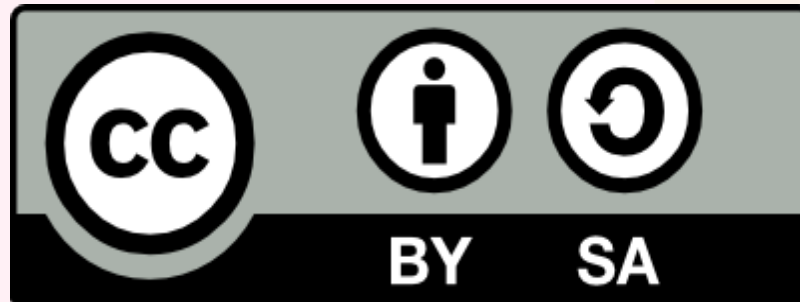


# Data Structures & Algorithms

**Dr. Pierre Vignéras**

<http://www.vigneras.name/pierre>



This work is licensed under a Creative Commons Attribution-Share Alike 2.0 France.

See

<http://creativecommons.org/licenses/by-sa/2.0/fr/>  
for details

# Class, Quiz & Exam Rules

- No entry after the first 10 minutes
- No exit before the end of the class
- Unannounced Quiz
  - After (almost) each end of a chapter/concept
  - At the beginning of a class
  - Fixed timing (you may suffer if you arrive late)
  - Spread Out (do it quickly to save your time)
  - Papers that are not strictly in front of you will be considered as done
  - Cheaters will get '-1' mark

# Outline

- I. Introduction/Definitions
- II. Arrays
- III. Stacks & Queues
- I. Linked List
- I. Trees
- II. Priority Queues
- III. Sorting
- IV. Searching
- V. Balanced Trees
- VI. Hashing
- VII. Graphs
- VIII. Graphs Algorithms

**Standard Data Structures**

**Standard Algorithms**

**Standard**

**Outline**

# Introduction/Definitions

# Introduction/Definitions

- Data
- Algorithms
- Performance Analysis

# Data

- VCR Example : interactions through buttons on the control panel (PLAY, FFW, REW, REC);
  - we can't **interact** with the **internal** circuitry, the **internal representation** is **hidden** from the end-user ==> *Encapsulation*
  - Instructions Manual tells only **what** the VCR is supposed to do, not **how** it is implemented ==> *Abstraction*



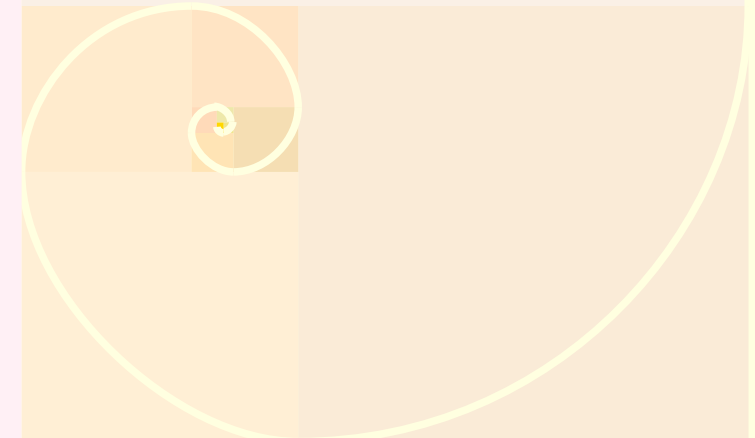
# Data

- Data Encapsulation or Information Hiding
  - is the concealing of the implementation details of a data object from the outside world.
- Data Abstraction
  - is the separation between the **specification** of a data object and its **implementation**
- Data Type
  - is a collection of **objects** and a set of **operations** that act on those objects



# Data

- Example: C++ fundamental data types
  - objects type: char, int, float and double
  - operations: +, /, -, \*, <, >, =, ==, ...
  - **Modifiers**
    - **short, long: amount of storage (8, 16, 32, 64 bits)**
    - **signed, unsigned: interpretation of the most significant bit of an *integer***





# Algorithms

- **An algorithm is a finite set of instructions that, if followed, accomplishes a particular task.**
  - (1)**Input:** Zero or more quantities are externally supplied
  - (2)**Output:** At least one quantity is produced
  - (3)**Definiteness:** Each instruction is clear and unambiguous
  - (4)**Finiteness:** If we trace out the instructions of an algorithm, then, for all cases, the algorithm terminates after a finite number of steps
  - (5)**Effectiveness:** every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in (3): it also must be feasible.

# Algorithms

- How to express algorithm? Many solutions
  - Natural language: must be well defined and unambiguous (what about portability?)
  - Graphic representations: flowcharts (only for small and simple algorithms)
  - Programming languages: low level implementation must be removed and replaced by natural language



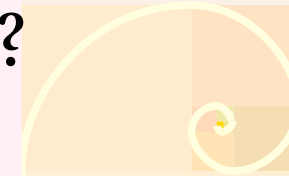
# Recursive Algorithms

- Limitation of recursion (only factorials, ackermann, fibonacci, ...)?
- A tool for theorician?
- Theorem: *"Any program that can be written using assignment, the if-else statement and the while statement can also be written using assignment, if-else and recursion."*
- Example: Fibonacci
  - $f(0)=f(1)=1$
  - $f(n) = f(n-1) + f(n-2)$



# Performance Analysis

- How to judge a program?
  - Does it do what we want it to do?
  - Does it work correctly according to original specifications of the task?
  - Is there documentation that describes how to use it and how it works?
  - Are the functions created in such way that they perform logical subfunctions?
  - Is the code readable?



# Performance Analysis

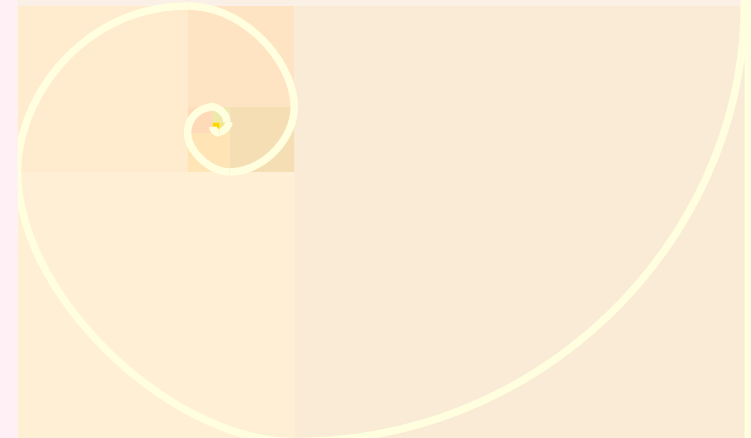
- From a performance point of view, we define two criteria:
  - Space complexity: the amount of memory needed by a program to run to completion
  - Time complexity: the amount of computer time needed by a program to run to completion
- Two phases in performance evaluation
  - performance analysis: a priori estimates;
  - performance measurement: a posteriori testing.

# Space Complexity

- The space needed by a program is seen to be the sum of two components
  - fixed part: independent of the characteristics (e.g. number, size) of the inputs and outputs
    - instruction space (space of the code itself)
    - space for constants, ...
  - variable part: dependent on the particular problem instance being solved, hence on the inputs and outputs characteristics
    - variables whose size depends on inputs/outputs,
    - recursion stacks (when it depends on inputs/outputs)

# Space Complexity

- $S(P) = c + S_P$ 
  - $c$  is constant, it represents the fixed part, it is not very interesting!
  - $S_P$  represents the variable part. Focus on it!
- Decide which characteristics to use to measure space requirements
  - Problem specific!



# Space Complexity

## Sum example

```
int sum(int* a, int n) {  
    int s = 0;  
    for (int i = 0; i < n; i++) {  
        s += a[i];  
    }  
    return s;  
}
```

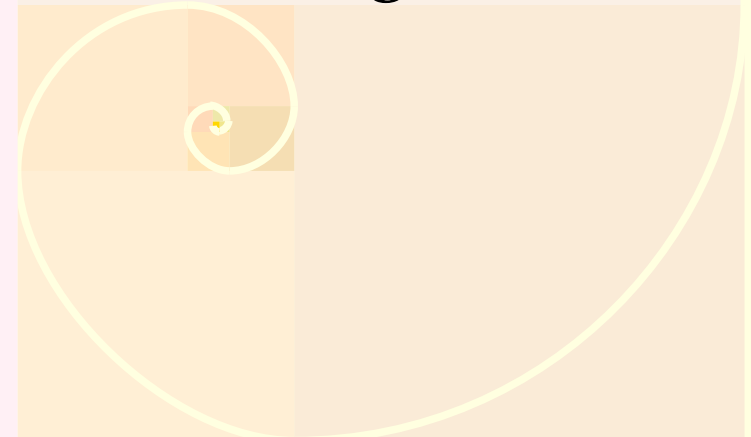
- Instance Characteristic:  $n$
- *How many space does it need?*
- *What would be the space required is the array is passed by copy?*

# Time Complexity

- $T(P) = c + T_P$ 
  - C is a constant representing the compile time
    - Do not take it into account!
  - $T_P$  represents the *runtime*, focus on it!
- Very hard to evaluate  $T_P$  exactly!
  - Suppose the compiler is well known
$$T_p(n) = C_a \cdot \text{Add}(n) + C_m \cdot \text{Mul}(n) + \dots$$
    - Time needed for addition, multiplication often depends on the actual numbers

# Time Complexity

- Try to guess the time complexity experimentally
  - program is typed, compiled and run on a specific machine. Execution time is physically clocked,
  - $T_{P(n)}$  is measured...
  - But, the value measured is inaccurate (multiuser systems, system load, number of running programs, ...)
- Consider only *steps*



# Steps

- A program step is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics
- Example:  
return  $(1+2+4) / (5+6+7) * a$ ;  
is a single step if  $a$  is independent of the instance characteristics.
- How to count steps?

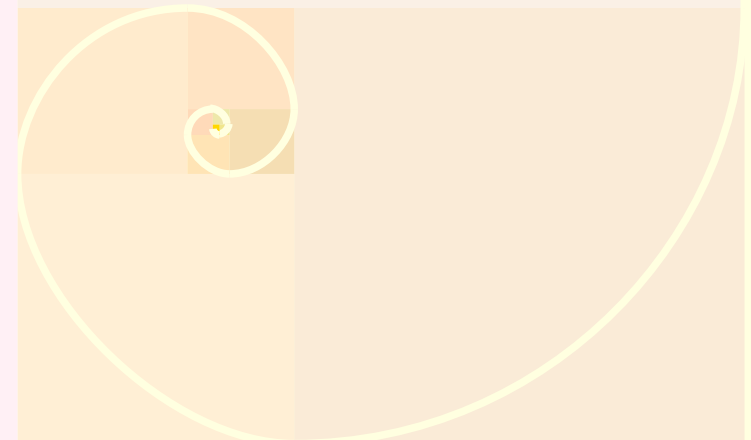
# Program Modification

- Introduce a new global variable in the original program that count the number of steps.
- Example : Sum



# Using a Step Table

- Create a table in which, for each line of code, you write the number of steps per execution and the frequency each statement is executed.
- Example : Sum

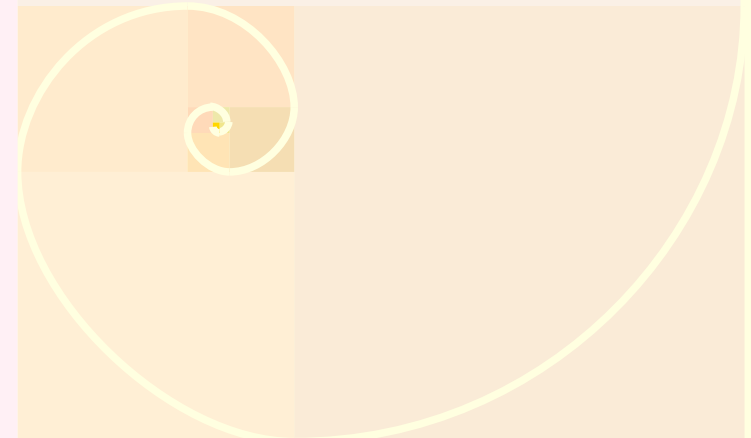


# Limitations of Exact Evaluation Performance

- Majority of real cases are not so simple
  - Time complexity may not depend only on the number of inputs/outputs but also on the value of one or many of them
    - Example: `int search(int* a, int n, int x);`
      - Instance characteristic:  $n$
      - $T_p(n)$  depends on  $a$ ,  $n$  and  $x$  !!
- Consider only three cases:
  - Best-case: minimum number of steps required
  - Worst-case: maximum number of steps possible
  - Average step count: guess !

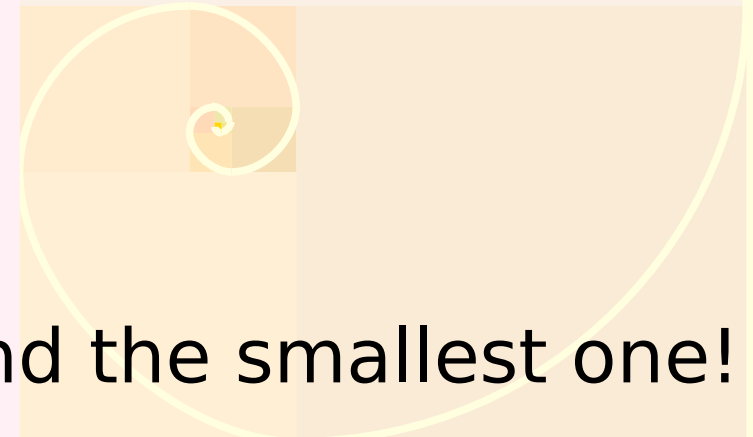
# Rough Comparisons

- Exact step count inaccurate anyway (what is a step?)
- Having a rough estimate is usually sufficient for comparison (but inexact) !!
  - A1 performs in  $c_1 \cdot n^2 + c_2 \cdot n$
  - A2 performs in  $c_3 \cdot n$
  - Which performs best?



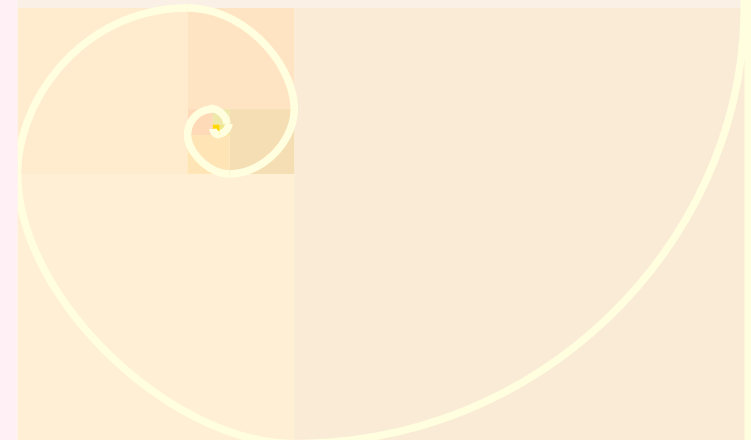
# Asymptotic Notation (O)

- $f(n) = O(g(n))$  iff there exist  $c > 0$  and  $n_0 > 0$  such that  $f(n) \leq c \cdot g(n)$  for all  $n, n \geq n_0$ 
  - $3n+2?$   $10n^2+4n+2?$
- $O(1)$ : constant time
- $O(\log(n))$ : logarithmic time
- $O(n)$ : linear time
- $O(n \cdot \log(n))$ : almost linear time
- $O(n^2)$ : quadratic time
- $O(n^3)$ : cubic time
- $O(2^n)$ : exponential time
- $g(n)$  is an upper bound, find the smallest one!



# Asymptotic Notation ( $\Omega$ )

- $f(n) = \Omega(n)$  iff there exist  $c > 0$  and  $n_0 > 0$  such that  $f(n) \geq c \cdot g(n)$  for all  $n, n \geq n_0$ 
  - $3n+2$ ?  $10n^2+4n+2$ ?
- $g(n)$  is a lower bound, find the largest one!
- Theorem: if  $f(n) = a_m n^m + \dots + a_1 n + a_0$ 
  - $f(n) = O(n^m)$
  - $f(n) = \Omega(n^m)$  **if  $a_m > 0$**

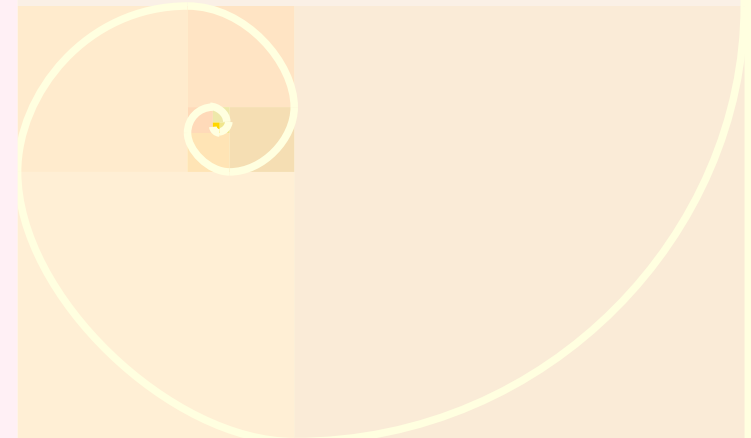


# Asymptotic Notation ( $\Theta$ )

- $f(n) = \Theta(n)$  iff there exist  $c_1 > 0$ ,  $c_2 > 0$ , and  $n_0 > 0$  such that  $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$  for all  $n$ ,  $n \geq n_0$ 
  - $3n+2$ ?  $10n^2+4n+2$ ?
- $g(n)$  is **both** an upper and lower bound of  $f(n)$
- Theorem: if  $f(n) = a_m n^m + \dots + a_1 n + a_0$ 
  - $f(n) = \Theta(n^m)$  **if  $a_m > 0$**
- **Example: sum**

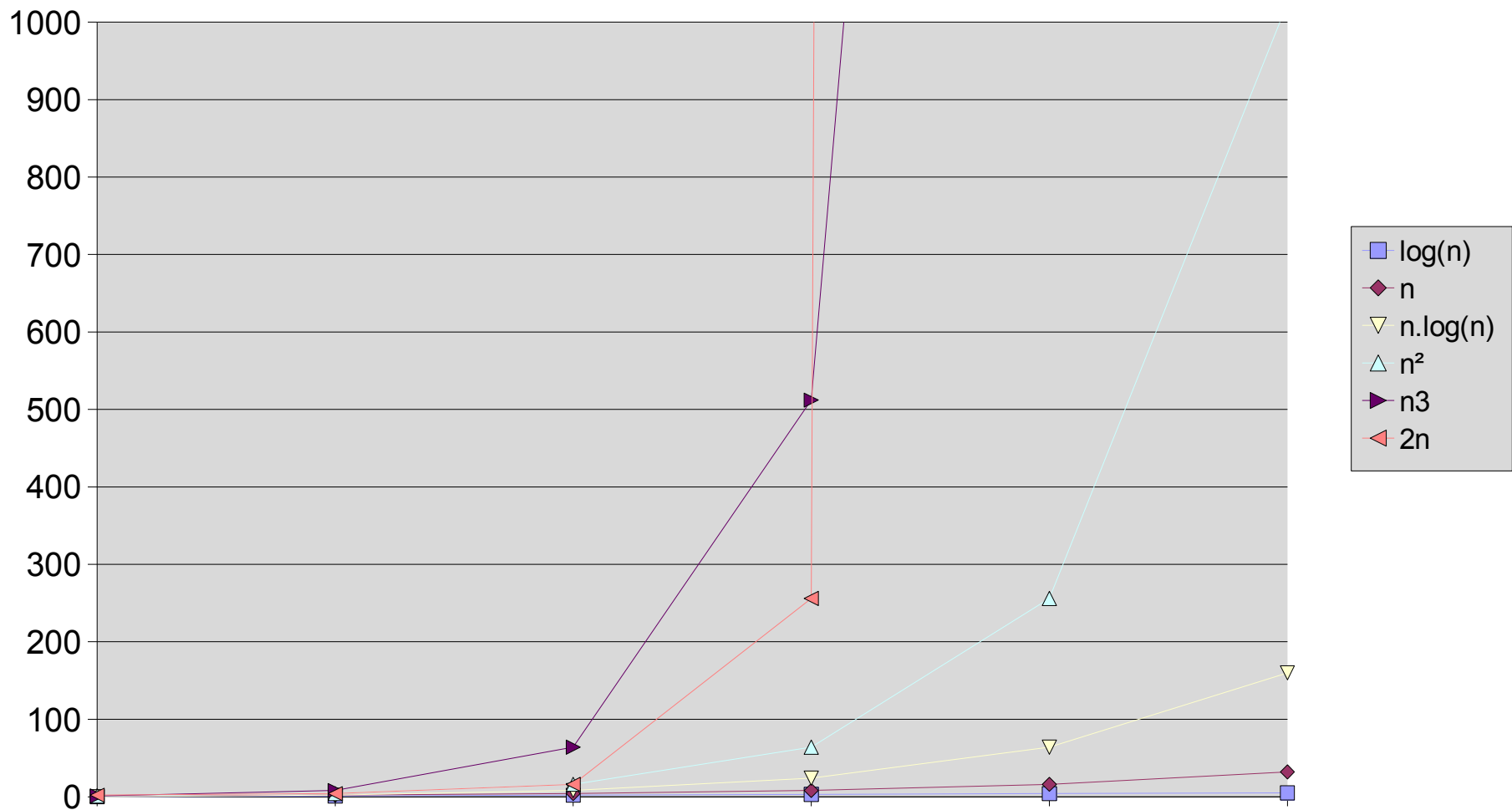
# Practical Complexities

$\log(n)$	$n$	$n \cdot \log(n)$	$n^2$	$n^3$	$2^n$
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296



# Practical Complexities

## Graph Overview



# Performance Measurement

- **Depends on several factors**
  - compiler used
  - architecture (processor, memory, disk, cache, ...)
  - operating system
  - load (number of users, number of running processus, etc.)
- **Hard to reproduce**
  - Averaging many experiments (10 and more)
  - Which values of  $n$ ? Higher  $n$   
==> conformance to asymptotic analysis.

# Performance Measurement

- **Needs a function time()**
  - **Accuracy?**
    - **To time a short event, it is necessary to repeat it several times and divide the total time for the event by the number of repetitions.**
- **What are we measuring?**
  - **best case, worst case or average?**
    - **Suitable test data need to be generated**
    - **Not always easy. Use random data if possible. Use a good random number generator.**

# Arrays

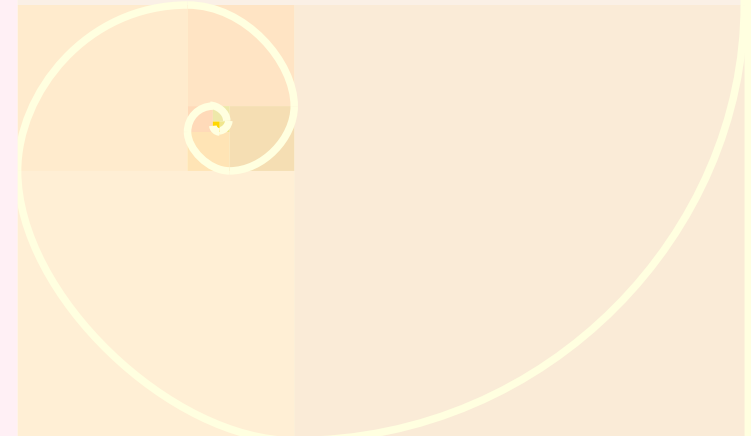
# Arrays

- Definition
  - A mapping  $\langle \text{index}, \text{element} \rangle$
- Operations
  - Creation/Deletion
  - Getting a value
  - Setting a value
- ***Random Access Order***
  - *get(i): 'x = a[i]'*
  - *set(i): 'a[i]=x'*
  - ***Warning: index bounds?***



# Array Data Structure Interface (C language)

```
----- C File: array.h -----  
typedef struct array* array;  
extern array array_new(int size);  
extern void array_delete(array a);  
extern void* array_get(array a, int i);  
extern void array_set(array a, int i, void* v);
```



# Using Arrays

- Ordered, linear list
  - Days of the week: (Sunday,... Saturday)
  - Values in a deck of cards (Ace, 2, ...,10, Jack, Queen, King)
  - Years France won the Cricket World Cup: '()'
    - an empty list is still a list!
- Operations on list
  - length, read from left (or right to left)
  - Get/Set the  $i$  th element ( $0 \leq i < n$ )
  - Insert/Delete at the  $i$  th position ( $0 \leq i < n$ )

# Polynomial Representation

- How to represent efficiently (space, time)

$$A(x) = 3x^2 + 2x + 4, \quad B(x) = x^{100} + 1$$

- Operations:

$$A(x) = \sum a_i \cdot x^i$$

$$B(x) = \sum b_j \cdot x^j$$

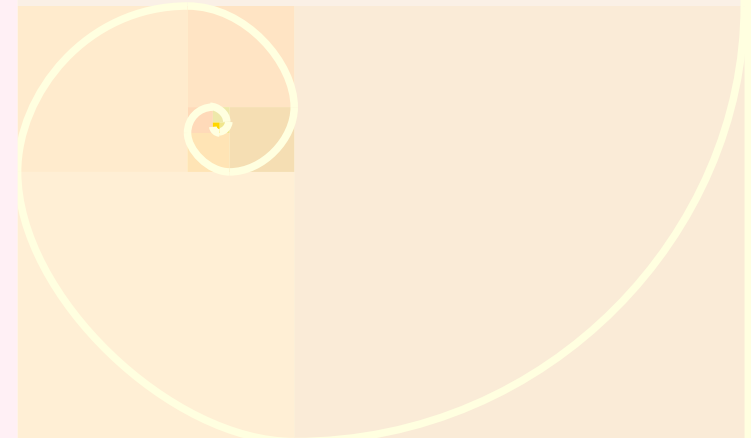
$$A(x) + B(x) = \sum (a_i + b_i) x^i$$

$$A(x) \cdot B(x) = \sum (a_i \cdot x^i \cdot \sum (b_j \cdot x^j))$$

# Polynomial Representation #1

```
struct poly {  
    int degree;    // degree < MaxDegree !!  
    float coef[MaxDegree + 1];  
};  
// MaxDegree: constant
```

- Very simple, inefficient!
  - Consider when  $\text{degree} \ll \text{MaxDegree}$
  - Complexity in (unused) space ?!



# Polynomial Representation #2

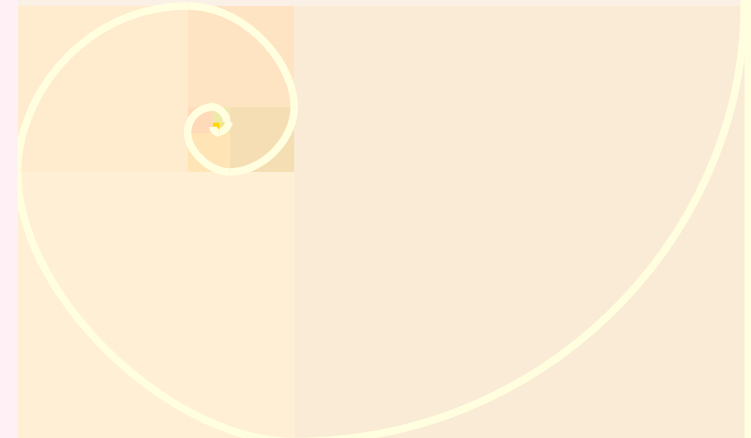
```
struct poly {
    int degree;    // degree < MaxDegree !!
    float* coef;
};
typedef struct poly* poly;
poly poly_create(int d) {
    poly p = malloc(sizeof(*poly));
    p->degree = d;
    p->coef = malloc(d*sizeof(*p->coef));
    return p;
}
```

- Still inefficient (sparse polynomial):  $B(x) = x^{100} + 1$

# Polynomial Representation #3

```
struct monom {
    int degree;
    float coef;
};
static struct monom GlobalArray[MaxTerms];
static int free;
struct poly {
    int start, end;
};
```

- $A(x)=3x^2+2x+4$ ,  $B(x)=x^{100}+1$  representations



# Polynomial Representations

- Which representation is the best
  - Space complexity?
  - Time complexity?
  - May depend on polynomials used (sparse)
- Global (static) variable representing maximum instances of a data structure is bad design
  - Dynamism is the key!! Provide it!
  - Allocate an array of monoms for each polynomial
    - Space complexity?
    - Time complexity (addition for example)?

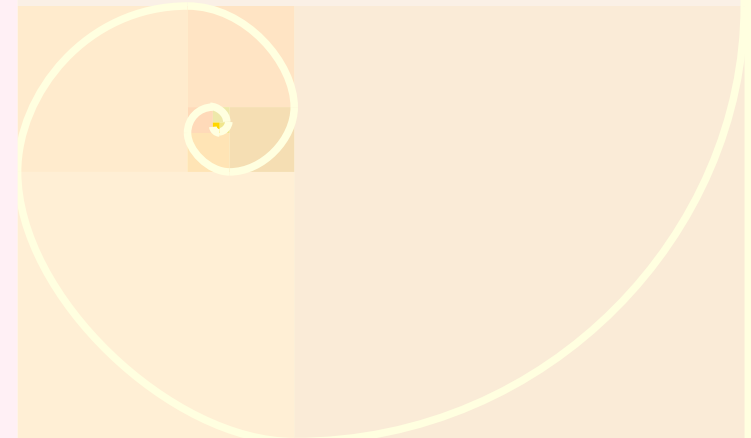
# Multidimensional Arrays

- Memory is a single array of word cell
- Any data has a word array internal representation
- Represents explicitly multidimensional array into a single array
  - Example: 2 dimensional array
    - $A[][]$ : dimension  $(n,p)$  (row, columns)
    - $A[i][j] \rightarrow a[k], k = i*p+j$



# Strings

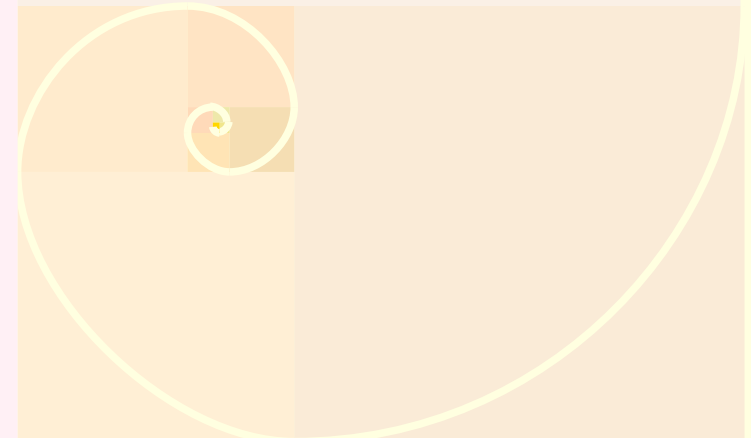
- Internal string representations
  - arrays of « char »
  - size of the string?
    - field of a structure (Java)
    - s[0] (Pascal)
    - Ends by a special character (C language: '\0')
- Operations
  - length(), replace(),
  - concat(), delete()
  - find()



# String Pattern Matching

## Simple Algorithm

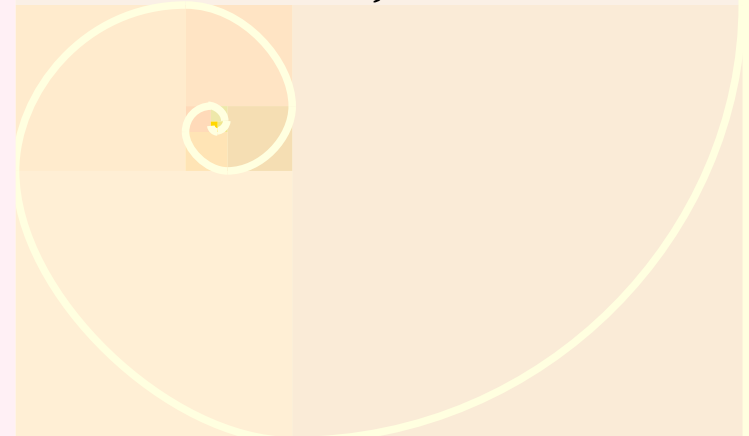
- Two strings 's' and 'p'
- 'p' is a pattern to be searched for in 's'
- `int find(char* s, char* p)`
  - returns -1 if 'p' is empty or if 'p' is not a substring of 's'
  - returns index 'i' such that 'p' matches the substring of 's' at position 'i' otherwise
- Simple Algorithm



# String Pattern Matching

## Simple Algorithm

- Improvement:  $\text{while}(i \leq |s| - |p|)$
- Space Complexity:  $O(1)$
- Time Complexity (comparisons):
  - Best case:  $O(|p|)$
  - Worst case:  $O((|s| - |p|) \cdot |p|)$
  - Common case:
    - $|s| \gg |p|$ , Complexity  $\sim \Omega(|s|)$  (Lower bound)



# String Pattern Matching

## Knuth, Morris, Pratt

- Keeping memory
  - $S = 100101$
  - $P = 100000$
  - On a failure of length  $j$ , where shall we start our next comparison in  $S$ ?
  - We know that  $j$  characters of  $S$  match  $P$
  - None of the  $j-1$  other characters of  $S$  can match the first character of  $P$
- Start comparing the  $j$ th character after the current one in  $S$

# String Pattern Matching

## Knuth, Morris, Pratt

- Definition
  - Alphabet  $A$  of symbols (characters)
  - String 'x', where 'x[i]' is the 'i'th character of 'x'
  - (Proper) Prefix, (Proper) Suffix, Border
- Example:  $x=abacab$ 
  - Proper Prefix: (), a, ab, aba, abac, abaca
  - Proper Suffix: (), b, ab, cab, acab, bacab
  - Border: (), ab  $\rightarrow |()|=0, |ab|=2$
- () is always a border of any non empty string, it has no border itself

# String Pattern Matching

## Knuth, Morris, Pratt

- Example:

