Jin-Soo Kim
(*jinsoo.kim@snu.ac.kr*)

Systems Software &
Architecture Lab.

Seoul National University

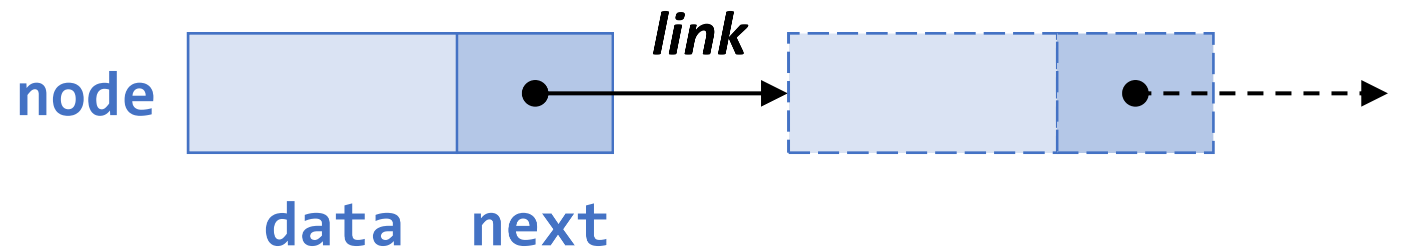
Spring 2022

Lists

Self-referential Structures

- Structures with pointer members that refer to the structure containing them

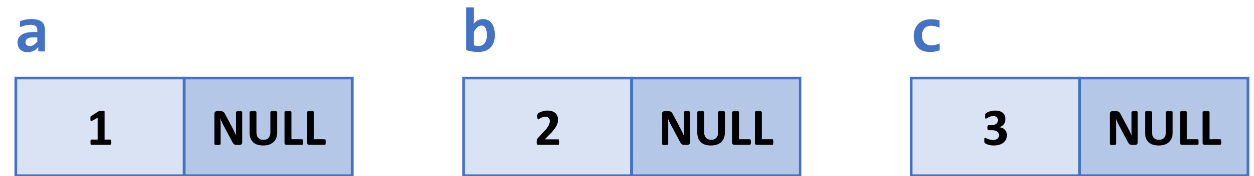
```
struct list {
    int data;
    struct list *next;
} node;
```

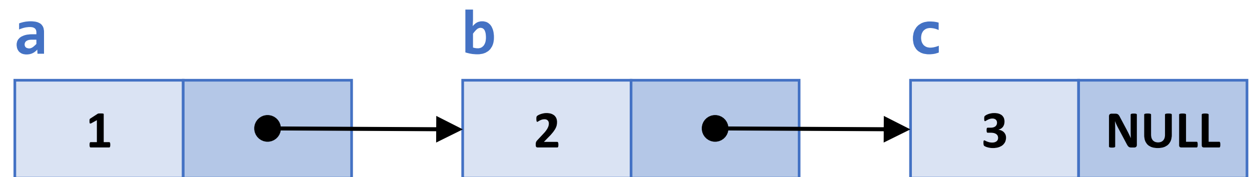


- The pointer variable `next` contains either an address of the memory space for the successor list element or NULL (defined as 0)
- Each structure is linked to a succeeding structure by way of the member `next`

Example

```c
struct list a, b, c;
a.data = 1;
b.data = 2;
c.data = 3;
a.next = b.next = c.next = NULL;


a.next = &b;
b.next = &c;


printf("%d\n%d\n",
       a.next->data,
       a.next->next->data);
```





Linear Linked Lists

- A linear linked list is like a clothes line on which the data structures hang sequentially

- Head pointer addresses the first element
- Each element points at a successor element
- The last element has a link value NULL

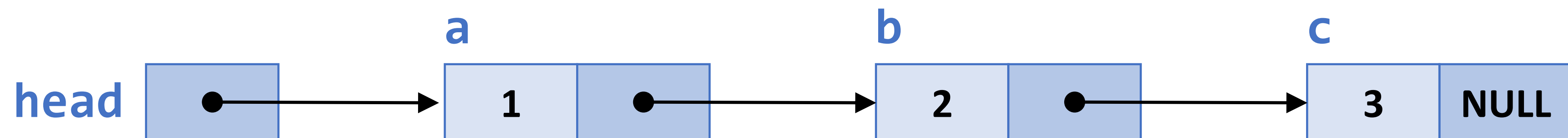
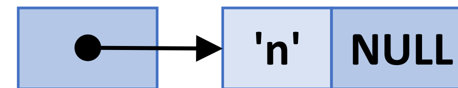


Image Source: https://pixabay.com/vectors/clothesline-clothes-dry-silhouette-155864/

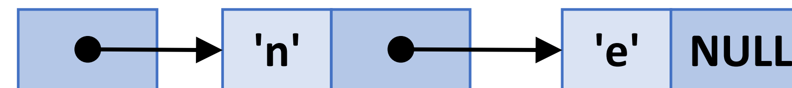
Example

```c
typedef char DATA;
struct linked_list {
    DATA d;
    struct linked_list *next;
};
typedef struct linked_list ELEMENT;
typedef ELEMENT *LINK;

LINK head;

head = malloc(sizeof(ELEMENT));
head->d = 'n';
head->next = NULL;

head->next = malloc(sizeof(ELEMENT));
head->next->d = 'e';
head->next->next = NULL;

head->next->next = malloc(sizeof(ELEMENT));
head->next->next->d = 'w';
head->next->next->next = NULL;
```

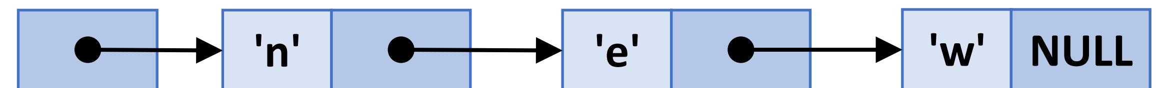
head



head



head



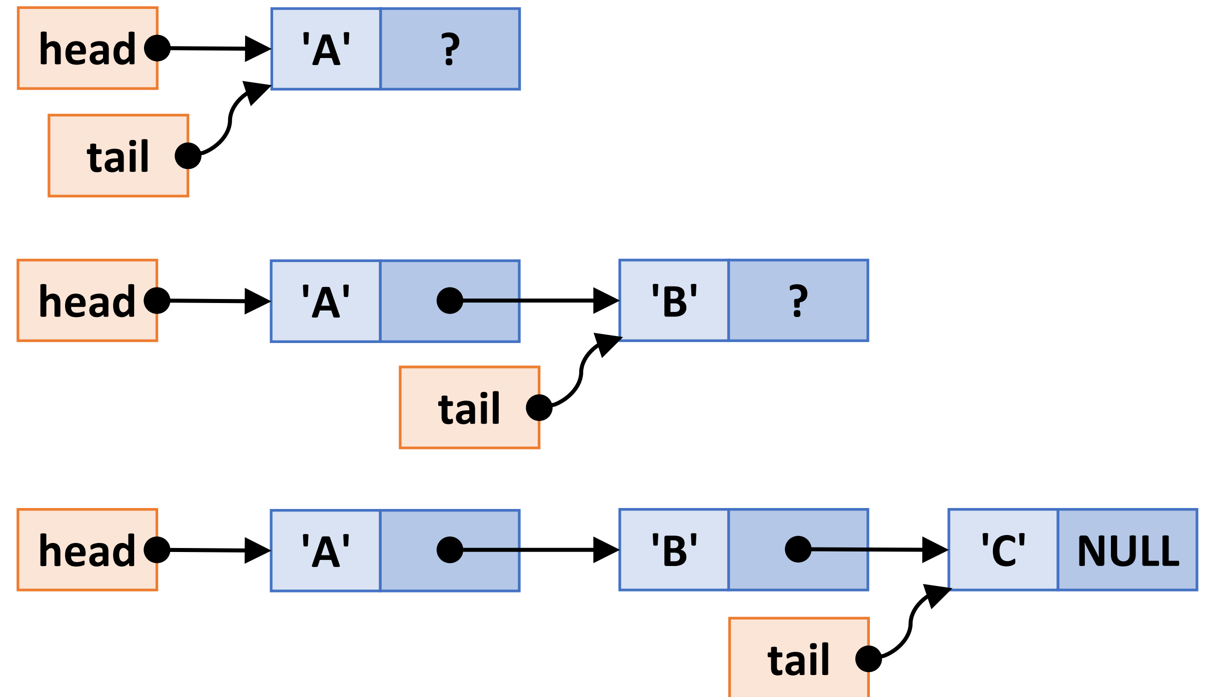
(Linear) List Operations

- Creating a list
- Counting the elements
- Looking up an element
- Concatenating two lists
- Inserting an element
- Deleting an element

