CSE 230 Intermediate Programming in C and C++

Structures and List Processing

Fall 2017

Stony Brook University

Instructor: Shebuti Rayana

http://www3.cs.stonybrook.edu/~cse230/

Self-referential Structure

- Self-referential structures have pointer members that refer to the structure itself. Such data structures are called dynamic data structures.
- Unlike arrays or simple variables that are normally allocated at block entry, dynamic data structures often require storage management routines to explicitly obtain and release memory.

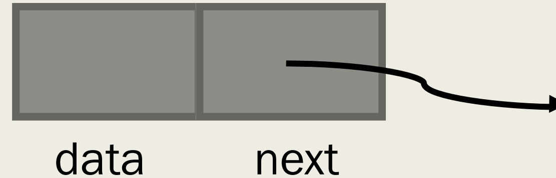
Example: Self-referential Structure

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

- This declaration can be stored in two words of memory.
  - The first word stores the member data, and
  - the second word stores the member next.
- The pointer variable next is called a link. Each structure is linked to a succeeding structure by way of the member next. These structures are conveniently displayed pictorially with links shown as arrows.

Example (cont.)

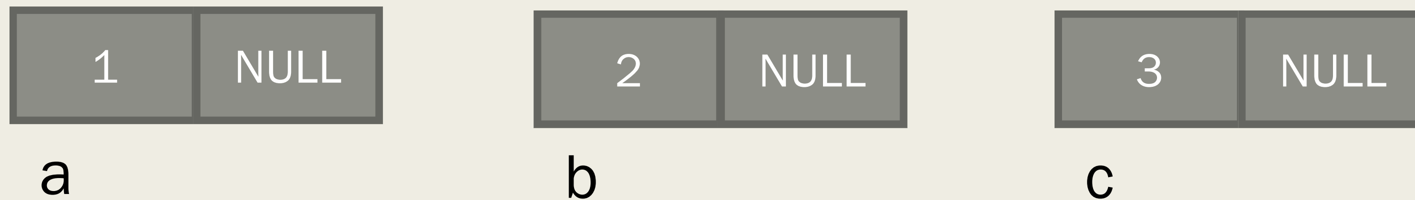
A structure of type `struct list`



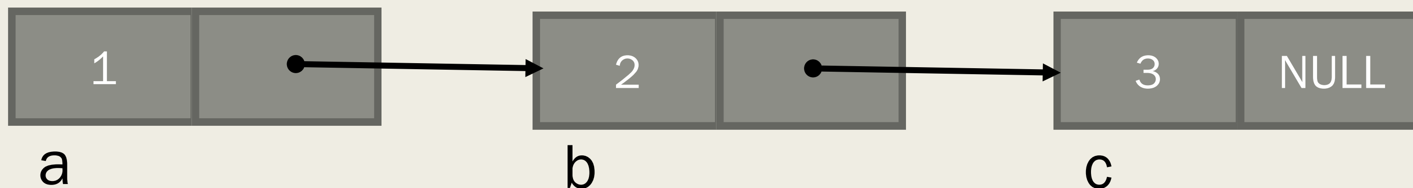
- The pointer variable next contains either an address of the location in memory the successor list element, or the special value NULL defined as 0.
- – NULL is used denote the end of the list.
- Declaration: `struct list a, b, c;`
- Initialization:

```
a.data = 1; b.data = 2; c.data = 3;

a.next = b.next = c.next = NULL
```

Example (cont.)



- Chaining: `a.next = &b;  b.next = &c;`

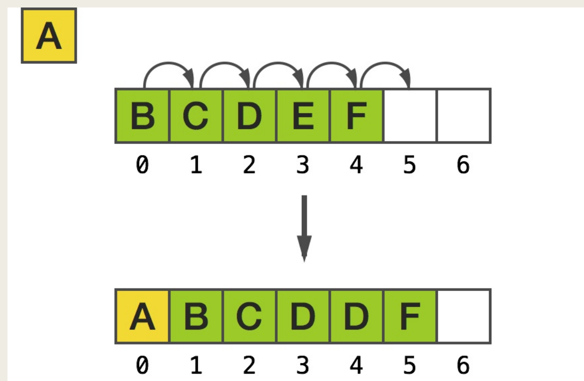


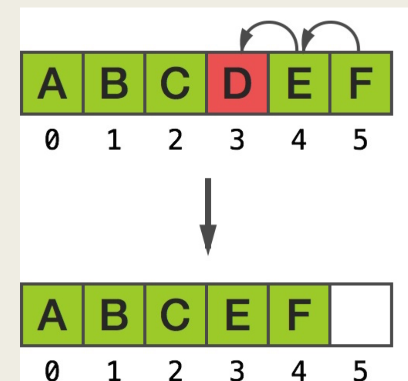
- Accessing elements:

```
a.next -> data
a.next -> next -> data
```

Motivation of using Linked List

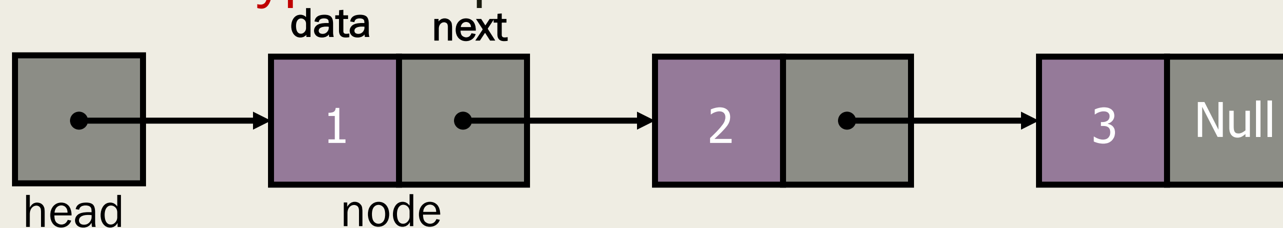
- Consider an array: 1, 4, 10, 19, 25
- If we want to insert 7 between 4 and 10, what to do?
- Disadvantages of using arrays to store data:
  - arrays are static structures and cannot be easily extended or reduced to fit the data set
  - Expensive to maintain: insertions and deletions are costly