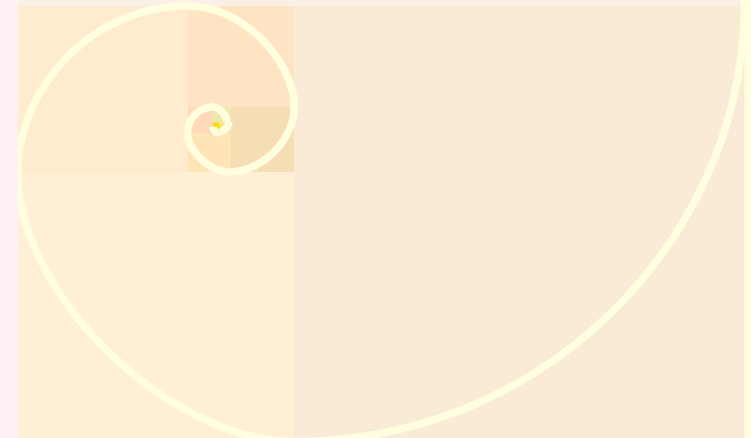
0	1	2	3	4	5	6	7	8	9
a	b	c	a	b	c	a	b	d	
a	b	c	a	b	d				
			a	b	c	a	b	d	

- Pattern shifted by 3, resuming at 5
- Shift distance determined by the widest border of the matching prefix of the pattern
  - matching prefix: `abcab`,  $w = 5$ ,
  - widest border: `ab`,  $w = 2$
  - Shift distance:  $d = 5 - 2 = 3$

# String Pattern Matching

## Knuth, Morris, Pratt

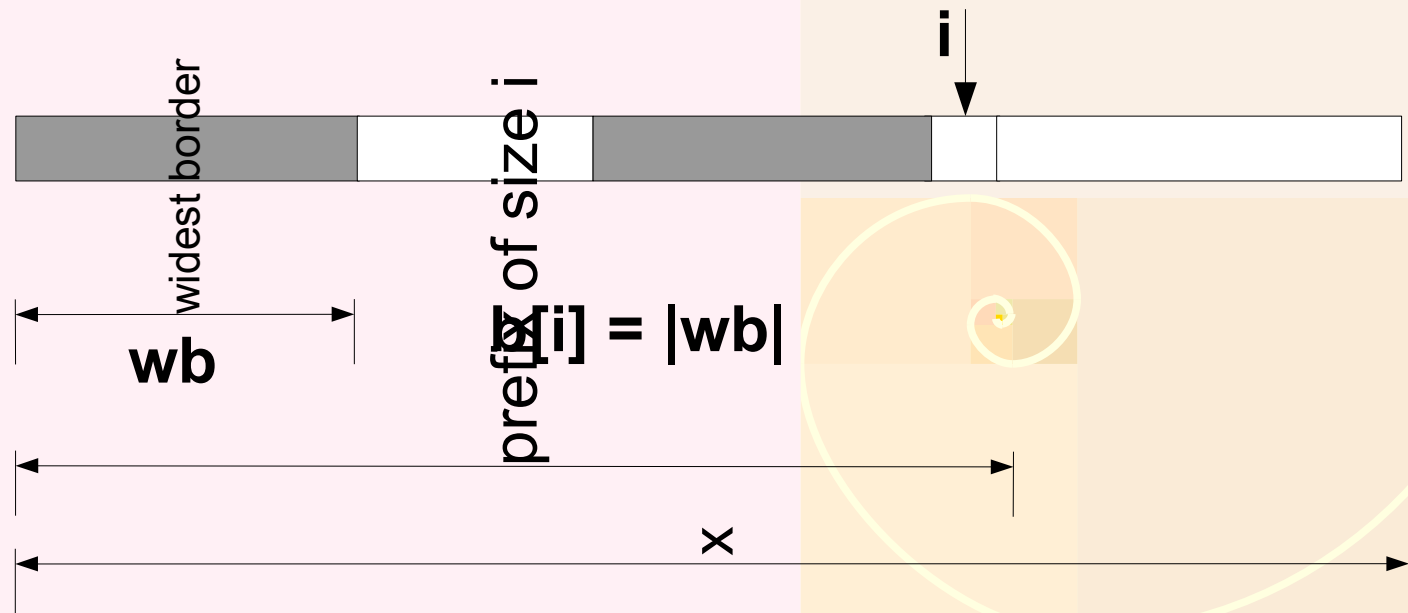
- Two phases
- Preprocessing phase:
  - compute the width of the widest border of each prefix of the pattern
- Searching phase
  - compute the shift distance according to the prefix that has matched



# String Pattern Matching

## Knuth, Morris, Pratt

- Preprocessing phase: compute  $b[]$ ,  $|b|=|p|+1$
- $b[i]$  = width of the widest border of the prefix of length 'i' of the pattern ( $i=0,\dots,|p|$ ).
- $b[0] = -1$  (the prefix '()' of length 'i=0' has no border)



# String Pattern Matching

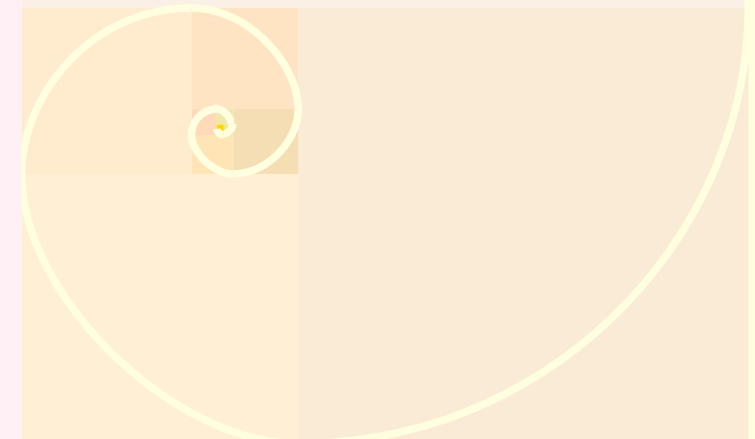
## Knuth, Morris, Pratt

- Computing  $b[]$

0	1	2	3	4	5	6
a	b	a	b	a	a	
-	0	0	1	2	3	1

0	1	2	3	4	5	6	7	8	9	10
a	b	c	a	b	c	a	c	a	b	
-	0	0	0	1	2	3	4	0	1	2

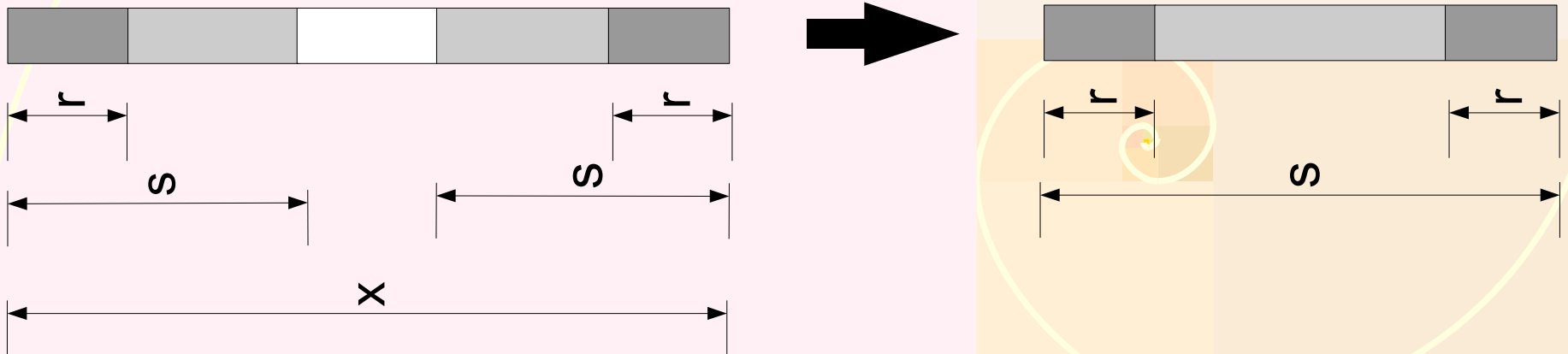
0	1	2	3	4	5	6	7	8
a	b	a	b	b	a	a	a	
-	0	0	1	2	0	1	1	1



# String Pattern Matching

## Knuth, Morris, Pratt

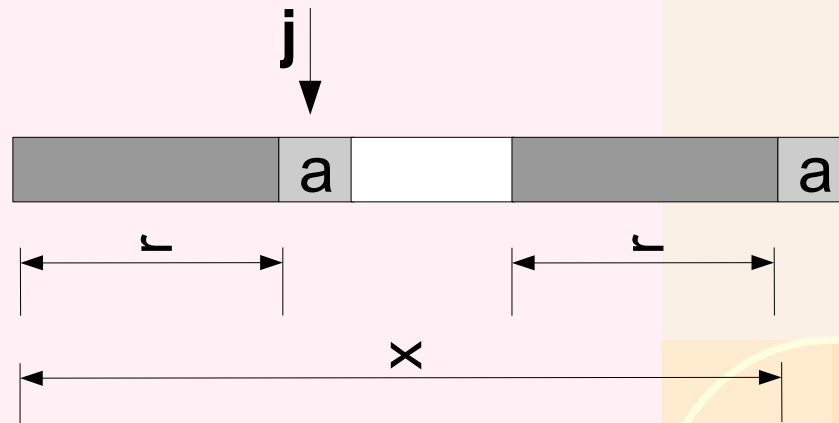
- Theorem:
  - if 'r', 's' are borders of 'x',  $|r| < |s|$ ,
  - then 'r' is a border of 's'
  - if 's' is the widest border of 'x', the next widest border 'r' of x, is the widest border of 's'



# String Pattern Matching

## Knuth, Morris, Pratt

- Def: 'x': string, 'a': character.
- A border 'r' of 'x' can be **extended by 'a'** if 'ra' is a border of 'xa'



A border 'r', of width 'j' of 'x' can be extended by 'a' if  $x[j]=a$

# String Pattern Matching

## Knuth, Morris, Pratt

- Suppose we already know  $b[0], \dots, b[i]$ 
  - To compute  $b[i+1]$  we search a border of width  $j < i$  of the prefix ' $p[0] \dots p[i-1]$ ' that can be extended by character  $p[i]$
  - This happens when  $p[b[j]] = p[i]$
  - If this is the case, then  $b[i+1] = b[j] + 1$
  - The border list is in decreasing order
    - $j = b[i], j = b[b[i]], \dots$

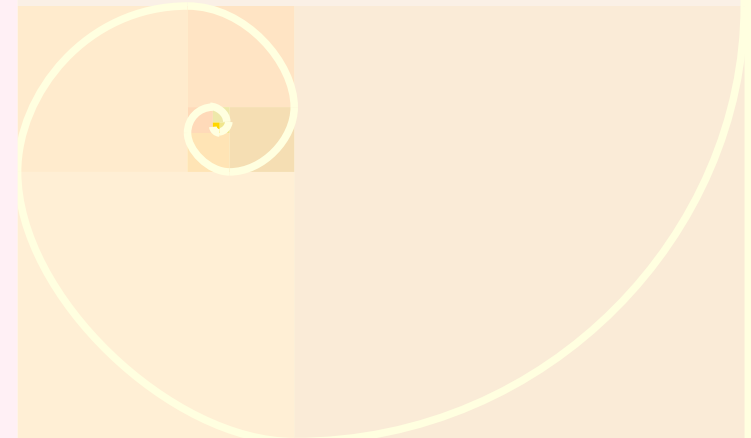


# String Pattern Matching

## Knuth, Morris, Pratt

- Algorithm for the creation of the array 'b[]'

```
void kmpPreProcess(char p[]) {
    int i = 0, j = -1;
    b[0] = -1; // Array allocated dynamically and returned
    while (i < |p|) {
        while (j >= 0 && // j == -1 ==> STOP !!
              p[i] != p[j]) { // mismatch
            j = b[j]; // Find the widest border
        }
        i++; j++;
        b[i]=j; // b[i+1] = b[j]+1
    }
}
```

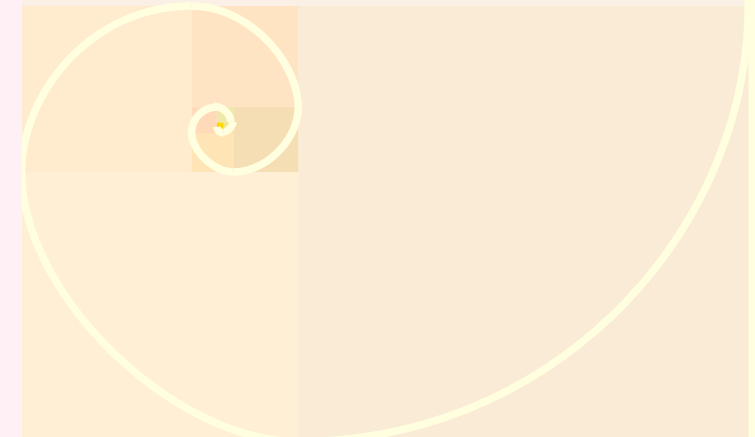


# String Pattern Matching

## Knuth, Morris, Pratt

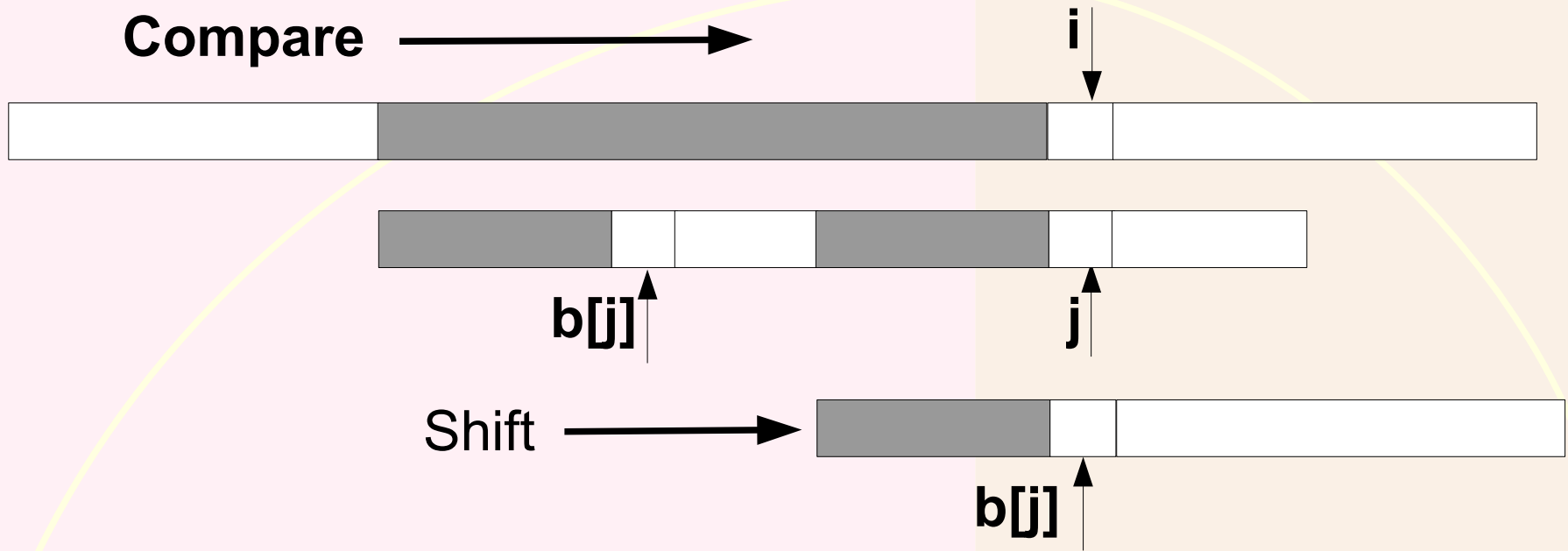
- Searching algorithm

```
void kmpSearch(char t[], char p[]) {  
    int i = 0, j = 0;  
    while (i < |t|) {  
        while (j >= 0 && // j == -1 ==> STOP !!  
              t[i] != p[j]) { // mismatch  
            j = b[j]; // Shift the pattern!!  
        }  
        i++;j++;  
        if (j == |p|) return i - j;  
    }  
    return -1;  
}
```



# String Pattern Matching

## Knuth, Morris, Pratt



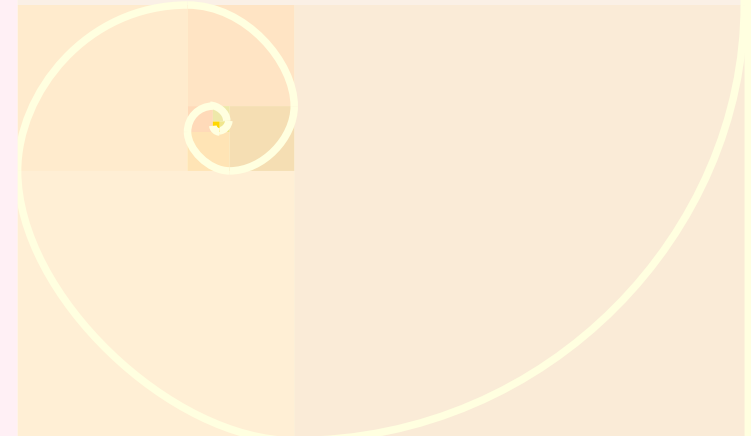
0	1	2	3	4	5	6	7	8	9
a	b	a	b	b	a	b	a	a	
a	b	a	b	a	c				
		a	b	a	b	a	c		
			a	b	a	b	a	c	

Matching prefix size = 4,  
 widest border = 2,  
 shift =  $4 - 2 = 2$ ,

Matching prefix size = 2,  
 widest border = 0,  
 shift =  $2 - 0 = 2$

# KMP Algorithm Complexities

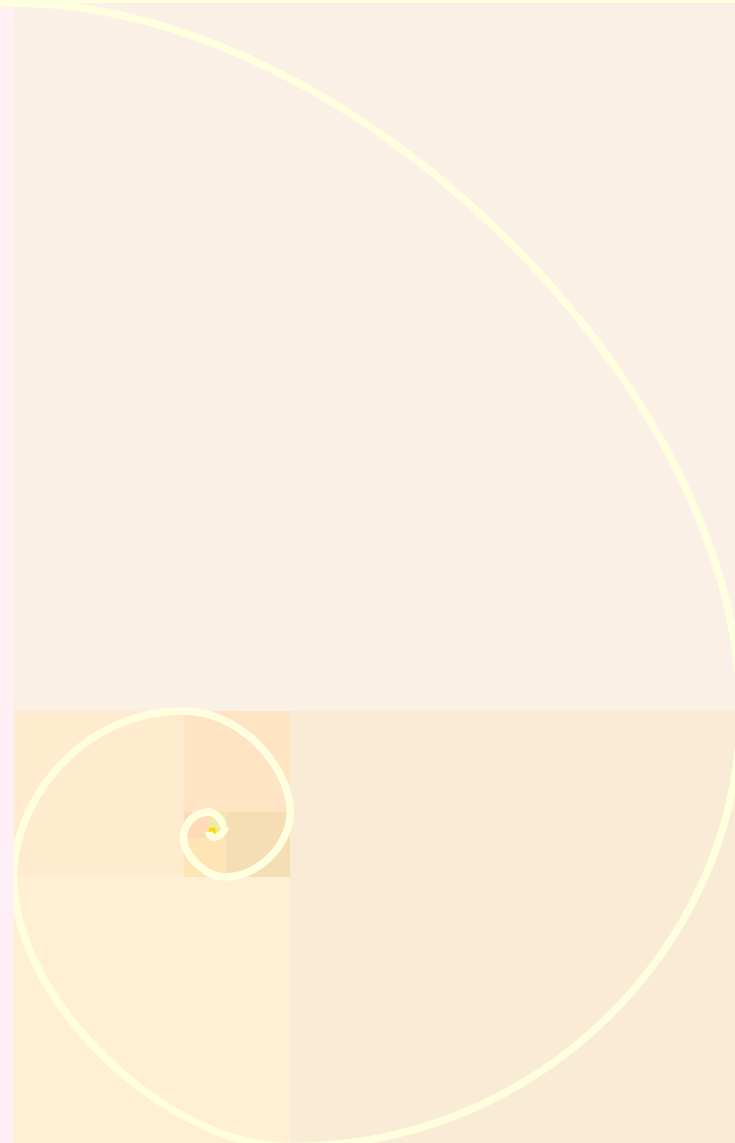
- Space
  - The array  $b[] \Rightarrow O(|p|+1)$
- Time: how many characters comparisons
  - PreProcessing: focus on the inner while loop
    - decreases 'j' by at least '1' until 'j = -1' ( $b[j] < j$ )
    - 'j' is increased exactly '|p|' times by the outer loop
    - $\Rightarrow$  'j' cannot be decreased more than '|p|' times:  $O(|p|)$
  - Search
    - Same argument:  $O(|s|)$
  - Total:  $O(|s|+|p|)$



# Stacks & Queues

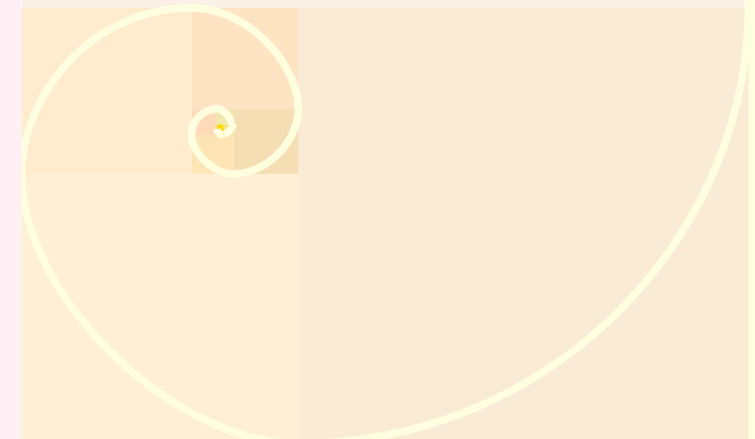
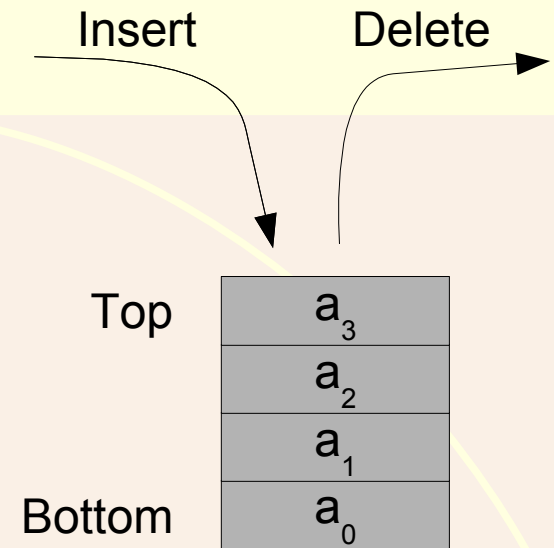
# Stacks and Queues

- Widely used data structures
- Ordered List of element
- Easy to implement
- Easy to use



# Stacks

- $S=(a_0, \dots, a_{n-1})$ 
  - $a_0$  is the *bottom* of the stack
  - $a_{n-1}$  is the *top* of the stack
  - $a_i$  is *on top* of  $a_{i-1}$  ( $0 < i < n$ )
- Insertions and deletions are made at the *top*
- *Last In First Out (LIFO)* list
  - Example: stack of plates



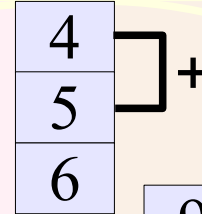
# Stack Interface

- Basic operations
  - add() also called push()
  - delete() also called pop()
  - isEmpty()
- Optional Operation
  - isFull() (when the stack as a maximum capacity)
- Basic implementation using an array
  - How to prevent a stack to become full?

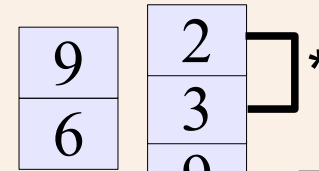
# Stack Use: evaluation of expression

•  $6 + (((5 + 4) * (3 * 2)) + 1) = ?$

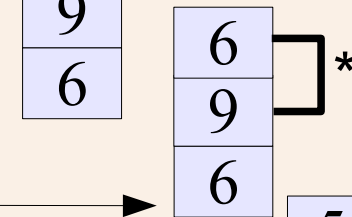
– push(6), push(5), push(4)



– push(pop()+pop())



– push(3), push(2)

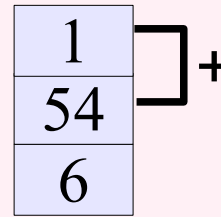


– push(pop()\*pop())

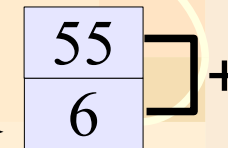


– push(pop()\*pop())

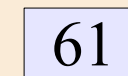
– push(1)



– push(pop()+pop())



– push(pop()+pop())



# Expression notation

- Infix
  - operators are *in-between* their operands
    - $(3+2)*5 = 25$  --> Needs parenthesis
- Postfix (HP calculators)
  - operators are *after* their operands
    - $3 2 + 5 * = 25$
- Prefix
  - operators are *before* their operands
    - $* + 3 2 5 = 25$
- Order of operands is the same

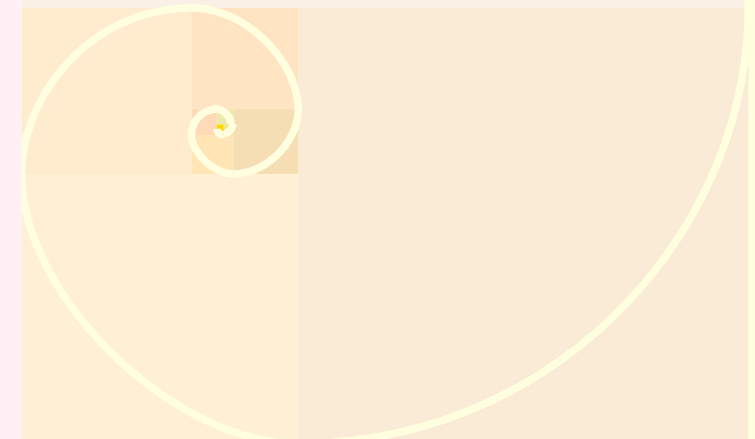
# Stack Use:

## Conversion from infix to postfix

```
// Return the postfix notation of a fully bracketed  
// infix expression  
// ((2+3)*5) is ok, (2+3)*5 is not  
char* convert(char* s) {  
    char* t = new char[|s|]; // |t| < |s|  
    for (int i = j = 0; i < |s|; i++) { // i:s[], j:t[]  
        if (s[i] == ')') t[j++] = pop();  
        else if (s[i] == '+') push(s[i]);  
        else if (s[i] == '*') push(s[i]);  
        else if (isDigit(s[i])) t[j++] = s[i];  
    }  
    t[j] = '\\0';  
    return t;  
}
```

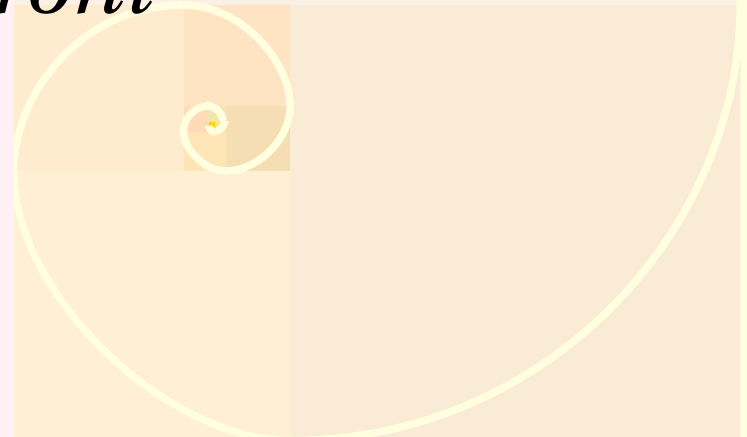
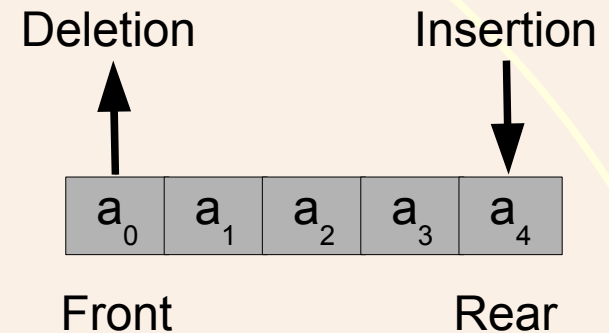
# Evaluation of postfix expression

```
// Evaluate a postfix expression such as 23+5*
int compute(char* s) { // s is postfix
    int r = 0;
    for (int i = 0; i < |s|; i++) {
        if (s[i] == '+') push(pop() + pop());
        else if (s[i] == '*') push(pop() * pop());
        else if (isDigit(s[i])) push(valueOf(s[i]));
    }
    return pop();
}
```



# Queues

- $Q=(a_0, \dots, a_{n-1})$ 
  - $a_0$  is the *front* of the queue
  - $a_{n-1}$  is the *rear* of the queue
  - $a_i$  is *behind*  $a_{i-1}$  ( $0 < i < n$ )
- Insertions take place at the *rear*
- Deletions take place at the *front*
- *First In First Out (FIFO)* list
  - Example: queue of persons



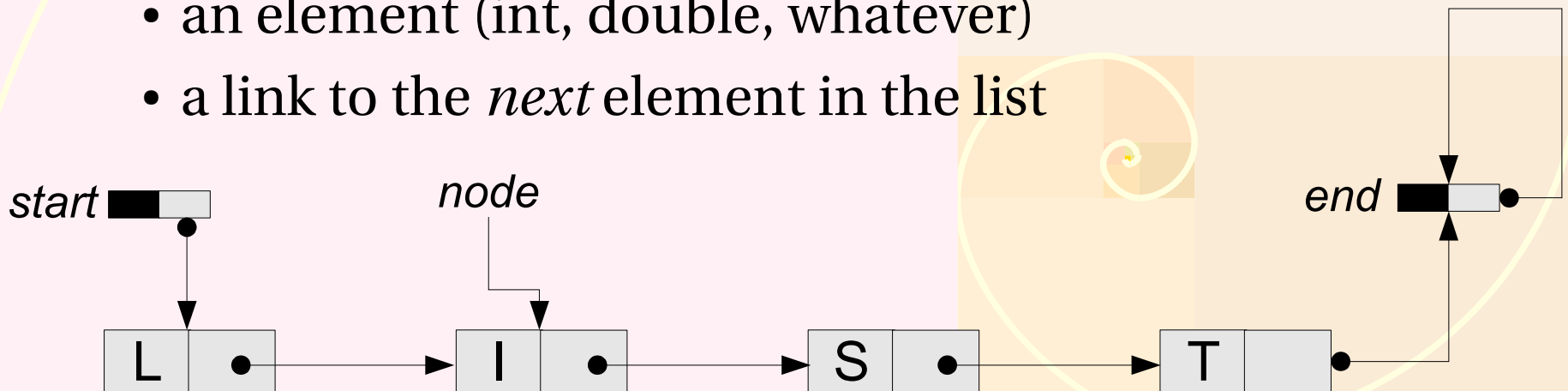
# Queue Interface

- Basic operations
  - add()
  - delete()
  - isEmpty()
- Optional Operation
  - isFull() (when the queue has a maximum capacity)
- Basic implementation using an array
  - How to prevent a queue to become full?

