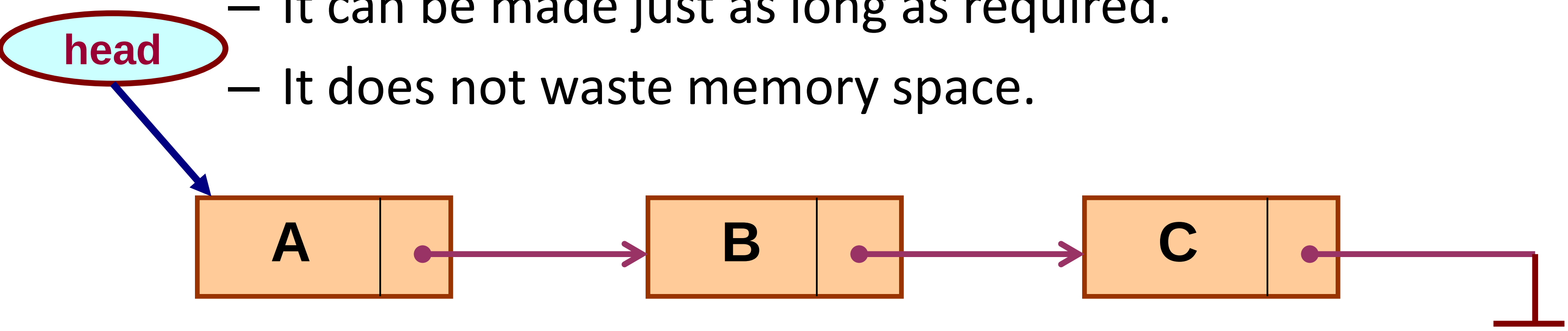
Linked List

Introduction

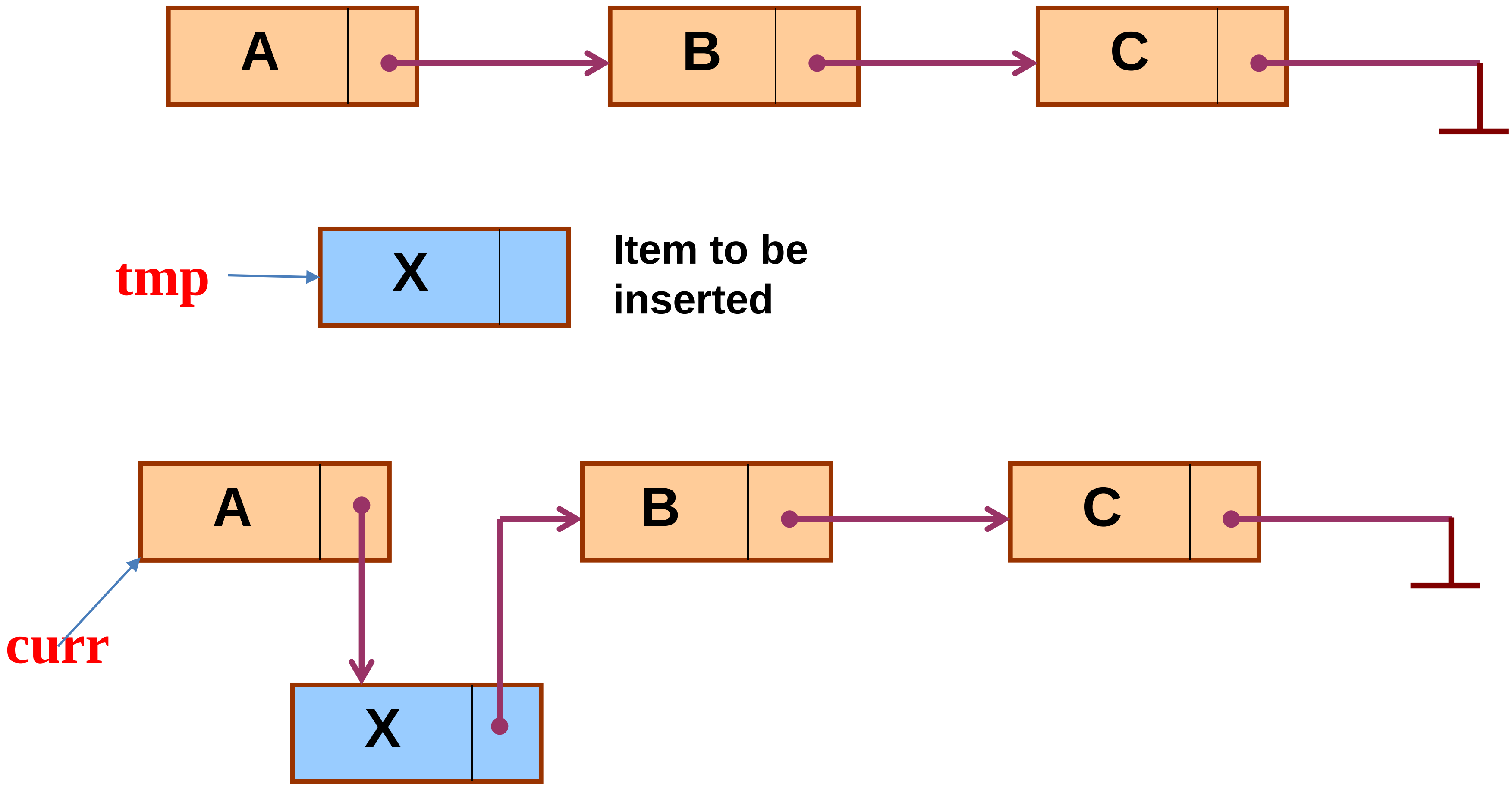
- A linked list is a data structure which can change during execution.
  - Successive elements are connected by pointers.
  - Last element points to `NULL`.
  - It can grow or shrink in size during execution of a program.
  - It can be made just as long as required.
  - It does not waste memory space.



- Keeping track of a linked list:
  - Must know the pointer to the first element of the list (called *start*, *head*, etc.).

- Linked lists provide flexibility in allowing the items to be rearranged efficiently.
  - Insert an element.
  - Delete an element.