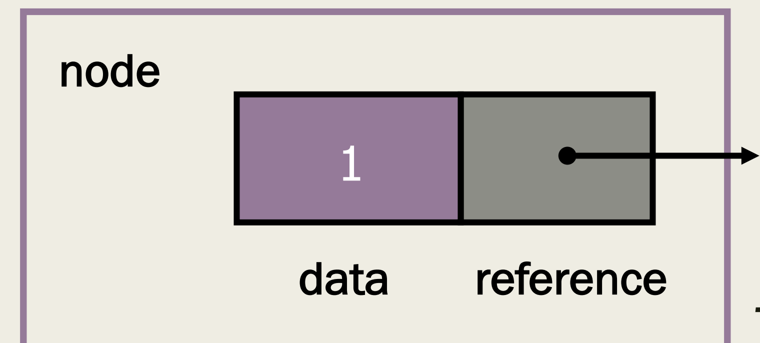


Linked List

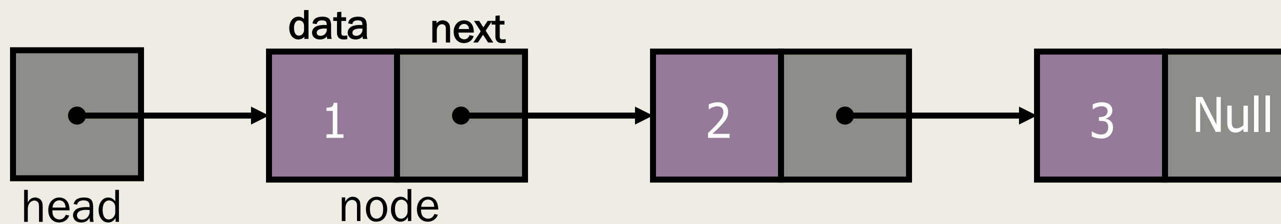
- A linear data structure where each element is a separate structure type. Sequence of elements called Nodes



- Every node (except the last) contains pointer to the next.
- Components of a node:
  - Data: stores the relevant information
  - Link: stores the address/pointer to the next node



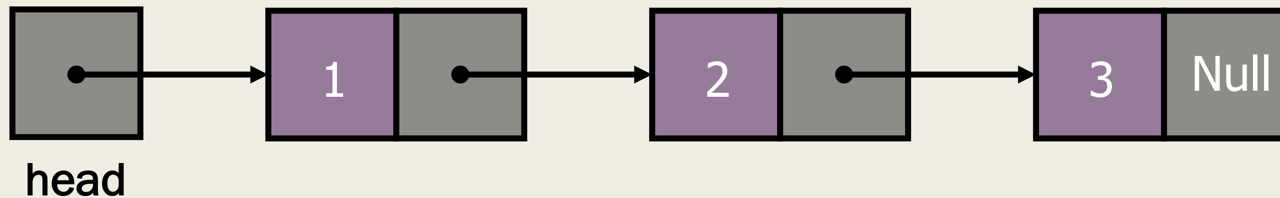
Linked Lists: Anatomy



- **Head**: a special pointer, points to the first node. Head is not a separate node, but a pointer to the first node. If the list is empty then the head is a null pointer.
- The last node points to none, as such **Null**

Linked List: Example

Conceptual Picture



Actual Picture

Word Address	Content
1000	2 2nd content
1002	1008
1004	1 1st content
1006	1000
1008	3 3rd content
1010	0
head 1012	1004

Linked List: More Terminologies

- Some lists may have a special link called the **tail** that points the last node in a list.
- A **cursor** is a link that points to one of the nodes of the list.
- A node's **successor** is the next node in the sequence
  - The last node has no successor
- A node's **predecessor** is the previous node in the sequence
  - The first node has no predecessor
- A list's **length** is the number of elements in it
  - A list may be empty (i.e. head = tail = Null)

Linked List

- A linked list is a dynamic data structure.
  - The number of nodes in a list is not fixed and can grow and shrink on demand (unlike arrays).
- One disadvantage of a linked list against an array
  - Does not allow direct access to the individual elements.
  - In order to retrieve an element, we have to follow the pointers and traverse one by one [O(n)]

Why linked list? not array?

- Linked lists are more complex to code and manage than arrays, but they have some distinct advantages.

– Efficient memory usage

- An array needs a continuous memory block, but a node of a list can be anywhere, it just points to the next element

– Dynamic: a linked list can easily grow and shrink in size.

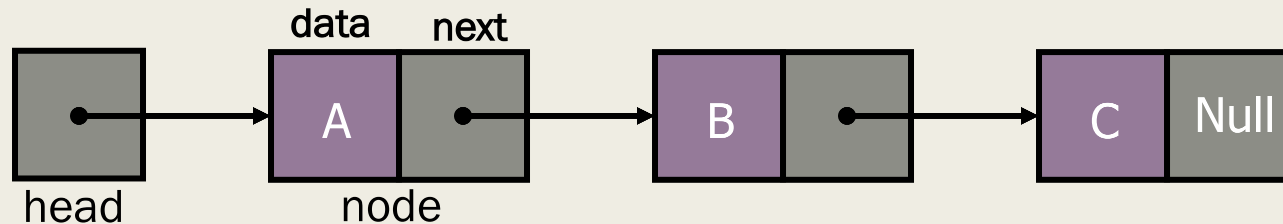
- We don't need to know how many nodes will be in the list. They are created in memory as needed. In contrast, the size of an array is fixed

– Easy and fast insertions and deletions

- To insert or delete an element in an array, we need to copy to temporary variables to make room for new elements or close the gap caused by deleted elements.
- With a linked list, no need to move other nodes. Only need to reset some pointers.