# Linked List

# Characteristics

- Insertion and deletion of elements in constant time  $O(1)$ 
  - Contrary to arrays (linear time  $O(n)$ )
- Accessing an element is in linear time  $O(n)$ 
  - Contrary to arrays (constant time  $O(1)$ )
- Composed of *nodes* where a node is:
  - an element (int, double, whatever)
  - a link to the *next* element in the list



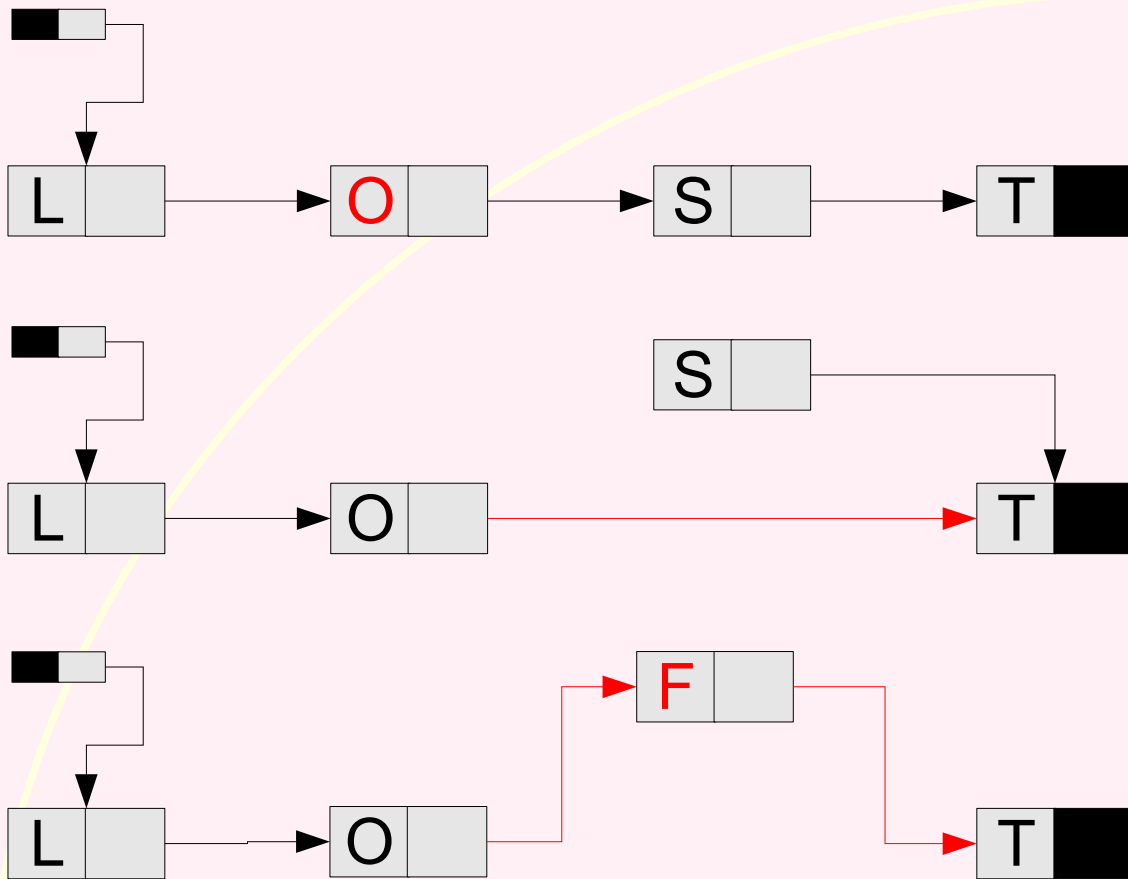
# Operations

## IV. Linked List

Modification  
(LIST --> LOST)

Deletion  
(LOST --> LOT)

Insertion  
(LOT --> LOFT)



# Dynamic implementation

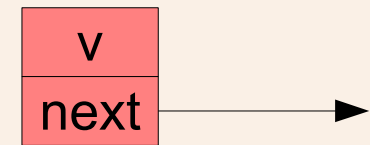
- Use a structure (or a class) to represent a node

// Always use pointers alias

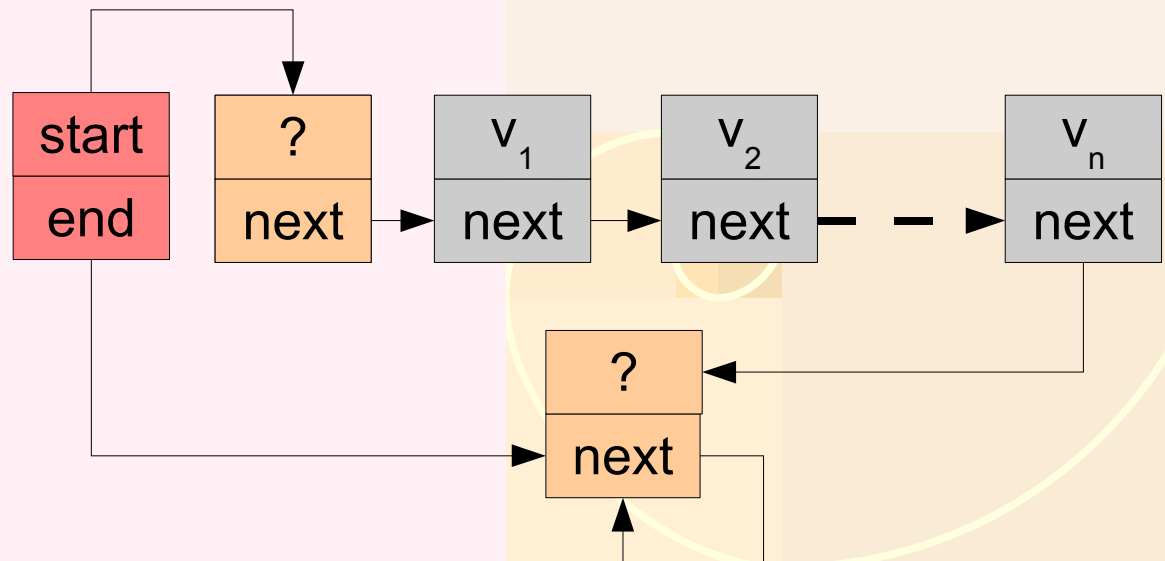
```
typedef struct node* node; // 'node' == 'struct node*'
```

```
typedef struct list* list; // 'list' == 'struct list*'
```

```
struct node {  
    char v; // the value of this element  
    node next; // the next node  
};
```



```
struct list {  
    node start;  
    node end;  
};
```



# Dynamic Implementation

- Node creation

```
node newNode(char v) {  
    node n = malloc(sizeof(*n));  
    n->v = v;  
    n->next = NULL; // Must be set by the caller.  
    return n;  
}
```

- Write the deleteNode() function.

- List creation

```
list newList() {  
    list l = malloc(sizeof(*l));  
    l->end = newNode(0, NULL); // Value has no meaning  
    l->end->next = end; // loop !!  
    l->start = newNode(0, end); // Value has no meaning  
    return l;  
}
```

# Dynamic Implementation

- Insertion

```
void insertAfter(list l, char v, node n) {  
    node new = newNode(v, n->next);  
    n->next = new;  
}
```

- Deletion

```
void DeleteNext(list l, node n) {  
    node t = n->next;  
    n->next = t->next;  
    deleteNode(t);  
}
```

- Interface

- Pass the list in argument even if unused
- Interface must be independent of implementations

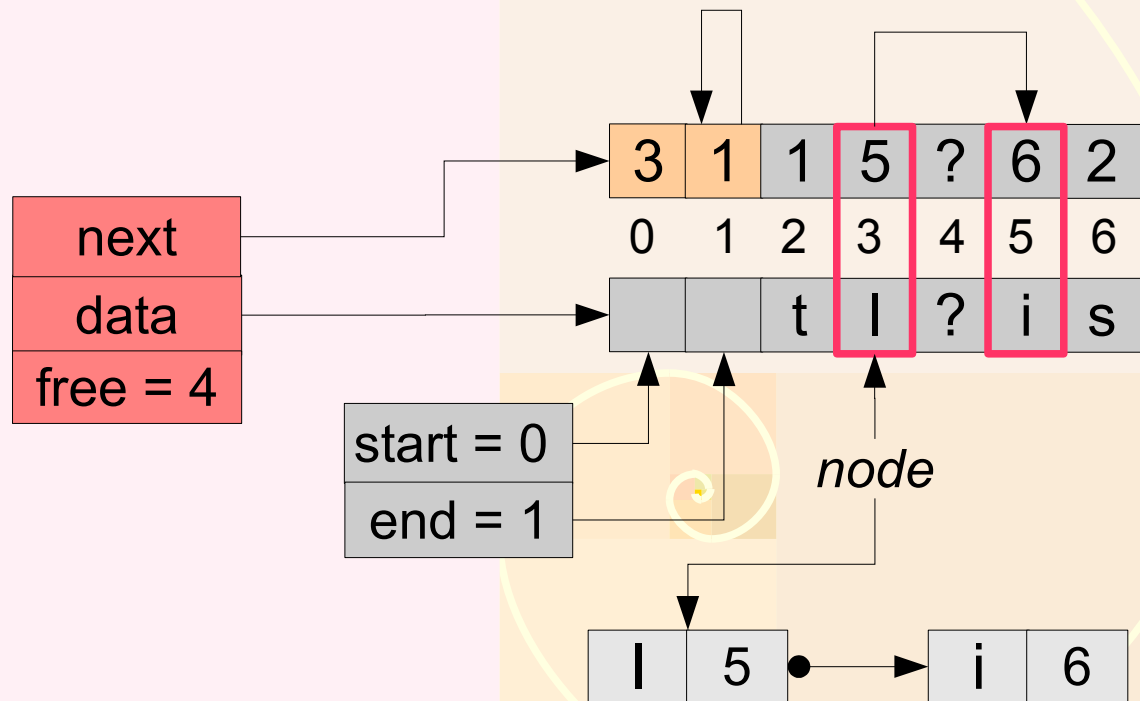
# Static implementation

- Using arrays
  - one array contains data,
  - others contain next links. (multiple links on data)

```
struct node{
  char v;
  int next;
};

struct list{
  int *next;
  char *data;
  int free;
};

#define START 0
#define END 1
```



# Static Implementation

```
list newList() {
    int size = MAX + 2; // start & end
    list l = malloc(sizeof(*l));
    l->next = malloc(size * sizeof(int));
    l->data = malloc(size * sizeof(char));
    l->next[START] = END; l->next[END] = START;
    l->free = 2;
    return l;
}

void insertAfter(list l, char v, node n) {
    l->data[free] = v;
    l->next[free] = l->next[n->index];
    l->next[n->index] = free++;
}

void deleteNext(list l, node n) {
    l->next[n->index] = l->next[l->next[n->index]]; }
}
```

# Static Implementation

- How to handle free cells more efficiently?
  - 'free' is only incremented until it reached the array size.
  - then, how to use cells that have been removed from the list in the middle of the array
- Use a 'free' list
  - Multiple list on the same data
  - This is how 'malloc()' and 'free()' actually works
  - This is also how the kernel works: memory is a (big) array

# Stack implementation using a (linked) list

```
// Independent of the actual implementation of the list!
struct stack { list l; };

stack newStack() {
    stack s = malloc(sizeof(*s));
    s->l = newList();
}

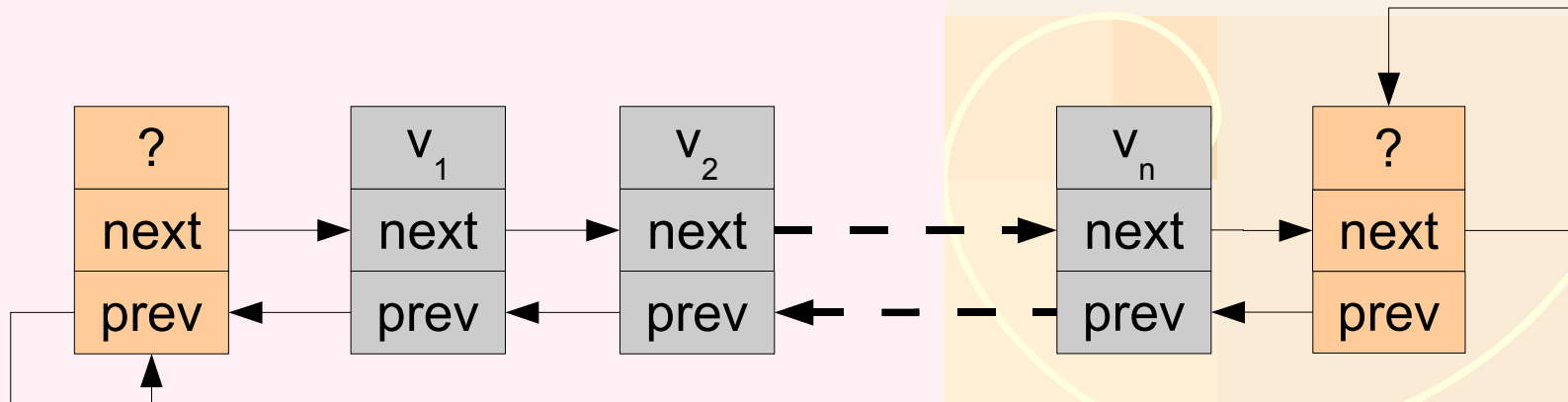
void push(stack s, char v) {
    insertAfter(s->l, v, s->l->start);
}

char pop(stack s) {
    assert(!isEmpty(s->l));
    // Write these 2 functions
    node top = getNextNode(s->l->start);
    char v = getNodeValue(top);
    deleteNext(s->l, s->l->start);
    return v;
}
```

# Double Linked List

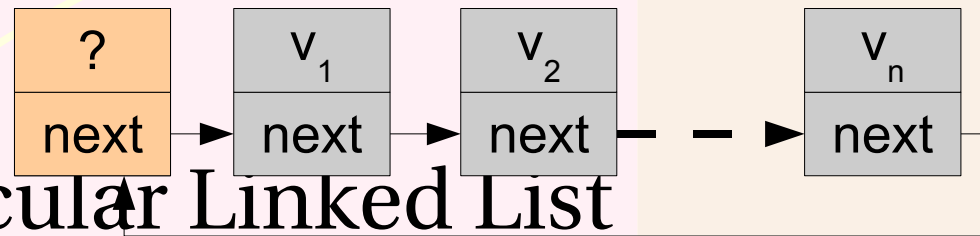
- Problems of single linked list
  - moving only in one direction leads to problem on deletion or searching
    - the preceding node must be known
- Use two links per node (space complexity?)

```
struct node {  
    char v;  
    node *next, *prev;  
}
```

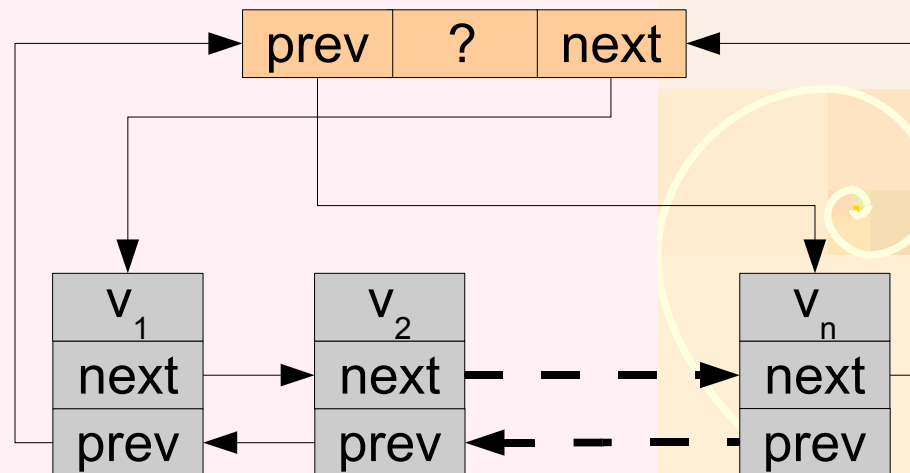


# Circular List

- Single Circular Linked List



- Double Circular Linked List



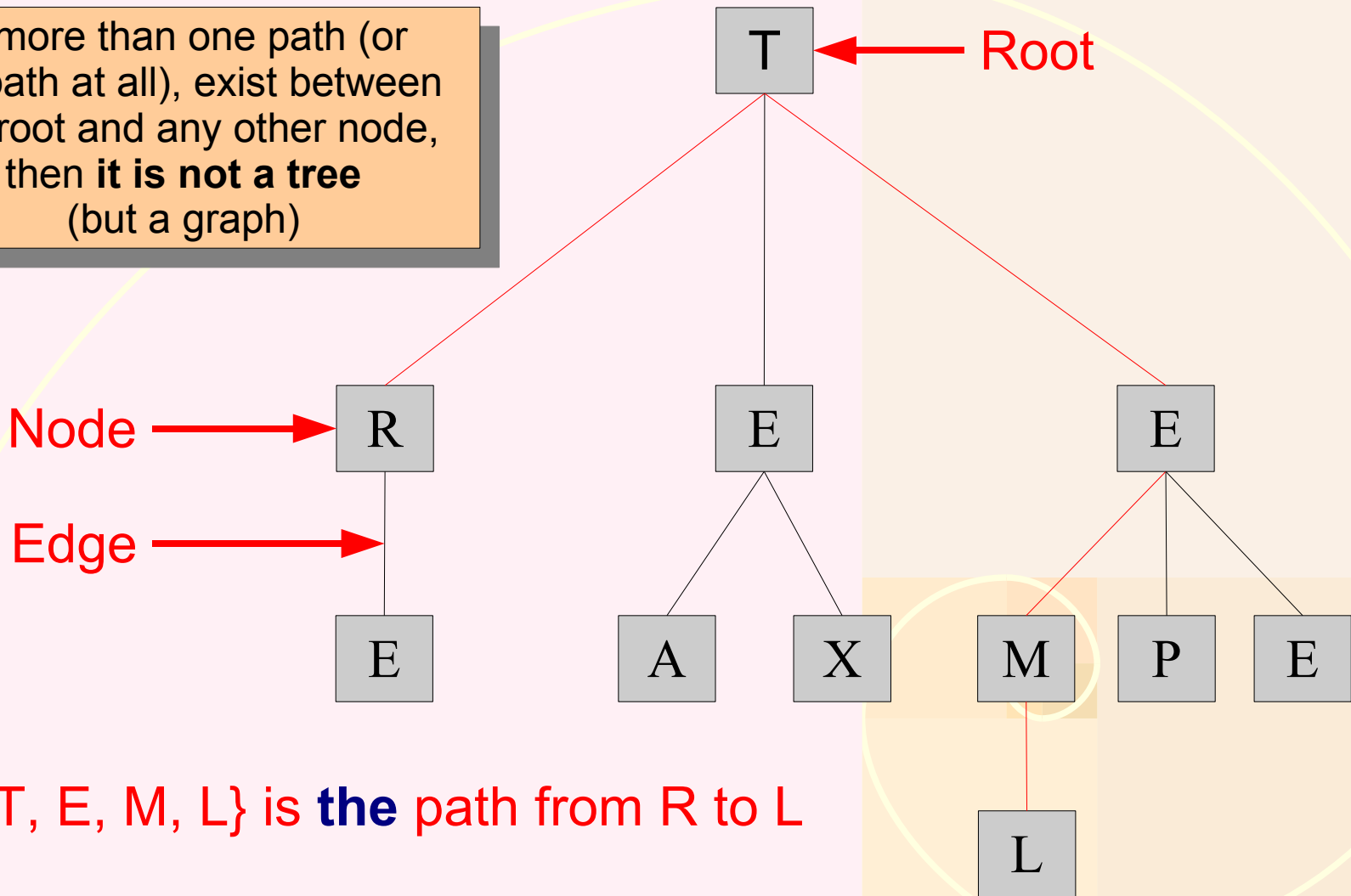
# Trees

# Glossary

- Tree
  - A non-empty finite set of *nodes* and *edges* that follow some conditions
- Node
  - Simple object that contains some data
- Edge
  - A link between two nodes
- Path
  - A list of distinct nodes in which 2 successive nodes are linked by an edge

# Glossary

If more than one path (or no path at all), exist between the root and any other node, then **it is not a tree** (but a graph)



# Glossary

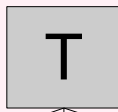
- M is the *father* of L
- R is the *child* of T
- {A, X} and {M, P, E} are siblings

If N is the number of nodes, (N-1) is the number of edges

- Degree(Node)
  - number of children
- $\text{deg}(T) = 3, \text{deg}(M) = 1$

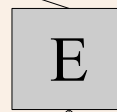
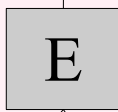
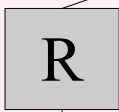
Level

1



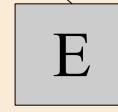
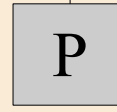
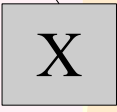
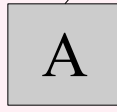
Root

2



Leaf Node →

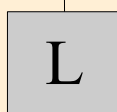
3



- Degree(Tree)
  - maximum degree of its node
- degree = 3

Depth: 4

4



V. Trees

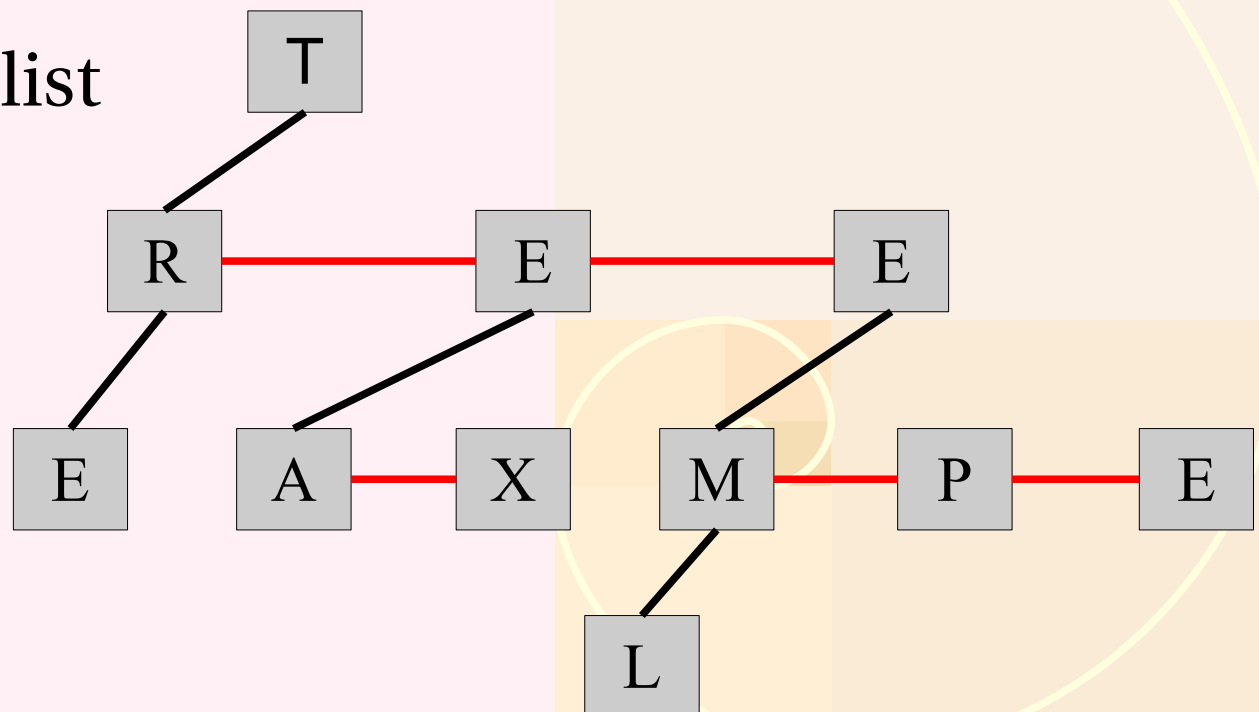
# Representations

- Depends on the needs
  - If you just need to go from a child to its parent, use two arrays
    - $a[k]$  = value of the node  $k$  (e.g. a character)
    - $father[k]$  = index of father of node  $a[k]$
    - $a[father[k]]$  = value of the father of node  $k$

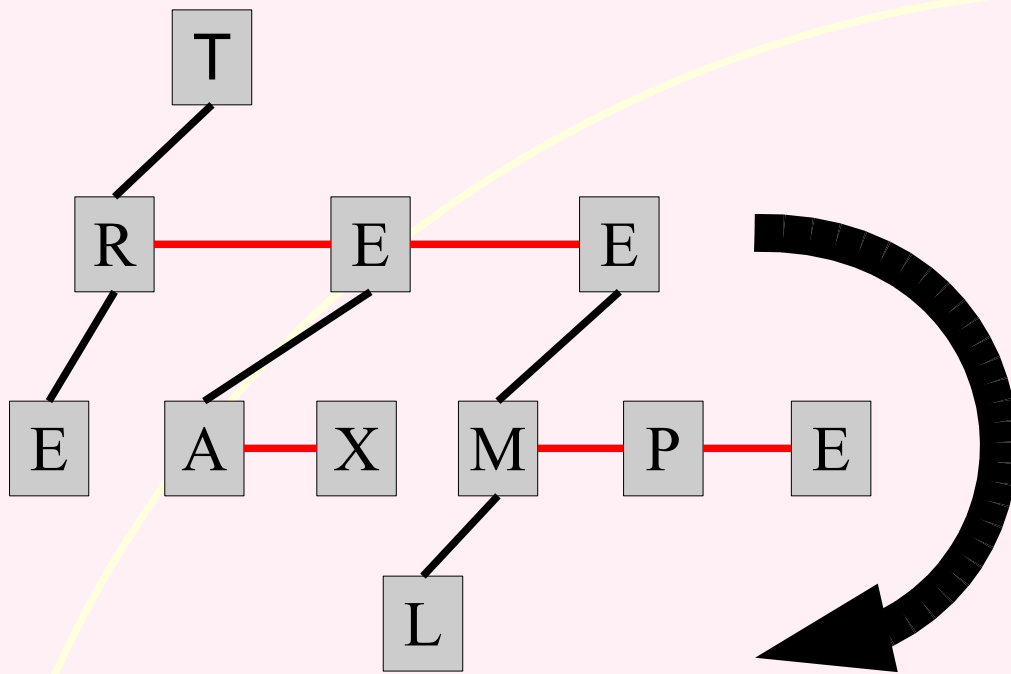
k	0	1	2	3	4	5	6	7	8	9	10
a[k]	T	R	E	E	E	X	A	M	P	L	E
father[k]	0	0	0	0	1	2	2	3	3	7	3

# Representations

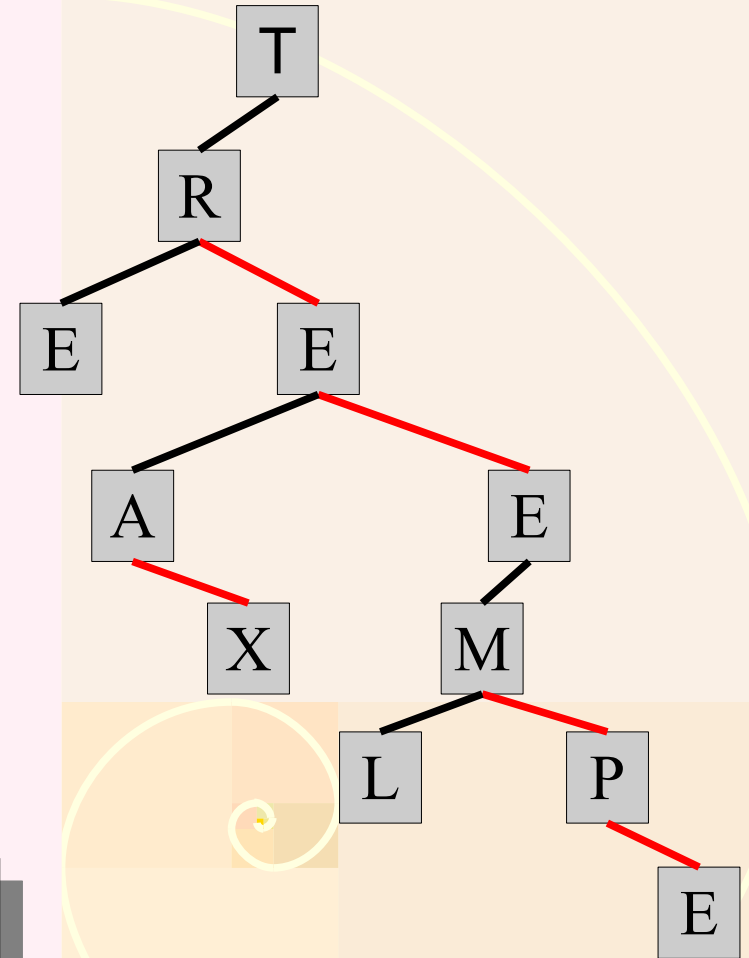
- If you need to go down, from parents to children
  - use (dynamic) linked lists to keep track of children
  - one brother list
  - one children list



# Representations



Rotate



Any tree can be converted into a **2-degree** tree.

# Binary Tree

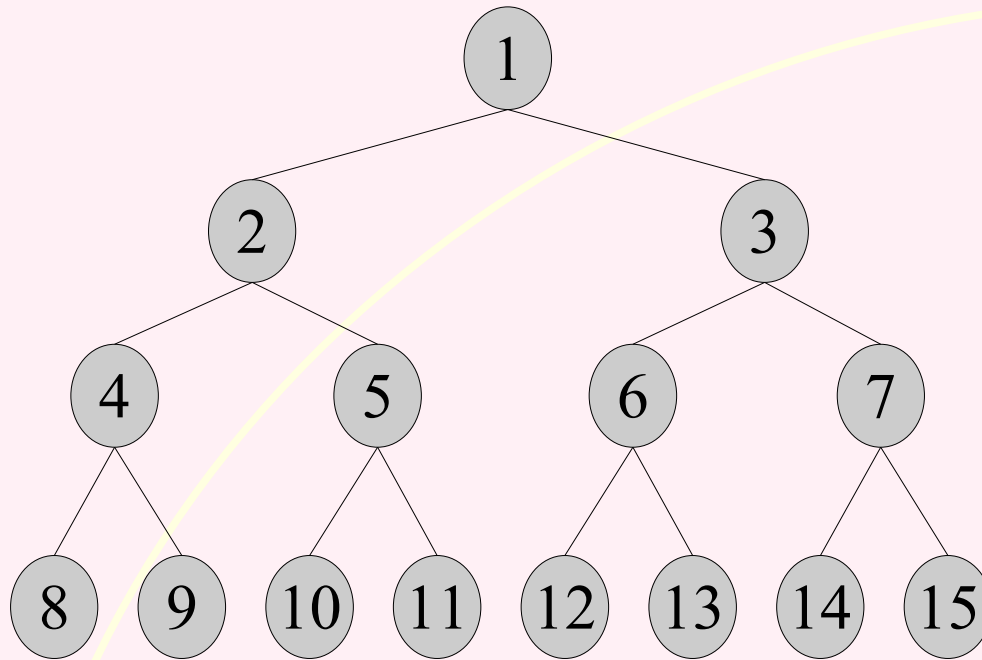
- 2-degree tree are so important that they have a special name: *Binary Tree*
- A binary tree is a finite set of nodes that is either **empty** or as a **root and two disjoint binary trees** called *left subtree* and *right subtree*.
- Recursion in the definition
- Algorithm on binary trees is often expressed recursively



# Binary Tree Characteristics

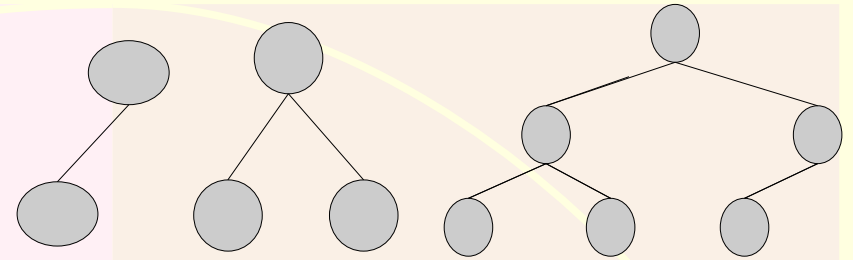
- Maximum number of nodes at level 'i':  $2^{(i-1)}$
- Maximum number of nodes in a binary tree of depth 'k':  $(2^k - 1)$ 
  - Proof by induction
- A *full* binary tree of depth 'k' is a binary tree of depth 'k' having  $(2^k - 1)$  node
- A binary tree with 'n' nodes and depth 'k' is *complete* iff its node correspond to the nodes numbered from '1' to 'n' in the full binary tree of depth 'k'.

# Full and Complete Binary Trees

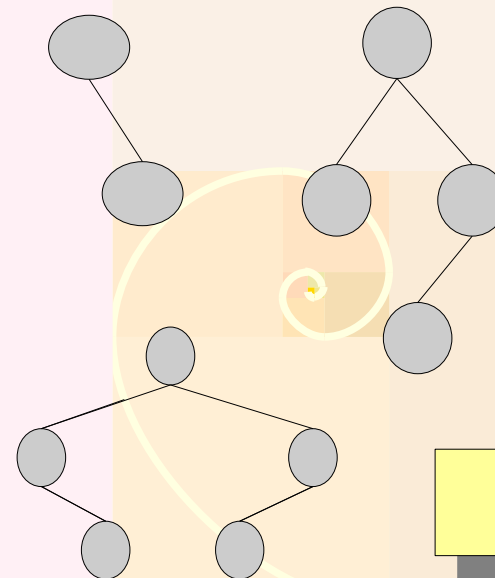


Full

Height of a complete binary tree with 'n' nodes:  $\lceil \lg(n+1) \rceil$



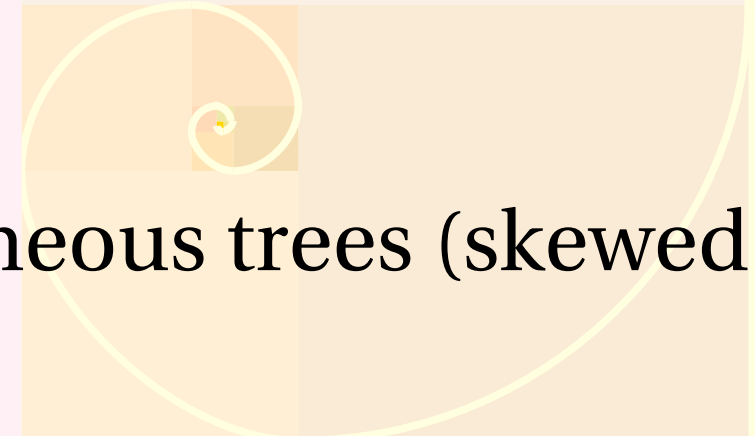
Complete



Normal

# Representations: array

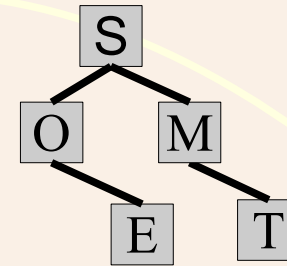
- If you know the number of nodes 'n', you may use an array 't' of width '(n+1)'
  - $\text{Parent}(i) = t[\text{ceil}(i/2)]$ ,  $i \neq 1$ ; If ( $i == 1$ ),  $i$  is the root node and has no parent.
  - $\text{LeftChild}(i) = t[2.i]$  if ( $2*i \leq n$ ); If ( $2*i > n$ ),  $i$  has no left child.
  - $\text{RightChild}(i) = t[2.i+1]$  if ( $2.i+1 \leq n$ ); If ( $2.i+1 > n$ ),  $i$  has no right child.
- Ideal for complete trees
- Waste of space for miscellaneous trees (skewed trees)



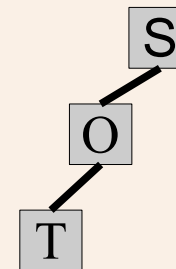
# Representations: array

## Examples

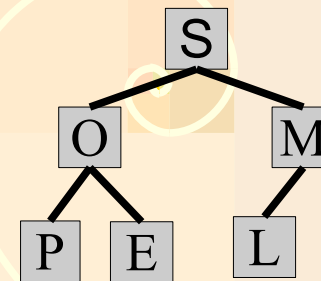
k	0	1	2	3	4	5	6	7
a[k]	-	S	O	M	-	E	-	T



k	0	1	2	3	4	5	6	7
a[k]	-	S	O	-	T	-	-	-



k	0	1	2	3	4	5	6	7
a[k]	-	S	O	M	P	E	L	-

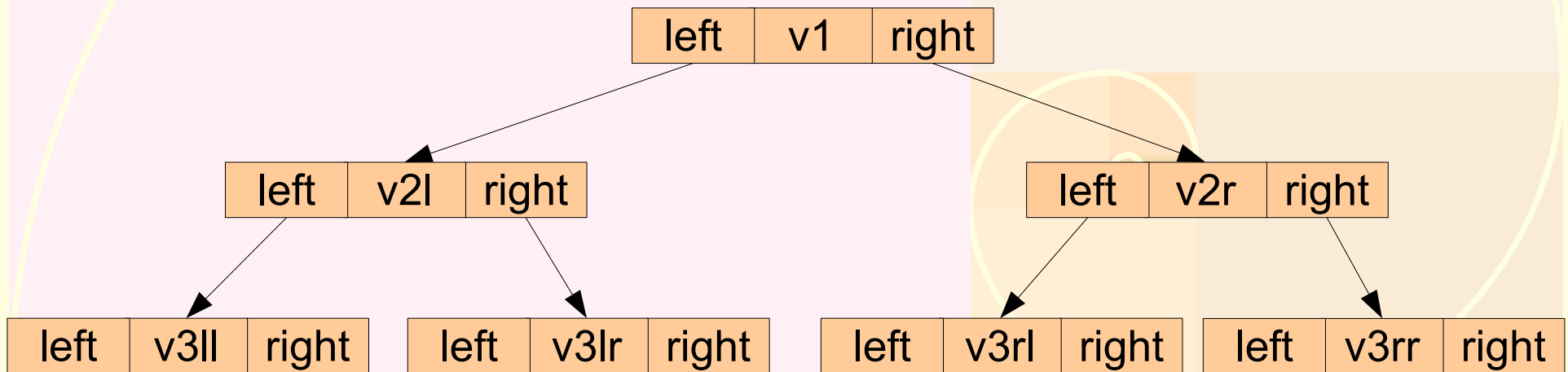


# Representations: Linked

- Use pointers to represents left and right childs

```
struct node{
    char v;
    node left, right;
};
struct tree{
    node root;
}
```

Exercice: write the newNode() and newTree() function!