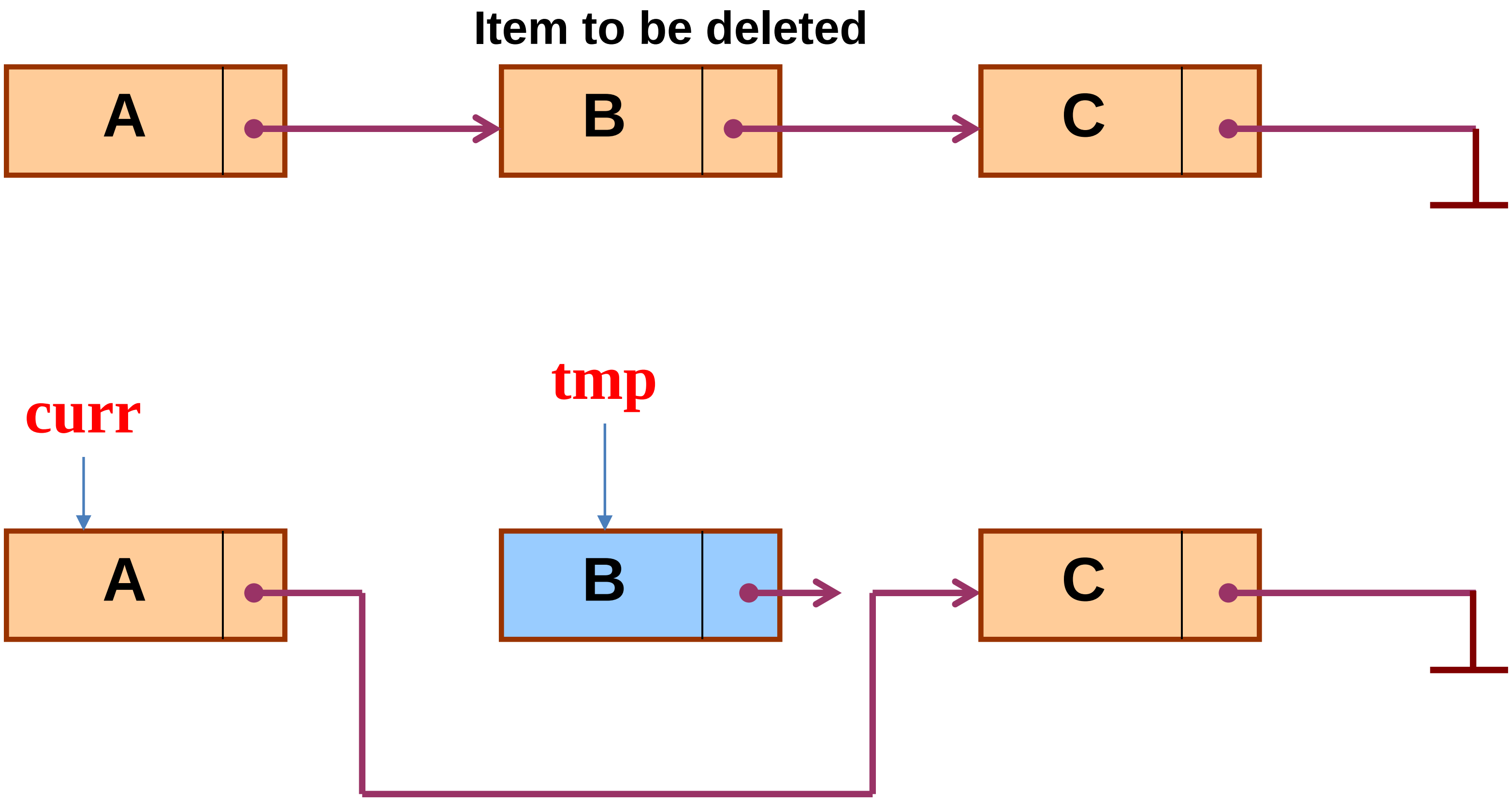
Illustration: Insertion



Pseudo-code for insertion

```
typedef struct nd {
   struct item data;
   struct nd * next;
   } node;

void insert(node *curr)
{
node * tmp;

tmp=(node *) malloc(sizeof(node));
tmp->next=curr->next;
curr->next=tmp;
}
```

Illustration: Deletion



Pseudo-code for deletion

```
typedef struct nd {
   struct item data;
   struct nd * next;
   } node;

void delete(node *curr)
{
node * tmp;
 tmp=curr->next;
curr->next=tmp->next;
free(tmp);
}
```

In essence ...

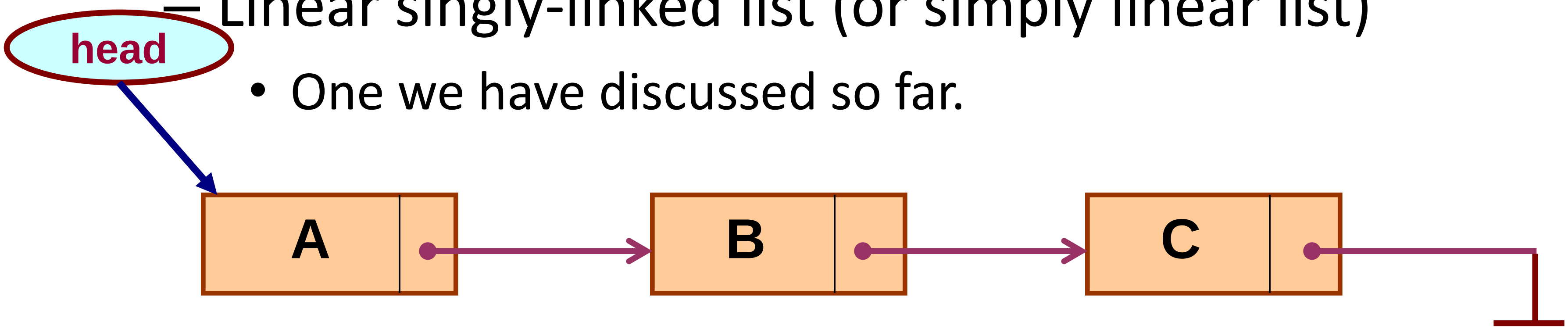
- For insertion:
  - A record is created holding the new item.
  - The next pointer of the new record is set to link it to the item which is to follow it in the list.
  - The next pointer of the item which is to precede it must be modified to point to the new item.
- For deletion:
  - The next pointer of the item immediately preceding the one to be deleted is altered, and made to point to the item following the deleted item.