Types of Linked Lists

- A Singly or Linear Linked List



- A Doubly Linked List:

  – a list that has two pointers: one to the next node and another to previous node

- Circular linked list where last node of the list points back to the first node (or the head) of the list.

Linear Linked List Implementation

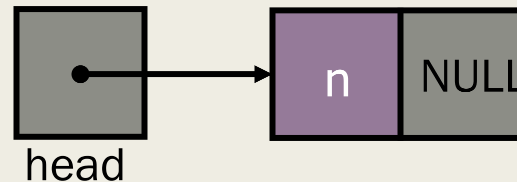
```
In file list.h

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

typedef char DATA;    /* will use char in examples */

struct linked_list {
       DATA d;
       struct linked_list *next;
};

typedef struct linked_list ELEMENT;
typedef ELEMENT            *LINK;
```

Storage Allocation

obtains a piece of memory from the system adequate to store an ELEMENT and assigns its address to the pointer head.

```
LINK head;
head = malloc(sizeof(ELEMENT));
head -> ‘n’;
head -> next = NULL;
```

Creating a linked list

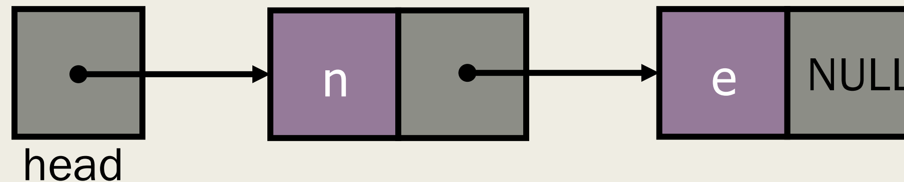


Adding more elements

- A second element is added by the assignments

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

A two element linked list

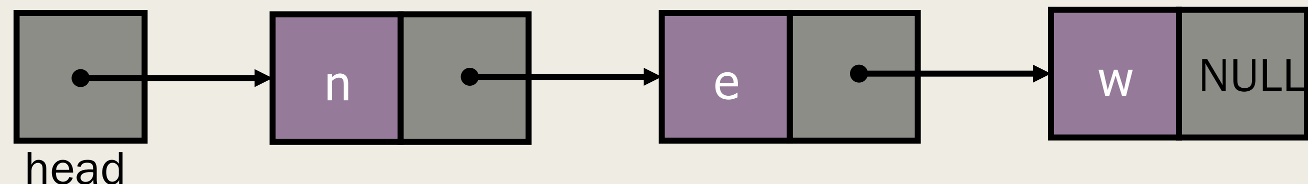


Adding more elements

- A third element is added by the assignments

```
head -> next -> next =
malloc(sizeof(ELEMENT));

head -> next -> next -> d = ’w';

head  -> next -> next -> next = NULL;
```

A two element linked list



Linear List Operations

- Some of the basic operations are:
  - Creating a list
  - Counting the elements
  - Looking up an element
  - Concatenating two lists
  - Inserting an element
  - Deleting an element
- This operations can be implemented with both recursion and iteration.
- The data d in could be redefined as an arbitrarily complicated data structure.

Creating a List from a String using Recursion

```
/* List creation using recursion. */

#include <stdlib.h>
#include "list.h"

LINK string_to_1ist(char s[])
{
        LINK head;
        if(s[0] == '\0')  /* base case */
                return NULL;
        else {
                head = malloc(sizeof(ELEMENT));
                head -> d = s[0];
                head -> next = string_to_list(s + 1);
                return head;
        }
}
```

Creating a List from a String: Iterative solution

```c
/* List creation using iteration. */

#include <stdlib.h>
#include "list.h"

LINK s_to_l(char s[])
{
	LINK head = NULL, tail;
	int i;
	if (s[0] != '\0'){ /* first element */
		head = malloc(sizeof(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;
}
```

Counting Elements in a List

```
/* Count a list recursively. */
int count(LINK head)
{
        if (head == NULL)
                return 0;
        else
                return (1 + count(head -> next));
}

/* Count a list iteratively. */
int count_it(LINK head)
{
        int cnt = 0 ,
        for ( ; head != NULL; head = head -> next)
                ++cnt;
        return cnt;
}
```

Keep in mind that head is passed “call-by-value”, so that invoking count_it() does not destroy access to the list in the calling environment