Creating a List using Iteration (1)

- `LINK str_to_list(char s[]);` /* function prototype */
- `str_to_list("ABC");` /* function call */

```
LINK head, tail;

head = malloc(sizeof(ELEMENT));
head->d = s[0];
tail = head;

tail->next = malloc(sizeof(ELEMENT));
tail->d = s[1];
tail = tail->next;

tail->next = malloc(sizeof(ELEMENT));
tail->d = s[2];
tail = tail->next;

tail->next = NULL;
```



Creating a List using Iteration (2)

```
LINK str_to_list(char s[]) {
    LINK head = NULL, tail;
    int i;

    if (s[0] != '\0') {
        head = malloc(sizeof(ELEMENT));     /* first element */
        head->d = s[0];
        tail = head;
        for (i = 1; s[i] != '\0'; i++) {    /* add to tail */
            tail->next = malloc(sizeof(ELEMENT));
            tail = tail->next;
            tail->d = s[i];
        }
        tail->next = NULL;                  /* end of list */
    }
    return head;
}
```

Creating a List using Iteration (3)

```
/* Create a list from the back */

LINK str_to_list2(char s[])
{
    LINK head = NULL, tmp;
    char *p = s + strlen(s) - 1;

    while (s <= p) {
        tmp = malloc(sizeof(ELEMENT));
        tmp->d = *p--;
        tmp->next = head;
        head = tmp;
    }
    return head;
}
```

Creating a List using Recursion

```
LINK str_to_list(char s[])
{
    LINK head;

    if (s[0] == '\0')
        return NULL;

    head = malloc(sizeof(ELEMENT));
    head->d = s[0];
    head->next = str_to_list(s+1);
    return head;
}
```

Counting Elements

- Using iteration

```
/* Count a list iteratively */

int count(LINK head)
{
    int cnt = 0;

    for (; head != NULL;
           head = head->next)
        cnt++;
    return cnt;
}
```

- Using recursion

```
/* Count a list recursively */

int count(LINK head)
{
    if (head == NULL)
        return 0;
    else
        return 1 + count(head->next);

    /*
    return head?
        1 + count(head->next) : 0;
    */
}
```

Concatenating Two Lists

- Concatenate list a and b with a as head

```
/* Concatenate two lists iteratively */

void concatenate(LINK a, LINK b)
{
    assert(a != NULL);
    while (a->next)
        a = a->next;
    a->next = b;
}
```

```
/* Concatenate two lists recursively */

void concatenate(LINK a, LINK b)
{
    assert(a != NULL);
    if (a->next == NULL)
        a->next = b;
    else
        concatenate(a->next, b);
}
```

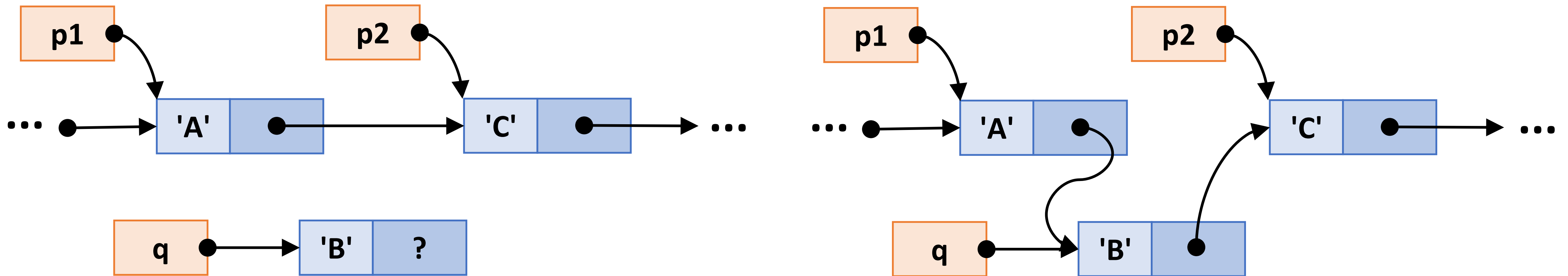
Recursive Functions for List Processing

```
void generic_recursion(LINK head)
{
    if (head == NULL)
    {
        /* do the base case */
    }
    else
    {
        /* do the general case and recur with
           generic_recursion(head->next)       */
    }
}
```

Inserting an Element

- Insert q between p1 and p2

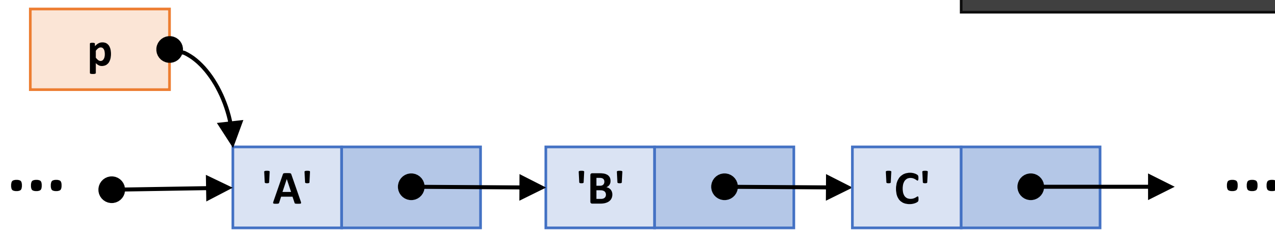
```
void insert(LINK p1, LINK p2, LINK q)
{
    assert(p1->next == p2);
    p1->next = q;
    q->next = p2;
}
```

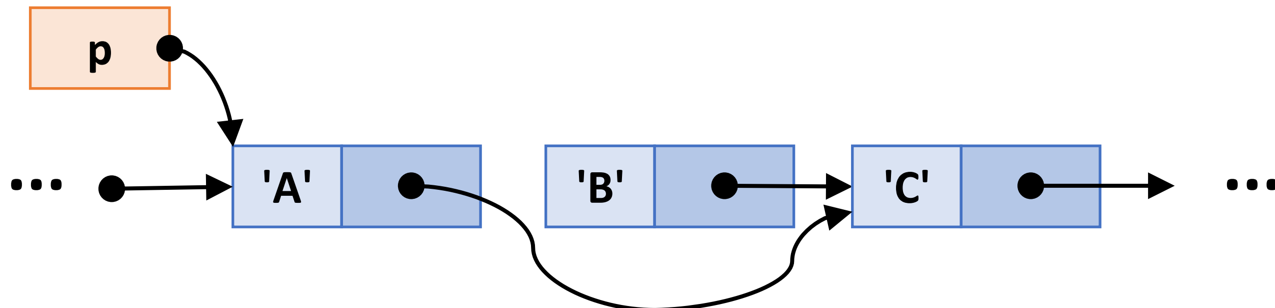


Deleting an Element

- Delete an element after p

```
void delete(LINK p)
{
    assert(p->next != NULL);
    p->next = p->next->next;

    /* element containing B becomes a garbage */
}
```





Deleting a List

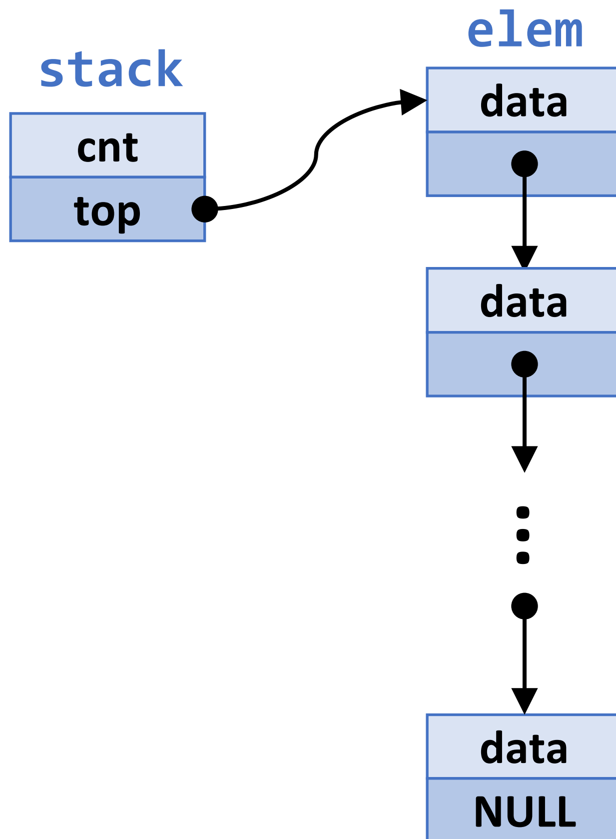
- You must release the storage using `free()` if the node was allocated by `malloc()` or `calloc()`