Array versus Linked Lists

- Arrays are suitable for:
  - Inserting/deleting an element at the end.
  - Randomly accessing any element.
  - Searching the list for a particular value.
- Linked lists are suitable for:
  - Inserting an element.
  - Deleting an element.
  - Applications where sequential access is required.
  - In situations where the number of elements cannot be predicted beforehand.

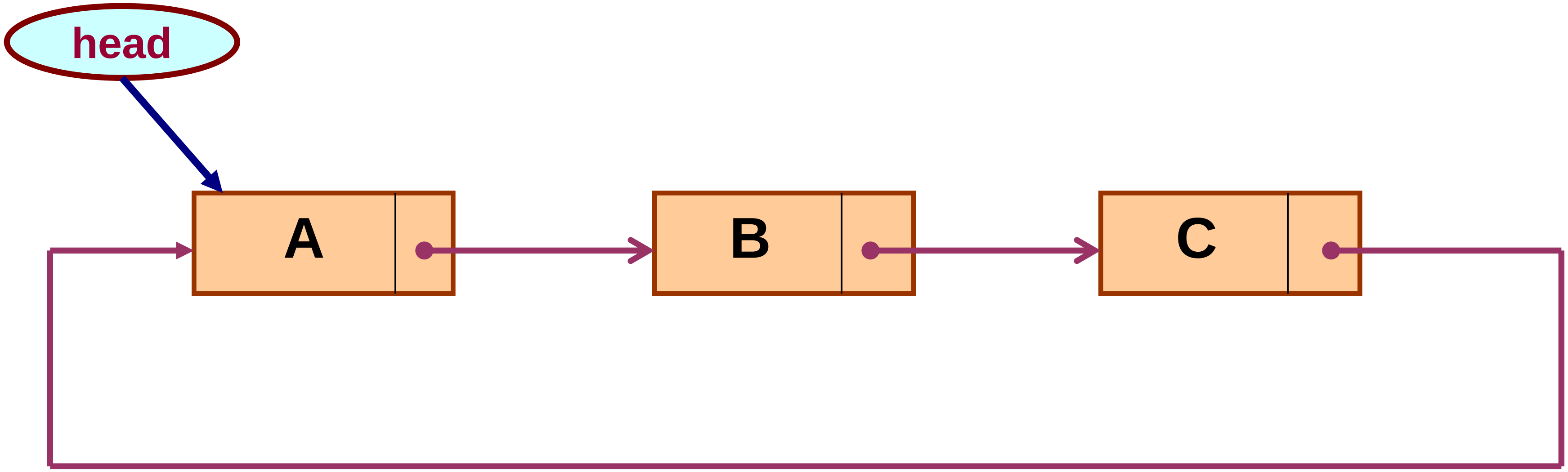
Types of Lists

- Depending on the way in which the links are used to maintain adjacency, several different types of linked lists are possible.
  - Linear singly-linked list (or simply linear list)
    - One we have discussed so far.



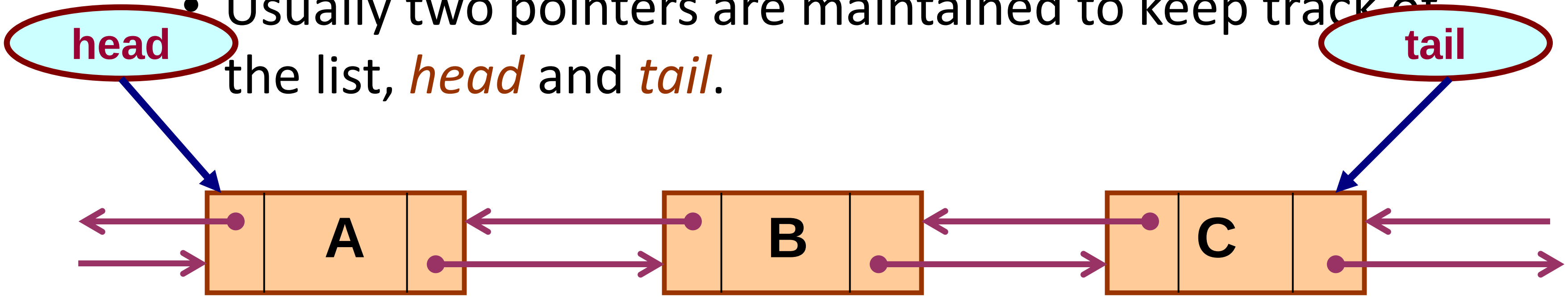
– Circular linked list

  - The pointer from the last element in the list points back to the first element.



– Doubly linked list

- Pointers exist between adjacent nodes in both directions.
- The list can be traversed either forward or backward.
- Usually two pointers are maintained to keep track of the list, *head* and *tail*.