# Binary Tree Traversal

- Visit each node of a tree exactly once
  - On visit, perform an operation on the data
- Convention: always visit **left before right**
- In order: LVR (recursive)
  - move Left, Visit the node, move Right
- Pre order: VLR (recursive)
  - Visit the node, move Left, move Right
- Post order: LRV -> Guess! (recursive)
- Level order: visit by level (non-recursive)

# Binary Tree Traversal Examples

$$6 + (((5 + 4) * (3 * 2)) + 1)$$

- In Order:

$$6 + 5 + 5 * 3 * 2 + 1$$

- Pre Order:

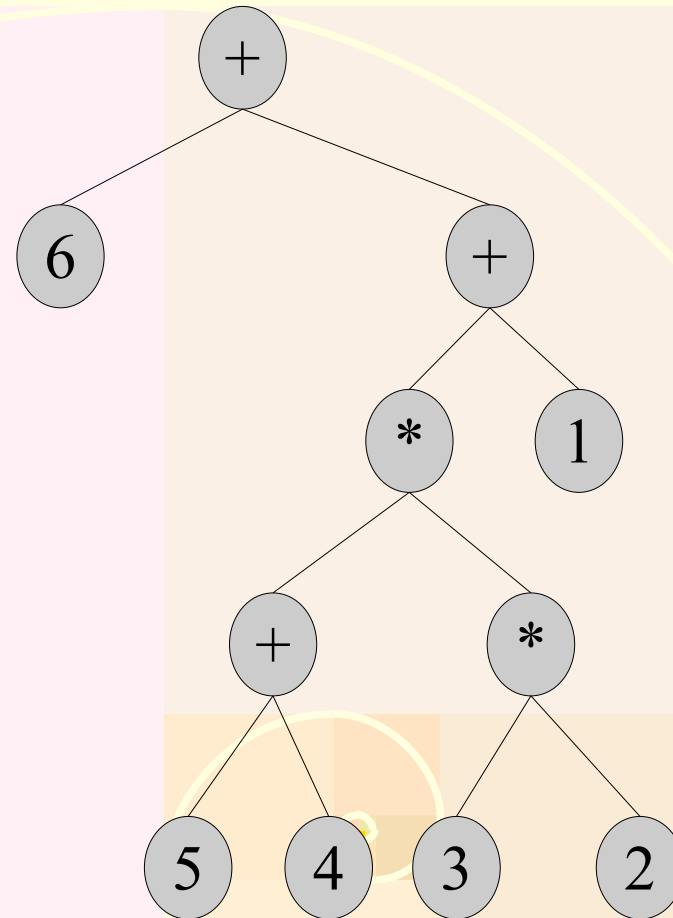
$$+ 6 + * + 5 4 * 3 2 1$$

- Post Order:

$$6 5 4 + 3 2 * * 1 + +$$

- Level Order:

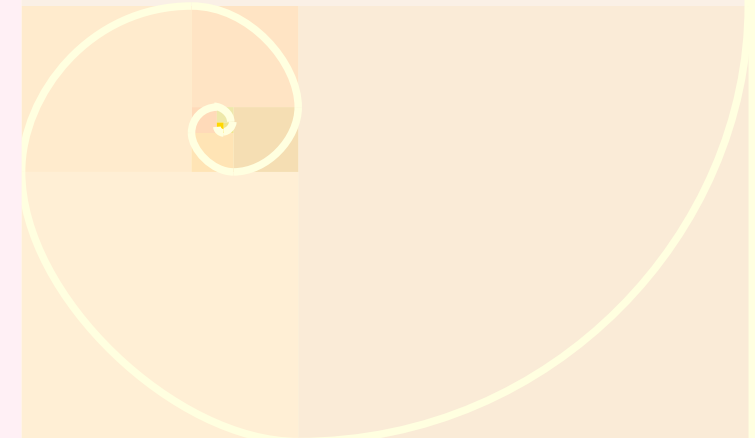
$$+ 6 + * 1 + * 5 4 3 2$$



# Binary Tree Traversal Implementations

```
void inOrder(node root) { // implicit use of a stack
    if (root == NULL) return;
    inOrder(root->left);
    process(root->v); // Do something with the value
    inOrder(root->right);
}
void preOrder(node root) { // implicit use of a stack
    if (root == NULL) return;
    process(root->v); // Do something with the value
    preOrder(root->left);
    preOrder(root->right);
}
```

Exercice 1: write postOrder()!



# Binary Tree Traversal

## Non-recursive implementations

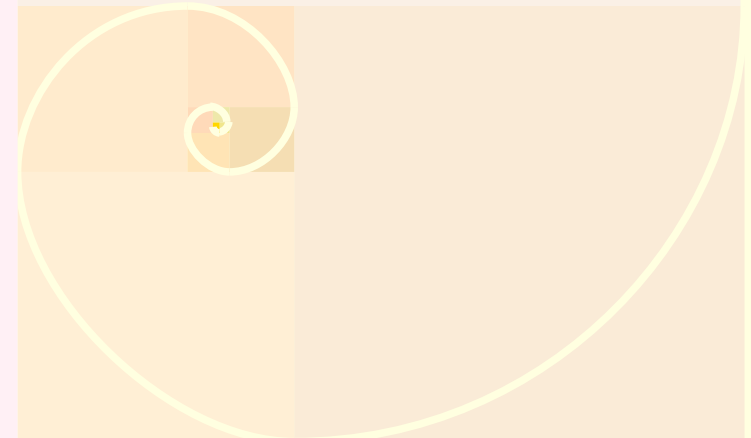
```
void inOrder(node n) {
    stack s = newStack(); // needs a stack
    while (1) { // Infinite loop
        while (n != NULL) { // Push n on the (top of the) stack
            push(n);
            n = n->left; // Move down on left child
        }
        if (isEmpty(s)) break; // Nothing else to do
        n = pop(s); // pop the last inserted child
        process(n); // Do something with the node
        n = n->right; // Move right then
    }
}
```

Exercice 1 (easy): write preOrder() non-recursive version

Exercice 2 (difficult): write postOrder()

# Binary Tree Traversal Implementations

```
void levelOrder(node root) {
    node n = root;
    while (n != NULL) {
        process(n);
        if (n->left != NULL) addQueue(n->left);
        if (n->right != NULL) addQueue(n->right);
        n=deleteQueue();
    }
}
// No need of a stack.
// Needs a queue.
```



# Priority Queues

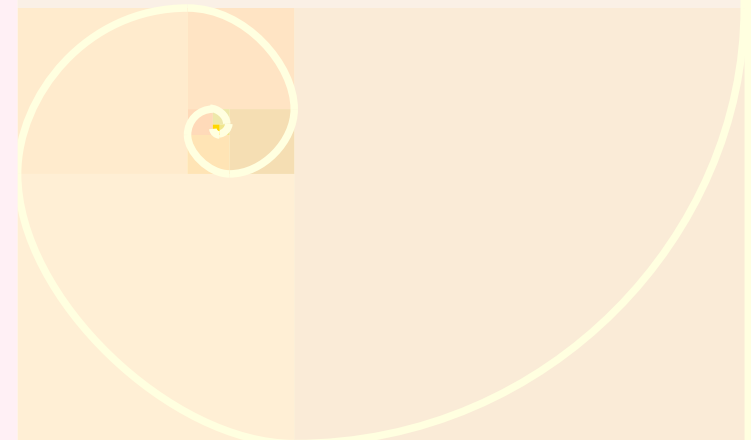
# Definition

- A max (resp. min) priority queue is a queue that provides a `deleteMax()` (resp. `deleteMin()`) operation.
  - each element in the queue has:
    - a value
    - a priority that is called a *key*
- The `deleteMax()` (resp. `deleteMin()`) operation delete the element in the queue with the maximal priority (resp. minimal) instead of the first inserted one as with the `delete()` operation of ordinary queues (FIFO).

# Priority Queues

## Basic Implementations

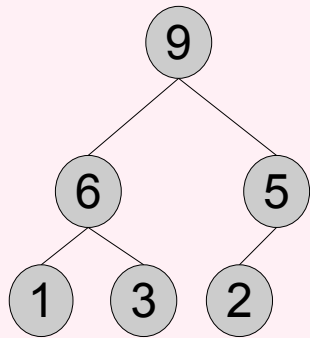
- Using a non-ordered list
  - insertion() in constant time:  $O(1)$
  - deleteMax() in linear time:  $O(n)$
- Using an ordered-list
  - insertion() in linear time:  $O(n)$
  - deleteMax() in constant time:  $O(1)$



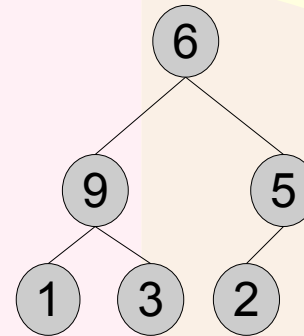
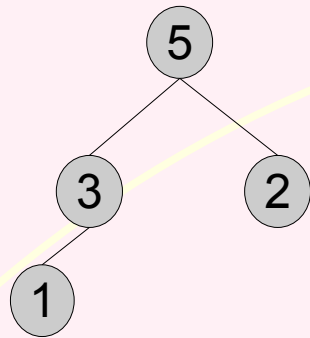
# {max, min}-Heap

- A max (min) tree is a tree in which the key value in each node is no smaller (larger) than the key values in its children (if any)
- A *max heap* is a complete binary tree that is also a max tree
- A *min heap* is a complete binary tree that is also a min tree
- The root of a max (min) heap is the largest (smallest) key in the tree
- Complete binary tree: use an array for storage

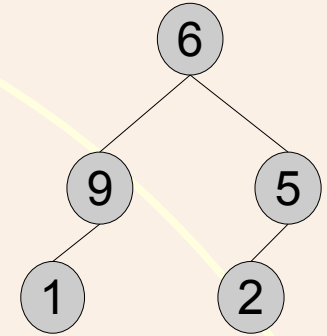
# Max-Heap Examples



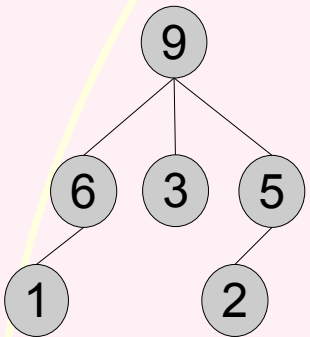
max-heaps



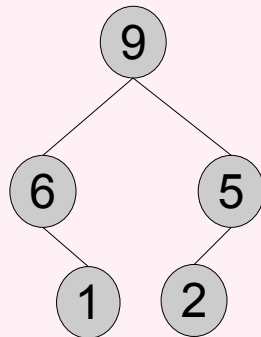
complete-binary tree



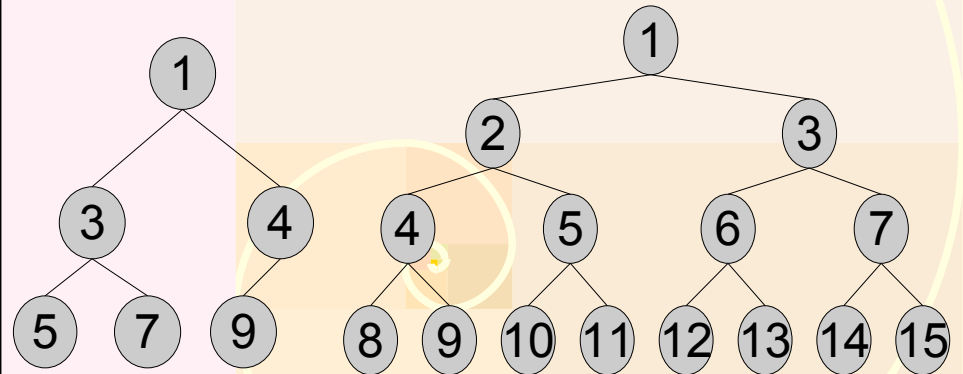
binary-tree



max-tree  
(not-binary)



max-tree  
(not-complete)



min-heaps

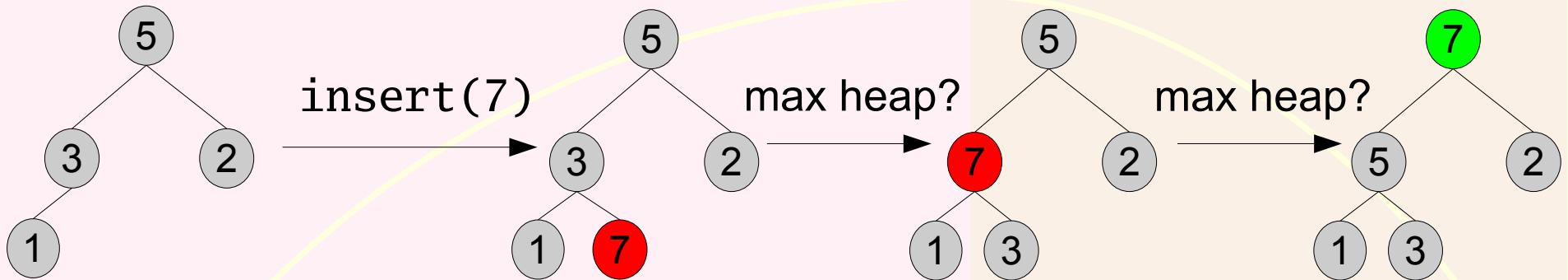
# Heap implementation

```
struct element {
    char value;
    int key;
}
struct heap {
    element *a; // backed by an array
    int n;      // size of the tree
};

heap newHeap() {
    heap h = malloc(sizeof(*h));
    h->n = 0; // Empty
    h->a = malloc(MAX_SIZE*sizeof(*a));
    return h;
}

// Exercice: write the freeHeap() function
```

# insert() Implementation



```
void insert(heap h, element e) {  
    assert(!heap_full(h)); // Implement this function  
    n++; // increase the size of the heap  
    int i = n; // start from the last 'node' in the tree  
    while(1) { // infinite loop  
        if (i == 1) break; // We have reached the root  
        element father = h->a[i/2];  
        if (e.key <= father.key) break; // Position found at 'i'  
        h->a[i] = h->a[i/2]; // Move the value from parent to 'i'  
        i = i/2; // Next insertion point is the father  
    }  
    h->a[i] = e; // Insert the element at its right position  
}
```



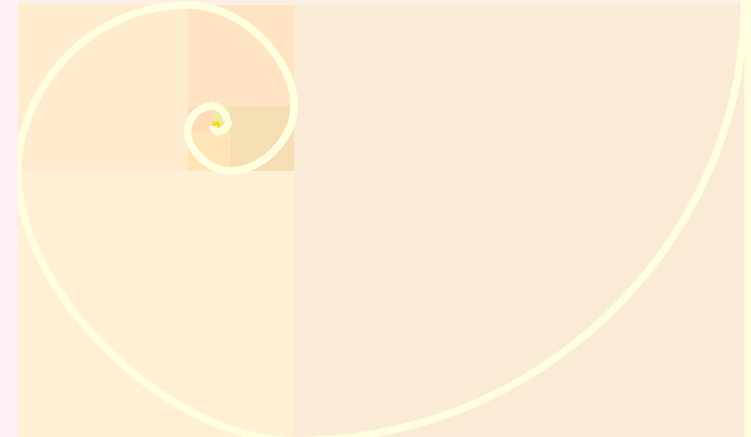
# Heap Implementation Complexities

- Space
  - insert() & deleteMax():  $O(1)$
- Time
  - insert():  $O(\lg(n))$ 
    - moves up from a leaf **toward** the root
      - maximum number of nodes visited = height(tree) =  $\lceil \lg(n+1) \rceil$
    - At each node,  $O(1)$  operation
  - deleteMax():  $O(\lg(n))$ 
    - moves down from the root **toward** a leaf
    - same argument

# Sorting

# Introduction

- 20% of computer time is about sorting
- Many different algorithms with different time and space complexities
  - None is the best
- Simple algorithms are very efficient in common cases
- Complex algorithms have better asymptotic time complexities
- Some algorithms are well understood whereas others are not
  - Still a research area



# Terminology

- We consider a sequential list (linked list or array) of *elements*
  - each element has a *key*
  - keys are used for sorting
- Example: class list
  - elements are “students” record containing many fields
    - name, id, average
  - Each field may be a key for a given sort

# Terminology

- A sort is said
  - *internal*: if it takes place in the memory
  - *external*: if only part of the list can be stored in memory
- A sort is said *stable* if elements with equal keys in the input list is kept in the same order in the output list
  - Most simple sorting algorithm are stables whereas most complex ones are not
  - Example: list of students sorted by name
    - you sort this list by the average mark
    - students with same average mark are still in order

# Selection Sort

- Find the final position 'k' of element at position 'i'
- swap element 'i' and 'k'

```
void sort_selection(int * t, int N) { // From 1 to N !!
    int min;
    for (int i = 1; i < N; i++) {
        min = i;
        for (int j = i+1; j <= N; j++) {
            if (t[j] < t[min]) min = j;
        }
        swap(t, i, min);
    }
}
```



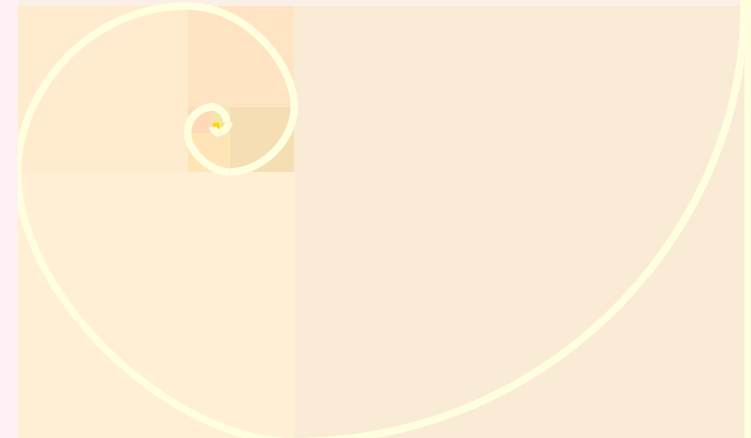
# Selection Sort Analysis

- Space complexity
  - $O(1)$
- Time complexity
  - comparisons:  $(N-1)+(N-2)+1 = N(N-1)/2 = O(N^2)$
  - movements:  $N = O(N)$
- Performance of this algorithm does not depend on the datas
  - Worst case, best case and average case are roughly the same!
    - number of assignments may vary (min = j)

# Insertion Sort

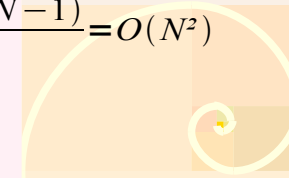
- For the given element at position 'i', move each greater elements on its left to its right
- Insert element 'i' at the free position

```
void sort_insertion(int * t, int N) { // From 1 to N !!
    for (int i = 2; i <= N; i++) {
        int j = i, v = t[i];
        while (j > 1 && t[j-1] > v) {
            t[j] = t[j-1];
            j--;
        }
        t[j] = v;
    }
}
```



# Insertion Sort Analysis

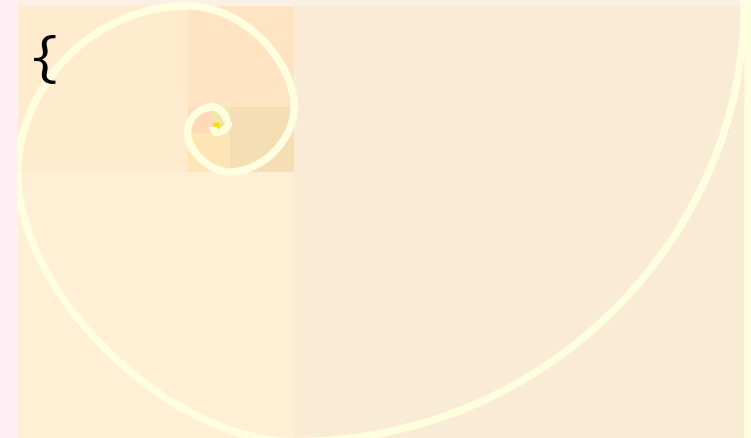
- Space complexity
  - $O(1)$
- Time complexity
  - comparisons:
    - Improvement: the test ( $j>1$ ) is almost always **true**
    - $t[0]=\text{MIN\_KEY}$ , remove the test
    - *worst case* :
    - average case is  $\sum_{i=1}^{N-1} (i+1) = 2+3+\dots+N = \frac{(N+2)(N-1)}{2} = O(N^2)$
  - movements:
    - *worst case* :  $O(N^2)$
    - average case  $\sim N^2/2 = O(N^2)$



# Shell Sort

- Reorder the list to obtain an ordered sublist when considering every 'h'-th elements (for a given h)
- Series of decreasing values of 'h'

```
void sort_shell(int * t, int N) { // From 1 to N !!
    for (int h = N / 9; h > 0; h = h / 3) {
        for (int i = h; i <= N; i++) {
            int j = i, v = t[i];
            while (j > h && t[j-h] > v) {
                t[j] = t[j-h];
                j = j-h;
            }
            t[j] = v;
        }
    }
}
```



# Shell Sort Analysis

- Space complexity
  - $O(1)$
- Time complexity
  - comparisons & movements:
    - Depends on the series used
      - Some are better than others
    - Still unknown in the general case
- Very efficient algorithm for some well known series
  - 1, 4, 13, 40, 121, ...:  $h=3*h+1 : O(N^{3/2})$  comparisons

$N^{3/2}$

# Merging arrays

- Given 2 ordered lists  $s$  and  $t$ , merge them in a list  $u$  so that  $u$  is ordered

```
void merge(int * s, int N, int *t, int P) {
    int * u = malloc((n+p) * sizeof(*u));
    int i = N, j = P;
    s[0] = t[0] = INT_MIN;
    for (int k = N+P; k > 0; k--) {
        u[k] = (s[i] > t[j]) ? s[i--] : t[j--];
    }
}
```

# Merge Sort (array)

```
int s[MAX]; // Bad design!
void sort_merge(int *t, int l, int r) {
    int i, j, k, m;
    if (r <= l) return;
    // divide and conquer
    m = (l+r)/2;
    sort_merge(t, l, m);
    sort_merge(t, m+1, r);
    // create s = t[l]...t[m]t[r]...t[m+1]
    for (i = m; i >= l; i--) s[i]=t[i];
    for (j = m; j < r; j++) s[r+m-j] = t[j+1];
    // merge the two sublists
    for (k = i = l, j = r; k <= r; k++) {
        t[k] = (s[i] < s[j]) ? s[i++]:s[j--];
    }
}
```

# Merging lists

- Given 2 ordered lists s and t, merge them in a list u so that u is ordered

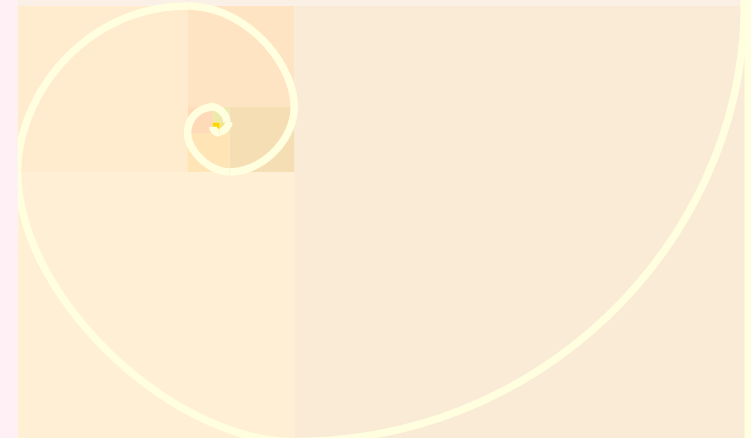
```
list merge(list s, list t) {
    list u = newList(); node up = start(u)->next;
    node sp = start(s)->next, tp = start(t)->next;
    end(s)->key = end(t)->key = INT_MAX;
    do {
        if (key(sp) <= key(tp)) {
            up->next = sp; up = sp; sp=sp->next;
        }else{
            up->next = tp; up = tp; tp=tp->next;
        }
    } while(up != end(s) && up != end(t));
    start(u) = end(u)->next; free(end(u));
    return u;
}
```

# Merge Sort (list)

```
node end; // Any list must end with this node.
node sort_merge(node u) {
    node s, t; // 's': start of first list
    if (u->next = end) return u;
    s = u; t = u->next->next; // 't': search the end
    // Shift 't' 2 times more than 'u'
    while(t != end) {
        u = u->next; t=t->next->next;
    }
    // Makes 't' the start of the second list
    t = u->next; // 'u': end of the first list
    // Makes 's' the start of the first list
    u->next = end; // 's' must end with 'end'
    // Exercice: write this merge() function
    return merge(sort_merge(s), sort_merge(t));
}
```

# Merge Sort Analysis

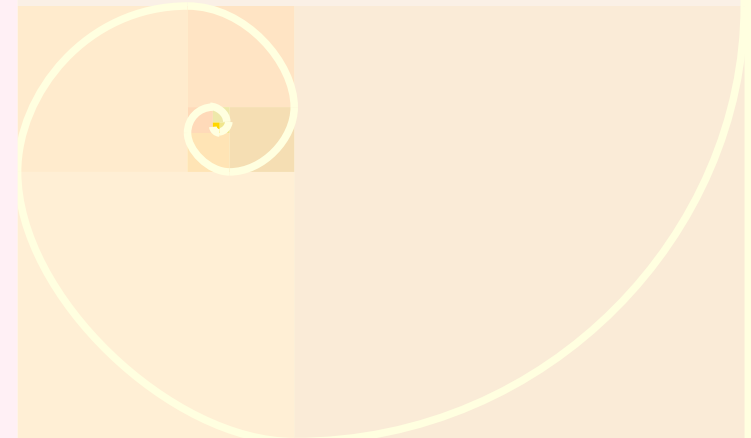
- Space complexity
  - Arrays:  $O(n)$ , List:  $O(1)$
- Time complexity
  - comparisons:  $O(n \cdot \log(n))$
  - Both in the worst and in the average case.
- This algorithm is *stable*
- Very efficient algorithm
  - Requires some space!



# Heap Sort

- Insert all elements of the list in a (max-)heap
- Delete each element one after the other and insert it a the next free position.

```
void sort_heap(int * t, int N) {  
    heap h = newMaxHeap();  
    for (int i = 1; i <= N; i++) heap_insert(h, t[i]);  
    for (int i = N; i >= 1; i--) t[i] = heap_deleteMax(h);  
}
```



# Heap Sort Analysis

- Space complexity
  - Using a heap:  $O(n)$
  - Using an heap backed by the given array:  $O(1)$
- Time complexity
  - comparisons:  $O(2n \cdot \log(n))$
- Efficient algorithm
  - Less efficient than merge sort
  - Does not need additional space

# Quick Sort

- Find an element called '*pivot*' and partition the list so that:
  - any elements at the left of the pivot are lesser
  - any elements at the right of the pivot are greater
- Sort the two sublists at the left and the right of the pivot

```
void sort_quick(int * t, int l, int r) {  
    if (l > r) return;  
    int i = partition(t, l, r);  
    sort_quick(t, l, i-1);  
    sort_quick(t, i+1, r);  
}
```

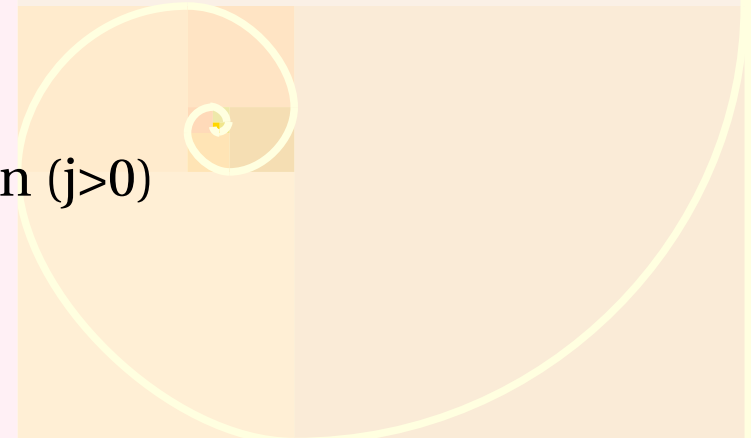
# Quick Sort

```
int partition(int * t, int l, int r) {
    int i = l-1, j=r, v=t[r];
    for(;;) {
        while (t[++i] < v);
        while (t[--j] > v); // check j>0 --> median
        if (i >= j) break;
        swap(t, i, j);
    }
    swap(t, i, r);
    return i;
}
```

SORTINGCHARACTERS  
R R E I N G C H A R A C S T S T  
A A C C I N G R H O R R E S T T  
A A C E N G R H O R R I  
H G I N O R R R  
G H N O R R R  
A A C C E G H I N O R R R S S T T

# Quick Sort Analysis

- Space complexity (a stack is used)
  - worst case:  $O(n)$ , average case is  $O(\log(n))$
- Time complexity
  - Worst case is  $O(n^2)$
  - Average case is  $O(n \cdot \log(n))$  // Best one!!
  - Improve performance by choosing a better pivot
    - random
    - median of (left, middle, right)
      - sort them to prevent the condition ( $j > 0$ )
- Unstable !!

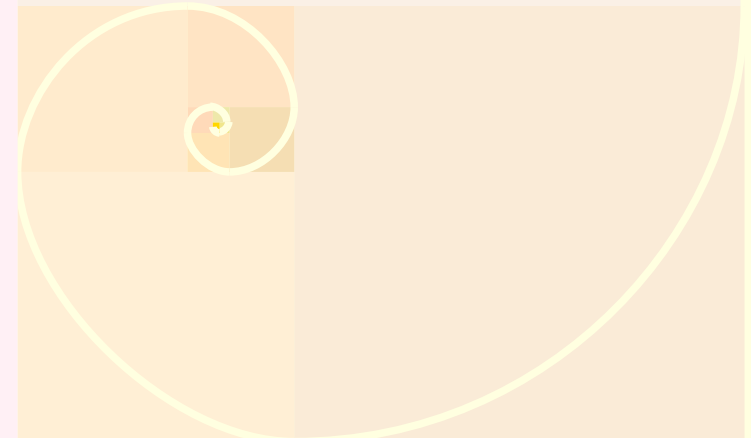


# How fast can we sort?

- Time complexity of simple algorithms
  - $O(n^2)$  but very efficient for small 'n'
- Complex algorithm
  - $O(n \cdot \log(n))$ 
    - space requirement in  $O(n)$  (merge sort)
    - worst case in  $O(n^2)$ , unstable (quick sort)
    - Good compromise: (heap sort) –  $O(2n \cdot \log(n))$
- It can be shown that  $\Omega(n \cdot \log(n))$  comparisons is an average minimum
- But...

# Radix Sort

- How do you sort a deck of cards?
  - Most-Significant-Digit-First (MSD)
    - sort by suits value first --> 4 piles (*bin-sort* using *bins*)
    - sort each bin by face value independently
  - Least-Significant-Digit-First (LSD)
    - bin-sort by face value first --> 13 bins
    - stack each bins,
    - bin sort according to suit



# Radix Sort Example

Consider the binary representation of key

5	101	010	000	000	0
2	010	110	100	001	1
7	111	000	101	010	2
6	110	100	001	011	3
0	000				4
1	001	101	010	100	5
4	100	111	110	101	6
3	011	001	111	110	7
		011	011	111	

# Radix Sort Analysis

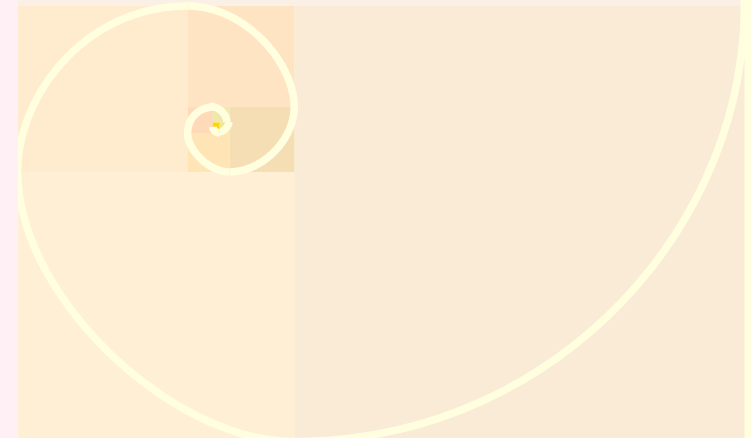
- Instance Characteristics
  - $d$ : the number of digits (keys)
  - $r$ : the radix
- Space complexity
  - ' $n$ ' elements contained in ' $r$ ' bins each pass:  $O(r+n)$
- Time complexity
  - Steps:  $O(d(n+r)) = O(nb)$ , ' $b$ ': number of bits
    - ' $n$ ' insertion into bins for each ' $d$ ' digits
    - ' $r$ ' concatenation of bins for each ' $d$ ' digits
- Linear sort?  $b \sim \log(n)$  !!

# External Sorting

- Consider a huge list that does not fit entirely in memory (usually on disk or tape)
- Access cost to any element is much more important than comparing
- Access may be constrained by the underlying storage device
  - hard drives allow random access
  - tapes only allow sequential access
- Costs of external sort algorithms depend essentially on input/output operations
  - *block* is the unit of data that is read from or written to a storage device at one time.