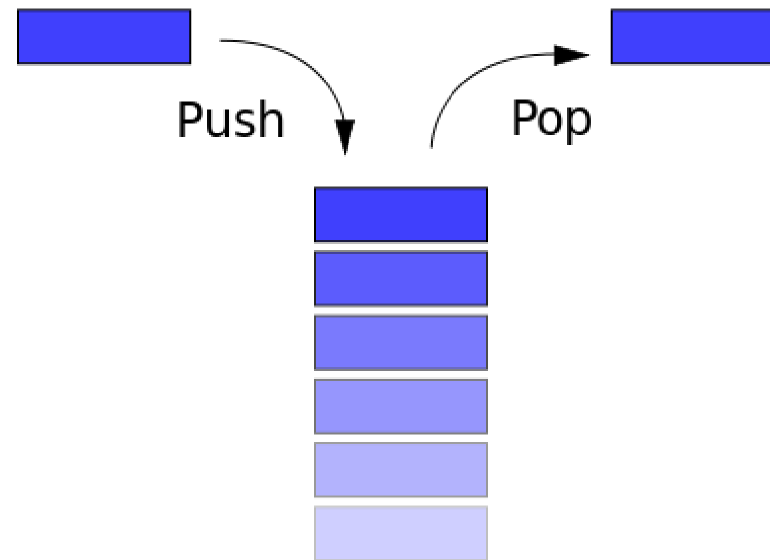
```
/* Delete a list iteratively */

void delete_list(LINK head)
{
    LINK tmp;

    while (head) {
        tmp = head;
        head = head->next;
        free(tmp);
    }
}
```

```
/* Delete a list recursively */

void delete_list(LINK head)
{
    if (head != NULL)
    {
        delete_list(head->next);
        free(head);
    }
}
```

Stack

- A data structure with the LIFO (Last-In, First-Out) principle





Stack: `stack.h`

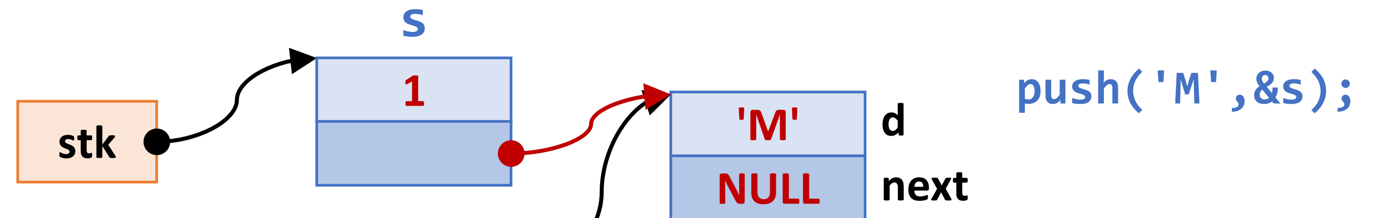
```c
#include <stdio.h>
#include <stdlib.h>

#define     EMPTY   0
#define     FULL    10000

typedef char data;
typedef enum {FALSE, TRUE} boolean;

/* an element on the stack */
struct elem {
    data d;
    struct elem *next;
};
typedef struct elem elem;
```

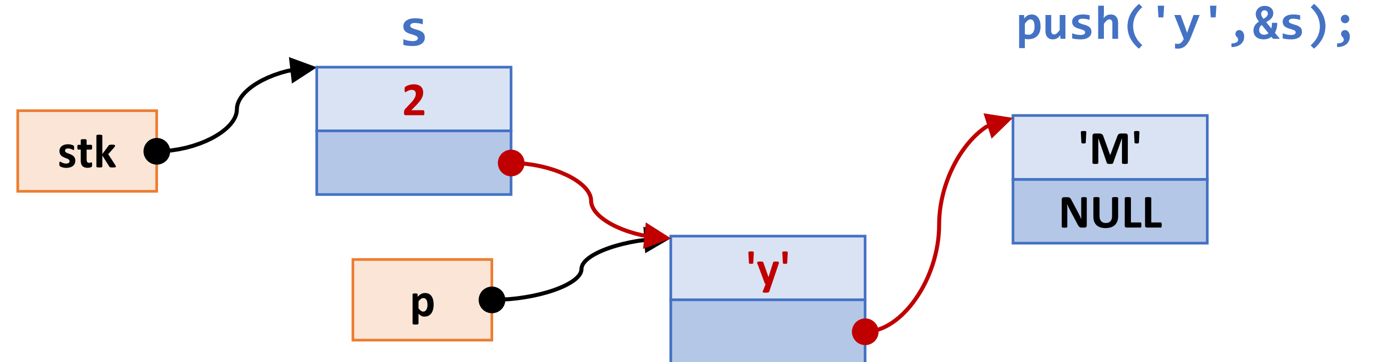
```c
/* stack head */
struct stack {
    int cnt;    /* count of the elements */
    elem *top; /* pointer to the top element */
};
typedef struct stack stack;

void initialize(stack *stk);
void push(data d, stack *stk);
data pop(stack *stk);
data top(const stack *stk);
boolean empty(const stack *stk);
boolean full(const stack *stk);
```

Stack: `initialize()` and `push()`

```
void initialize(stack *stk)
{
    stk->cnt = 0;
    stk->top = NULL;
}
```

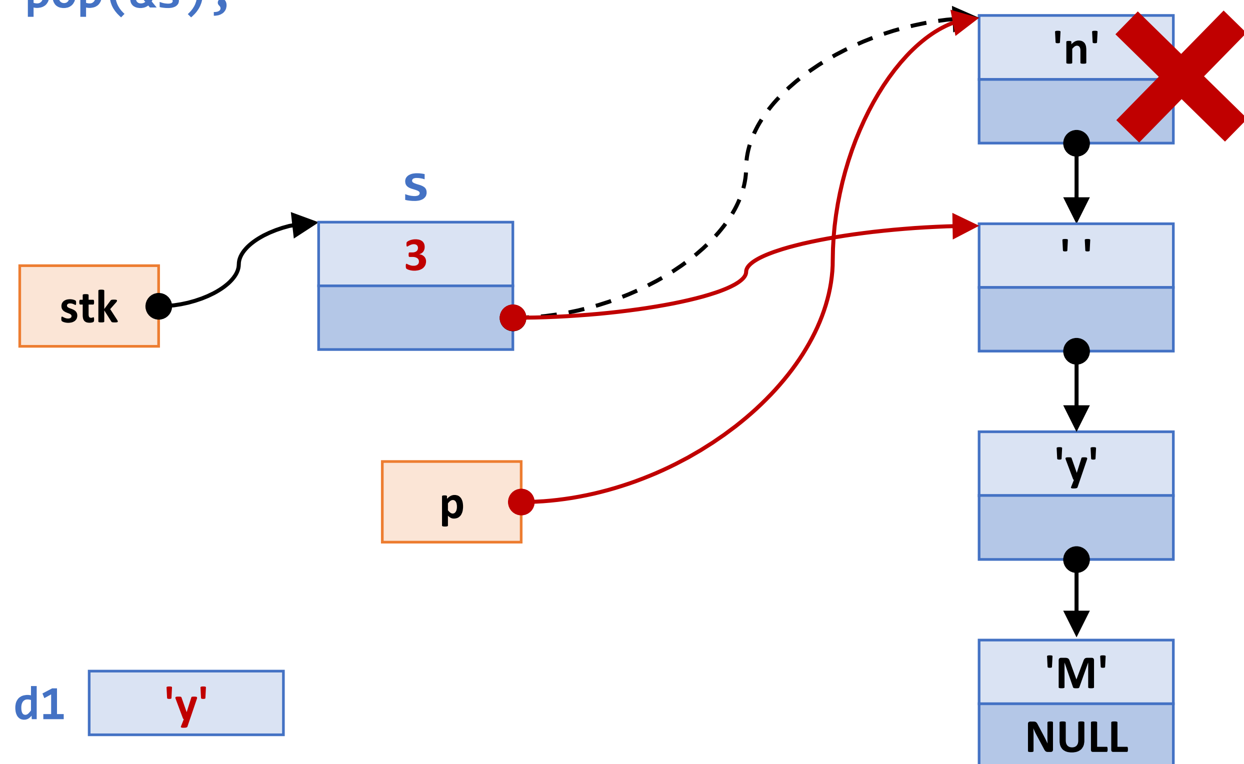
```
void push(data d, stack *stk)
{
    elem *p;

    p = malloc(sizeof(elem));
    p->d = d;
    p->next = stk->top;
    stk->top = p;
    stk->cnt++;
}
```





