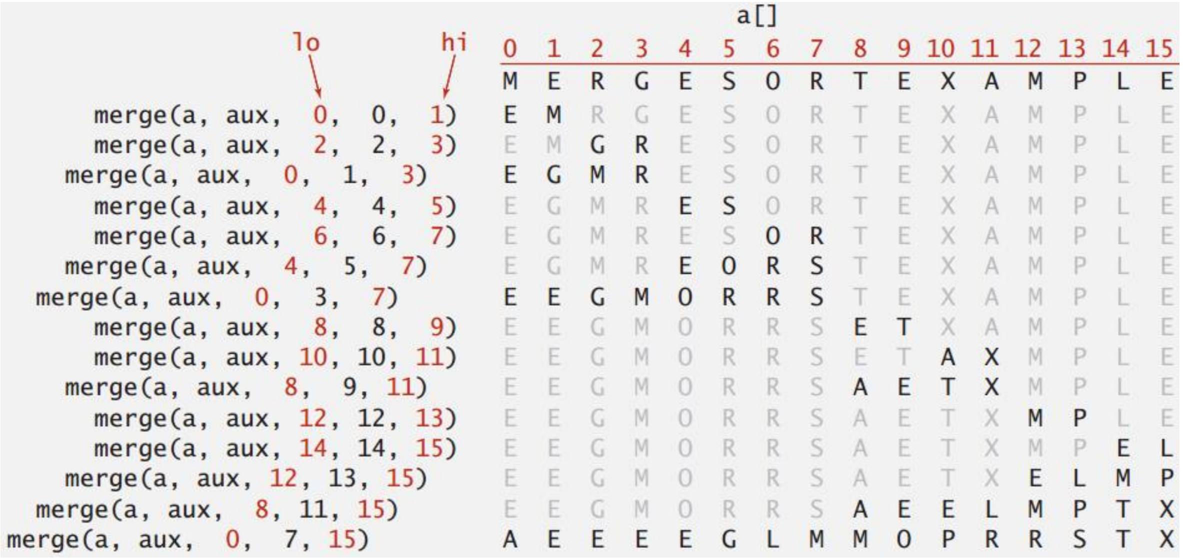
Java implementation:

Mergesort:



```
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);
   }
}
```

Mergesort: trace



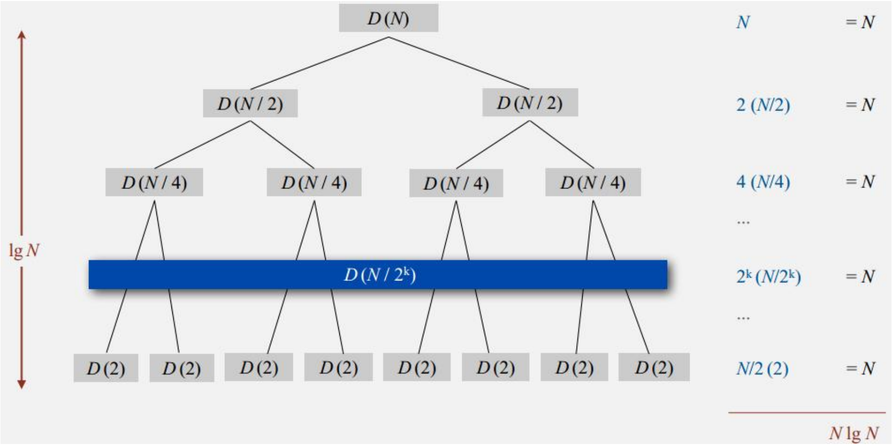
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 0, 1, 3)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 4, 4, 5)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux,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