Printing List

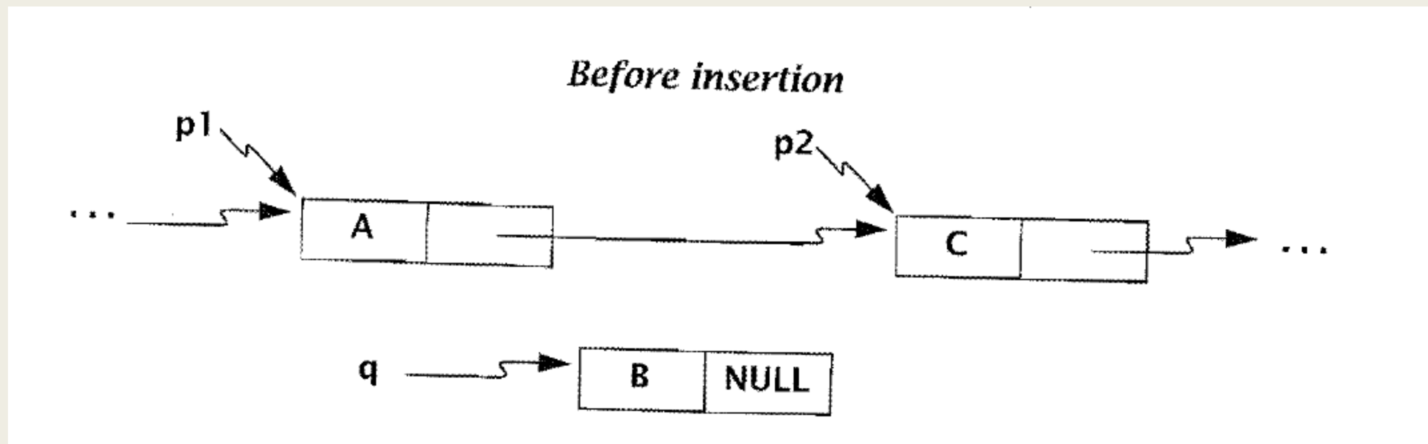
```
/* Print a list recursively. */

void print_list(LINK head)
{
        if (head == NULL)
                printf("NULL")'
        else {
                printf("%c -->", head -> d);
                print_list(head -> next);
        }
}
```

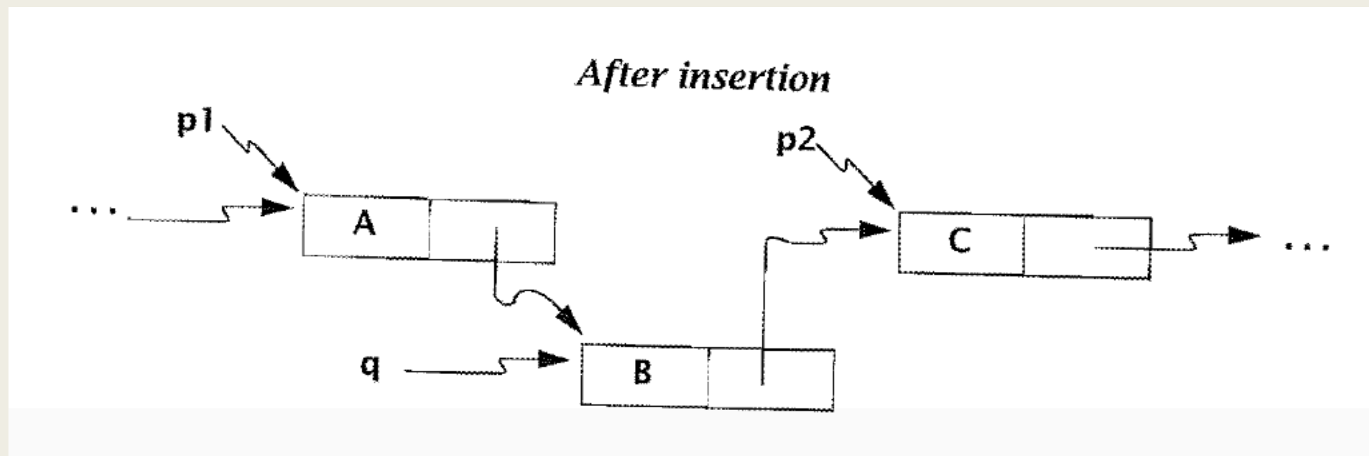
Concatenating two Lists

```
/* Concatenate list a and b with a as head. */
void concatenate(LINK a, LINK b)
{
        assert(a != NULL);
        if (a -> next == NULL)
                a -> next = b;
        else
        concatenate(a -> next, b);
}
```

Insertion

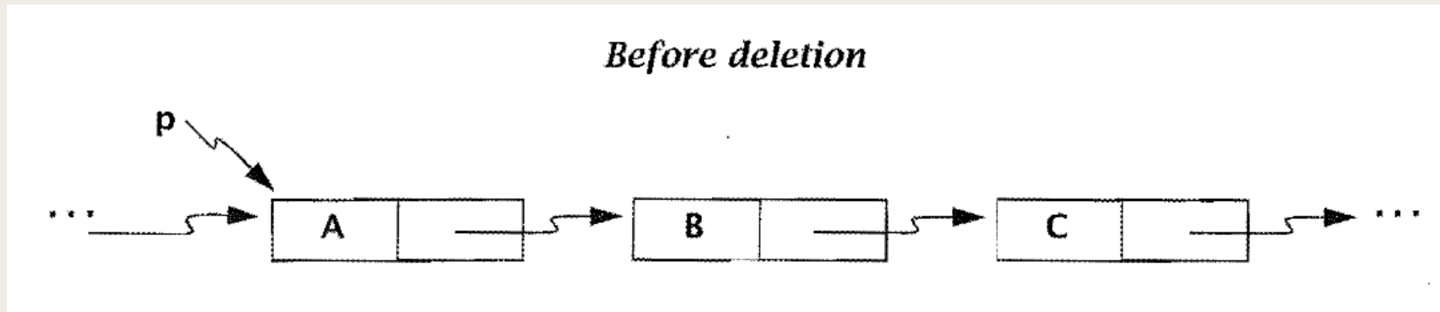
```
/* Inserting an element in a linked list. */
void insert(LINK p1, LINK p2, LINK q)
{
        assert(pl -> next == p2);
        p1 -> next = q;
        q  -> next = p2;
}
```