Stack: pop( )

```
data pop(stack *stk)
{
    data d1;
    elem *p;

    d1 = stk->top->d;
    p = stk->top;
    stk->top = stk->top->next;
    stk->cnt--;
    free(p);      /* !!! */

    return d1;
}
```



Stack: `top()`, `empty()`, and `full()`

```
data top(const stack *stk)
{
    return stk->top->d;
}


boolean empty(const stack *stk)
{
    return ((boolean) (stk->cnt == EMPTY));
}


boolean full(const stack *stk)
{
    return ((boolean) (stk->cnt == FULL));
}
```

Stack: main()

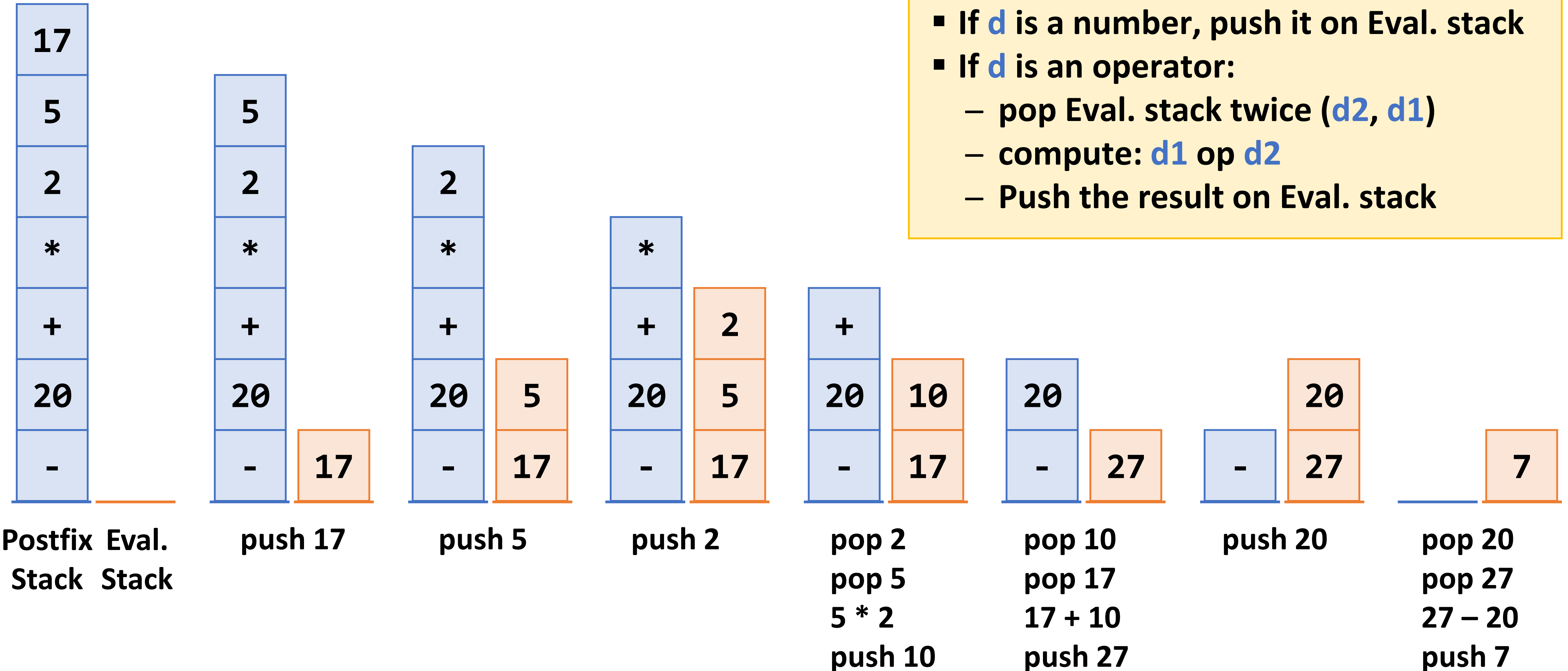
```
#include "stack.h"
int main(void) {
    char str[] = "My name is Joanna Kelly!";
    int i;
    stack s;

    initialize(&s);                              /* initialize the stack */
    printf("In the string: %s\n", str);
    for (i = 0; str[i] != '\0'; i++)
        if (!full(&s))
            push(str[i], &s);                    /* push a char on the stack */

    printf("From the stack: ");
    while (!empty(&s))
        putchar(pop(&s));                        /* pop a char from the stack */

    putchar('\n');
    return 0;
}
```

Mathematical Notations

- Infix: operators are placed between operands (ordinary notation)
  - `3 + 5`
  - `17 + 5 * 2 - 20`
- Polish notation or prefix notation: operators precede their operands
  - `+ 3 5`
  - `- + 17 * 5 2 20`
- Reverse polish notation or postfix notation: operators follow their operands
  - `3 5 +`
  - `17 5 2 * + 20 -`

Stack Evaluation of Postfix Notation

- `17 5 2 * + 20 -`



- Pop the Postfix stack: d
- If d is a number, push it on Eval. stack
- If d is an operator:
  - pop Eval. stack twice (d2, d1)
  - compute: d1 op d2
  - Push the result on Eval. stack

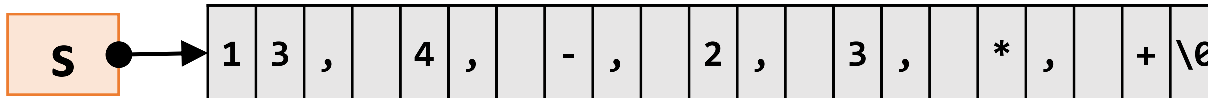
Postfix: postfix.h

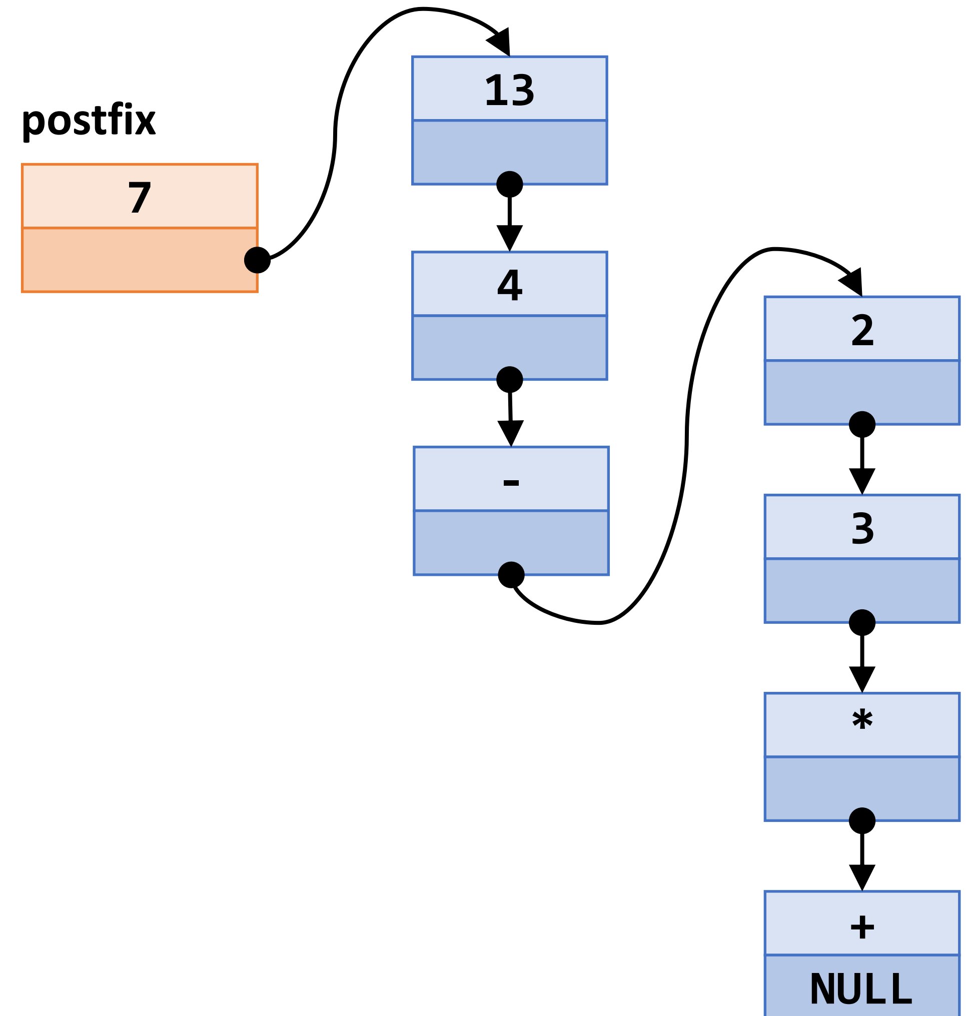
```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

#define     EMPTY   0
#define     FULL    10000

typedef struct {
    enum {OP, VALUE} kind;
    union {
        char op;
        int  val;
    } u;
} data;

struct elem {
    data    d;
    struct elem   *next;
};
typedef struct elem elem;
```

```
typedef enum {FALSE, TRUE} boolean;

struct stack {
    int     cnt;
    elem    *top;
};
typedef struct stack stack;

void initialize(stack *stk);
void push(data d, stack *stk);
data pop(stack *stk);
data top(const stack *stk);
boolean empty(const stack *stk);
boolean full(const stack *stk);

int eval(stack *postfix);
void fill(stack *stk, const char *str);
void parse(stack *stk, const char *str);
```