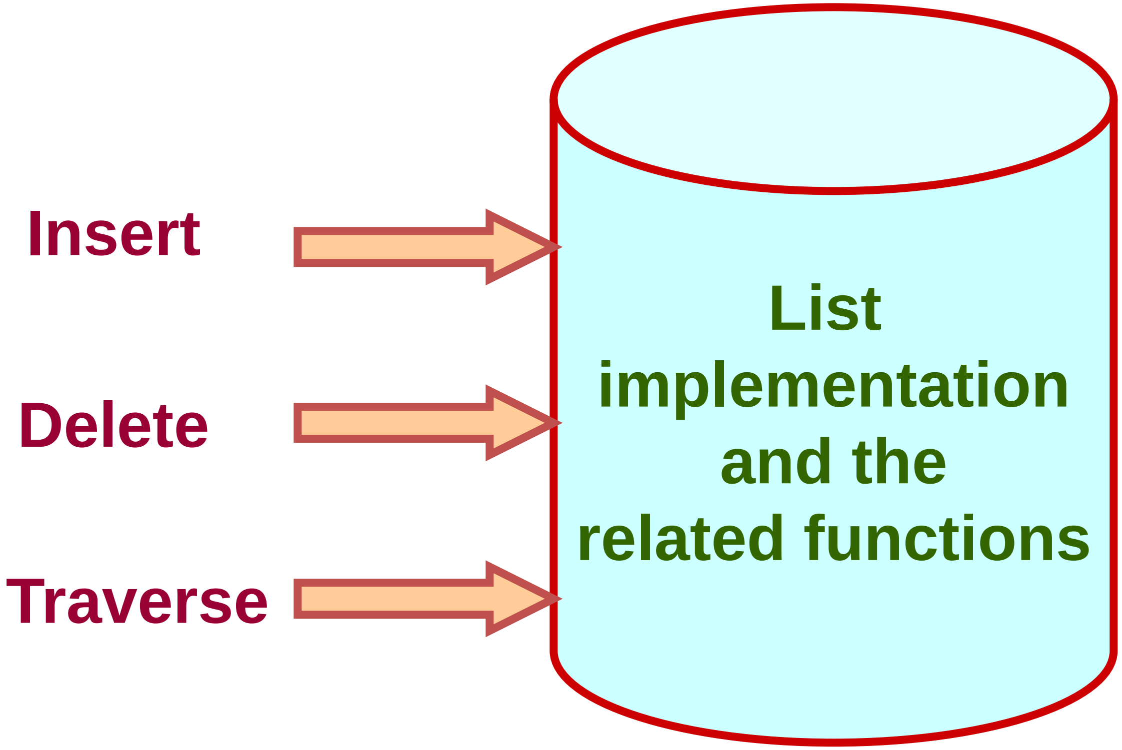
Basic Operations on a List

- Creating a list
- Traversing the list
- Inserting an item in the list
- Deleting an item from the list
- Concatenating two lists into one

List is an Abstract Data Type

- What is an abstract data type?
  - It is a data type defined by the user.
  - Typically more complex than simple data types like *int*, *float*, etc.
- Why abstract?
  - Because details of the implementation are hidden.
  - When you do some operation on the list, say insert an element, you just call a function.
  - Details of how the list is implemented or how the insert function is written is no longer required.

Conceptual Idea



Example: Working with linked list

- Consider the structure of a node as follows:

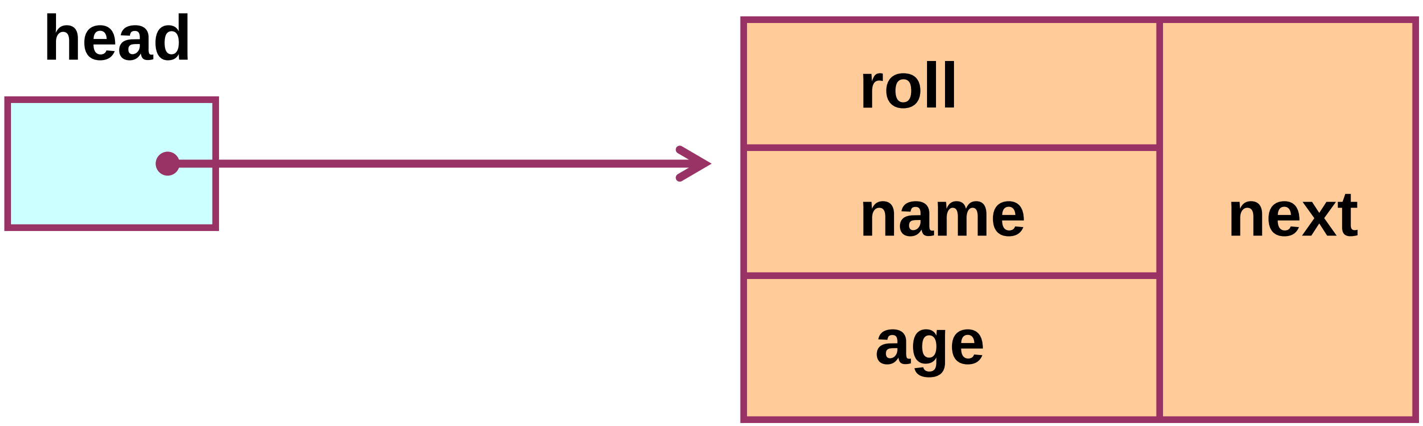
```
struct stud {
              int   roll;
              char  name[25];
              int   age;
              struct stud *next;
            };

  /* A user-defined data type called "node" */

typedef struct stud node;
node *head;
```

Creating a List

How to begin?

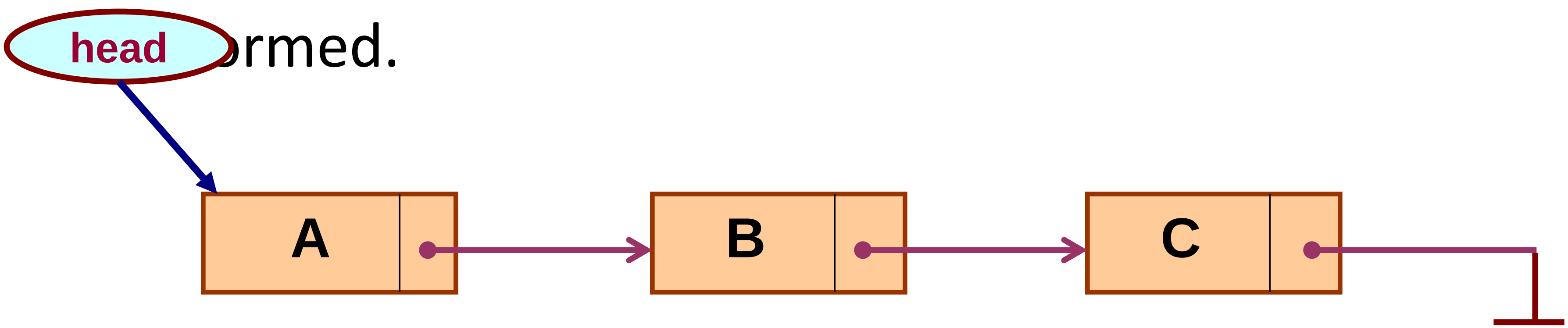
- To start with, we have to create a node (the first node), and make `head` point to it.

```
head = (node *)
  malloc(sizeof(node));
```