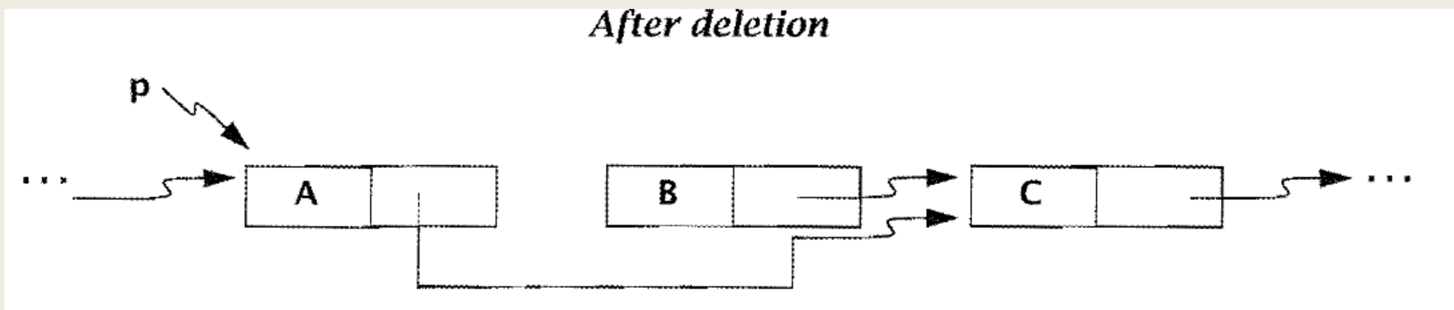




Deletion

```
p -> next = p -> next -> next;
```



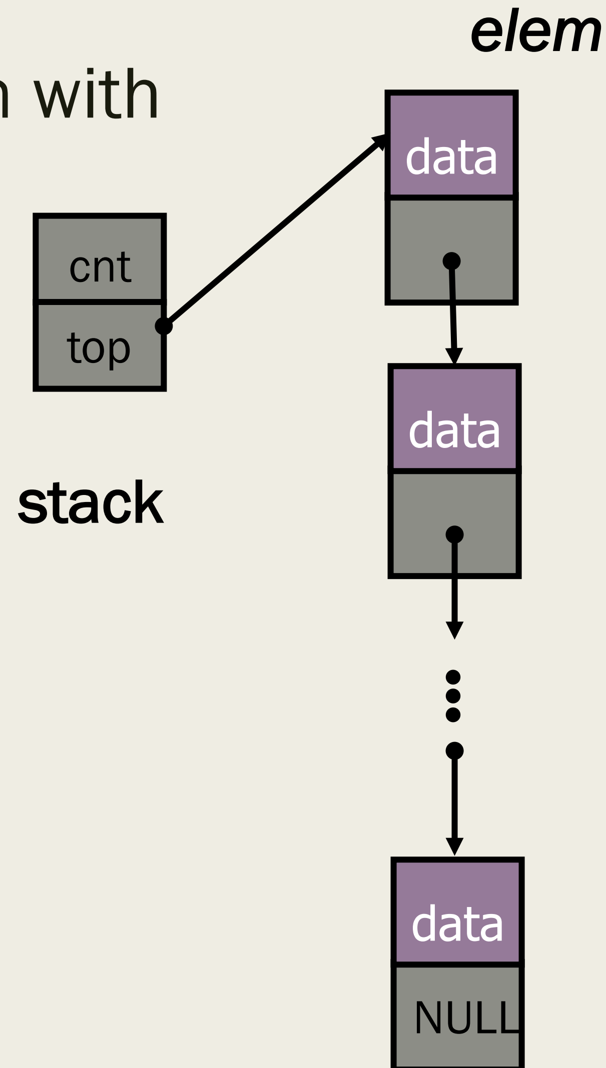


Deletion of a whole List

```
/* Recursive deletion of a list. */

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

Stack

- Stack implementation with
  - Fixed size arrays ✔
  - Linked lists



A stack implementation

Stack: Implementation (stack.h)

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

#define EMPTY 0
#define FULL 10000

typedef char data;
typedef enum {false, true} boolean;

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

typedef struct elem elem;

struct stack {
       int cnt;        /* a counter of the elements */
       elem *top;      /* a 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(stack *stk);
boolean empty(const stack *stk);
boolean full(const stack *stk);
```

Basic stack routines

```c
/* The basic stack routines. */

#include "stack.h"

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++;
}

data pop (stack *stk)
{
        data d;
        elem *p;
        d = stk  -> top  -> d;
        p = stk -> top;
        stk -> top = stk -> top -> next;
        stk -> cnt--;
        free(p);
        return d;
}
```

```c
data top (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: Implementation (test code)

```c
/* Test the stack implementation by reversing a string. */
#include <stdio.h>
int main(void)
{
	char str[] = "My name is Laura Pohll";
	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 off the stack */
	putchar('\n');
	return 0;
}
```