Postfix: main()

```
#include "postfix.h"

void main(void)
{
    char *s = "13, 4, -, 2, 3, *, +";
    stack postfix;

    fill(&postfix, s);
    printf("result: %d\n", eval(&postfix));
}
```





Postfix: parse()

```
void parse(stack *stk, const char *str) {
    const char *p = str;
    boolean b1, b2;   data d;    char c1, c2;

    initialize(stk);
    while (*p != '\0') {
        while (isspace(*p) || *p == ',') p++;
        b1 = ((c1=*p)=='+' || c1=='-' || c1=='*');
        b2 = ((c2=*(p+1))==',' || c2=='\0');
        if (b1 && b2) {
            d.kind = OP;
            d.u.op = c1;
        }
        else {
            d.kind = VALUE;
            sscanf(p, "%d", &d.u.val);
        }
        if (!full(stk)) push(d, stk);
        while (*p != ',' && *p != '\0') p++;
    }
}
```

1	3	,		4	,		-	,		2	,		3	,		*	,		+	\0

Postfix: fill()

```
void fill(stack *stk, const char *str)
{
    data d;
    stack tmp;

    initialize(stk);
    parse(&tmp, str);

    while (!empty(&tmp))
    {
        d = pop(&tmp);
        if (!full(stk))
            push(d, stk);
    }
}
```

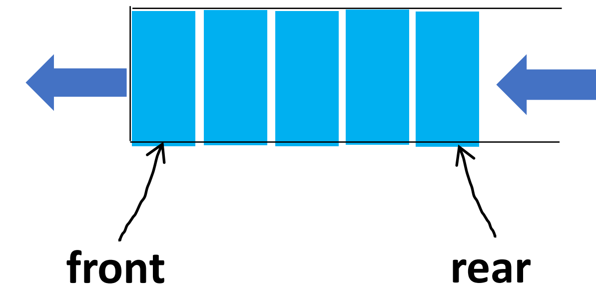
1	3	,		4	,		-	,		2	,		3	,		*	,		+	\0

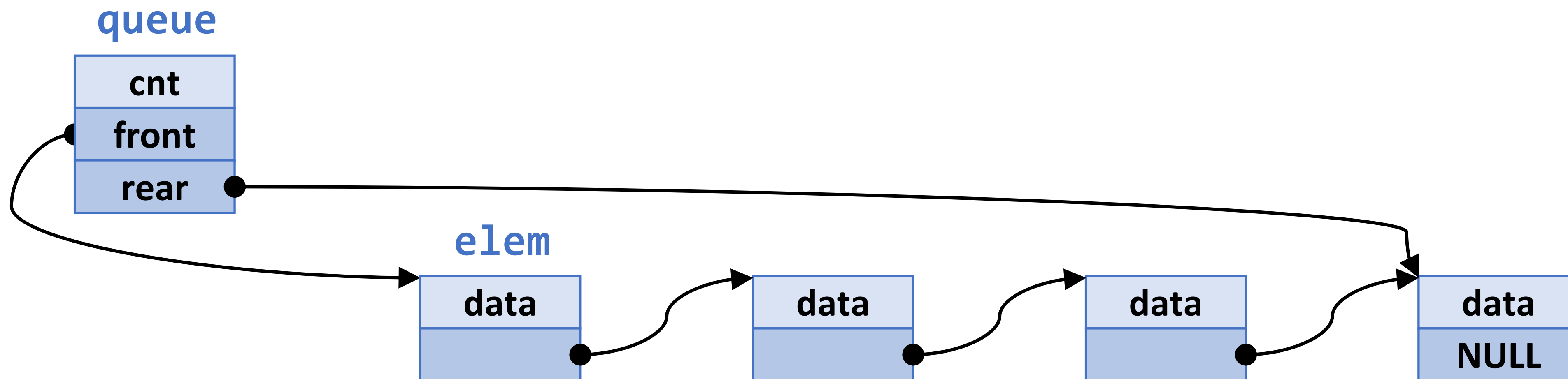
Postfix: eval()

```
int eval(stack *stk) {
    data d, d1, d2;
    stack eval;

    initialize(&eval);
    while (!empty(stk)) {
        d = pop(stk);
        switch (d.kind) {
            case VALUE: push(d, &eval); break;
            case OP:    d2 = pop(&eval);  d1 = pop(&eval);
                        switch (d.u.op) {
                            case '+': d.u.val = d1.u.val + d2.u.val; break;
                            case '-': d.u.val = d1.u.val - d2.u.val; break;
                            case '*': d.u.val = d1.u.val * d2.u.val;
                        }
                        d.kind = VALUE;
                        push(d, &eval);
        }
    }
    d = pop(&eval);
    return d.u.val;
}
```

Queue

- A data structure with the FIFO (First-In, First-Out) principle
- Two pointers, front and rear
- Insertion occurs at the rear of the list
- Deletion occurs at the front of the list





Queue: queue.h

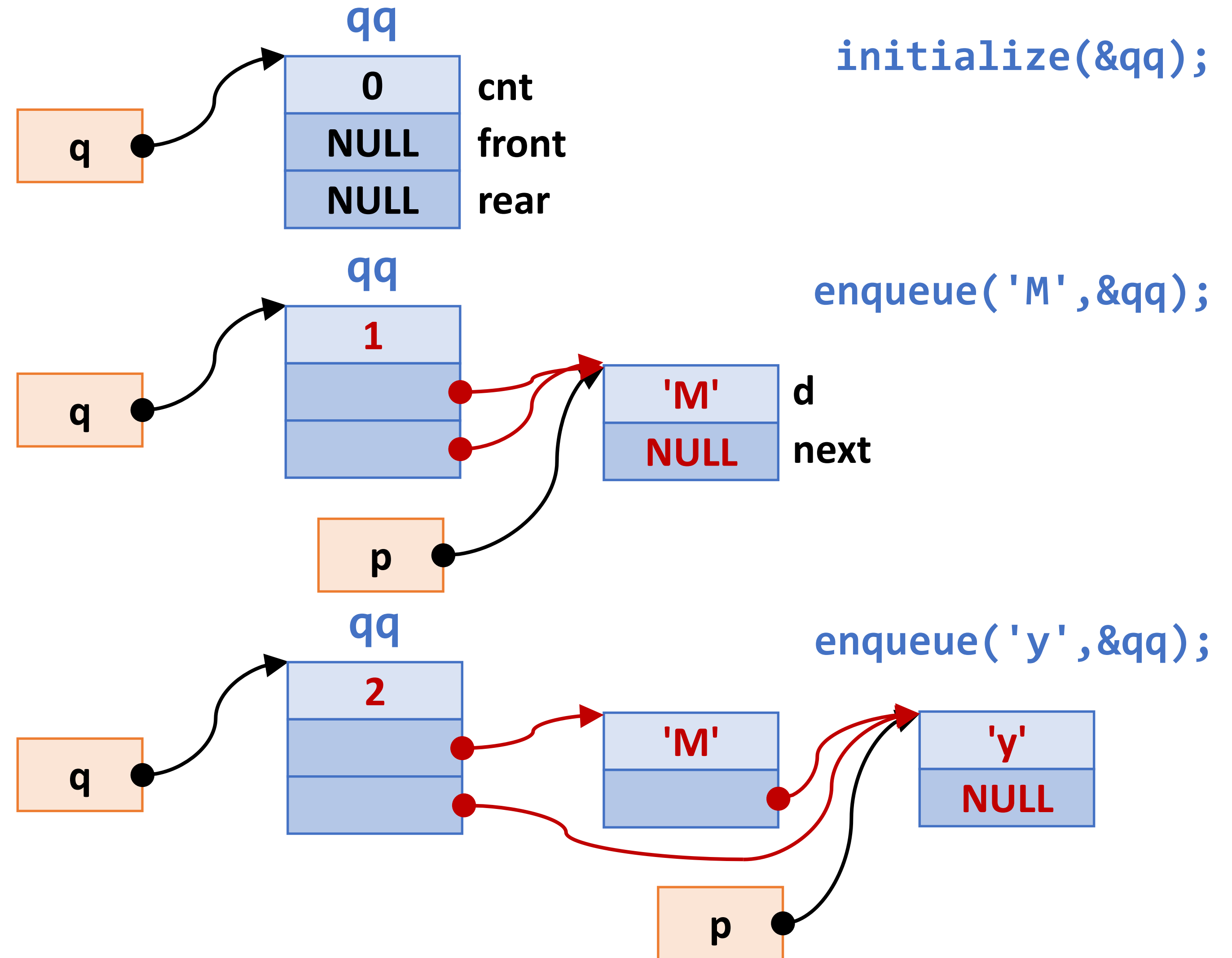
```c
#include <stdio.h>
#include <stdlib.h>

#define     EMPTY   0
#define     FULL    10000

typedef char data;
typedef enum {FALSE, TRUE} boolean;

/* an element on the queue */
struct elem {
    data d;
    struct elem *next;
};
typedef struct elem elem;
```

```c
/* queue head */
struct queue {
    int  cnt;      /* count of the elements */
    elem *front;   /* ptr to front element */
    elem *rear;    /* ptr to rear element */
};
typedef struct queue queue;

void initialize(queue *q);
void enqueue(data d, queue *q);
data dequeue(queue *q);
data front(const queue *q);
boolean empty(const queue *q);
boolean full(const queue *q);
```