Contd…

- If there are n number of nodes in the initial linked list:
  – Allocate n records, one by one.
  – Read in the fields of the records.
  – Modify the links of the records so that the chain is formed.



```c
node *create_list()
{
    int  k, n;
    node  *p, *head;

    printf  ("\n How many elements to enter?");
     scanf ("%d", &n);

    for  (k=0; k<n; k++)
    {
        if (k == 0) {
          head = (node *) malloc(sizeof(node));
          p = head;
        }
        else {
               p->next  = (node *) malloc(sizeof(node));
               p = p->next;
             }

        scanf ("%d %s %d", &p->roll, p->name, &p->age);
    }

    p->next  =  NULL;
    return (head);
}
```

- To be called from `main()` function as:

```
node *head;
………
head = create_list();
```

Traversing the List

What is to be done?

- Once the linked list has been constructed and *head* points to the first node of the list,
  - Follow the pointers.
  - Display the contents of the nodes as they are traversed.
  - Stop when the *next* pointer points to NULL.

```
void display (node *head)
{
  int  count = 1;
  node  *p;

  p = head;
  while (p != NULL)
  {
    printf ("\nNode %d: %d %s %d", count,
                     p->roll, p->name, p->age);
    count++;
    p = p->next;
  }
  printf ("\n");
}
```

- To be called from `main()` function as:

```
node *head;
………
display (head);
```

Inserting a Node in a List

How to do?

- The problem is to insert a node *before a specified node*.
  - Specified means some value is given for the node (called *key*).
  - In this example, we consider it to be `roll`.
- Convention followed:
  - If the value of roll is given as *negative*, the node will be inserted at the *end* of the list.

Contd…

- When a node is added at the beginning,
  - Only one next pointer needs to be modified.
    - *head* is made to point to the new node.
    - New node points to the previously first element.
- When a node is added at the end,
  - Two next pointers need to be modified.
    - Last node now points to the new node.
    - New node points to NULL.
- When a node is added in the middle,
  - Two next pointers need to be modified.
    - Previous node now points to the new node.
    - New node points to the next node.

```
void insert (node *head)
{
    int  k = 0, rno;
    node *p, *q, *new;

    new = (node *) malloc(sizeof(node));

    printf ("\nData to be inserted: ");
      scanf ("%d %s %d", &new->roll, new->name, &new->age);
    printf ("\nInsert before roll (-ve for end):");
      scanf ("%d", &rno);

    p = head;

    if (p->roll == rno)       /* At the beginning */
    {
        new->next = p;
        head = new;
    }
```

```
else
   {
      while ((p != NULL) && (p->roll != rno))
        {
             q = p;
             p = p->next;
        }

        if  (p == NULL)        /* At the end */
        {
             q->next = new;
             new->next = NULL;
        }
       else if  (p->roll  == rno)
                              /* In the middle */
                {
                     q->next = new;
                     new->next = p;
                }
   }
}
```

The pointers q and p always point to consecutive nodes.

- To be called from `main()` function as:

```
node *head;
………
insert (&head);
```

Deleting a node from the list

What is to be done?

- Here also we are required to delete a specified node.
  - Say, the node whose `roll` field is given.
- Here also three conditions arise:
  - Deleting the first node.
  - Deleting the last node.
  - Deleting an intermediate node.

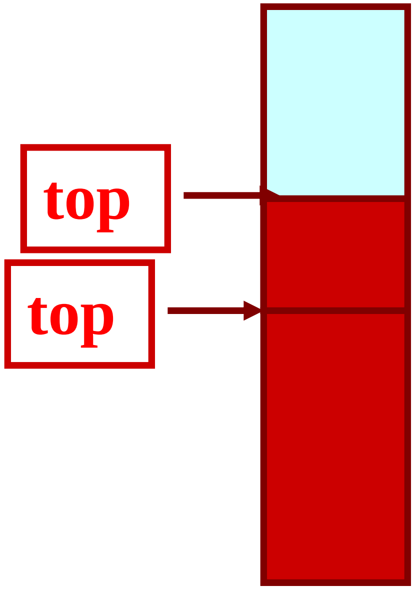
```
void  delete (node *head)
{
    int  rno;
    node  *p, *q;

    printf ("\nDelete for roll :");
      scanf ("%d", &rno);

    p = head;
    if  (p->roll == rno)
             /* Delete the first element */
    {
        head = p->next;
        free (p);
    }
```

```
 else
    {
        while  ((p != NULL) && (p->roll != rno))
        {
            q = p;
            p  =  p->next;
        }

        if  (p == NULL)       /* Element not found */
           printf ("\nNo match :: deletion failed");

        else if (p->roll == rno)
                          /* Delete any other element */
             {
                 q->next  =  p->next;
                 free (p);
             }
    }
}
```