```
In the string: My name is Laura Pohll
From the stack: llhoP aruaL si eman yM
```

Stack: Application 1

- Balanced Parenthesis: An arithmetic expression has balanced parenthesis if and only if:
- the number of left parentheses of each type is equal to the number of right parentheses of each type
- each right parenthesis of a given type matches to a left parenthesis of the same type to its left and all parentheses in between are balanced correctly.
- Example:
  - ({A + B} –C) Balanced
  - ({A + B) –C} Not balanced
  - ({A + B} –[C / D]) Balanced
  - ((({A + B} –C) / D)) Not balanced

Algorithm:
check for balanced parenthesis

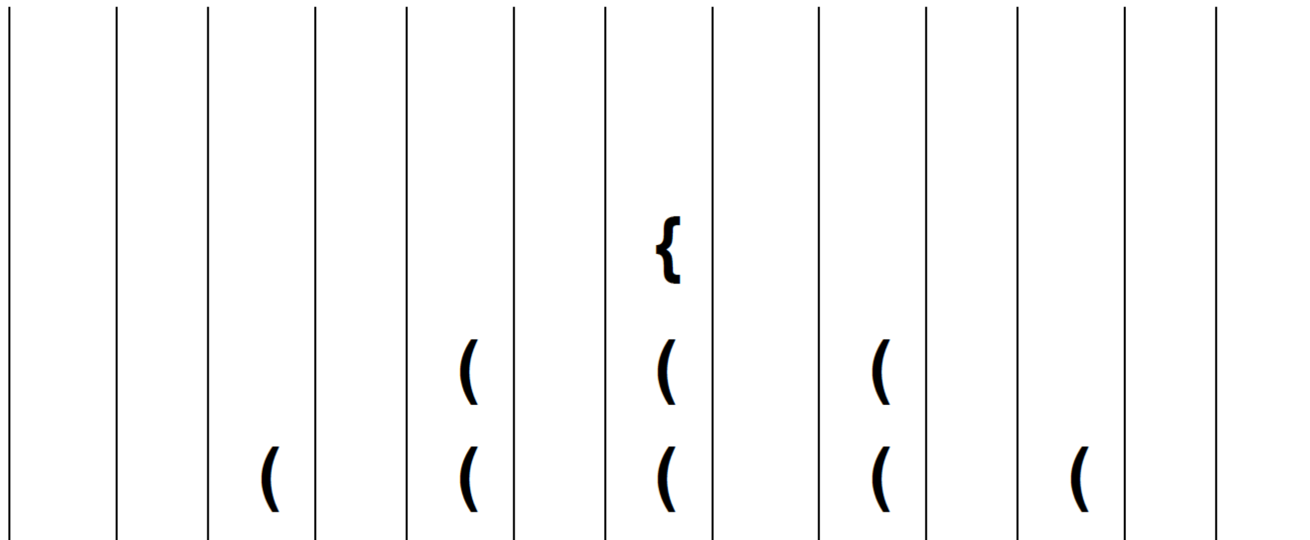
Scan the expression from left to right.

- For each left parenthesis that is found, push on the stack
- For each right parenthesis that is found,
  - If the stack is empty, return false(too many right parentheses)
  - Otherwise, pop the top parenthesis from the stack:
    - If the left and right parentheses are of the same type, discard. Otherwise, return false.

If the stack is empty when the scan is complete, return true. Otherwise, return false. (too many left parentheses)

Trace

- (({A + B} – C) / D))
- Stack trace:



Stack: Application 2

- Evaluating Expressions: An expression is fully parenthesized if every operator has a pair of balanced parentheses marking its left and right operands.
- Not fully-parenthesized:
  3 * (5 + 7) –9 (2 -4) * (5 -7) + 8
- Fully-parenthesized:
  ((3 * (5 + 7))–9)(((2 -4)*(5 -7))+ 8)

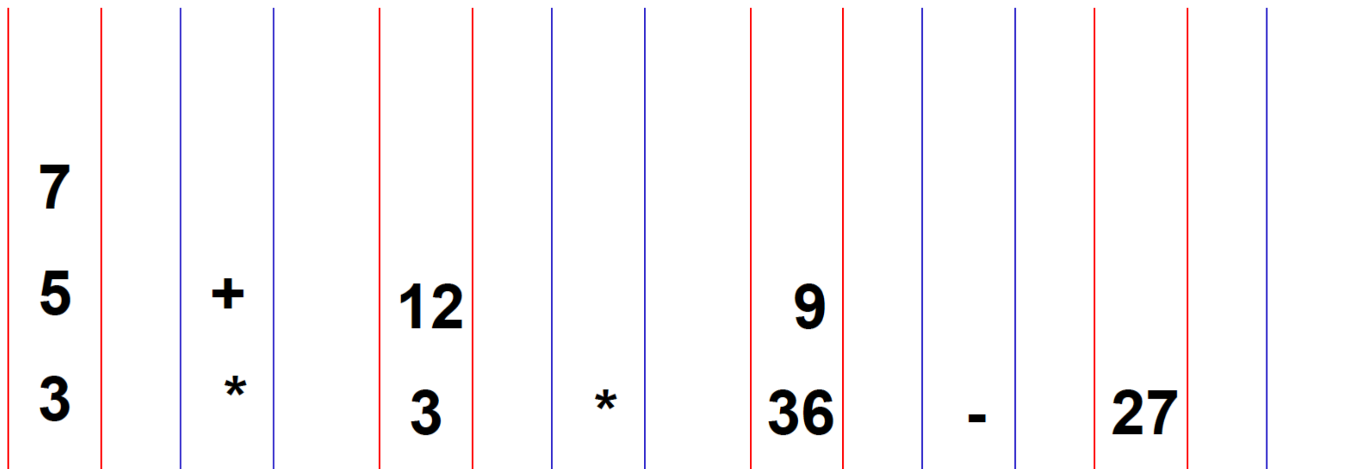
General Idea

- The first operation to perform is surrounded by the innermost set of balanced parentheses.
  - Example:((3 * (5 + 7)) –9)   First op: +
- By reading expression from left to right, first operator comes immediately before first right parenthesis.
- Replace that subexpression with its result and search for next right parenthesis, etc.
  - Example:((3 * 12) –9) = (36 –9) = 27

General Idea (cont.)

- How do we keep track of operands and operators as we read past them in the expression from left to right?
  - Use two stacks: one for operands and one for operators.
- When we encounter a right parenthesis, pop off one operator and two operands, perform the operation, and push the result back on the operand stack.

Trace



Algorithm

- Let each operand or operator or parenthesis symbol be a `token`.
- Let `NumStack` store the operands.
- Let `OpStack` store the operations.

For each token in the input expression do

If `token` = operand, `NumStack.push(token)`

If `token` = operator, `OpStack.push(token)`

If `token = ")"`,

operand2←NumStack.pop()

operand1←NumStack.pop()

operator ←OpStack.pop()

result ←operand1 operator operand2

NumStack.push(result)

If token = "(", ignore token

After expression is parsed, answer ← NumStack.pop()

Stack: Application 3

- Arithmetic Expression:

Infix notation: operator is between its two operands

3 + 5 (5 + 7) * 9 5 + (7 * 9)

Prefix notation: operator precedes its two operands

+ 3 5 * + 5 7 9 + 5 * 7 9

Postfix notation: operator follows its two operands

3 5 + 5 7 + 9 * 5 7 9 * +

Precedence of Operators

- Multiplication and division (higher precedence) are performed before addition and subtraction (lower precedence)
- Operators in balanced parentheses are performed before operators outside of the balanced parentheses.
- If two operators are of the same precedence, they are evaluated left to right.

Example

- Infix expression:

A + B * (C * D – E / F) / G – H

6 4 1 3 2 5 7

- What is prefix equivalent?

– + A / * B – * C D / E F G H

7 6 5 4 3 1 2

- What is postfix equivalent?

A B C D * E F / – * G / + H –

1 2 3 4 5 6 7

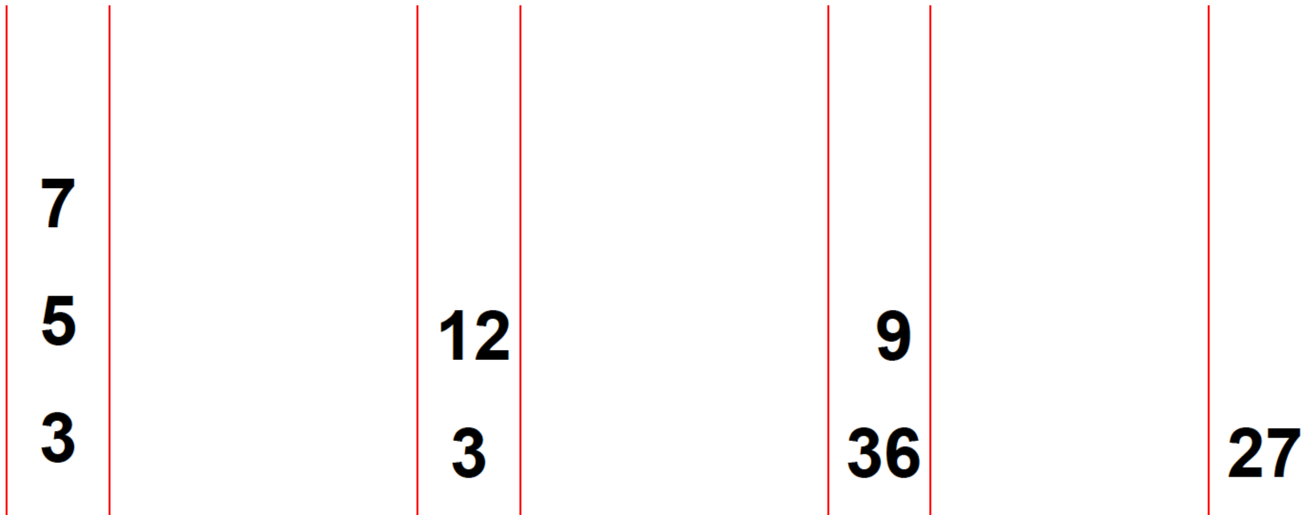
Evaluating a Postfix Expression

- Let each operand or operator be a `token`.
- Let `NumStack` store the operands.