Queue: initialize() and enqueue()

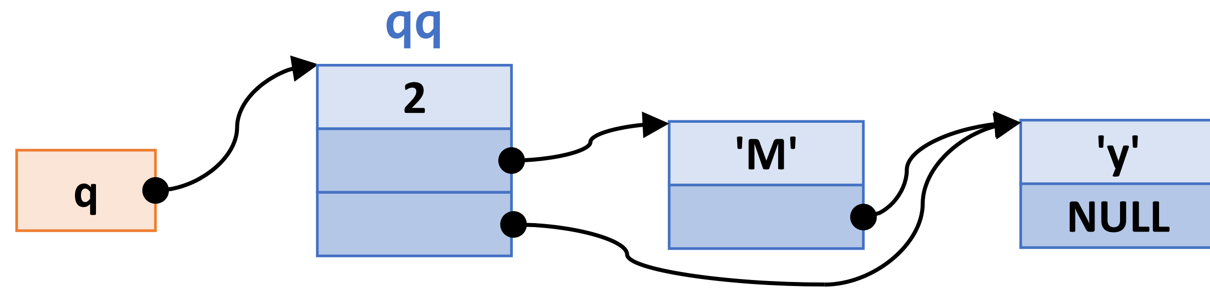
```
void initialize(queue *q) {
    q->cnt = 0;
    q->front = q->rear = NULL;
}
```

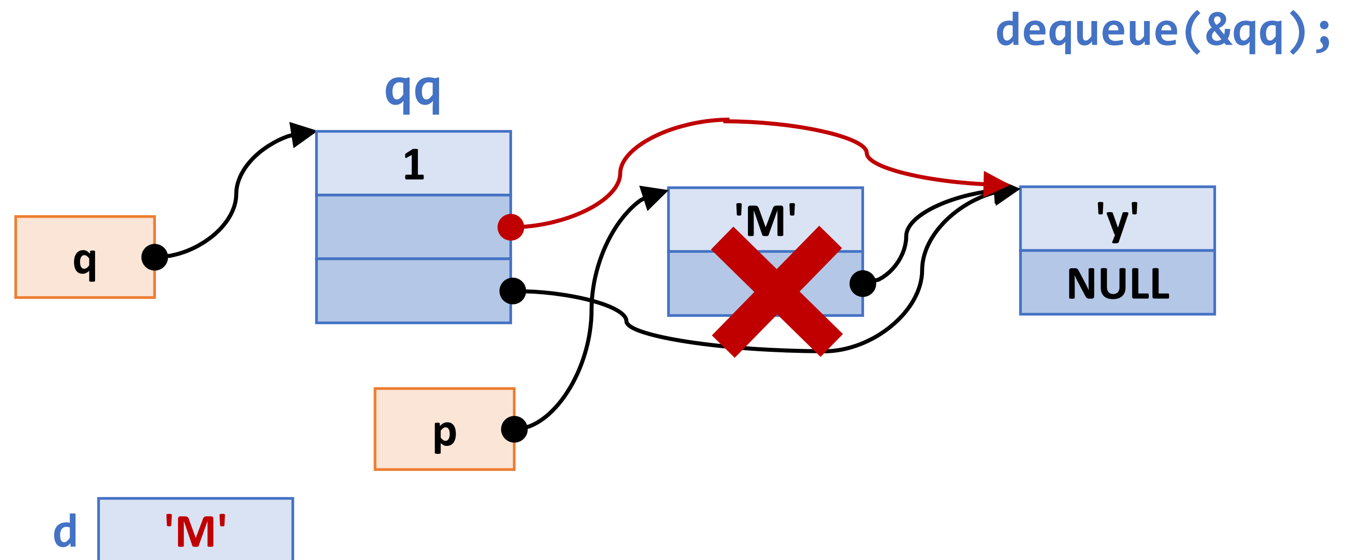
```
void enqueue(data d, queue *q) {
    elem *p;
    p = malloc(sizeof(elem));
    p->d = d;
    p->next = NULL;
    if (!empty(q)) {
        q->rear->next = p;
        q->rear = p;
    }
    else
        q->front = q->rear = p;
    q->cnt++;
}
```



Queue: dequeue()

```
data dequeue(queue *q)
{
    data d;
    elem *p;

    d = q->front->d;
    p = q->front;
    q->front = q->front->next;
    q->cnt--;
    free(p);
    return d;
}
```





Queue: front(), empty(), and full()

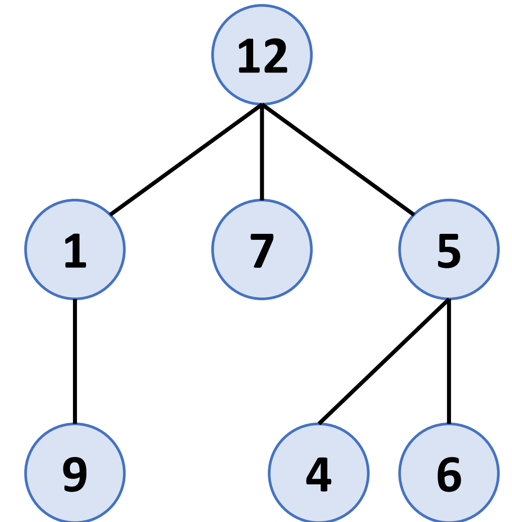
```
data front(const queue *q)
{
    return q->front->d;
}

boolean empty(const queue *q)
{
    return ((boolean) (q->cnt == EMPTY));
}

boolean full(const queue *q)
{
    return ((boolean) (q->cnt == FULL));
}
```

Queue: main()

```c
#include "queue.h"

int main(void) {
    char str[] = "My name is Joanna Kelly!";
    int i;
    queue qq;

    initialize(&qq);      /* initialize the queue */
    printf("In the string: %s\n", str);

    for (i = 0; str[i] != '\0'; i++)
        if (!full(&qq))
            enqueue(str[i], &qq);

    printf("From the queue: ");
    while (!empty(&qq))
        putchar(dequeue(&qq));
    putchar('\n');
    return 0;
}
```