Mergesort: 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

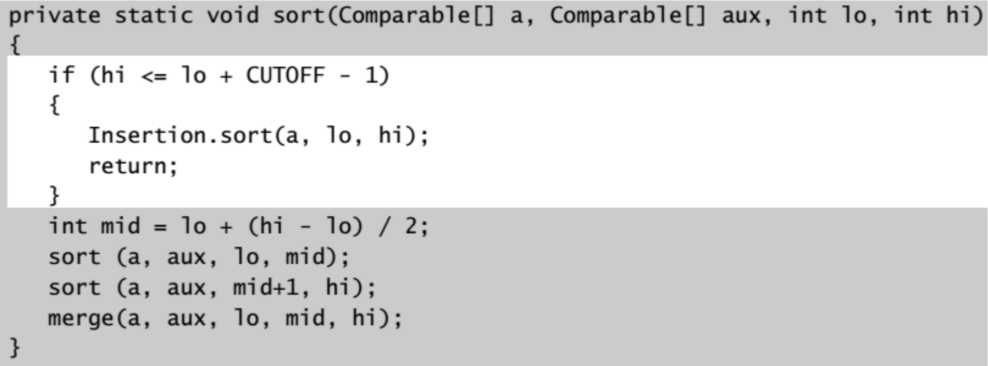


Mergesort analysis: memory (array accesses)

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

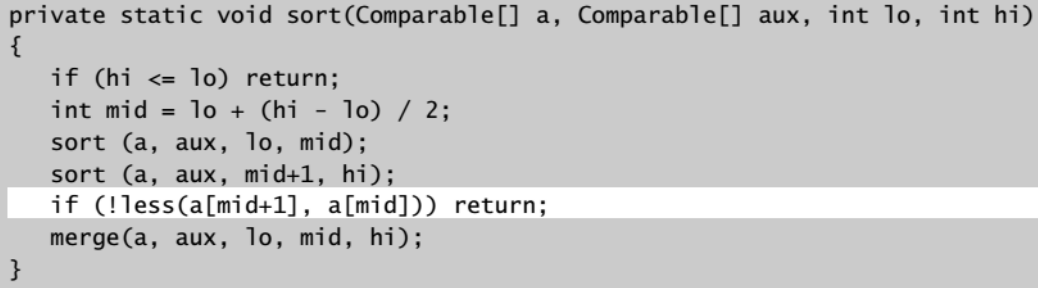
Mergesort: practical improvements

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



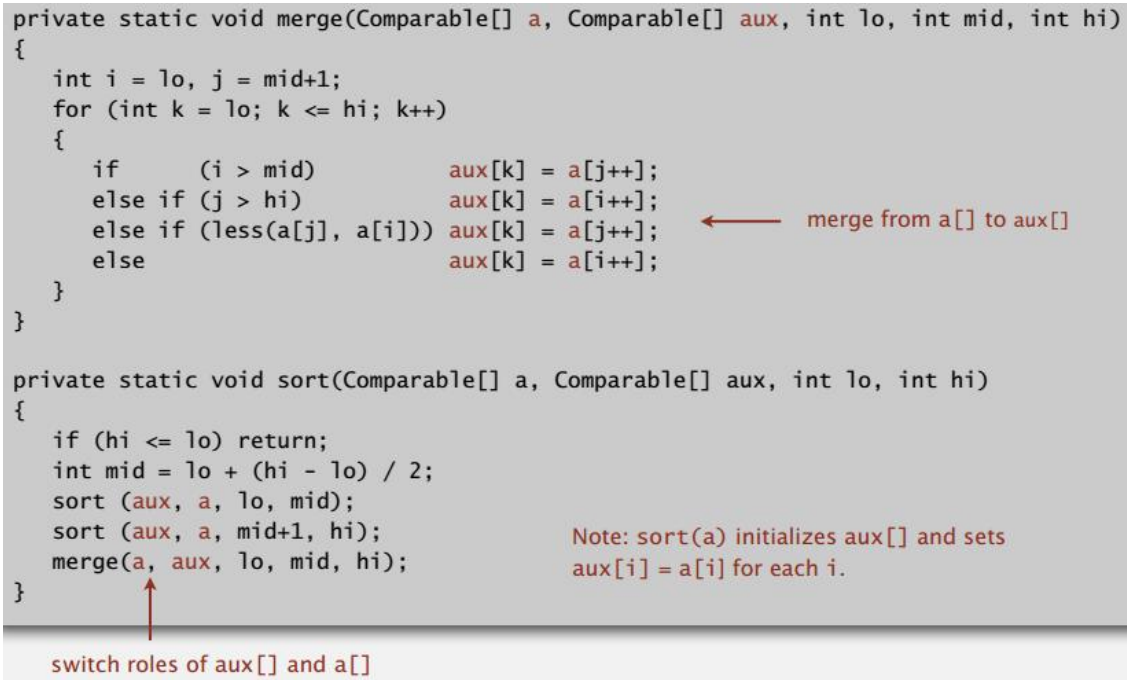
```
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
   if (hi <= lo + CUTOFF - 1)
   {
      Insertion.sort(a, lo, hi);
      return;
   }
   int mid = lo + (hi - lo) / 2;
   sort (a, aux, lo, mid);
   sort (a, aux, mid+1, hi);
   merge(a, aux, lo, mid, hi);
}
```