Stack Implementations: Using Array and Linked List

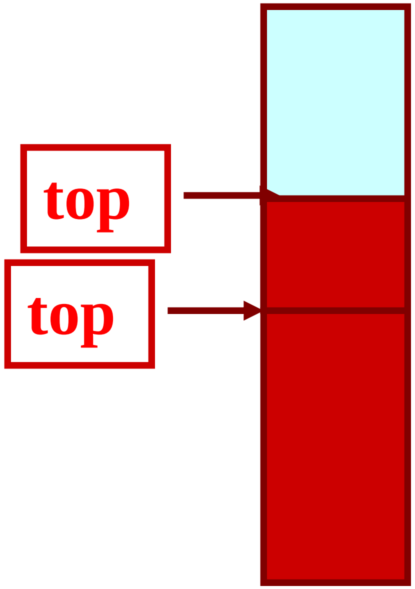
STACK USING ARRAY

PUSH



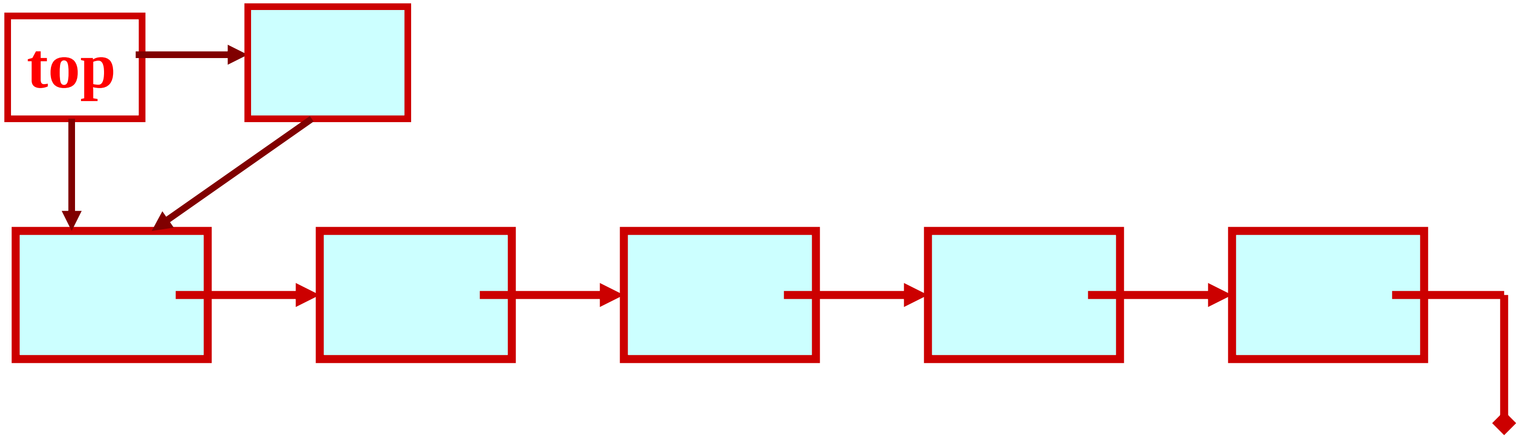
STACK USING ARRAY

POP



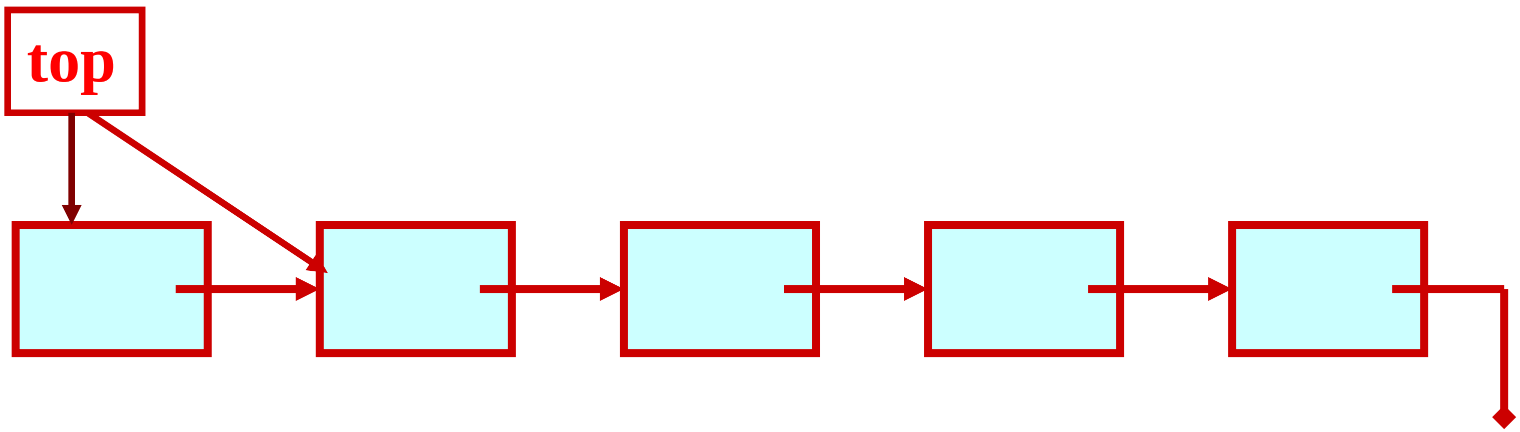
Stack: Linked List Structure

PUSH OPERATION



Stack: Linked List Structure

POP OPERATION



Basic Idea

- In the array implementation, we would:
  - Declare an array of fixed size (which determines the maximum size of the stack).
  - Keep a variable which always points to the "top" of the stack.
    - Contains the array index of the "top" element.
- In the linked list implementation, we would:
  - Maintain the stack as a linked list.
  - A pointer variable `top` points to the start of the list.
  - The first element of the linked list is considered as the stack top.

Declaration

```
#define MAXSIZE 100

struct lifo
{
   int st[MAXSIZE];
   int  top;
};
typedef struct lifo
                stack;
stack s;
```

ARRAY

```
struct lifo
{
   int value;
   struct lifo *next;
};
typedef struct lifo
                stack;

stack *top;
```

LINKED LIST

Stack Creation

```
void create (stack *s)
{
    s->top = -1;

    /* s->top points to
       last element
       pushed in;
       initially -1 */
}
```

ARRAY

```
void create (stack **top)
{
    *top = NULL;

    /* top points to NULL,
       indicating empty
       stack           */
}
```

LINKED LIST

Pushing an element into the stack

```
void push (stack *s, int element)
  {
      if (s->top == (MAXSIZE-1))
      {
          printf ("\n Stack overflow");
          exit(-1);
      }
      else
      {
          s->top ++;
          s->st[s->top] = element;
      }
  }
```

ARRAY

```
void push (stack **top, int element)
{
    stack *new;

    new = (stack *) malloc(sizeof(stack));
    if (new == NULL)
    {
       printf ("\n Stack is full");
       exit(-1);
    }

    new->value = element;
    new->next = *top;
    *top = new;
}
```

LINKED LIST

Popping an element from the stack

```
int pop (stack *s)
  {
     if (s->top == -1)
     {
        printf ("\n Stack underflow");
        exit(-1);
     }
     else
     {
        return (s->st[s->top--]);
     }
  }
```

ARRAY

```
int pop (stack **top)
{
    int t;
    stack *p;

    if (*top == NULL)
    {
        printf ("\n Stack is empty");
        exit(-1);
    }
    else
    {
        t = (*top)->value;
        p = *top;
        *top = (*top)->next;
        free (p);
        return t;
    }
}
```

LINKED LIST

Checking for stack empty

```
int isempty (stack *s)
{
   if (s->top == -1)
          return 1;
   else
          return (0);
}
```

ARRAY

```
int isempty (stack *top)
{
   if (top == NULL)
        return (1);
    else
        return (0);
}
```

LINKED LIST

Checking for stack full

```
int isfull (stack *s)
{
   if (s->top ==
           (MAXSIZE-1))
        return 1;
   else
        return (0);
}
```

ARRAY

- Not required for linked list implementation.
- In the `push()` function, we can check the return value of `malloc()`.
  - If -1, then memory cannot be allocated.