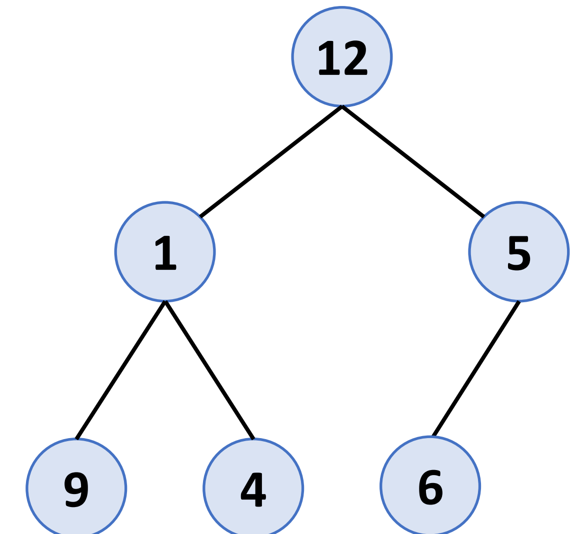
Binary Tree

- A tree T is a set of nodes storing elements in a parent-child relationship that satisfies the followings:
  - T has a special "root" node r that has no parent
  - Each node v of T, such that $v \neq r$, has a unique parent node w



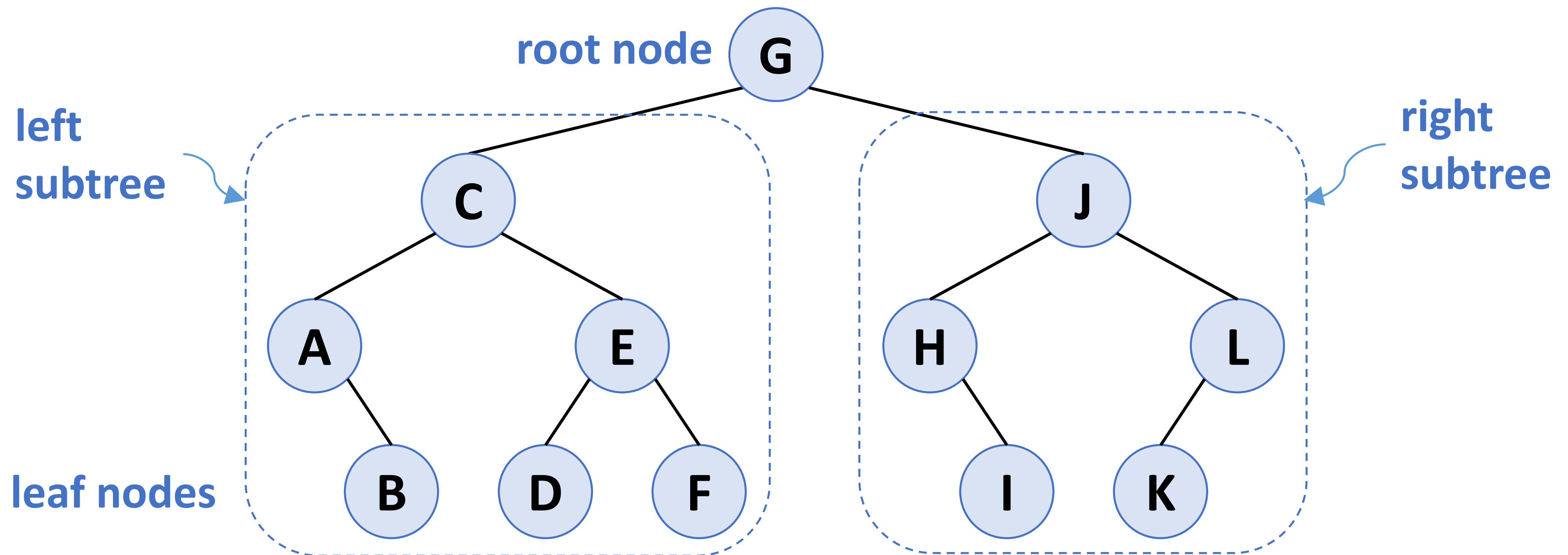
- A tree is ordered if there is a meaningful linear order among the children of each node

- A binary tree is an ordered tree where every node has at most two children (a left child and a right child)



Binary Search Tree (BST)

- A binary search tree is a binary tree such that
  - the key of the left subtree is less than the key value of its parent node
  - the key of the right subtree is greater than the key value of its parent node



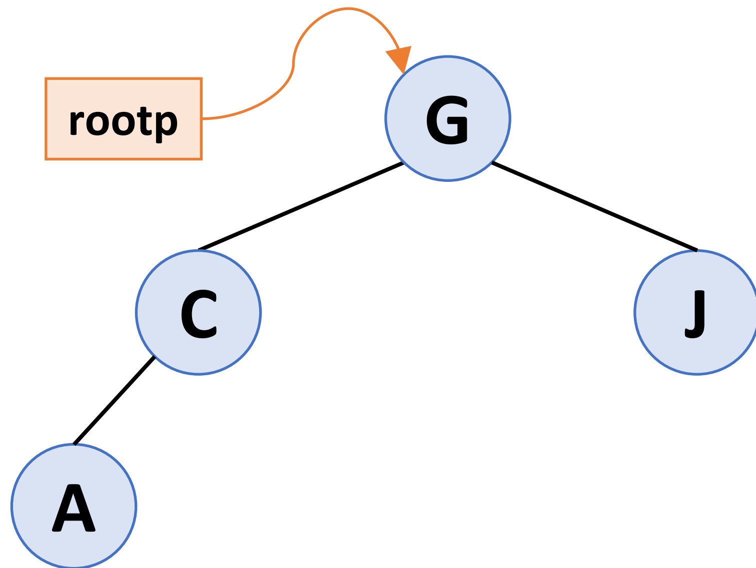
BST: bst.h and newnode()

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

typedef int DATA;

struct node {
    DATA key;
    struct node *left;
    struct node *right;
};
typedef struct node NODE;

void insert(NODE *root, DATA key);
int  lookup(NODE *root, DATA key);
void inorder(NODE *root);
void preorder(NODE *root);
void postorder(NODE *root);
```

```c
NODE *newnode(DATA key)
{
    NODE *node = malloc(sizeof(NODE));
    node->key = key;
    node->left = node->right = NULL;
    return node;
}
```

BST: insert()

- An iterative version

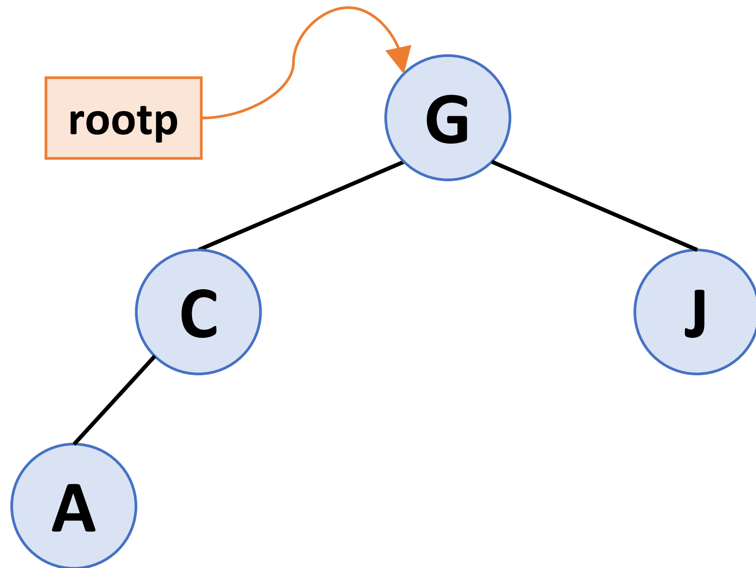



```
NODE *rootp = NULL;
insert(&rootp, 'G');
insert(&rootp, 'C');
insert(&rootp, 'J');
insert(&rootp, 'A');
```

```
void insert(NODE **root, DATA key)
{
    NODE *parent, *current;

    parent = NULL;
    current = *root;
    while (current) {
        parent = current;
        if (key <= current->key)
            current = current->left;
        else
            current = current->right;
    }
    if (parent == NULL)
        *root = newnode(key);
    else if (key <= parent->key)
        parent->left = newnode(key);
    else
        parent->right = newnode(key);
}
```

BST: insert2()

- A recursive version

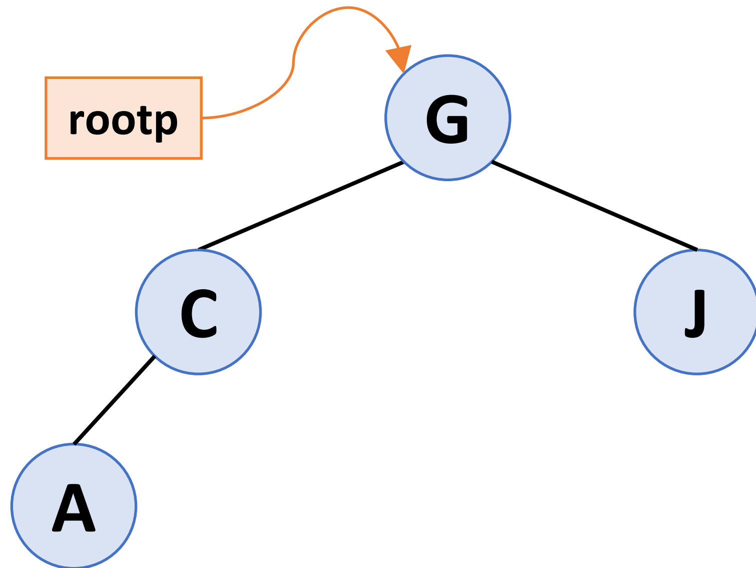