- 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++];
      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);
   merge(a, aux, lo, mid, hi);
}
```

merge from a[] to aux[]

Note: sort(a) initializes aux[] and sets aux[i] = a[i] for each i.

switch roles of aux[] and a[]

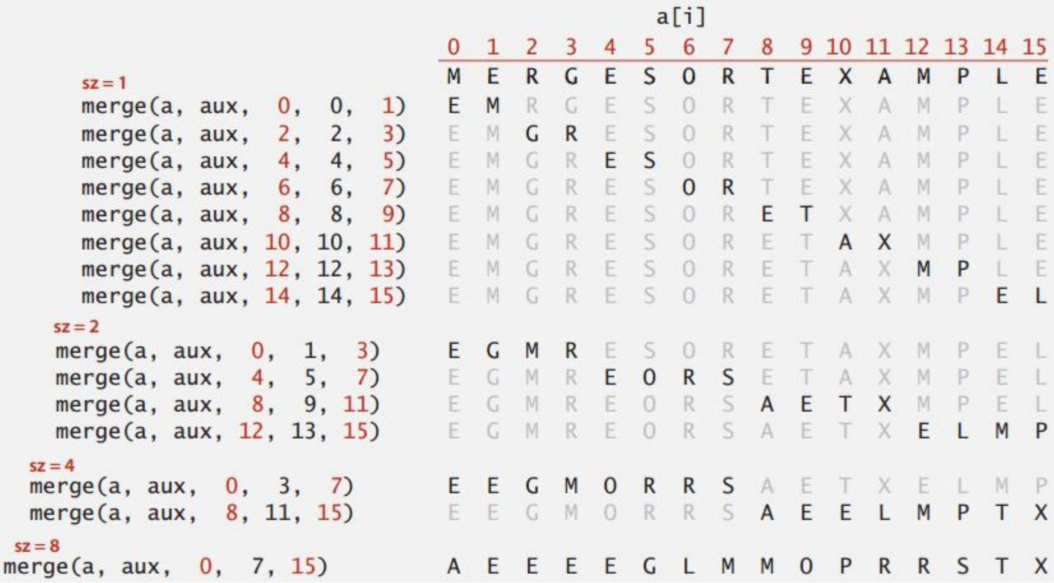
Complexity of sorting

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

Bottom-up Mergesort

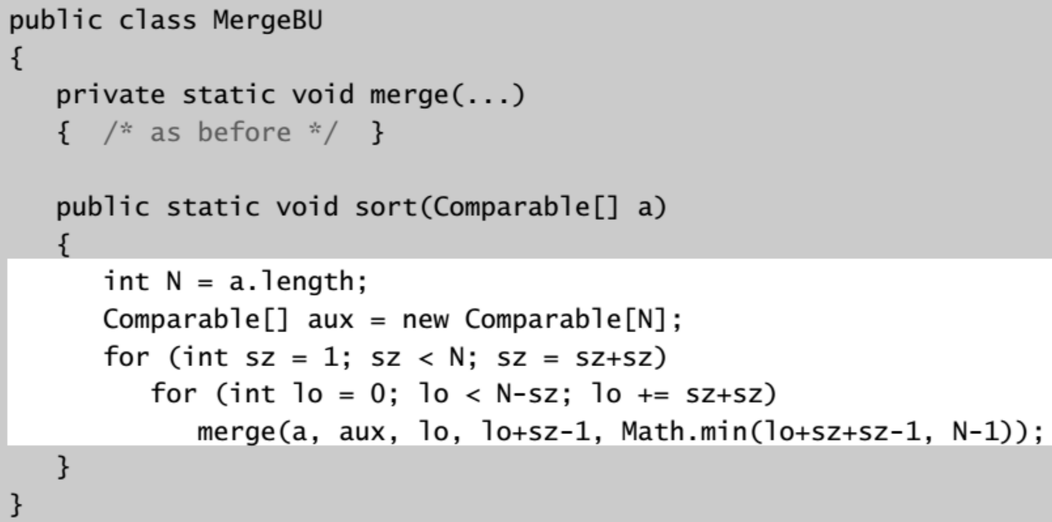
Basic plan:

- 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



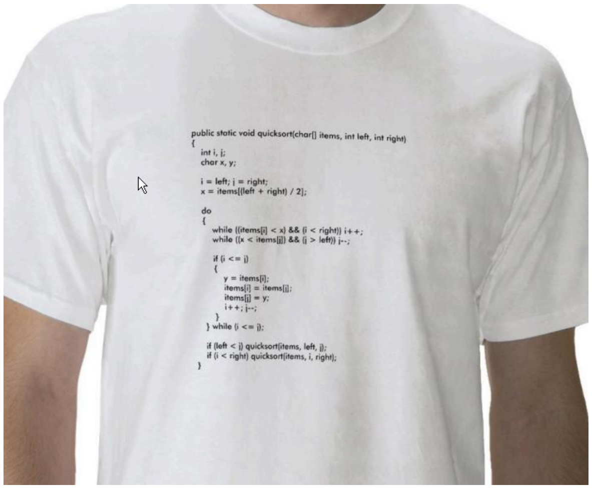
```
public class MergeBU
{
   private static void merge(...)
   {  /* as before */  }