# External Sorting

- Reduce the Number of times a data element is moved from the storage device to the memory and *vice-versa*
  - Perform these movements as fast as allowed by the hardware
- Hierarchy of memory
  - cache (level 1, 2, 3, ...)
  - RAM
  - Hard Drive
  - Tape



# External Merge Sort

- Divide the original file into  $m$  *runs*
  - $|run| = \text{RAM}$
  - sort each run using an internal sort
- merge sorted runs in several phases
  - $p$  tape are needed for input
  - each phase produces new runs of bigger size
    - they are written on  $p$  output tape
- End when you only have one big run of the original file size
  - original file size =  $n$

# External Merge Sort

ANEXAMPLEFOREXTERNALMERGESORT

$n=29$

$m=4$

$p=3$

tape 1 AENX EETX EORS

tape 2 ALMP ALNR T

tape 3 EFOR EGMR

3-merge

tape 4 AAEEFLMNOPRX

tape 5 AEEGLMNRRTX

tape 6 EORST

3-merge

tape 1 AAAEEEEEEFGLLMNNOOPRRRRTTSXX

tape 2

tape 3

Each  $p$ -merge divide the number of runs by  $p$

VII. Sorting

# External Merge Sort Analysis

- Space complexity
  - 2.p-tapes
  - all the RAM !!
- Time complexity
  - The initial sort produce approximately 'n/RAM' runs.
  - If 'p' tapes are used, the number of phases is roughly  $\log_p (N/\text{RAM})$
- Sorting a 10 Gb file, with a 512 Mb computer and 4 tapes requires 2 phases.

# Searching

# Introduction

- Fundamental operation
- Finding an element in a (huge) set of other elements
  - Each element in the set has a *key*
- Searching is the the looking for an element with a given key
  - distinct elements may have (share) the same key
  - how to handle this situation?
    - first, last, any, listed, ...
- May use a *specialized* data structure

# Sequential Search

- Store elements in an array
  - Unordered

```
// return first element with key 'k' in 't[]';  
// return 'NULL' if not found  
// 't[]' is from 1 to 'N'  
element find(element* t, int N, int k) {  
    t[0].key = k; t[0].value = NULL; // sentinel  
    int i = N;  
    while (t[i--].key != k);  
    // 'i' has been decreased!  
    return t[i + 1];  
}
```



# Sequential Search Analysis

- Generic simple algorithm
- Space complexity
  - $O(1)$
- Time complexity
  - Worst case:  $N + 1$  comparisons
  - Best case: 1 comparison
  - Average case (successful):  $(1+2+\dots+N)/N = (N+1)/2$

# Sequential Search in a (sorted) Linked List

- Keep the list sorted
  - Easy to implement with linked list (*exercice: do it!*)!

```
// return first node with key 'k' in 'l';  
// return 'NULL' if not found  
// 'l' is sorted  
node find(list l, int k) {  
    node z = list_end(l);  
    node_setKey(z, k); // sentinel  
    for (node n = list_start(l);  
         node_getKey(n) > k;  
         n = node_next(n));  
    if (node_getKey(n) != k) return NULL;  
    return n;  
}
```

# Sequential Search in a (sorted) Linked List

- Space complexity
  - $O(1)$
- Time complexity
  - Best case: 1 comparison
  - Average case (successful): same as the sequential search in unordered list (array):  $(N+1)/2$
  - Worst case (unsuccessful):
    - consider the sentinel as part of the list
    - then a search is always “successful” (finding the sentinel at least)
    - Hence:  $(N+2)/2$

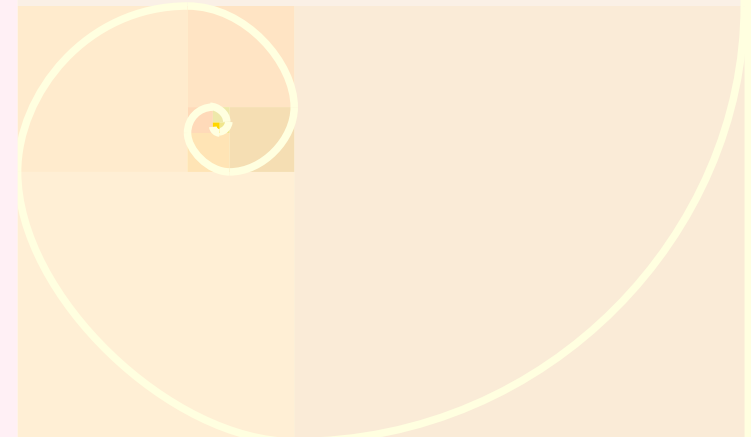
# Sequential Search Improvements

- Static caching
  - Use the *relative access frequency* of elements
    - store the *most often accessed* elements at the first places
- Dynamic caching
  - For each access, move the element to the *first* position
    - Needs a linked list data structure to be efficient
- Very difficult to analyse the complexity in theory
  - Very efficient in practice

# Dichotomic Search

- *divide and conquer* algorithm
- Constraint: the list must be *ordered*

```
// return first element with key 'k' in 't[]';  
// return 'NULL' if not found  
// 't[]' is from 1 to 'N'. It is sorted  
element find(element* t, int N, int k) {  
    int l = 1, r = N, x;  
    while(l < r) {  
        x=(l+r)/2;  
        if (k == t[x]) return t[x];  
        if (k < t[x]) r=x-1;  
        else l=x+1;  
    }  
    return NULL;  
}
```

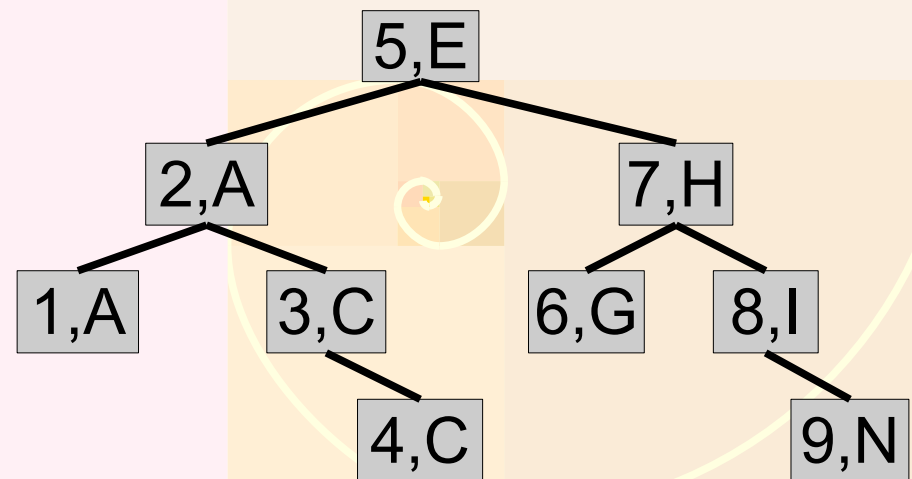


# Dichotomic Search Analysis

- Space Complexity:  $O(1)$
- Time Complexity
  - Best Case: 1 comparison
  - Worst Case and Average Case:  
 $C(N) = C(N/2) + 1$  comparisons,  $C(1) = 1$ ;  
 $C(N) = \lg(N) + 1$

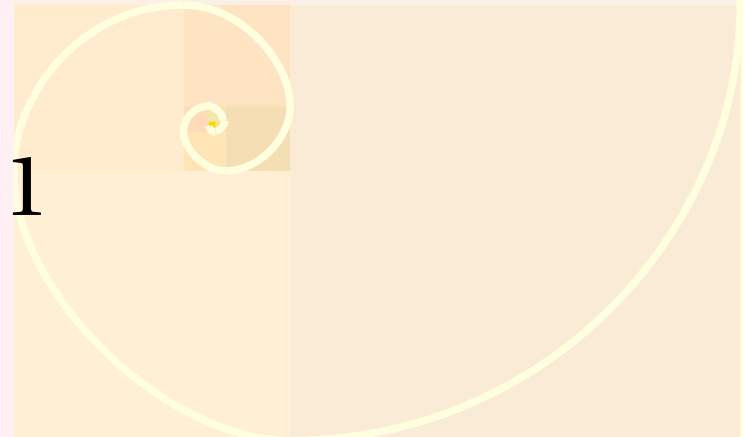
Comparisons can be represented by a binary tree

123456789  
AACCEGHIN



# Interpolation Search

- Improvement of the dichotomic search
  - Inspired by human way of searching into a dictionary
- Instead of  $x = l + (l+r)/2$ , use the searched key to estimate the location of the element:
  - $d = k - t[l].key$ : difference from the left key
  - $D = (r-l) / (t[r].key - t[l].key)$ : distribution coefficient
  - $x = l + d * D$ ;
- Time Complexity:  $\lg(\lg(N)) + 1$ 
  - For  $N = 10^9$ ,  $\lg(\lg(N)) < 5$  !

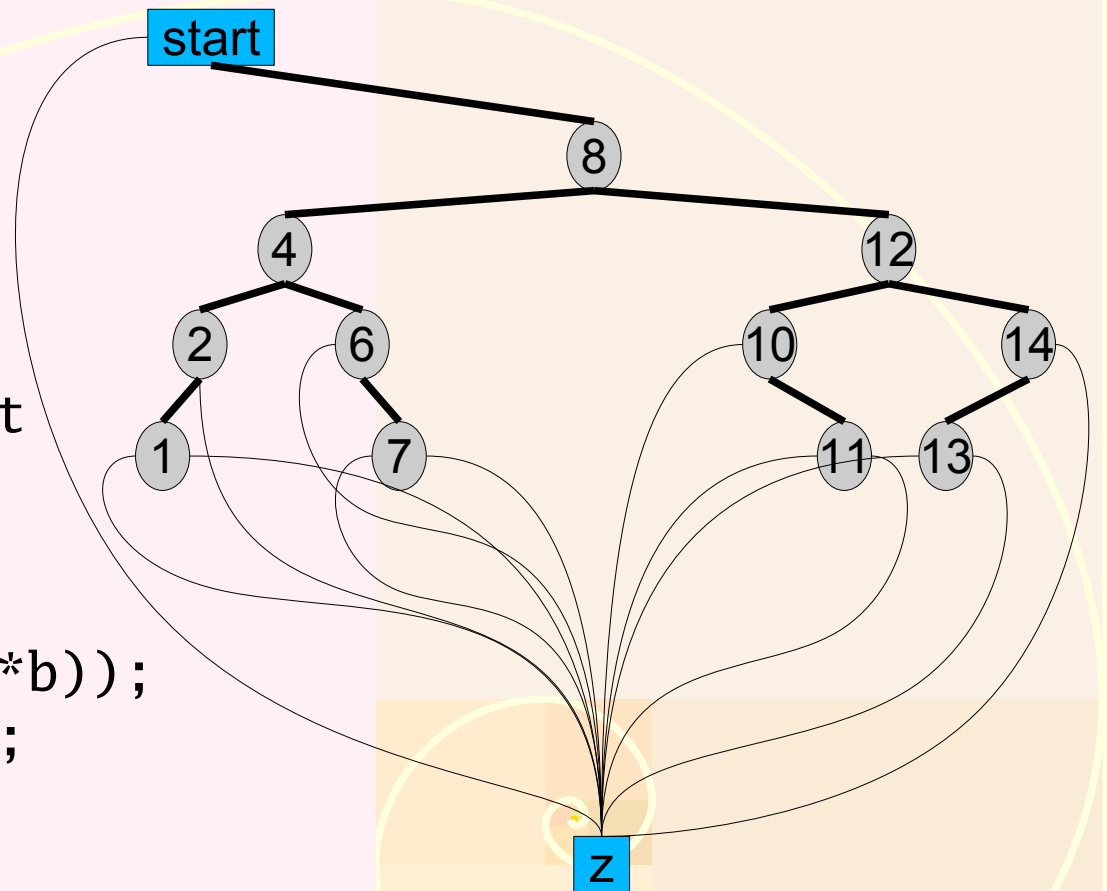


# Binary Search Tree

- Solves issues of preceding search algorithms
  - Insertion, deletion and searching can be done in  $\lg(N)$  steps in the average case
- Very simple data structure
  - easy to understand, easy to implement
- A BST is a binary tree such that for each node:
  - its *left child* has a key which is *strictly less* than its own
  - its *right child* has a key which is *greatest or equal* to its own

# BST Implementation

```
struct node{
    int k;
    char v;
    node left, right;
};
struct bst{
    node start; // smallest
    node z; // sentinel
}
bst newBST() {
    bst b = malloc(sizeof(*b));
    node start = newNode();
    start->k = MIN_INT;
    node z = newNode();
    z->left = z->right = z;
    b->start->left
    return b;
}
```



What is the output of an infix traversal of a BST?

# BST Implementation

```
void bst_insert(bst b, int k, char v) {
    node p = b->start, n=b->start->right;
    while (n != b->z) {
        p = n;
        n = (k < n->k) ? n->left : n->right;
    }
    n = newNode(); n->k = k; n->v = v;
    n->left = n->right = b->z;
    if (k < p->k) p->left = n;
    else p->right = n;
}

node find(bst b, int k) {
    node n = b->root; b->z->k=k; // sentinel
    while(k != n->k) {
        n = (k < n->k) ? n->left : n->right;
    }
    return n;
}
```

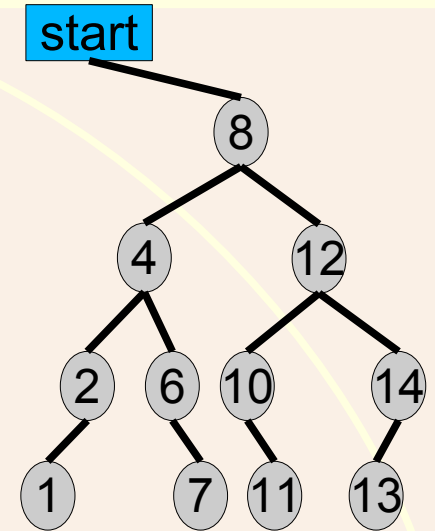
# BST Analysis

- Average number of comparisons for the searching in the following BST
  - ANEXAMPLEOFABST: 4.4
  - MENALXAEOPAFSBT: 4
  - AAABEEFLMNOPSTX: 8
  - AXATASBPEOENFML: 8
  - Average for the dichotomic search (10 elements):  
 $\lg(15)+1 = 4.9$
- Worst case can be linear!



# BST Deletion Implementation

```
void bst_delete(bst b, int k) {
    node p = b->start, n = b->start->right;
    b->z->k = k;
    while (k != n->k) {
        p = n;
        n = (k < n->k) ? n->left : n->right;
    }
    node t = n;
    if (t->right == b->z) n=n->left; //t:2,n:1,4->l:1
    else if (t->right->left == b->z) { //t:4
        n=n->right; n->left=t->left; //n:6, 6->l:2,8->l:6
    } else { //t:8,n:8
        node c = n->right; //c:12
        while(c->left->left != b->z) c=c->left;
        n=c->left; c->left=n->right; // n:10,12->l:11
        n->left = t->left; n->right = t->right; //10->l:4,10->r:12
    } // p:start, start->k = MIN_INT, start->r:10
    free(t);
    if (k < p->k) p->left=n; else p->right=n;
}
```



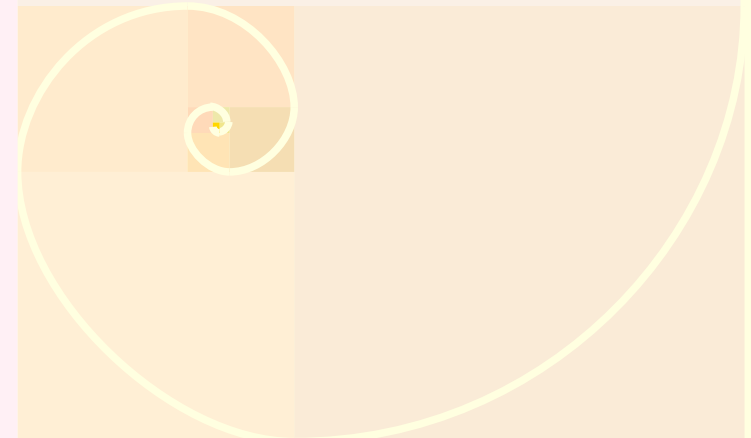
# Balanced Trees

# Concept

- BST has poor worst case performance
  - $O(N)$  comparisons
  - worst case is common in practice
- “Balanced” BST has good average performance
  - $O(\lg(N))$  comparisons
- Can we balance BST “automatically”?
  - [Adelson, Velskii and Landis, 1962]: AVL Tree

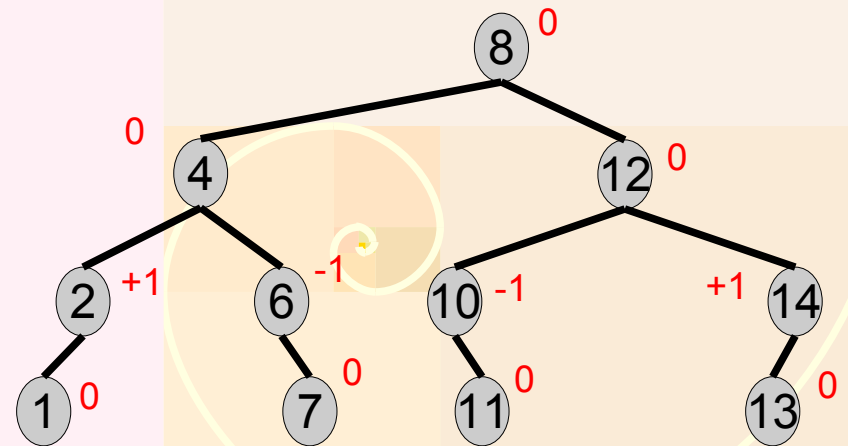
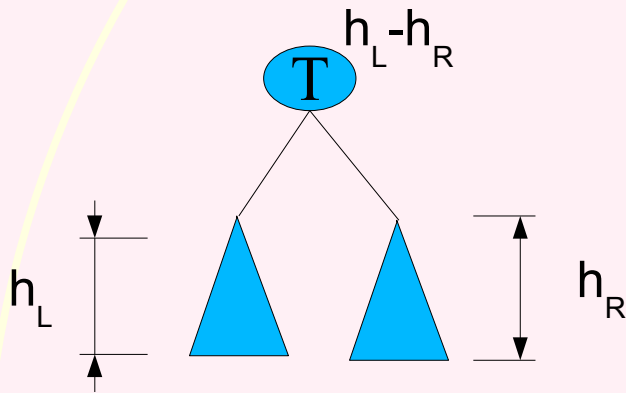
# AVL Tree Definition

- An empty tree is height-balanced
- If  $T$  is a non empty binary tree with  $T_L$  and  $T_R$  as its left and right subtrees respectively
- $T$  is *height-balanced* iff
  - $T_L$  and  $T_R$  are height-balanced
  - $|h_L - h_R| \leq 1$  where  $h_L$  and  $h_R$  are the heights of  $T_L$  and  $T_R$  respectively



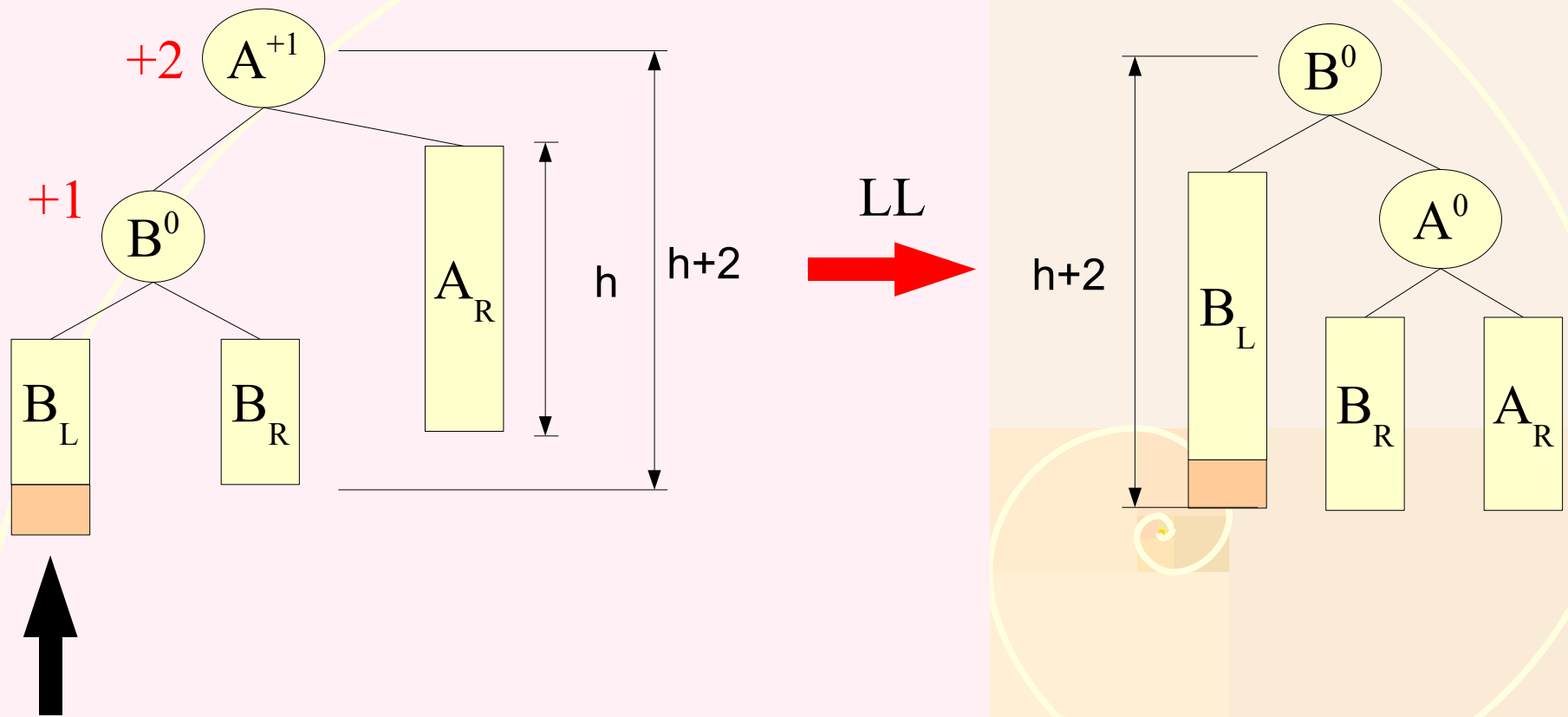
# AVL Tree Definition

- The *balance factor*  $BF(T)$  of a **node**  $T$  in a tree is:
  - $BF(T) = h_L - h_R$
- For any node  $T$  in an AVL tree,  $BF(T) = -1, 0$  or  $1$ .



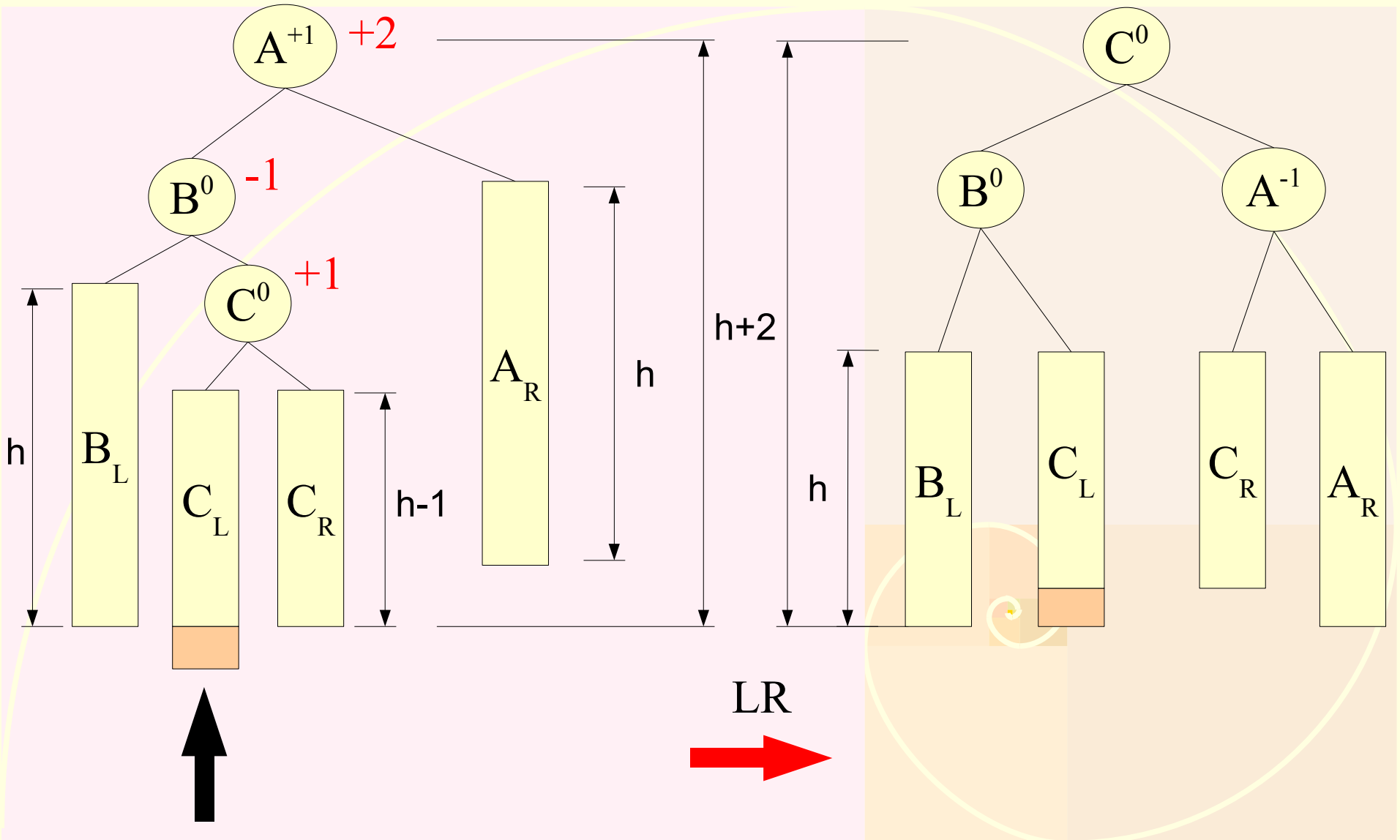
# AVL Transformations

## Left-Left Rotation



# AVL Transformations

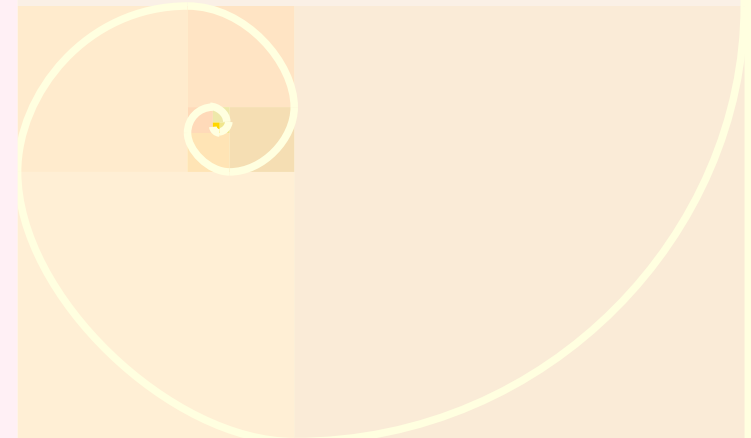
## Left-Right Rotation



# AVL Transformations

## Rotations

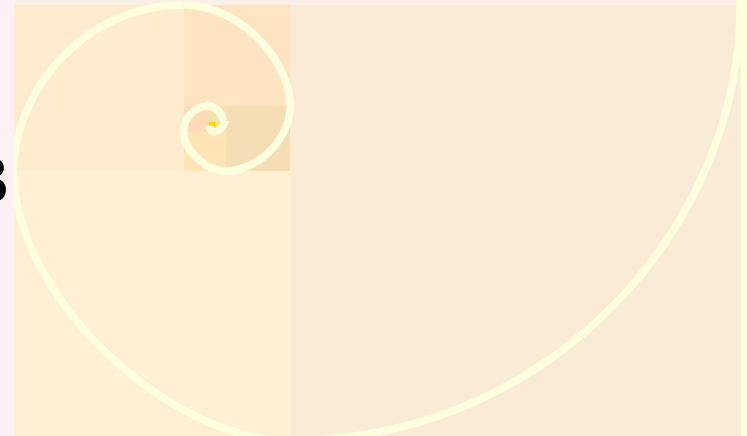
- Four kinds of rotations characterized by the nearest ancestor  $A$  of the inserted node  $Y$  whose  $BF(A)$  becomes  $\pm 2$ 
  - LL:  $Y$  is inserted in left subtree of the left subtree of  $A$
  - RR:  $Y$  is inserted in right subtree of the right subtree of  $A$
  - LR:  $Y$  is inserted in right subtree of the left subtree of  $A$
  - RL:  $Y$  is inserted in left subtree of the right subtree of  $A$
- LL and RR are symmetric
- LR and RL are symmetric



# AVL Transformations

## Examples

- AVL Tree after the following insertions
  - XTSPONMLFEEBAAA and AAABEEFLMNOPSTX
    - Solution (level order traversal): LBPAENTAAEFMOSX
  - AXATASBP EOENFML
    - Solution: EAPABMTAFNSXELO
  - ANEXAMPLE OF FABST
    - Solution: LEPAENTAAFMOSXB
  - MENALXAEOPAFSBT
    - Solution: MESAFOXAAELNPTB

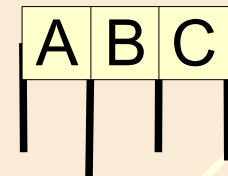
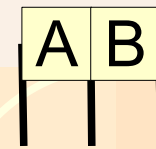
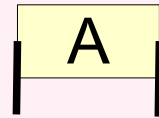


# Algorithm and Analysis

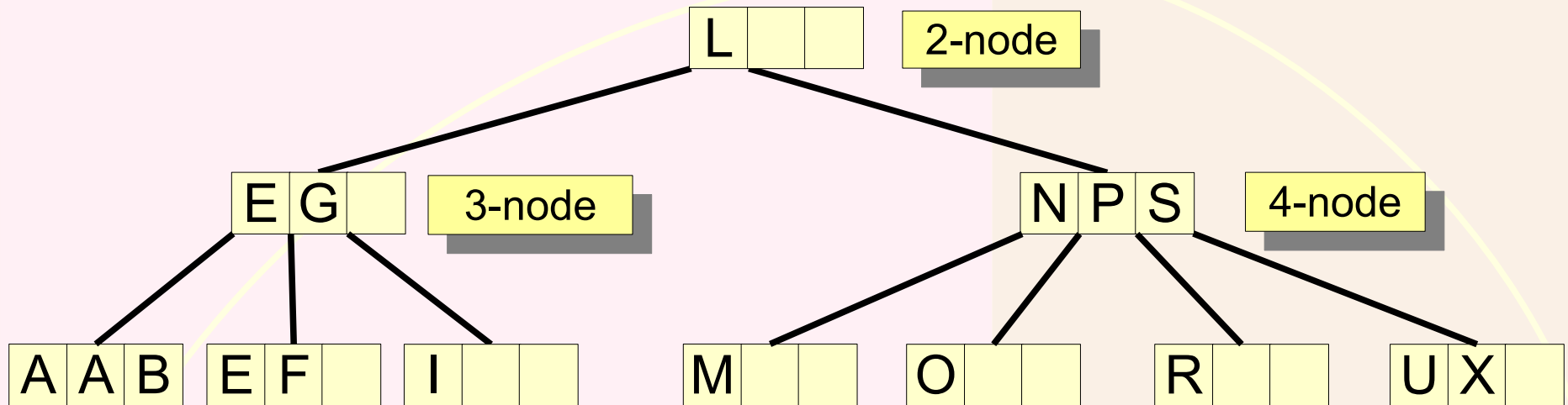
- Algorithm of insertion: 2 pages of C code!
  - Refer to the book
  - Time complexity:  $O(h)$ ,  $h$ : height of the tree
  - Same as in BST, but overhead of insertion is high (the constant hidden by the 'big O' notation is large)
  - in BST,  $h$  can be equal to ' $n$ ', in AVL,  $h < \lg(n)$
- Search is always in  $O(\log(n))$  on average
  - The BST worst case  $O(n)$  never happens, AVL trees are always balanced.

# 2-3-4 Trees

- Extension of BST to 4-degree trees
- Have good properties (always balanced)
- One node may contain
  - 1 key: it is a 2-node
    - 2 childs (less; greater)
  - 2 keys: it is a 3-node
    - 3 childs (less, middle, greater)
  - 3 keys: it is a 4-node
    - 4 childs (less, midLeft, midRight, greater)



# Searching in a 2-3-4 Tree



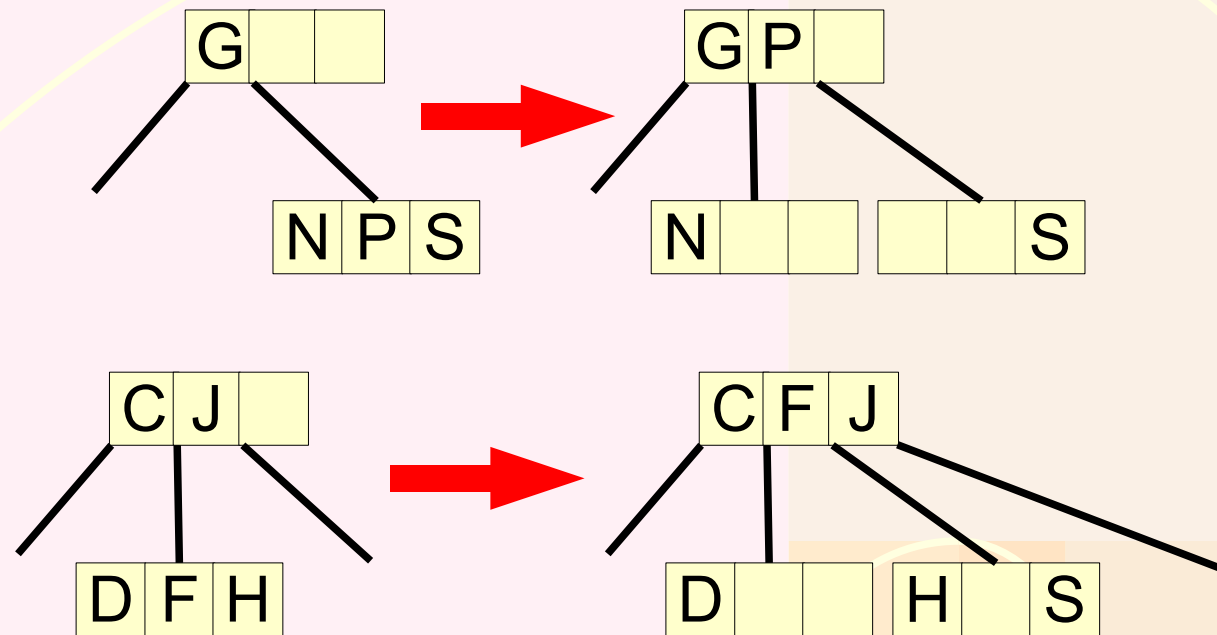
Algorithm for searching 'F':

- it is before 'L', hence it must be at the left of 'L';
- it is after 'E', so it must be at its right;
- it is before 'G', hence it is **in between**;
- it is after 'E', hence it must be at its right.
- Yes it is!