```
NODE *rootp = NULL;
insert2(&rootp, 'G');
insert2(&rootp, 'C');
insert2(&rootp, 'J');
insert2(&rootp, 'A');
```

```
void insert2(NODE **node, DATA key)
{
    if (*node == NULL)
        *node = newnode(key);
    else if (key <= (*node)->key) {
        if ((*node)->left)
            insert2(&((*node)->left), key);
        else
            (*node)->left = newnode(key);
    }
    else {
        if ((*node)->right)
            insert2(&((*node)->right), key);
        else
            (*node)->right = newnode(key);
    }
}
```

BST: `insert3()`

- Another recursive version

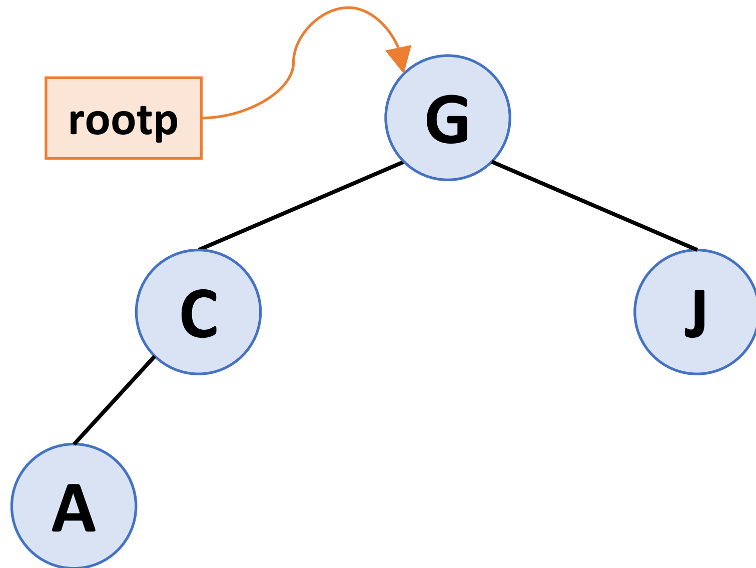



```
NODE *rootp = NULL;
rootp = insert3(rootp, 'G');
rootp = insert3(rootp, 'C');
rootp = insert3(rootp, 'J');
rootp = insert3(rootp, 'A');
```

```
NODE *insert3(NODE *node, DATA key)
{
    if (node == NULL)
        return newnode(key);

    if (key <= node->key)
        node->left = insert3(node->left, key);
    else
        node->right = insert3(node->right, key);
    return node;
}
```

BST: insert4()

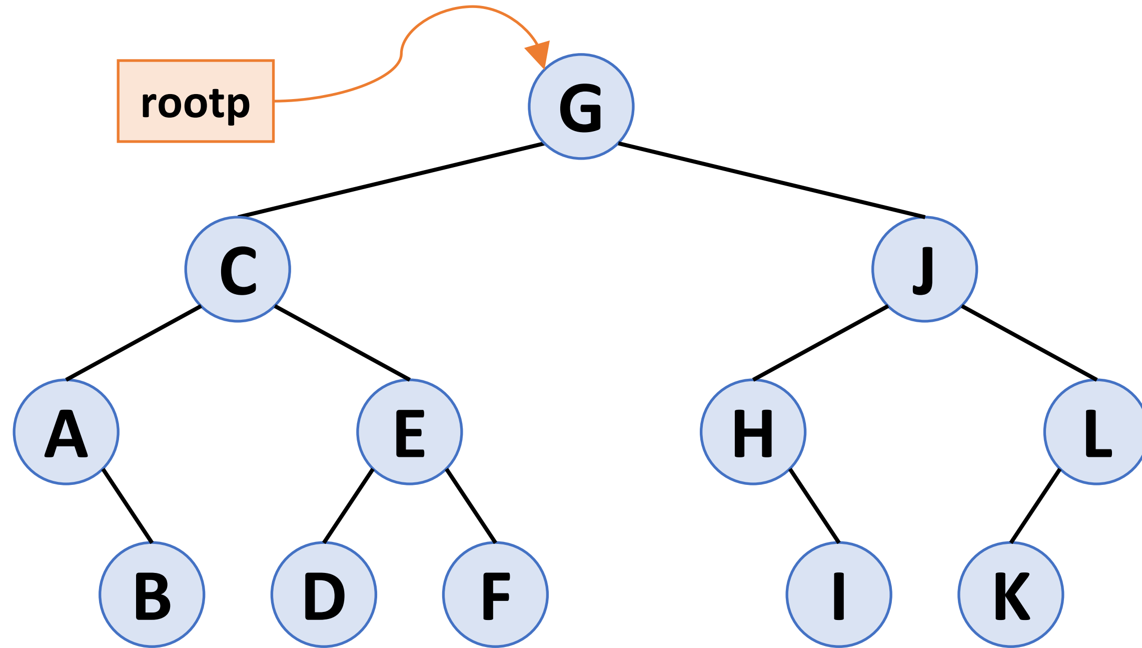
- Yet another recursive version




```
NODE *rootp = NULL;
insert4(rootp, &rootp, 'G');
insert4(rootp, &rootp, 'C');
insert4(rootp, &rootp, 'J');
insert4(rootp, &rootp, 'A');
```

```
void insert4(NODE *node, NODE **parent, DATA key)
{
    if (node == NULL)
        *parent = newnode(key);
    else if (key <= node->key)
        insert4(node->left, &node->left, key);
    else
        insert4(node->right, &node->right, key);
}
```

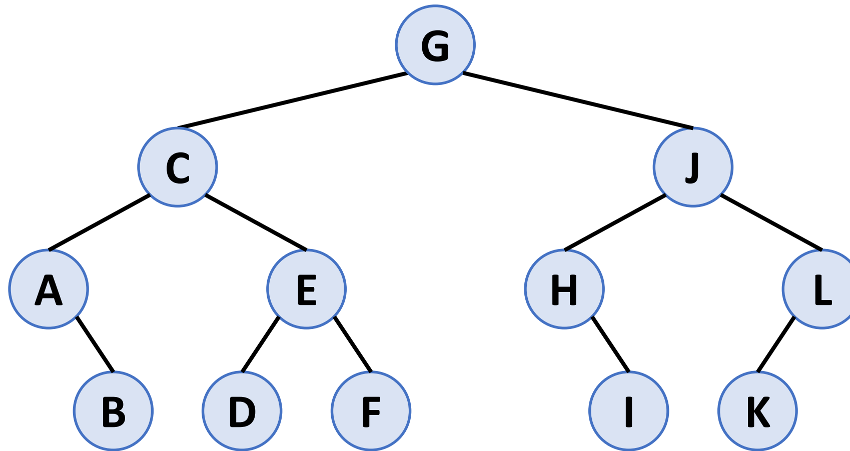
BST: lookup()




```
lookup(rootp, 'E');
lookup(rootp, 'K');
lookup(rootp, 'Z');
```

```
int lookup(NODE *node, DATA key)
{
    if (node == NULL)
        return 0;

    if (node->key == key)
        return 1;
    else if (key < node->key)
        return lookup(node->left, key);
    else
        return lookup(node->right, key);
}
```

BST: Tree Traversals



- Inorder
  - A B C D E F G H I J K L
- Preorder
  - G C A B E D F J H I L K
- Postorder
  - B A D F E C I H K L J G

```
void inorder(NODE *node) {
    if (node) {
        inorder(node->left);
        printf("%c ", node->key);
        inorder(node->right);
    }
}
```

```
void preorder(NODE *node) {
    if (node) {
        printf("%c ", node->key);
        preorder(node->left);
        preorder(node->right);
    }
}
```

```
void postorder(NODE *node) {
    if (node) {
        postorder(node->left);
        postorder(node->right);
        printf("%c ", node->key);
    }
}
```