   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));
   }
}
```

(Lecture 28) Sorting II

Quicksort

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.



```
public static void quicksort(char[] items, int left, int right)
{
   int i, j;
   char x, y;

   i = left; j = right;
   x = items[(left + right) / 2];

   do
   {
      while ((items[i] < x) && (i < right)) i++;
      while ((x < items[j]) && (j > left)) j--;

      if (i <= j)
      {
         y = items[i];
         items[i] = items[j];
         items[j] = y;
         i++; j--;
      }
   } while (i <= j);

   if (left < j) quicksort(items, left, j);
   if (i < right) quicksort(items, i, right);
}
```

Quicksort t-shirt

Quicksort partitioning demo

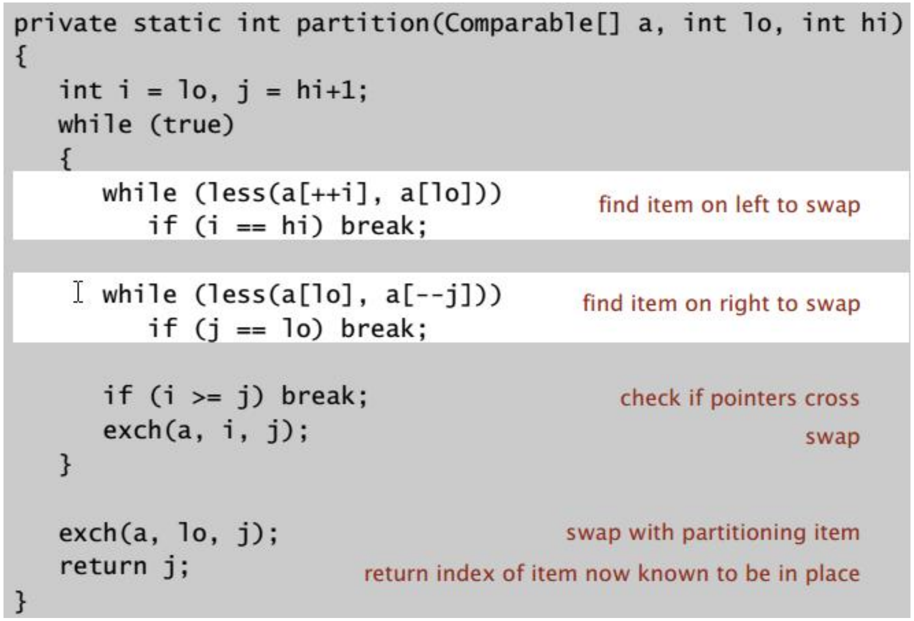
Repeat until ***i*** and ***j*** pointers cross.

- 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]))
         if (i == hi) break;

      while (less(a[lo], a[--j]))
         if (j == lo) break;

      if (i >= j) break;
      exch(a, i, j);
   }

   exch(a, lo, j);
   return j;
}
```