Searching for 'Q' leads to the following comparisons: L, N, P, S, R

# Split operation in 2-3-4 Tree

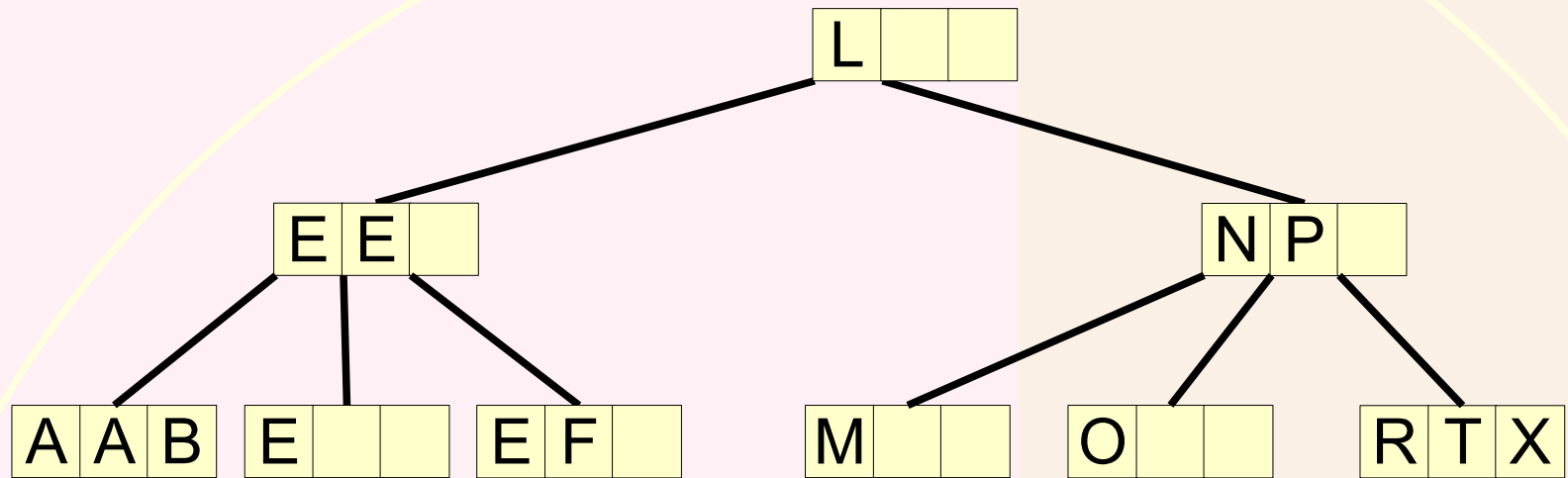
- Split 4-degree nodes into two 2-nodes



Split on top-bottom, root to leaf traversal!

# 2-3-4 Tree Insertion Example

- ANEXAMPLEOFBTREE

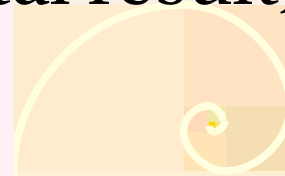


Try the following examples:

- AABEEEEFLMNOPRTX
- XTRPONMLFEEEEBA
- AXATBREPEOENEMFL

# 2-3-4 Tree Analysis

- Always balanced
- Searching
  - $O(\lg(n))$  comparisons
- Insertion
  - $O(\lg(n))$  comparisons
  - $\lg(n)$  splits in the worst case
  - 1 split in average (experimental result)
- Hard to implement
- Big overhead

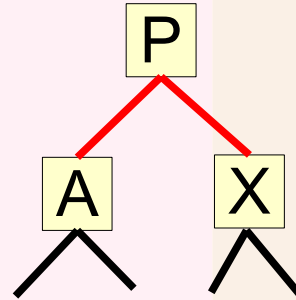
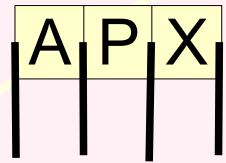


# Red-Black Tree (RBT)

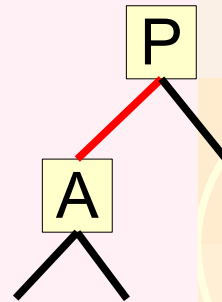
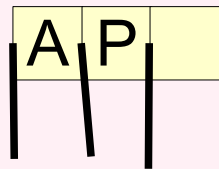
- Binary Tree representation of 2-3-4 Tree
  - Use one bit more by node (red/black color)
  - The color of a node represents the color of the link pointing to itself.
- Many characteristics
  - Always *almost* well-balanced
  - Never two consecutive red links on a path from root to any node
  - For any two such path, their number of black links are equals
- Easier to use and implement than 2-3-4 tree

# 2-3-4-Tree -- RBT Transformation

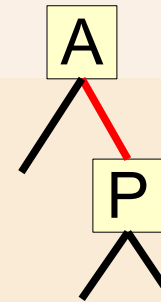
4-node



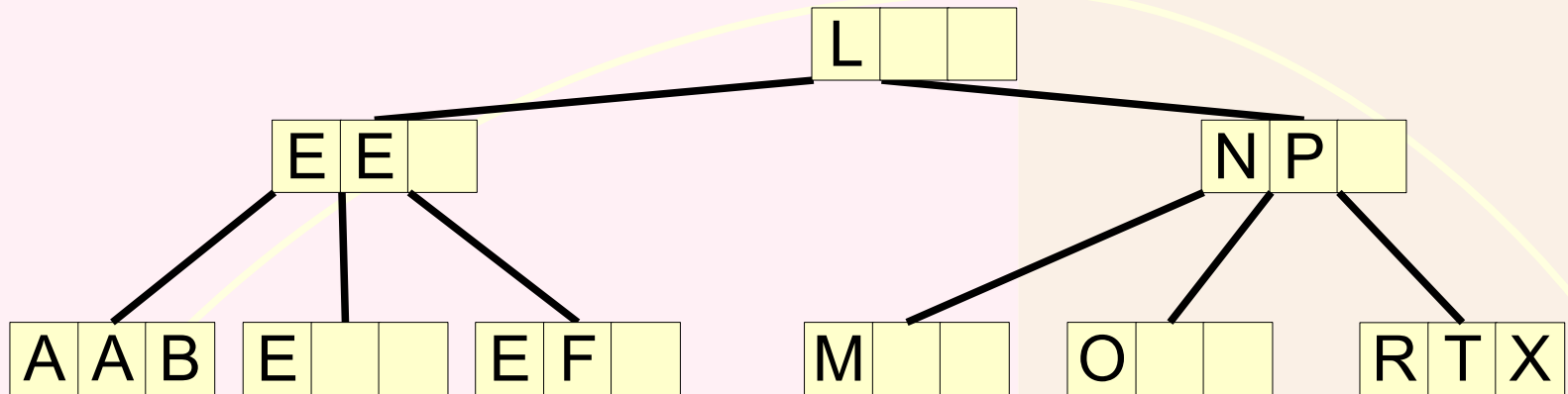
3-node



OR



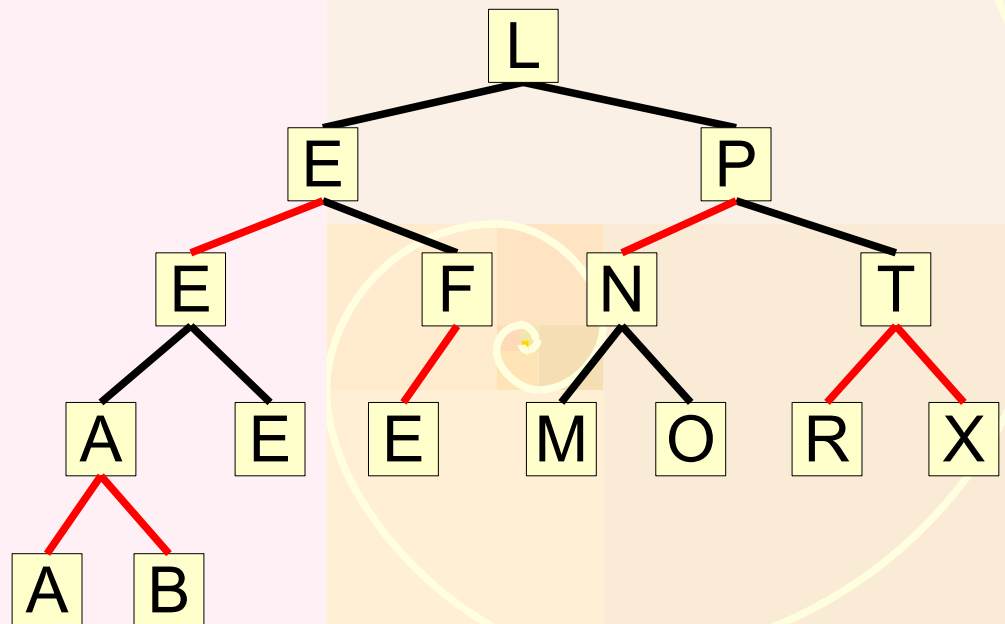
# 2-3-4-Tree -- RBT Transformation Example



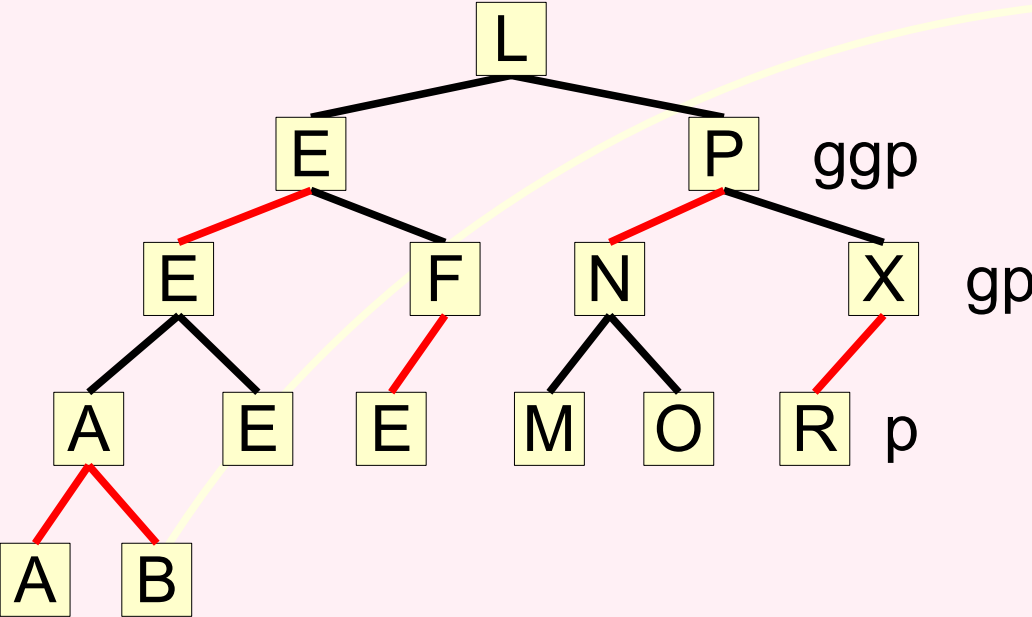
Try the following examples:

- AABEEEEFLMNOPRTX
- XTRPONMLFEEEEBA
- AXATBREPEOENEMFL

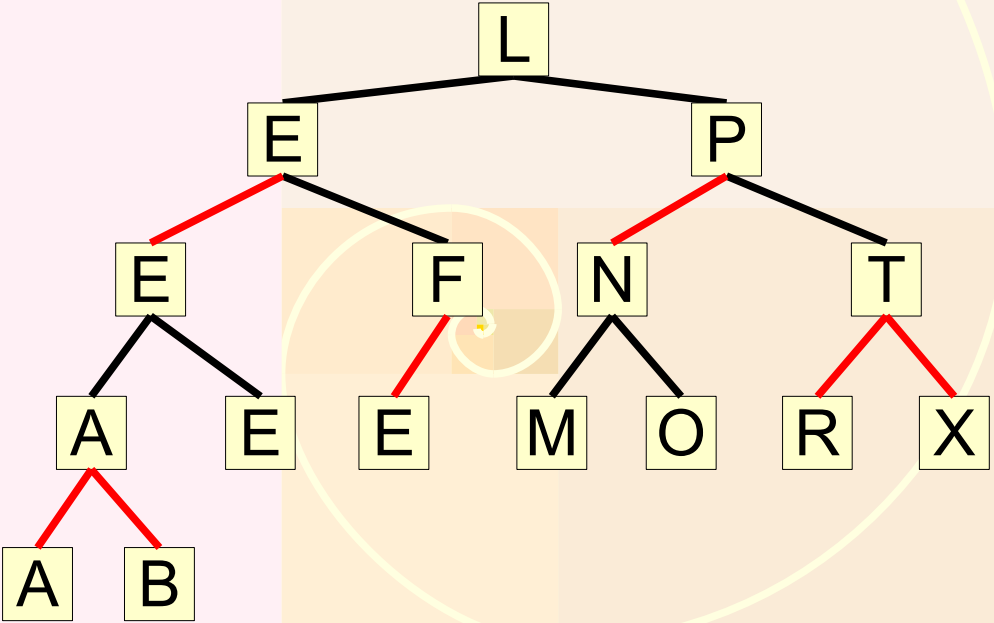
BST searching algorithm works as is!



# Insertion Example

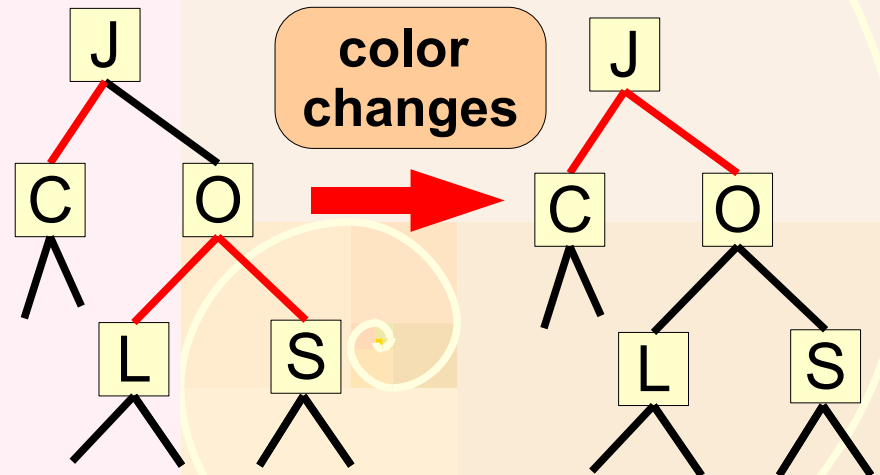
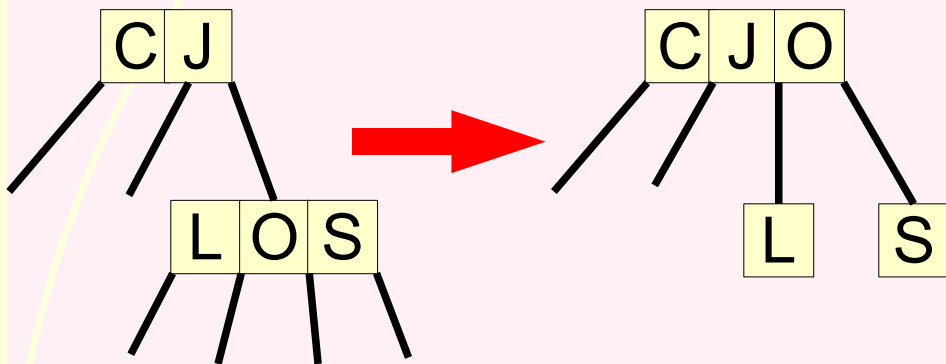
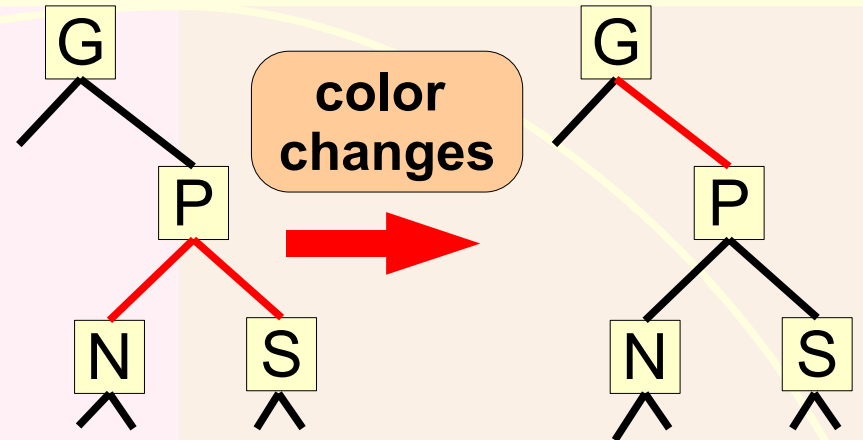
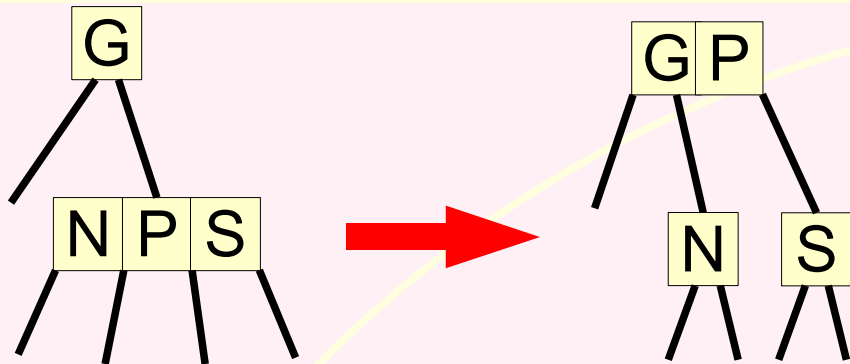


Insert 'T'



# Transformations (Easy cases)

## IX. Balanced Trees

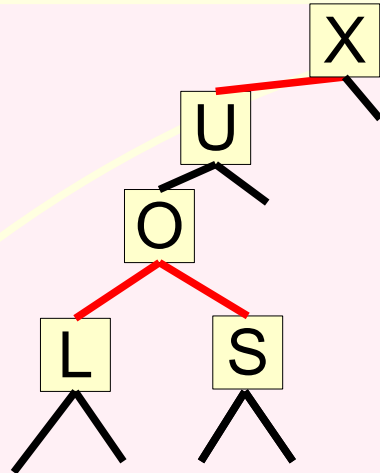
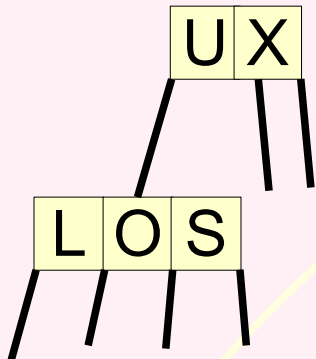


2-3-4 Trees

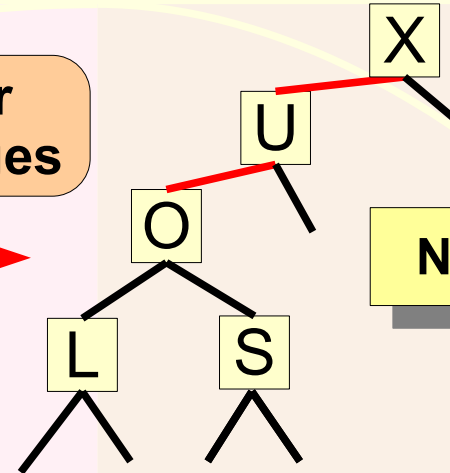
RBT

# Transformations (Hard cases)

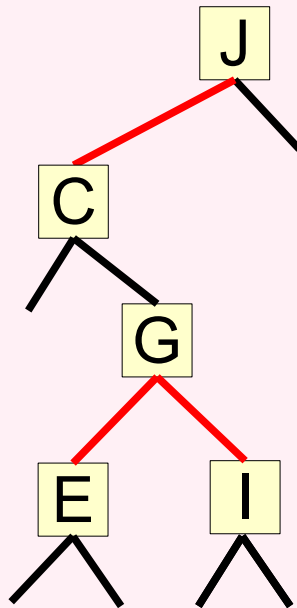
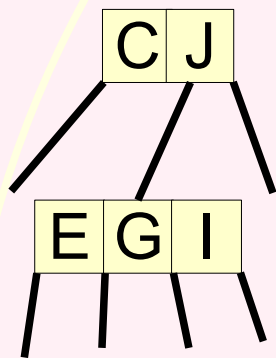
## IX. Balanced Trees



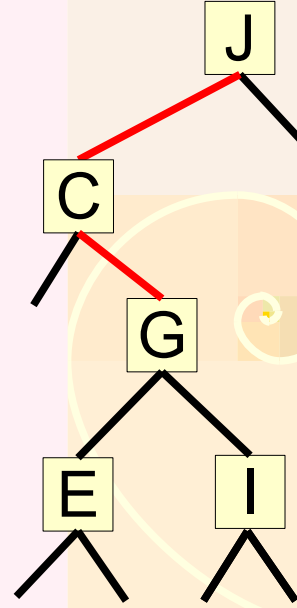
color changes



Not an RBT !!

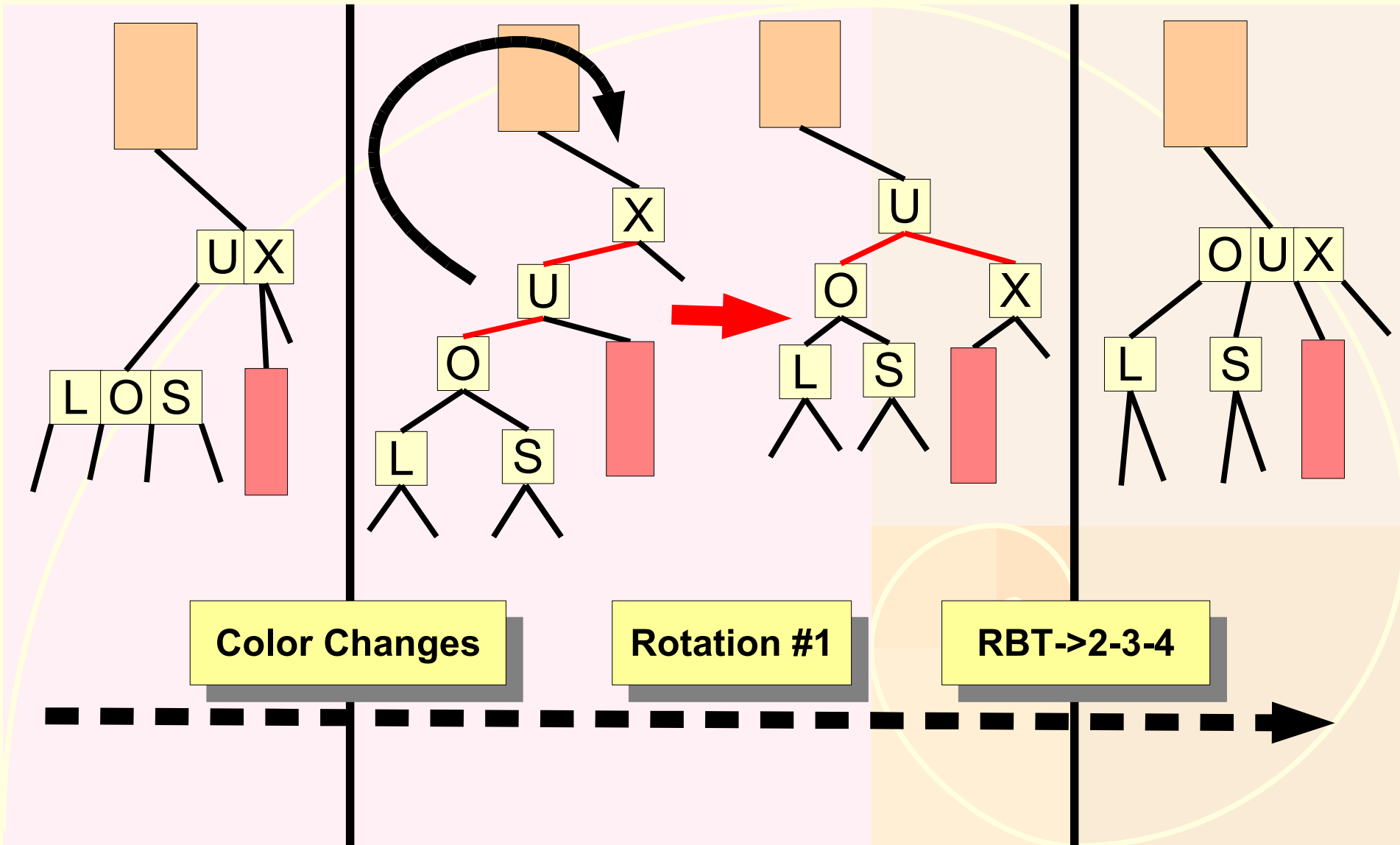


color changes

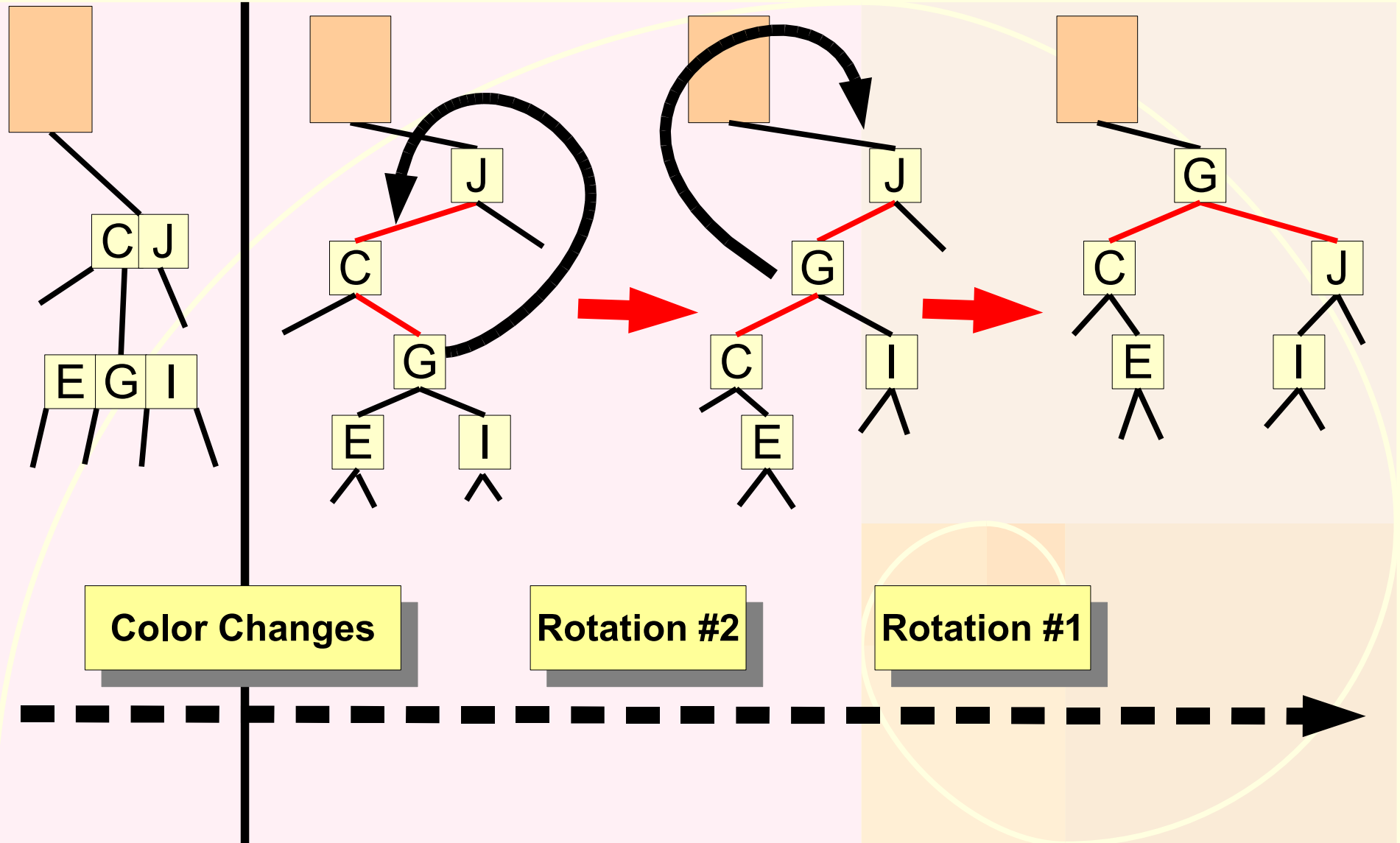


Not an RBT !!

# Transformation First Hard Case

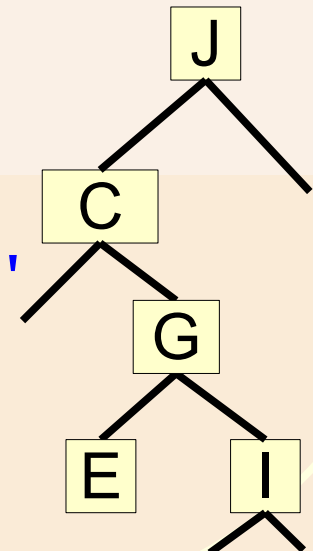
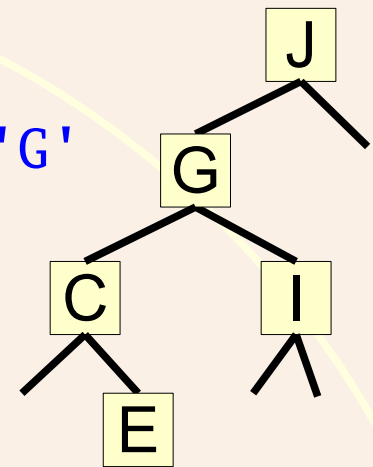


# Transformation Second Hard Case



# RBT Rotation

```
node rotate(int k, node y) { // k='D', y:'J'
  node c, gc;
  c = (k < y->k) ? y->left : y->right; // c:'G'
  if (k < c->k) {
    gc = c->left; // gc:'C'
    c->left=gc->right; // 'G'->left:'E'
    gc->right=c; // 'C'->right:'G'
  }else{
    gc=c->right;
    c->right=gc->left;
    gc->left=c;
  }
  if (k < y->k) y->left = gc; // 'J'->left:'C'
  else y->right = gc;
  return gc; // return 'C'
}
```



**Generic Function, not just for RBT  
(Does not use coloration)**

# RBT Split

```
node split(rbt b, // Red Black Tree
           int k, node n, // key, four-node to split
           node p, node gp, // parent, grand-parent of n
           node ggp) { // grand-grand-parent of n
    // Change colors (n is a four node (see rbt_insert()))
    n->red=TRUE; n->left->red=n->right->red=FALSE;
    if (p->red) { // Hard case #1
        gp->red=TRUE;
        if (k < gp->k != k < p->k) { // Hard Case #2
            p=rotate(k, gp); // Rotation #2
        }
        n=rotate(k, ggp); // Rotation #1
        n->red=FALSE;
    }
    b->start->right->red=FALSE;
}
```

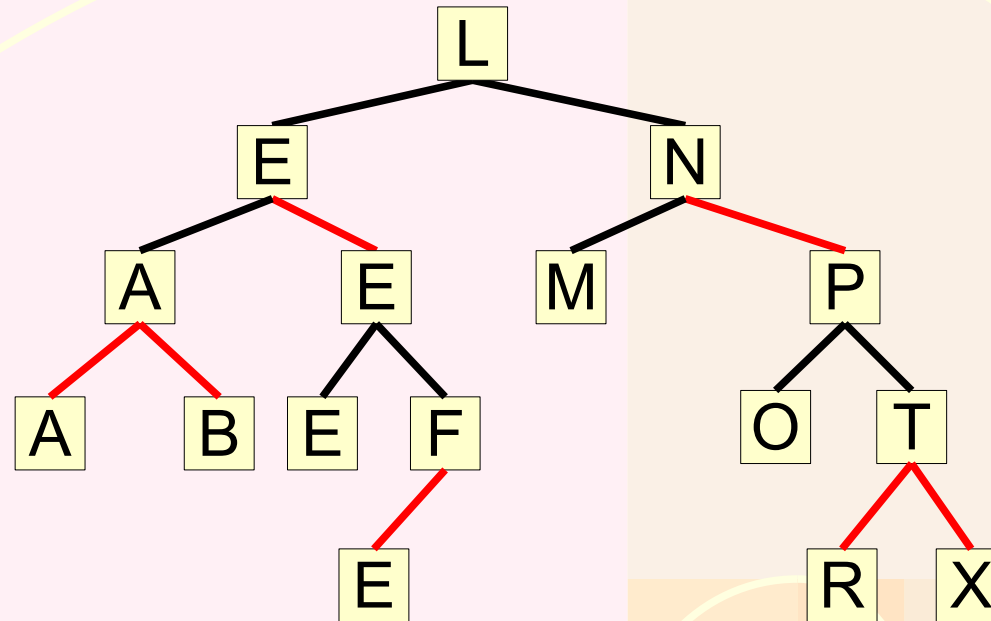
**Initialize the sentinel as a BLACK node!**

# RBT Insertion

```
void rbt_insert(rbt b, int k, char v) {
    node p=b->start, // parent of n
        gp=b->start, // grand-parent of n
        ggp, // grand-grand-parent of p
        n=b->start;
    while (n != b->z) {
        agp = gp; gp = p; p = n; // Update links
        n = (k < n->k) ? n->left : n->right; // Move
        if (n->left->red && n->right->red) {
            n = split(n,p,gp,agp);
        }
    }
    n = newNode(); n->k = k; n->v = v; // Insert
    n->left = n->right = b->z;
    if (k < p->k) p->left = n;
    else p->right = n;
    n = split(n,p,gp,agp);
}
```

# Red-Black Tree Insertion Examples

- ANEXAMPLEOFBTREE

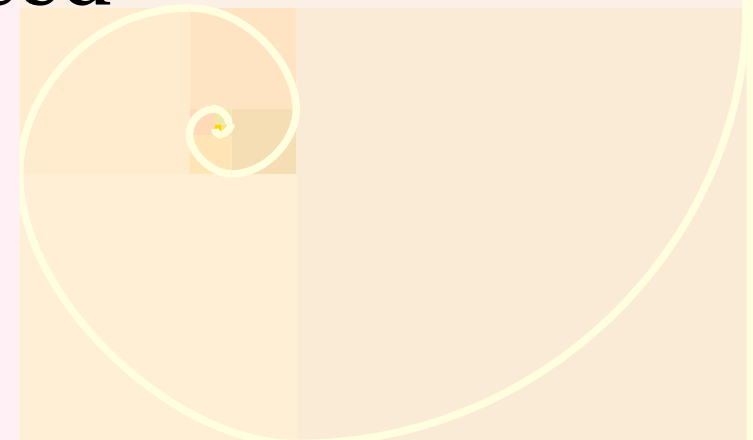


Try the following examples:

- AABEEEEFLMNOPRTX
- XTRPONMLFEEEEBA
- AXATBREPEOENEMFL

# Red-Black Tree Analysis

- Same as 2-3-4 Trees
  - Searching:  $O(\lg(n))$  comparisons (as efficient as the standard BST search)
  - Insertion:  $O(\lg(n))$  comparisons
  - less than 1 rotation on average
- Much less overhead than 2-3-4 Trees
- Always “almost” well balanced
  - Worst case is still in  $O(\lg(n))$



# B-Trees

- Generalization of 2-3-4 Trees
  - A node can have up to  $m-1$  keys (and so 2 to  $m$  links).
  - Split a node on insertion (top-bottom traversal)
- Widely used in “External Searching”
  - Reduce the number of disk access by the use of a high  $m$  value
- Used in several filesystems
  - XFS, ReiserFS
  - <http://www.namesys.com/>: a **must read**



# Hashing

# Hashing

- Very different method for searching
  - does not use key comparison as the core search engine (*Equal keys are usually not supported*)
    - transform (*hash*) the key into a number instead and use this number as an index in an array to store the element
      - Example: Inserting 'BAD' (hash function:  $h(x) = \text{place of } x \text{ in the alphabet modulo } 4$ , 4: size of the array)

0	1	2	3
A	B		D
      - Consider the example 'BAY': *collision!* ( $h(Y) == h(A)$ )
  - Two things are to be done
    - finding a good hash function
    - finding a good handling of collisions

# Hashing

- Compromise between time and size
  - if you have unlimited space, you can use an  $O(1)$  algorithm for searching using the memory address as the hash function.
  - if you have unlimited time, you can use a minimum of memory by using a sequential search.
- Hashing is in between this two extremes.