```
For each token in the input expression do
    If token = operand, NumStack.push(token)
    If token = operator,
        operand2←pop()
        operand1←pop()
        result ←operand1  operator  operand2
        NumStack.push(result)
answer ←pop()
```

Trace

- INFIX: 3 * (5 + 7) – 9
- POSTFIX: 3 5 7 + * 9 -
- Stack traces: operands



ANSWER

Translating Infix to Postfixt

- Let each operand, operator, or parenthesis be a token.
- Let OpStack store the operators.
- Let postfix string P = “” (empty string)

For each token in the input expression do

    If token = operand, append operand to P

    If token = operator, push(token)

    If token = “)”, append pop() to P

    If token = “(“, ignore

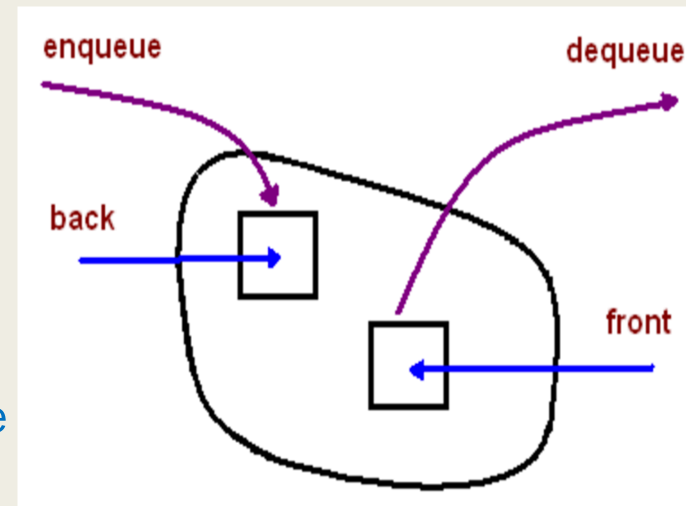
Trace

- **Infix: ((3 * (5 + 7)) – 9)**

Stack (sideways)	Postfix String
empty	3
*	3
*	3 5
* +	3 5
* +	3 5 7
*	3 5 7 +
empty	3 5 7 + *
–	3 5 7 + *
–	3 5 7 + * 9
empty	3 5 7 + * 9 –

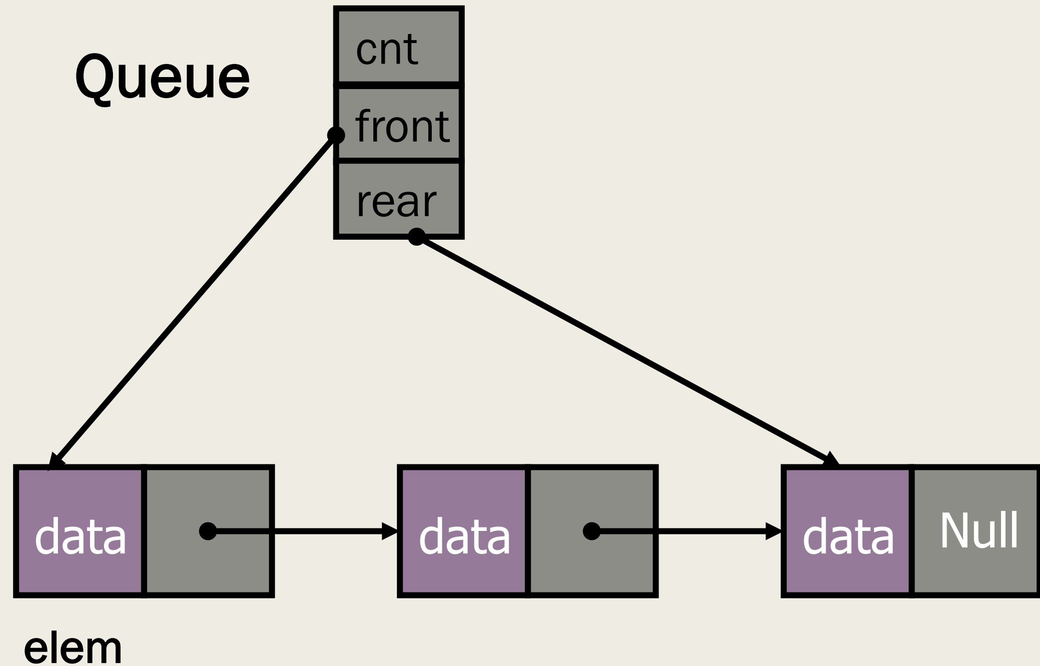
Queue

- Another ADT **queue** is a container of elements (a linear collection) that are inserted and removed according to the first-in first-out (FIFO) principle.
  - Example: a line of students in the SAC food court. New additions to a line made to the back of the queue, while removal (or serving) happens in the front.
- In the queue only two operations are allowed **enqueue** and **dequeue**
  - enqueue means to insert an item into the back of the queue
  - dequeue means removing the front item



The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Pictorial Representation of Queue



Queue: Implementation (queue.h)