find item on left to swap

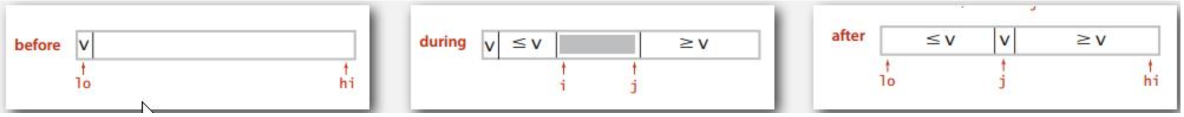
find item on right to swap

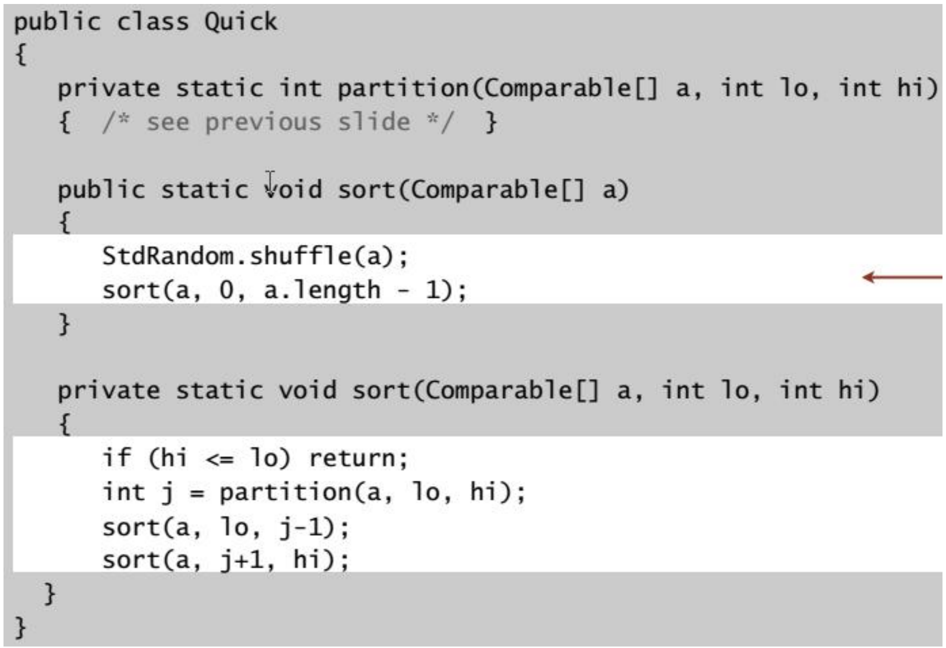
check if pointers cross

swap

swap with partitioning item

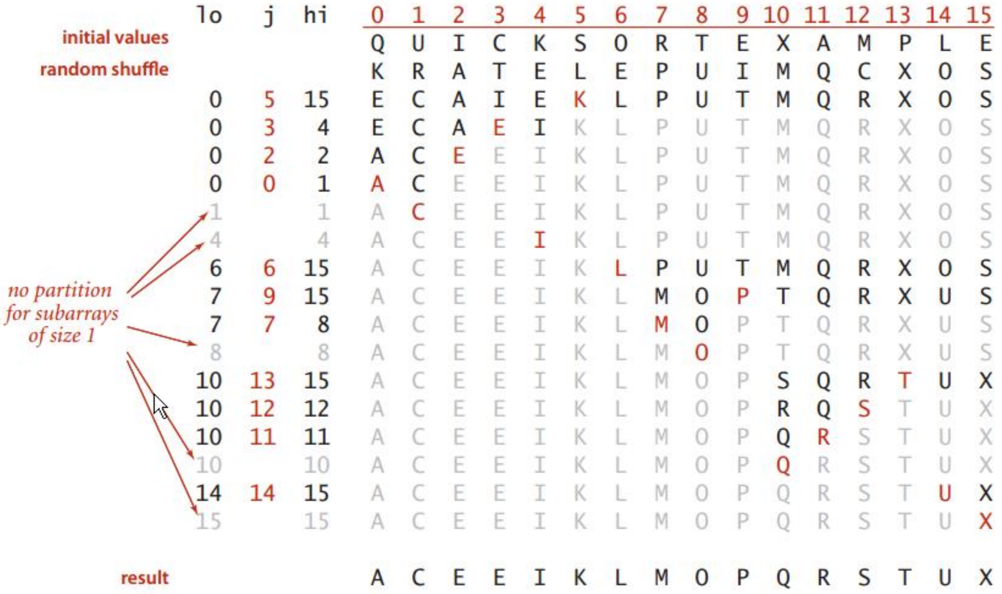
return index of item now known to be in place





```
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);
  }
}
```

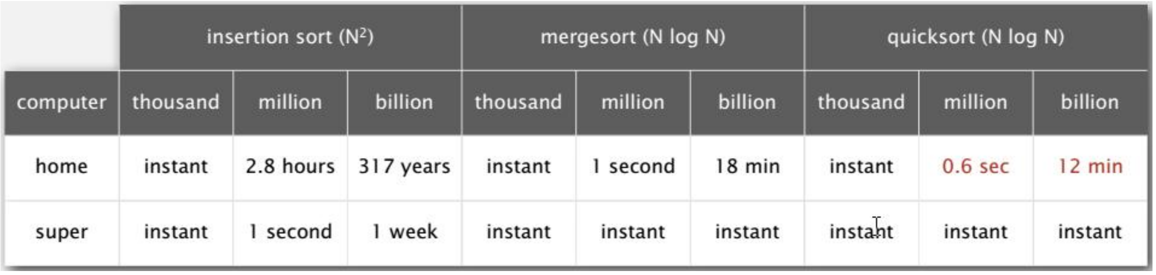
Quicksort trace



	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)