# Hash Function

- Requirements
  - easy to compute (fast)
  - minimize collisions
- Uniform hash function
  - if the size of the array is  $m$ , then for any random key  $x$ , the probability of ' $h(x)=i$ ' for all buckets  $i$  must be  $1/b$
- Uniform hash function is not a requirement but it is *a good* behavior.

$h(x)=\&x$ ; may be a good hash function (but not for strings)

# Hash Function

- key may be of any type (not just an integer)
  - First step: given a key 'k', return a integer 'f(k)=x'
  - Second step: return h(x), the hash value of 'k'
- $x: x_n \dots x_0$  - B:base  $f(x) = \sum_{i=0}^{i=n} x_i \cdot B^i$   
 Example:  $f(BAD) = 1 \cdot 26^2 + 0 \cdot 26^1 + 3 \cdot 26^0 = 677$
- $h_b(x) = x \% b$ , size of the array  $M \geq b$

To minimize collisions:

- $b=M$ ,
- $M > 20$ , prime number

	AN	EXAMPLE	OF	HASHING	USING	A	LONG	SENTENCE
	2 0	1	0	1	1	1	1	1
	5 3	4	4	4	0	0	4	2
	8 0	5	4	7	7	1	7	3
	11 8	1	1	7	1	10	10	4
	26 0	17	18	19	19	13	19	17

# Handling Collisions (Overflow Handling)

- Two main approaches
  - Open addressing
    - Linear Probing
  - Chaining



# Linear Probing

- Consider the hash table as an array.
- Insertion
  - For an element 'x', insert x at position  $h(x)$  in the array if it is free ;
  - otherwise, find the next free position in the array, and insert x at this place.
- Searching
  - Perform a sequential search for x, starting at  $h(x)$  until you reach a free position or x.

# Linear Probing

AN EXAMPLE OF HASHING (M=20)

Array: AAAXEEFHIGLMNOPNS

Average (success):  $32/18 = 1.8$

We simulate collisions by inserting equal keys

TRY ANOTHER EXAMPLE (M=20)

Array: AAXYEEHLMNOPRRTT

Average (success):  $27/17 = 1.5$

Average gets worse when table is full.

Solutions:

- Allocate a bigger array, and rehash everything
- Use a second hash function on failure

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

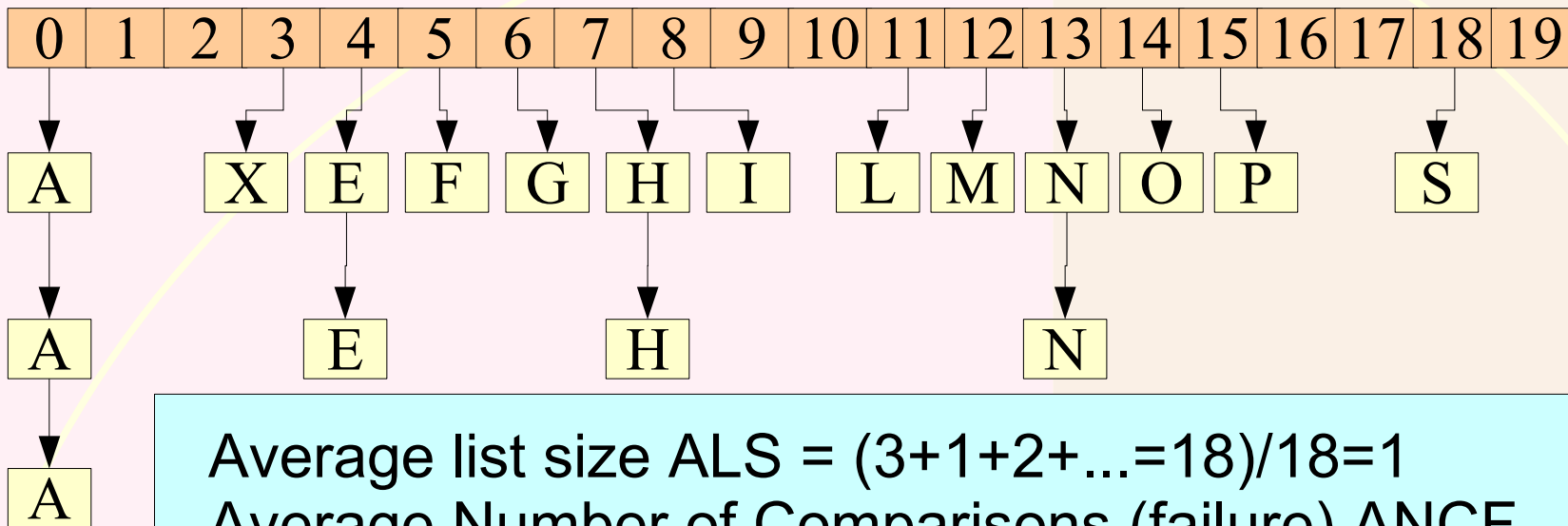
# Chaining

- Use an array of list
- list\* map
- Given a key 'x', map[h(x)] is a list
- Search for the key in this list
  - Sequential search is sufficient if the hash function is good (lists are small)



# Chaining

AN EXAMPLE OF HASHING (M=20)



Average list size ALS =  $(3+1+2+\dots=18)/18=1$   
 Average Number of Comparisons (failure) ANCF  
 ANCF = ALS (ALS/2 if lists are ordered)  
 Average Number of Comparisons (success) ANCS  
 ANCS =  $13*1+4*2+1*3/18 = 24/18=1.3$

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

# Hashing Analysis

- Space Complexity
  - Linear Probing:  $O(M)$ 
    - does not depend on  $N$  but  $M > N$
  - Chaining:  $O(M+N)$  (links)
- Time Complexity ( $\alpha = N/M$ , load factor)

**Linear Probing**

Success:  $\frac{1}{2} \left[ 1 + \frac{1}{1-\alpha} \right]$

Failure:  $\frac{1}{2} \left[ 1 + \frac{1}{1-\alpha} \right]$

**Chaining**

Success:  $1 + \frac{\alpha}{2}$

Failure:  $\alpha$

# Graphs

# Definitions

- Most widely used of all mathematical structures
  - roads, electrical circuits, networks, ...
- A graph  $G$  is defined by 2 sets  $V$  and  $E$ 
  - $V$  is a finite, non empty set of *vertices*
  - $E$  is a set of pairs of vertices; these pairs are called *edges*
  - $V(G)$ : set of vertices of graph  $G$
  - $E(G)$ : set of edges of graph  $G$
  - $G=(V,E)$  ; another notation

# Vocabulary

- *undirected graph*: the pair of vertices representing any edge is *unordered*
  - $(u,v)$  and  $(v,u)$  represent the same edge
- *directed graph*: the pair of vertices representing any edge is *ordered*
  - $\langle u,v \rangle$  and  $\langle v,u \rangle$  does not represent the same edge
  - in  $\langle u,v \rangle$ ,  $u$  is the head,  $v$  is the tail
- $(v,v)$  or  $\langle v,v \rangle$  is not allowed in normal graph (*graph with self edge*)
- a graph cannot have multiple instance of the same edge

# Characteristics & Vocabulary

- Maximum number of edges in an undirected graph with  $n$  vertex is  $n(n-1)/2$ 
  - in a directed graph:  $n(n-1)$
  - an  $n$ -vertex undirected graph with exactly  $n(n-1)/2$  edges is said to be **complete**
- $(u,v)$  an edge of  $E(G)$ 
  - $u$  and  $v$  are *adjacents*
  - $(u,v)$  is *incident* on both  $u$  and  $v$
- $\langle u,v \rangle$  a directed edge of  $E(G)$ 
  - $u$  is *adjacent to*  $v$ ,  $v$  is *adjacent from*  $u$

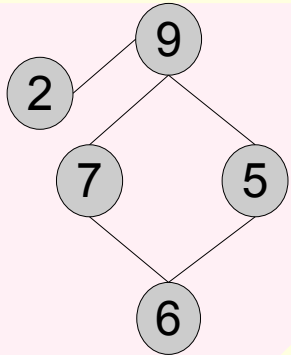
# Characteristics & Vocabulary

- A subgraph of  $G$  is a graph  $G'$  such that  $V(G') \subseteq V(G)$  and  $E(G') \subseteq E(G)$
- A *path* from  $u$  to  $v$  in  $G$  is a sequence of vertices  $u, i_1, \dots, i_k, v$  such that  $(u, i_1), (i_1, i_2), \dots, (i_k, v)$  are edges in  $E(G)$ 
  - length of a path is the number of edges in it
  - a *simple path* is a path in which all vertices except possibly the first and last are distinct
  - A cycle is a simple path in which the first and last vertices are the same.

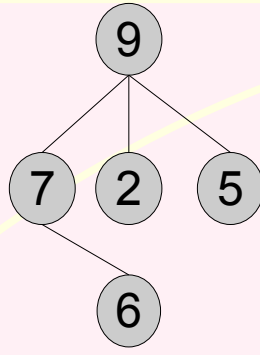
# Characteristics & Vocabulary

- In an undirected graph  $G$ , two vertices  $u$  and  $v$  are said to be **connected** iff there is a path in  $G$  from  $u$  to  $v$
- An undirected graph is said to be *connected* iff for every pair of distinct vertices  $u$  and  $v$  in  $V(G)$ , there is a path from  $u$  to  $v$  in  $G$ .
- A connected component  $H$  of an undirected graph  $G$  is a *maximal* connected subgraph
  - Maximal:  $G$  contains no graph that is both connected and properly contains  $H$ .

# Graph examples

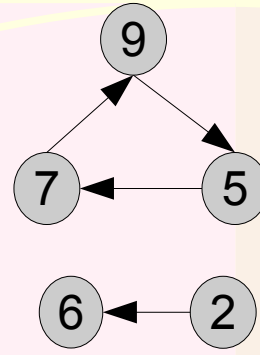


$V = \{2, 5, 6, 7, 9\}$   
 $E = \{(2, 9); (9, 7); (7, 6); (6, 5); (5, 9)\}$



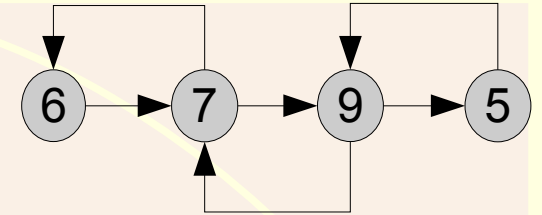
$V = \{2, 5, 6, 7, 9\}$   
 $E = \{(2, 9); (9, 7); (7, 6); (5, 9)\}$

Tree



$V = \{2, 5, 6, 7, 9\}$   
 $E = \{\langle 7, 9 \rangle; \langle 9, 5 \rangle; \langle 5, 7 \rangle, \langle 2, 6 \rangle\}$

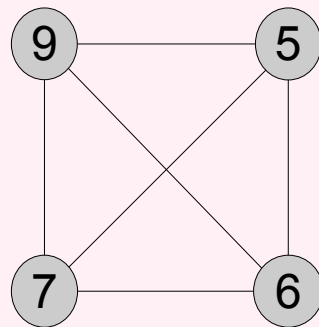
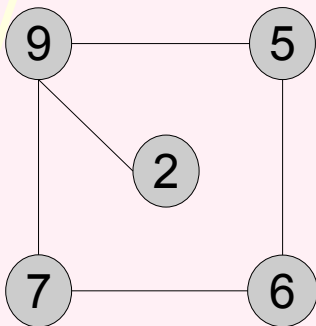
not-connected



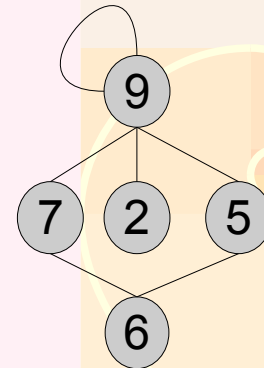
$V = \{5, 6, 7, 9\}$   
 $E = \{\langle 6, 7 \rangle; \langle 7, 9 \rangle; \langle 9, 5 \rangle; \langle 5, 9 \rangle; \langle 9, 7 \rangle; \langle 7, 6 \rangle\}$

linked list?

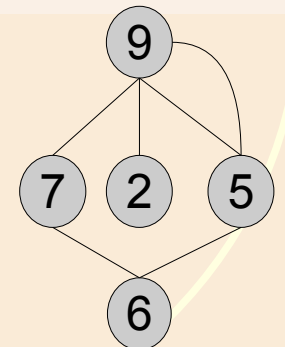
||



complete



graph with self edges



multigraph

# Characteristics & Vocabulary

- The degree of a vertex is the number of edges incident to that vertex.
- If  $G$  is a directed graph,
  - In-degree of a vertex  $v$  is the number of edges for which  $v$  is the head;
  - Out-degree is the number of edges for which  $v$  is the tail;
- When not otherwise mentioned, a graph is *undirected*
  - It may still be cyclic and not complete!

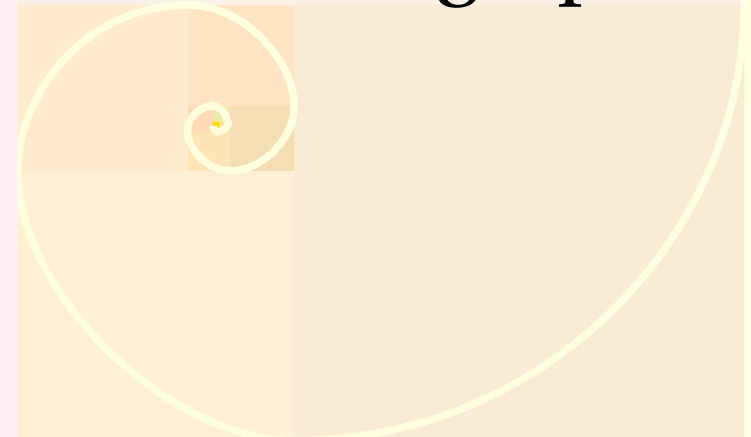
# Abstract Data Type (Interface)

```
----- C File: graph.h -----  
typedef struct graph* graph;  
typedef struct vertex* vertex;  
extern vertex vertex_new(char v);  
extern graph graph_new();  
extern void graph_delete(graph g);  
extern int graph_isEmpty(graph g);  
extern void graph_insertVertex(graph g, vertex v);  
extern void graph_deleteVertex(graph g, vertex v);  
extern void graph_insertEdge(graph g, vertex u,  
                             vertex v);  
extern void graph_deleteEdge(graph g, vertex u,  
                             vertex v);  
extern list graph_adjacent(graph g, vertex v);
```

# Representations: Adjacency Matrix

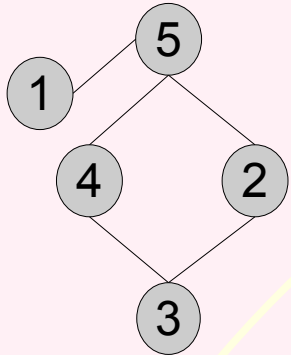
- $G(V,E)$  with  $n$  vertices,  $n \geq 1$
- Matrix  $M$  of dimension  $n \times n$
- $M[i][j] =$ 
  - **1** iff the edge  $(i,j)$  ( $\langle i,j \rangle$  if  $G$  is a directed graph) is in  $E(G)$ ,
  - **0** otherwise
- The adjacency matrix of an undirected graph is symmetric
  - $\text{degree}(i) =$

$$\sum_{j=0}^{n-1} A[i][j]$$



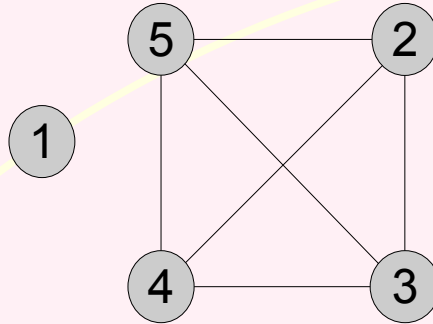
# Representations: Adjacency Matrix

## Examples (M)



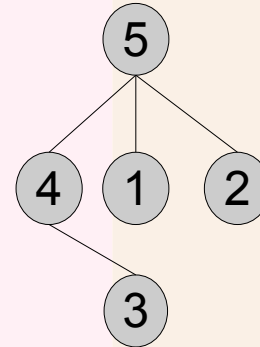
$V = \{1, 2, 3, 4, 5\}$   
 $E = \{(1, 5); (5, 4);$   
 $(4, 3); (3, 2);$   
 $(2, 5)\}$

0	0	0	0	1
0	0	1	0	1
0	1	0	1	0
0	0	1	0	1
1	1	0	1	0



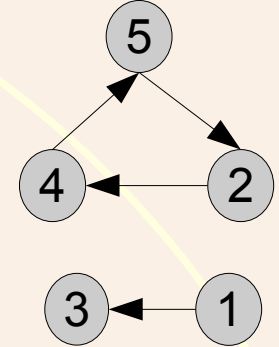
$V = \{1, 2, 3, 4, 5\}$   
 $E = \{(5, 4); (5, 2);$   
 $(4, 3); (4, 2);$   
 $(5, 3); (3, 2)\}$

0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0



$V = \{1, 2, 3, 4, 5\}$   
 $E = \{(1, 5); (5, 4);$   
 $(4, 3); (2, 5)\}$

0	0	0	0	1
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	0	1
1	1	0	1	0



$V = \{1, 2, 3, 4, 5\}$   
 $E = \{<1, 3>; <2, 4>;$   
 $<4, 5>, <5, 2>\}$

0	0	1	0	0
0	0	0	1	0
0	0	0	0	0
0	0	0	0	1
0	1	0	0	0

# Adjacency Matrix Representations Pros & Cons

- Pros (time)
    - Very efficient for basic operations: matrix backed by arrays are very efficient data structures for get()/set() operations
  - Cons (space)
    - Requires  $n^2$  entries of which:
      - $\frac{n^2-n}{2}$  in the case of undirected graph
      - $n^2-n$  in the case of directed graph
- are of no use (“-n” because the diagonal is always 0)

# Adjacency Matrix Representations Pros & Cons

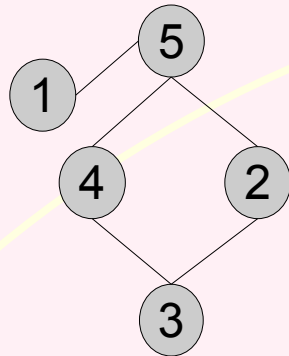
- Cons
  - For sparse matrix, representing graphs with few edges, most informations in the matrix is of no use (most 0)
  - Waste of space
    - $n^2$  space allocated
  - Waste of time
    - $O(n^2)$  algorithm (you have examine all entries)

# Representations: Adjacency Lists

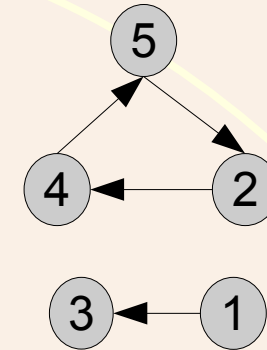
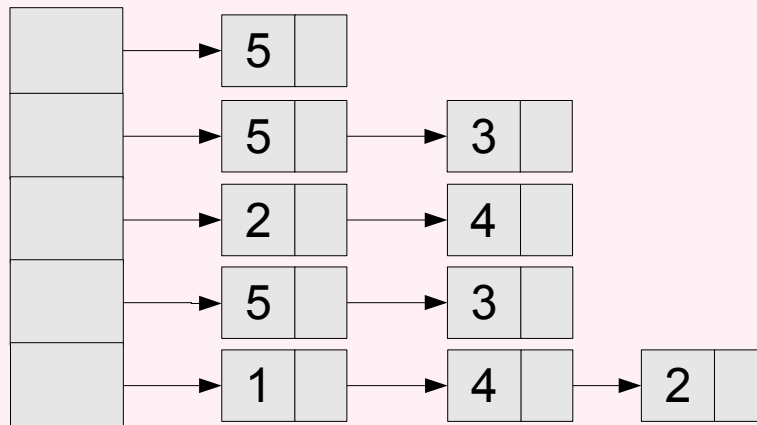
- For each vertex of the graph, store the vertices that are adjacent from it in a dedicated list, its adjacency list.
- Store each adjacency list in a global 'vertices' list
  - It may be an array if the number of vertices is fixed and known in advance
  - It may be a list backed by an array

# Representations: Adjacency Lists

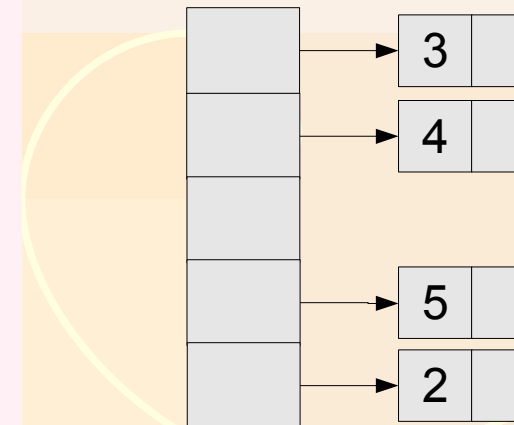
## Examples (L)



$V = \{1, 2, 3, 4, 5\}$   
 $E = \{(1, 5); (5, 4);$   
 $(4, 3); (3, 2);$   
 $(2, 5)\}$



$V = \{1, 2, 3, 4, 5\}$   
 $E = \{<1, 3>; <2, 4>;$   
 $<4, 5>, <5, 2>\}$



# Representations: Adjacency Lists

- For an undirected graph  $G$  ( $n$  vertices,  $e$  edges)
  - $n$  lists are needed,  $2 \cdot e$  list nodes
- For a directed graph  $G$  ( $n$  vertices,  $e$  edges)
  - $n$  lists are needed,  $e$  list nodes
- Size of the adjacency list of a vertex  $v$ 
  - $\text{degree}(v)$  in an undirected graph
  - $\text{out-degree}(v)$  of a digraph
- determining the number of edges of a graph is done in  $O(n+e)$  steps.

# Graph Traversals

## Depth-First Search

- Visit a vertex  $v$
- Select an unvisited vertex  $w$  adjacent to  $v$
- Initiate a DFS starting at  $w$
- When a vertex  $u$  is reached such that all its adjacent vertices have been visited, back up to the last vertex visited that has an unvisited vertex  $w$  adjacent to it
  - initiate a DFS starting at  $w$
- End when no unvisited vertex can be reached from any of the visited vertices.

# DFS General Implementation (recursive version)

```
void dfs(graph g) {
    int n = graph_vertices_nb(g); // Write this function
    int * visited = malloc(n*sizeof(int)); // booleans
    for (int i = 0; i < n; i++) visited[i] = 0;
    dfs_inner(g, visited, graph_start(g));
    free(visited);
}

void dfs_inner(graph g, int * visited, vertex v) {
    process(v); // Use the vertex (e.g. print it)
    visited[graph_index(g, v)] = 1;
    list adjacents = graph_adjacents(g, v);
    while(!list_isEmpty(adjacents)) {
        vertex w = list_deleteFirst(adjacents);
        if (!visited[graph_index(g, w)]) {
            dfs_inner(g, visited, w);
        }
    }
}
```

# DFS examples

- Examples (M), graph 1: [1,5,2,3,4];[5,1,2,3,4]
- Examples (M), graph 2: [1] ; [5,2,3,4]
  - How to process not-connected components?
- Examples (M), graph 3: [1,5,2,4,3] ; [5,1,2,4,3]
- Examples (M), graph 4: [1,3] ; [5,2,4]
- Examples (L), graph 1: [1,5,4,3,2] ; [5,1,4,3,2]
  - Different representations of the same graph may lead to different traversal order
- Examples (L), graph 2: [1,3] ; [5,2,4]

# DFS Space Complexity

- Allocation of the array of boolean 'visited'
  - size of vertices,  $n$
  - On each recursive call, the array of size ' $n$ ' is passed in parameter. Number of recursive calls is bounded by the number of vertices.
  - In this case, the space complexity is in  $O(n^2)$
- Reduce the space complexity by the use of a global array
  - Warning: thread-safety!!
  - In this case, space complexity is in  $O(n)$

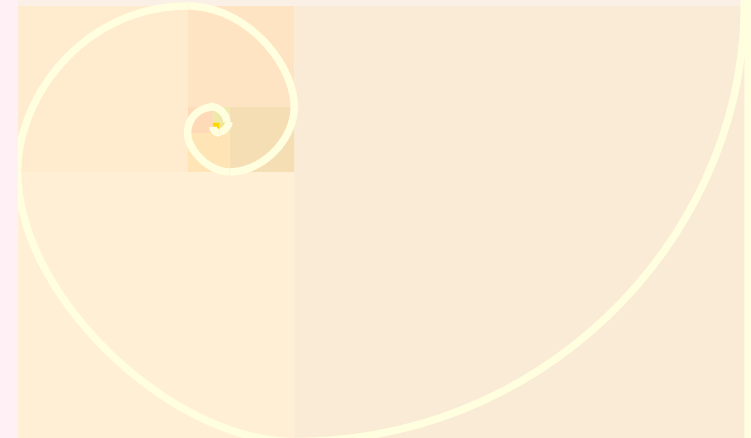
# DFS Time Complexity

- **Adjacency List Internal Graph Representation**
  - Determining the vertices adjacent to a given vertex consist in the traversal of a linked list
  - DFS examines each node in the adjacency list at most once
  - There are  $2 \cdot e$  list nodes
  - Time is in  $O(e)$
- **Adjacency Matrix Internal Graph Representation**
  - Determining the vertices adjacent to a given vertex is done in  $O(n)$
  - DFS examines  $n$  vertices at most
  - Time is in  $O(n^2)$

# Graph Traversals

## Breadth-First Search

- Visit a vertex  $v$
- Visit all unvisited vertices adjacent to  $v$
- Visit all unvisited vertices adjacent to any already visited vertices adjacent to  $v$
- etc...
- Whereas DFS needs a stack (implicit when recursive), BFS needs a queue



# BFS General Implementation

```
void bfs(graph g) {
    int n = graph_vertices_nb(g);
    int * visited = malloc(n*sizeof(int)); // booleans
    for (int i = 0; i < n; i++) visited[i] = 0;
    vertex start = graph_start(g);
    process(start); visited[graph_index(g, start)] = 1;
    queue q = newQueue(); queue_add(q, start);
    while(!isEmpty(q)) {
        vertex v = queue_remove(q);
        list adjacents = graph_adjacents(g, v);
        while(!list_isEmpty(adjacents)) {
            vertex w = list_deleteFirst(adjacents);
            if (!visited[graph_index(g, w)]) {
                process(w); visited[graph_index(g, w)] = 1;
                queue_add(q, w);
            }
        }
    }
    free(visited); }
```

# BFS examples

- Examples (M, graph 1: [1,5,2,4,3];[5,1,2,4,3])
- Examples (M, graph 2: [1] ; [5,2,3,4])
  - How to process not-connected components?
- Examples (M, graph 3: [1,5,2,4,3] ; [5,1,2,4,3])
- Examples (M), graph 4: [1,3] ; [5,2,4]
- Examples (L), graph 1: [1,5,4,2,3] ; [5,1,4,2,3]
  - Different representations of the same graph may lead to different traversal order
- Examples (L), graph 2: [1,3] ; [5,2,4]

# BFS Space Complexity

- Allocation of the array of boolean 'visited'
  - size of vertices,  $n$
  - Space complexity is in  $O(n)$



# BFS Time Complexity

- **Adjacency List Internal Graph Representation**
  - For each vertex  $v_i$ ,  $\deg(v_i)$  steps are needed for the inner while loop.
  - Total time is  $\deg(v_0) + \dots + \deg(v_n) = O(e)$
- **Adjacency Matrix Internal Graph Representation**
  - The inner loop requires  $O(n)$  steps
  - Each vertex enters the queue exactly once
  - Time is in  $O(n^2)$



# Graph Algorithms

# Connected Components

- Use the DFS (or BFS) algorithm to find all the connected components of a graph

// Modifications from slide #214

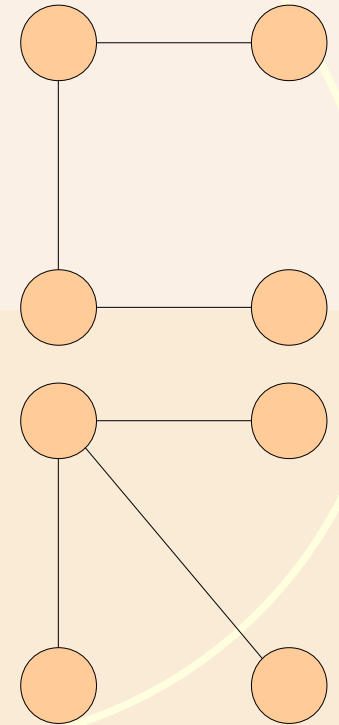
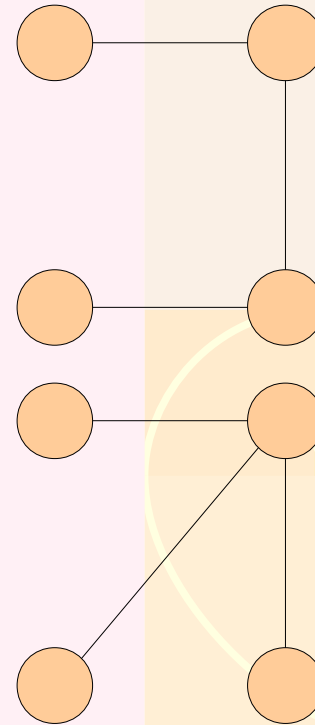
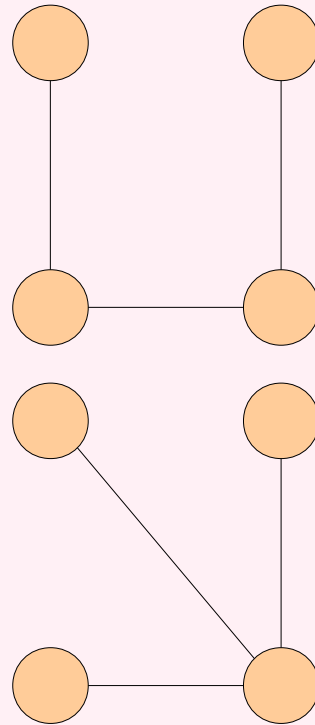
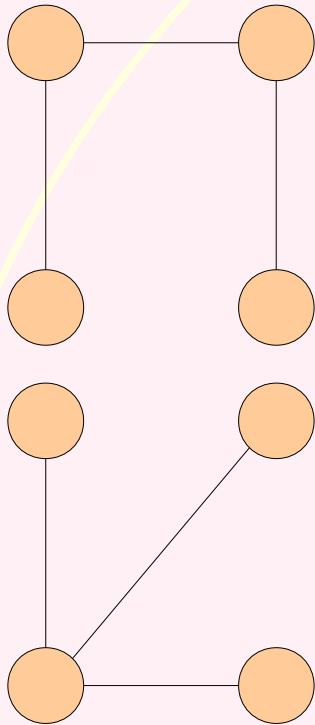
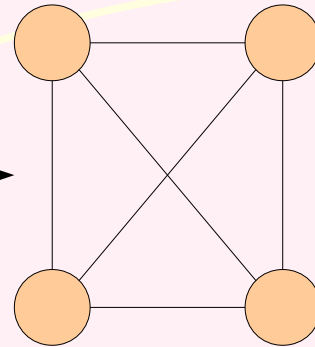
```
void dfs(graph g) {
    int n = graph_vertices_nb(g); // Write this function
    int * visited = malloc(n*sizeof(int)); // booleans
    for (int i = 0; i < n; i++) visited[i] = 0;
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            dfs_inner(g, visited, graph_start(g));
            processNewVisited(g, visited);
        }
    }
    free(visited);
}
```

# Spanning Trees

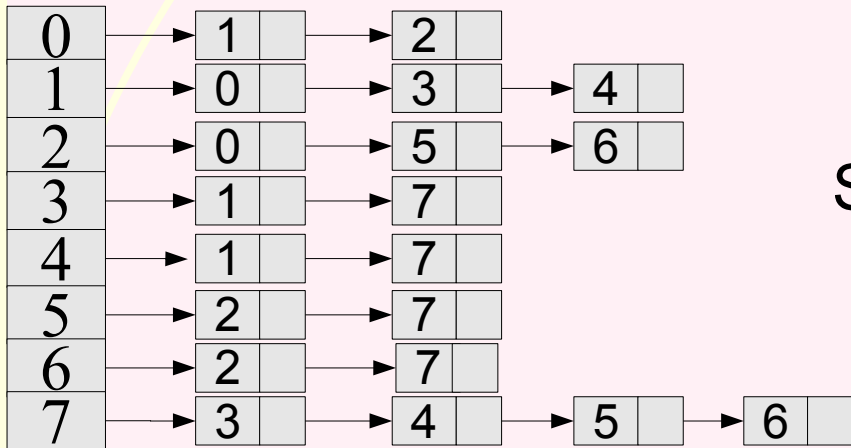
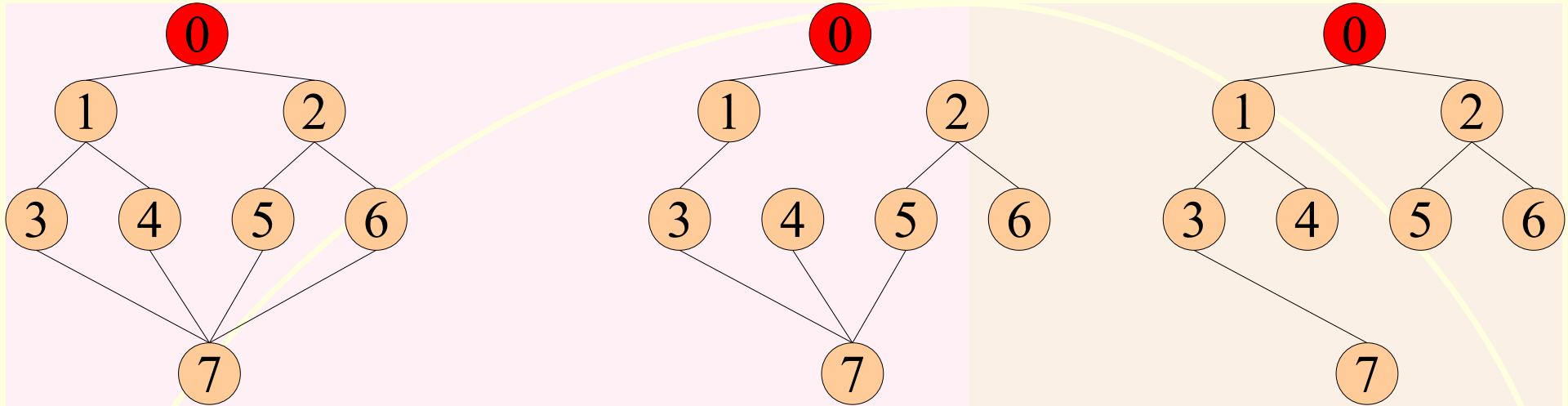
- If  $G$  is connected, BFS or DFS visits all nodes
- $G$  is partitioned in two sets  $T$  (Tree Edges) and  $N$  (Non-Tree edges)
  - Add ' $T=T \cup \{u,v\}$ ' in the `if()` clause of DFS or BFS
  - $T$  form a tree that includes all the vertices of  $G$
- Any tree consisting solely of edges in  $G$  and including all vertices in  $G$  is called a *spanning tree*.