```c
/* A linked list implementation of a queue. */

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

#define EMPTY 0
#define FULL 10000

typedef unsigned int data;
typedef enum{false, true} boolean;

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

/* an element in the queue */

typedef struct elem elem;

struct queue {
        int cnt;      /* a count of the elements */
        elem *front; /* pointer to the front element */
        elem *rear;  /* pointer to the rear element */
};

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

Basic queue routines

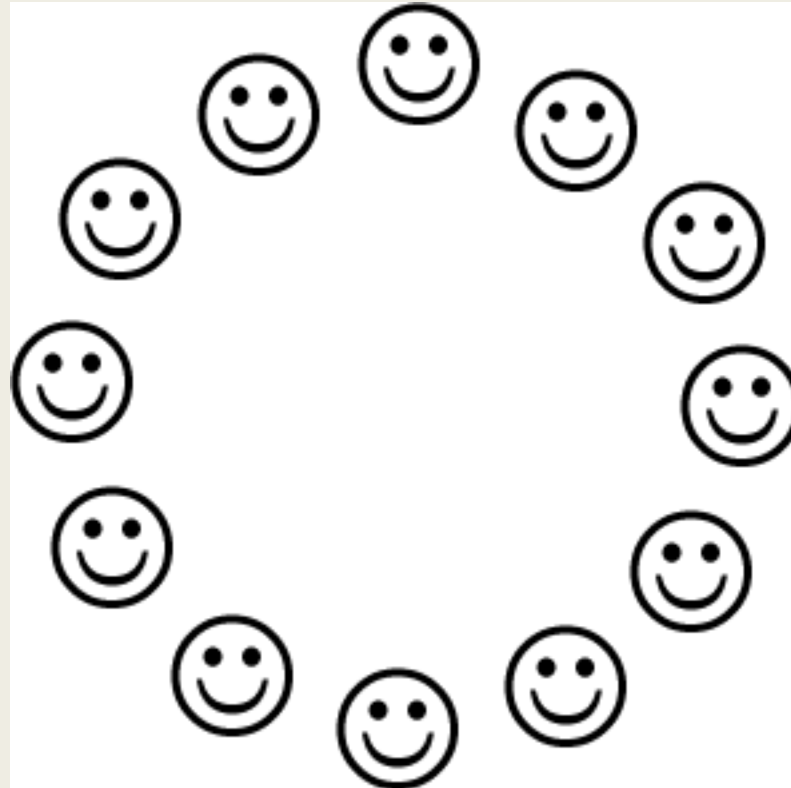
```c
/* The basic queue routines. */

#include "queue.h"

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


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;
}
```

```c
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++;
}


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));
}
```

Josephus Problem

- Suppose there are n children standing in a queue.
- Children are numbered from 1 to n in the clockwise direction.
- Choose a lucky number say m.
- They start counting in clockwise direction from the child designated as 1. The counting proceeds until the mth child is identified. mth child is eliminated from the queue.
- Counting for the next round begins from the child next to the eliminated one and proceeds until the mth child is identified. This child is then eliminated and the process continues.
- After few rounds of counting only one child is left and this child is declared as winner.

Josephus Problem



Can you solve this problem using a queue?