LINKED LIST

Example main function :: array

```
#include <stdio.h>
#define MAXSIZE 100

struct lifo
{
   int st[MAXSIZE];
   int  top;
};
typedef struct lifo stack;
```

```
main()
{
  stack A, B;
  create(&A);  create(&B);
  push(&A,10);
  push(&A,20);
```

```
  push(&A,30);
  push(&B,100);  push(&B,5);

  printf ("%d %d", pop(&A),
              pop(&B));

  push (&A, pop(&B));

  if (isempty(&B))
    printf ("\n B is empty");
}
```

Example main function :: linked list

```
#include <stdio.h>
struct lifo
{
    int value;
    struct lifo *next;
};
typedef struct lifo stack;
```

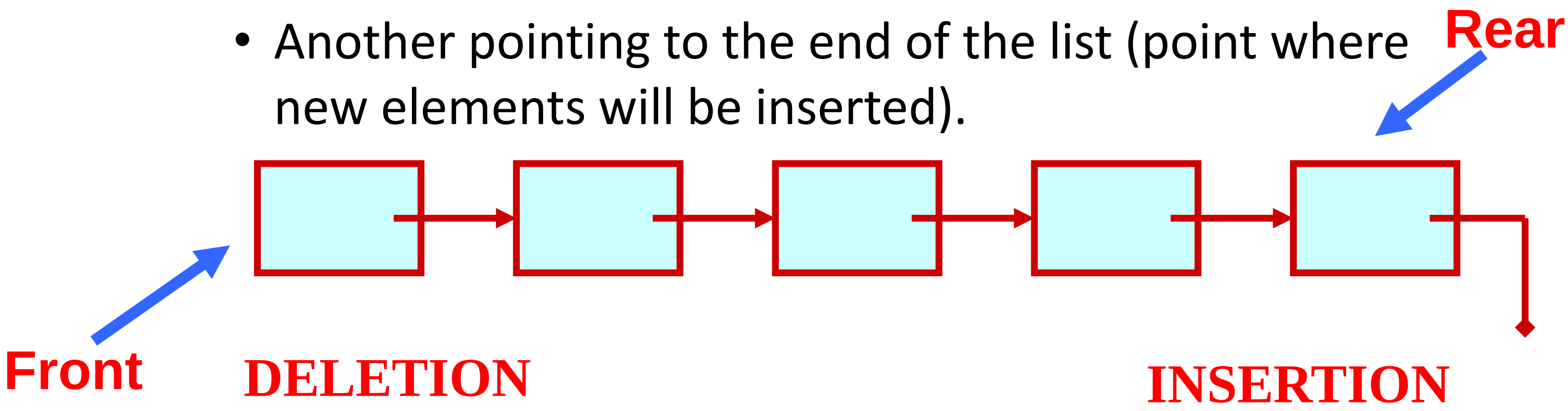
```
main()
{
  stack *A, *B;
  create(&A); create(&B);
  push(&A,10);
  push(&A,20);
```

```
  push(&A,30);
  push(&B,100);
  push(&B,5);

  printf (“%d %d”,
        pop(&A), pop(&B));

  push (&A, pop(&B));

  if (isempty(B))
    printf (“\n B is
  empty”);
}
```

Queue Implementation using Linked List

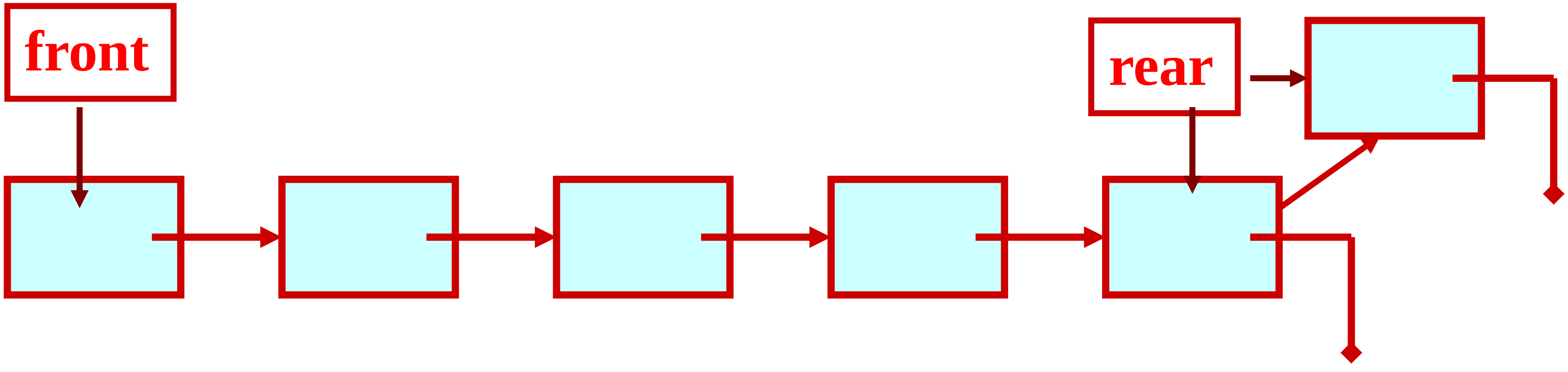
Basic Idea

- Basic idea:
  – Create a linked list to which items would be added to one end and deleted from the other end.
  – Two pointers will be maintained:
    - One pointing to the beginning of the list (point from where elements will be deleted).
    - Another pointing to the end of the list (point where new elements will be inserted).