# Spanning Tree Examples

Original Complete Tree



# Spanning Tree Examples

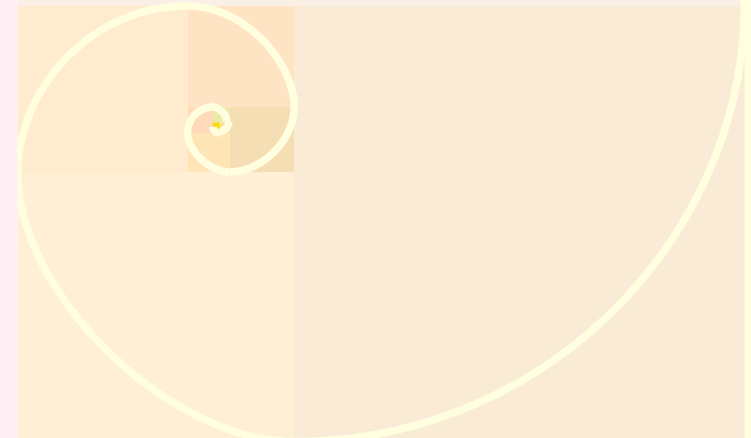


**Depth-First**  
Spanning Tree

**Breadth-First**  
Spanning Tree

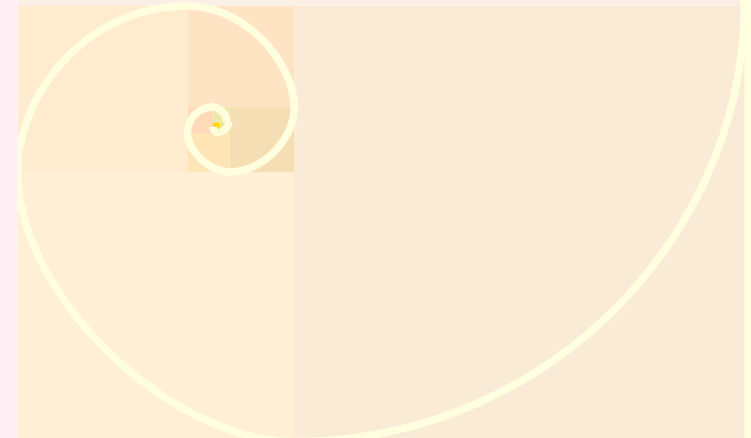
# Minimum-Cost Spanning Trees

- Edges has a cost (weight)
  - Use a field in Adjacency Lists, A number in matrix
- The cost of a spanning tree is the sum of the cost of the edges in the spanning tree.
- A *minimum-cost spanning tree* is a spanning tree of least cost.
- Three different algorithms
  - Prim, Kruskal and Sollin

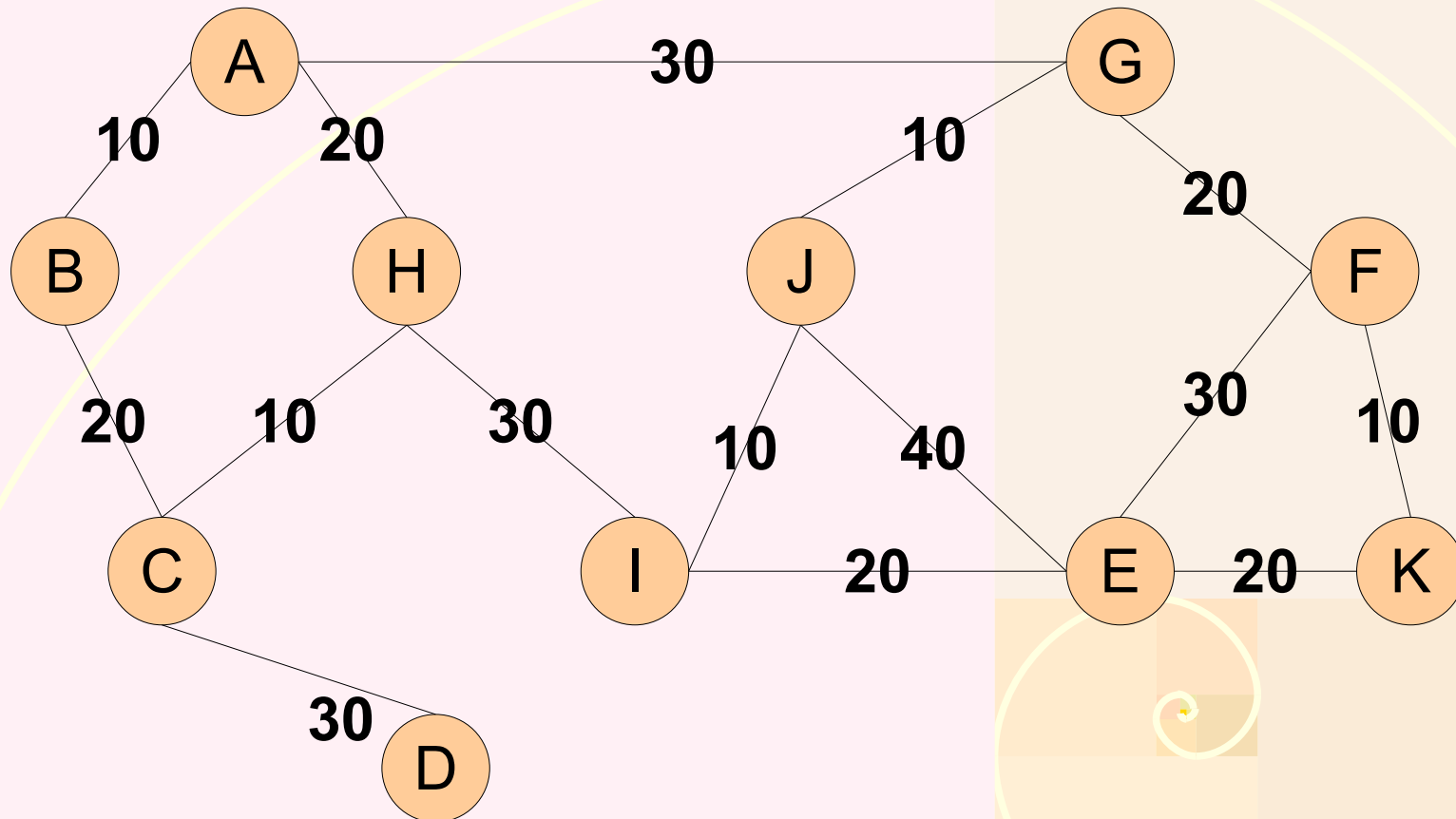


# Prim's Algorithm

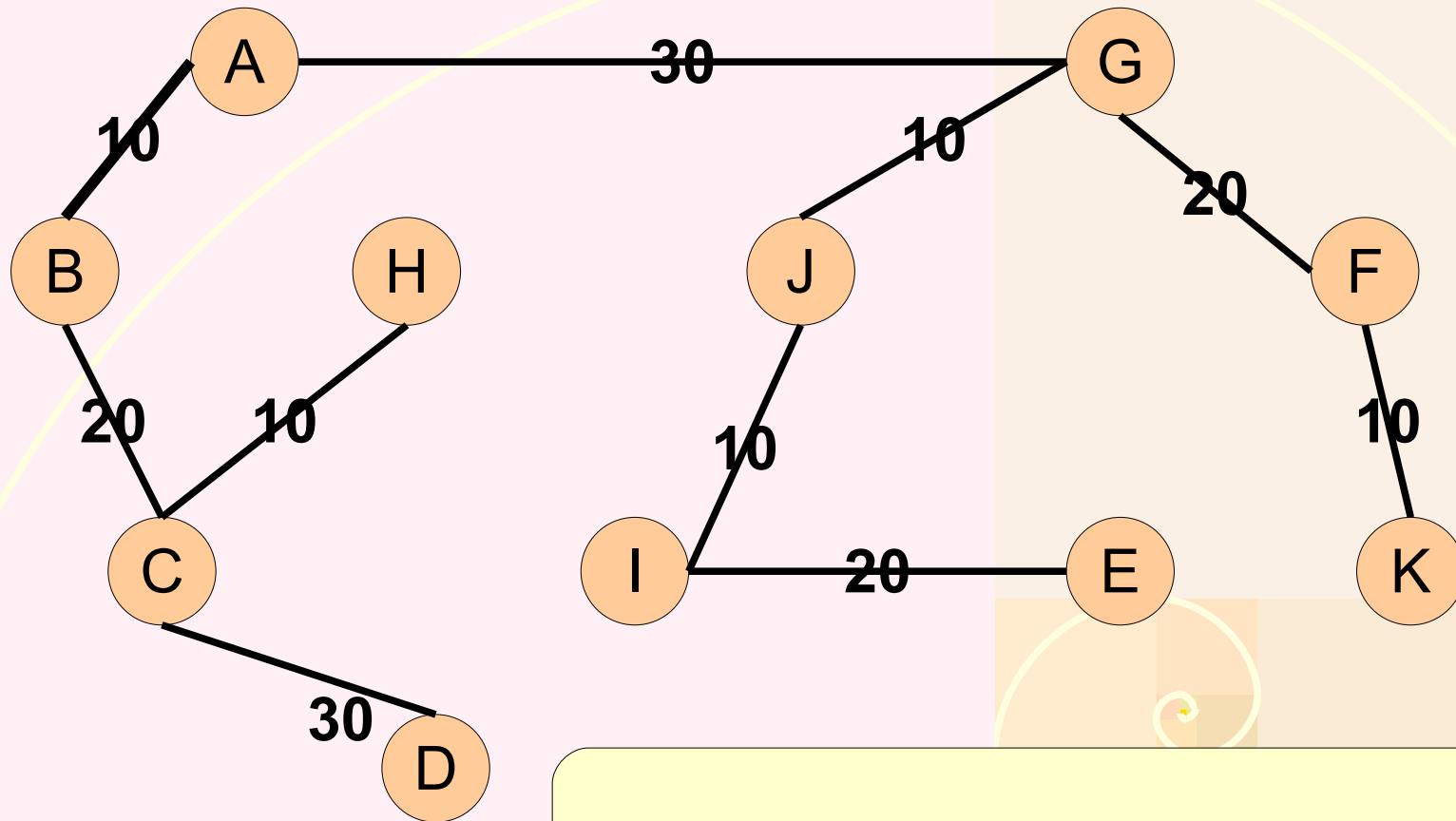
- Start from a tree  $T$  containing any vertex of  $G$
- Add the least-cost edge  $(u,v)$  to  $T$  such that:
  - $T \cup \{(u,v)\}$  is also a tree
  - Use a min-heap to find the least-cost edge
- Repeat until  $T$  contains  $n-1$  edges



# Prim's Algorithm Example



# Prim's Algorithm Example

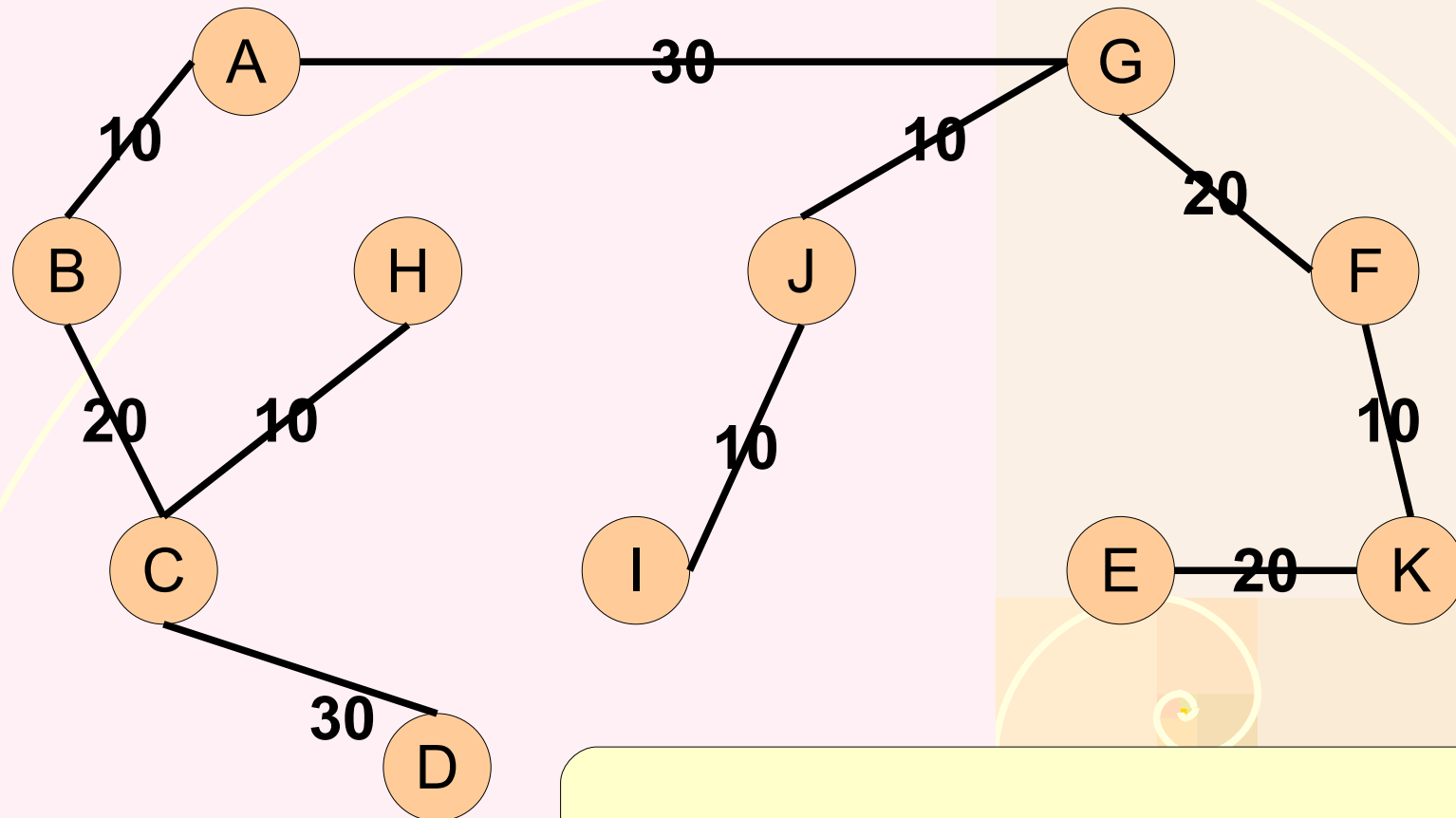


ABCHDGJIEFK=170

# Kruskal's Algorithm

- Start with two sets of edges
  - an empty set  $T$  (the result)
  - the set  $E$  containing all the edges of graph  $G$
- **remove** an edge in  $E$  with minimum cost
  - Use a min-heap for this purpose
- add this edge to  $T$  **if it does not form a cycle**
- End when  $T$  has  $n-1$  edges
  - or when no more edges is available ( $E$  is empty),  $G$  is not connected, there is no spanning tree

# Kruskal's Algorithm Example



AB,CH,IJ,JG,FK,BC,GF,EK,AG=170

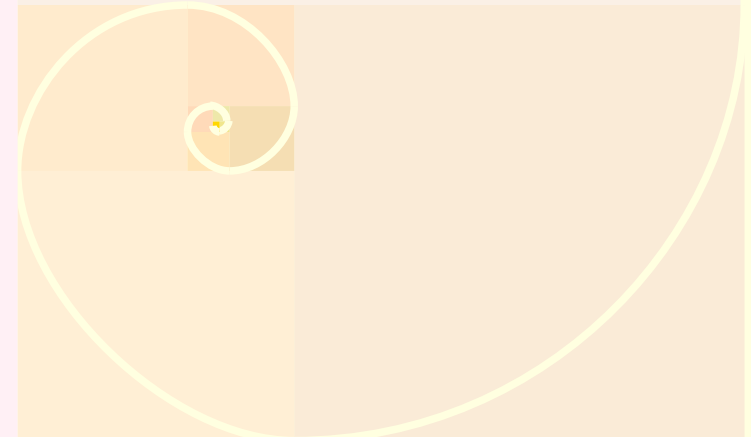
**End**

# Quiz

## Stacks & Lists

- Implements a stack backed by this list

```
list newList(); // Returns a new (empty) list
node start(list l); // Returns the 'start' element
node getNext(node n); // returns the next node of 'n'
char getValue(node n); // returns the value of the 'n'
int isEmpty(list l); // 1 ==>'l' is empty, '0' otherwise
void addAfter(list l,
              char v, // Adds 'v' in 'l' after 'n'
              node n);
// Deletes the node in 'l' which is just after 'n'
void deleteAfter(list l, node n);
```



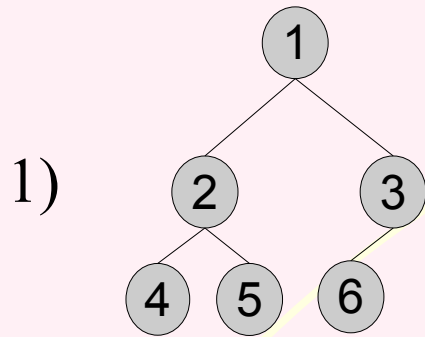
# Quiz

## Trees & Heaps

1. Gives the *complete binary tree* made of 6 nodes *labeled* from 1 to 6 in the ascending order.
2. Is it a full tree?
3. Gives a path from element 5 to 6
4. Gives the degree of node *labeled* 3
5. Gives the depth of this tree
6. Represents your tree in an array
7. Gives the output of the traversal of your tree in:
  - a. Post-order
  - b. Level-order
8. Gives the new *min-heap* that is the result of *deleteMin()*
9. Gives the new *min-heap* that is the result of *insert(1)*

# Quiz

## Trees & Heaps: Correction

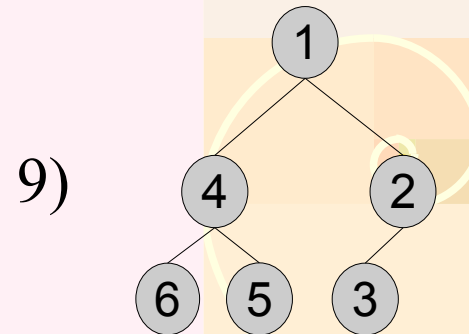
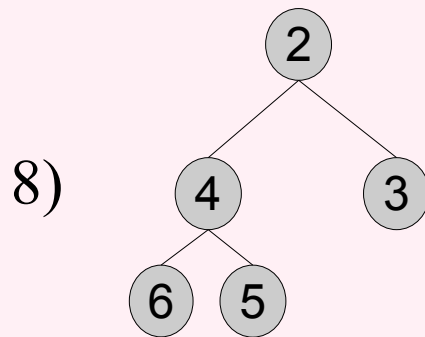


- 2) No  
 3) 5,2,1,3,6  
 4) 1  
 5) 3

6)

k	0	1	2	3	4	5	6	7
a[k]	-	1	2	3	4	5	6	-

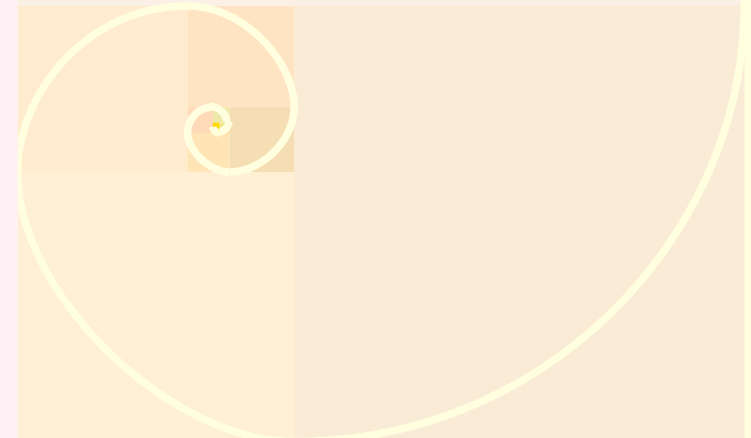
- 7)
- a) Post-order (LRV): 4,5,2,6,3,1  
 b) Level-order: 1,2,3,4,5,6



# Quiz

## Sorting

- Give the signature of a function implementing a sort algorithm on an array (3 points)
- Implement this function with the algorithm you like (5 points)
- Give the space and time complexity of your implementation (2 points)



# Quiz

## Searching

- 1) Give the signature of a function implementing a find algorithm on an array (3 points)
- 2) Implement this function with the algorithm you like (3 points)
- 3) Binary Search Tree
  - 1) Give the BST representation after the insertion of the following elements (consider alphabetical order): THISQUIZISVERYEASY (2 points)
  - 2) What is the average number of comparisons for a search in this tree? (2 points)

# Quiz

## Searching: Correction

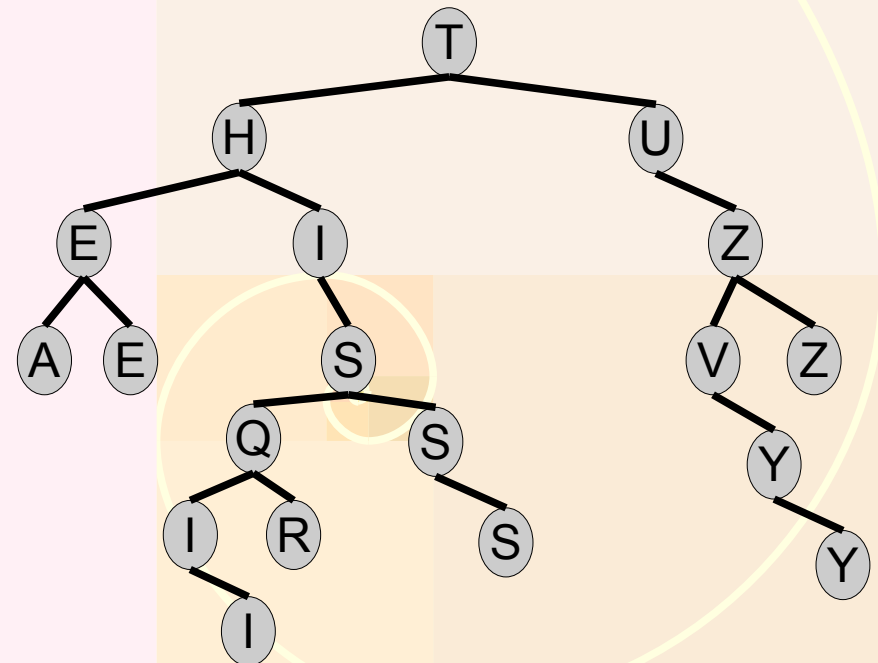
### 1) Many solutions

- // return the position (-1: not found)  
`int find(element* t, int n, key_type key);`
- // return a pointer on the element (NULL: not found)  
`element* find(element* t, int n, key_type key);`

### 2) See slides

### 3) Binary Search Tree

Average Comparisons:  $80/19 \approx 4.21$



# Quiz

## Balanced Trees

1) (3 points) Give the resulting AVL tree after the insertion of the following characters:

IMPROVEYOURAVERAGE

2) (3 points) Same question for a 2-3-4 Tree

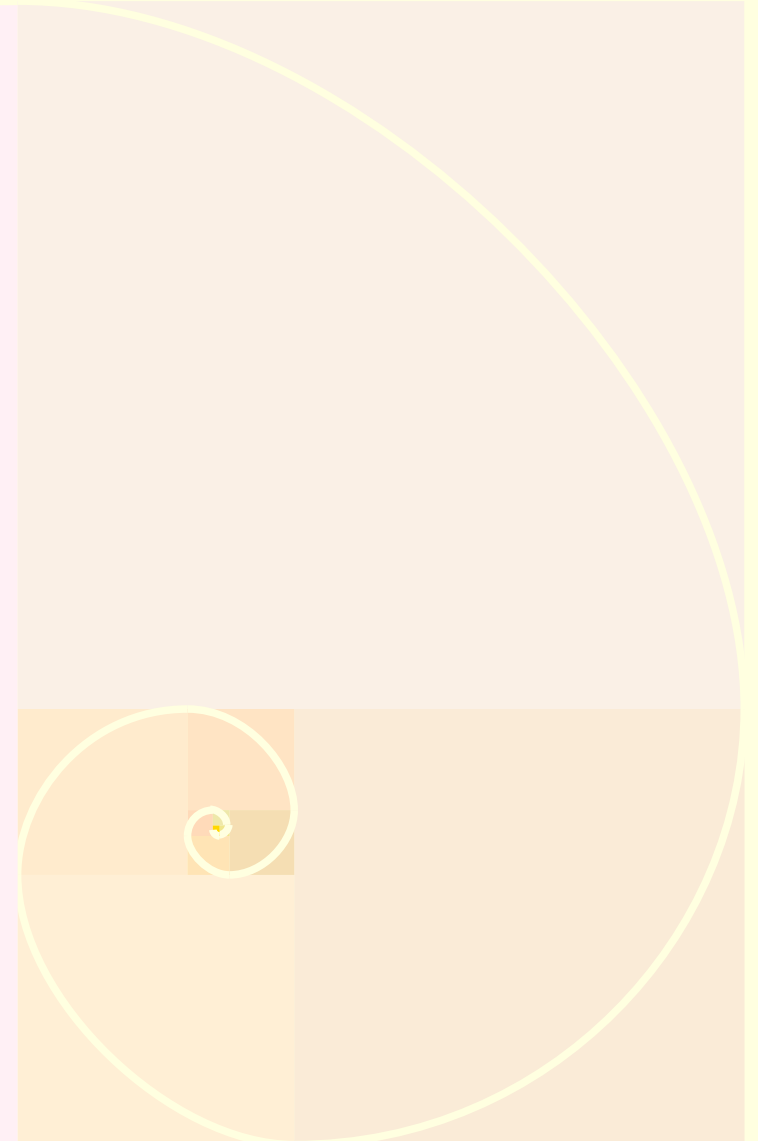
3) (3 points) Same question for a Red-Black Tree

1 extra point will be given  
for presentation!

# Quiz

## Balanced Trees: Correction

- 1) PMVEORYAGORUVAEIRE
- 2) PEGMRVAAEEIOORRVVY
- 3) PGVEMRYAEIORUVAEOR



# Main Quiz (6% of the grade)

- Write the function  $f(n)=n!$ 
  - Recursive version (1 point)
  - Iterative version (1 point)
  - Time complexity of both functions? (1 point)
- Polynoms & Lists
  - Provide a `polynom` class (or structure) that is backed by a list implementation
    - Write the **signature** of the list (member) functions you need; (1 point)
    - Write the (member) function required for creating, deleting and summing two polynoms. (2 points)

# Quiz

## Hashing

- Consider the hash function:  $h(c) = \text{rank}(c) \% 20$
- Represent the state of a 20 buckets hash table after inserting **ONEQUIZONHASHING**:
  - with linear probing (1 point)
  - with chaining (1 point)
- Gives the average number of comparisons on success in the two cases (2x1 points)

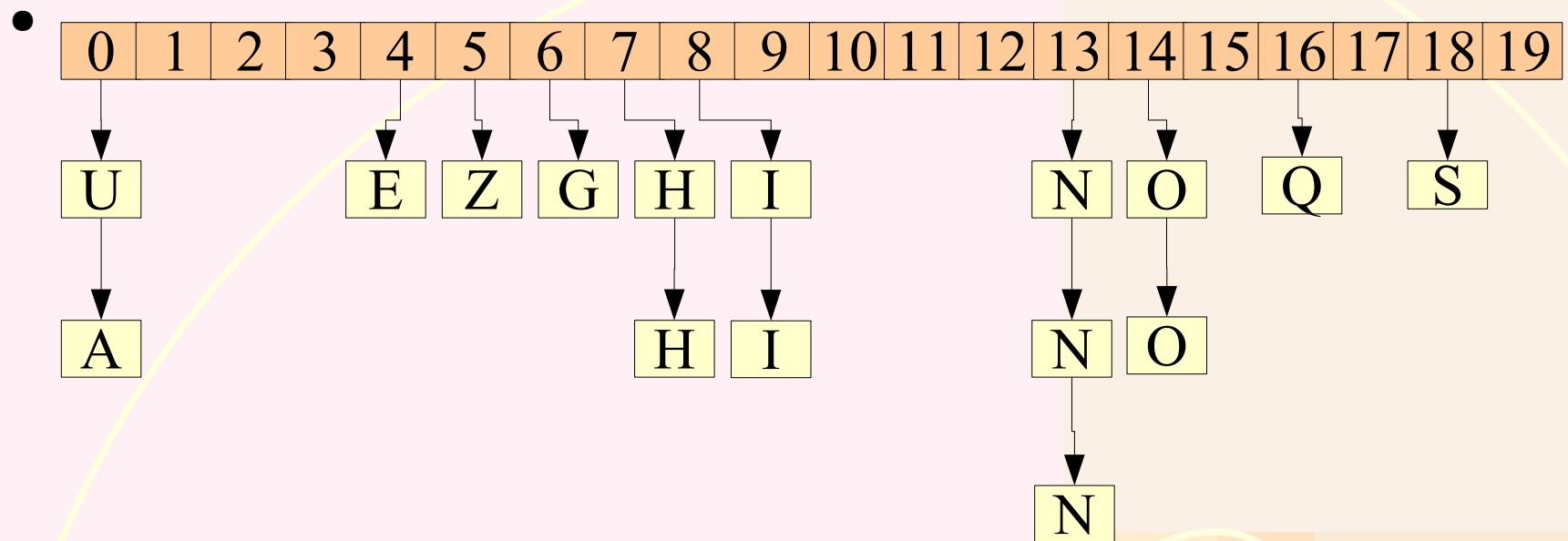
1 extra point will be given  
for presentation!

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

# Quiz

## Hashing: Correction

- UA--EZGHIHI--NOOQNSN: 31/16



Quiz

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25