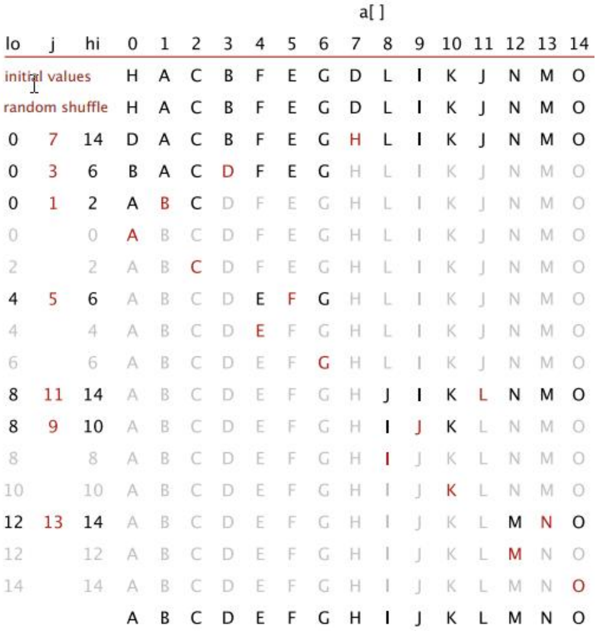
Quicksort: empirical analysis



	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

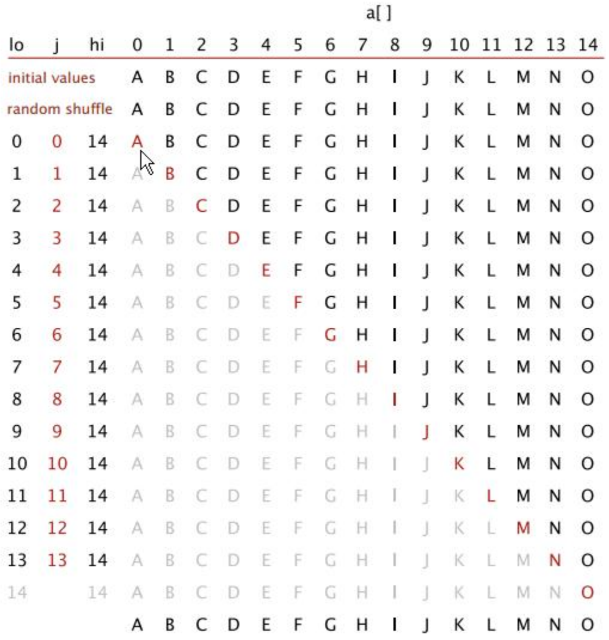
Best case: Number of compares is ~ ***N lg N***



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 ~ ***½ N^2***



			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 ln N***

Quicksort: summary of performance characteristics

Worst case: Number of compares is quadratic.

- ***$N + (N - 1) + (N - 2) + \cdots + 1 \sim \frac{1}{2} N^2$***
- 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.

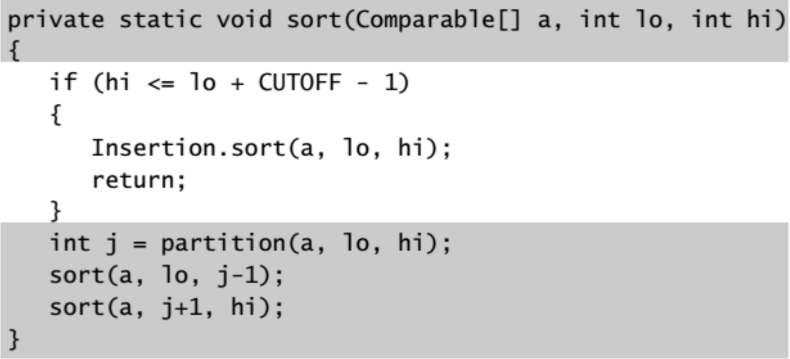
Quicksort is an **in-place** sorting algorithm.

Quicksort is **not stable**.

Quicksort: practical improvements

1- Insertion sort small subarrays:

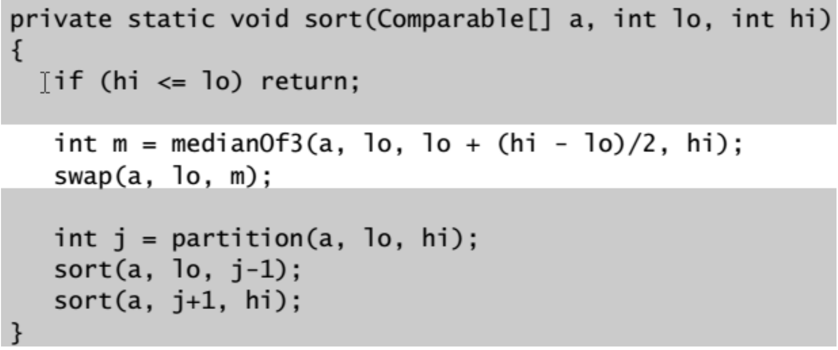
- Even quicksort has too much overhead for tiny subarrays.
- **Cutoff** to insertion sort for ≈ 10 items.
- Note: could delay insertion sort until one pass at end.



```
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);
}
```