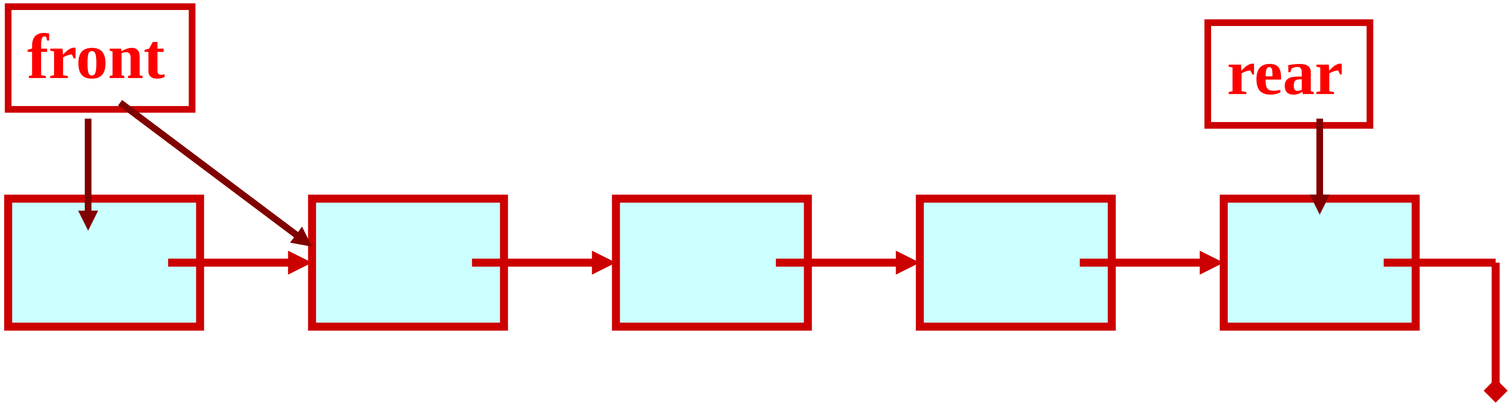
QUEUE: LINKED LIST STRUCTURE

ENQUEUE



QUEUE: LINKED LIST STRUCTURE

DEQUEUE



QUEUE using Linked List

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

struct node{
        char name[30];
        struct node *next;
    };

typedef struct node _QNODE;

typedef struct {
    _QNODE *queue_front, *queue_rear;
    } _QUEUE;
```

```
_QNODE *enqueue (_QUEUE *q, char x[])
{
_QNODE *temp;
temp= (_QNODE *)
        malloc (sizeof(_QNODE));
if (temp==NULL){
printf("Bad allocation \n");
return NULL;
}
strcpy(temp->name,x);
temp->next=NULL;
```

```
if(q->queue_rear==NULL)
{
q->queue_rear=temp;
q->queue_front=
     q->queue_rear;
}
else
{
q->queue_rear->next=temp;
q->queue_rear=temp;
}
return(q->queue_rear);
}
```

```
char  *dequeue(_QUEUE *q,char x[])
{
_QNODE *temp_pnt;

if(q->queue_front==NULL){
q->queue_rear=NULL;
printf("Queue is empty \n");
return(NULL);
}
```

```
else{
strcpy(x,q->queue_front->name);
temp_pnt=q->queue_front;
q->queue_front=
        q->queue_front->next;
free(temp_pnt);
if(q->queue_front==NULL)
q->queue_rear=NULL;
return(x);
  }
}
```

```
void init_queue(_QUEUE *q)
{
 q->queue_front= q->queue_rear=NULL;
}

int isEmpty(_QUEUE *q)
{
 if(q==NULL) return 1;
 else return 0;
}
```

```
main()
{
int i,j;
char command[5],val[30];
_QUEUE q;

 init_queue(&q);

command[0]='\0';
printf("For entering a name use 'enter <name>'\n");
printf("For  deleting  use 'delete' \n");
printf("To end the session use 'bye' \n");
while(strcmp(command,"bye")){
scanf(" %s",command);
```

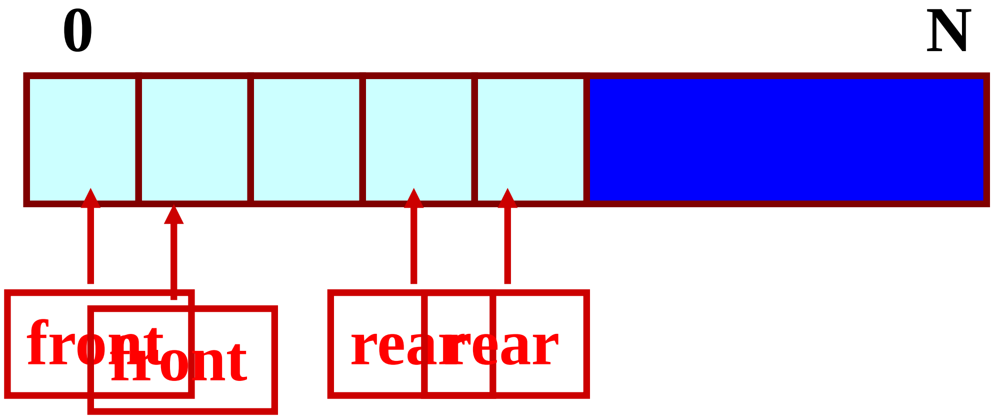
```c
if(!strcmp(command,"enter")) {
scanf("%s",val);
if((enqueue(&q,val)==NULL))
printf("No more pushing please \n");
else printf("Name entered %s \n",val);
}
```

```c
if(!strcmp(command,"delete")) {
if(!isEmpty(&q))
printf("%s \n",dequeue(&q,val));
else printf("Name deleted %s \n",val);
}
} /* while */
printf("End session \n");
}
```

Problem With Array Implementation

ENQUEUE DEQUEUE

Effective queuing storage area of array gets reduced.



Use of circular array indexing

Evaluation of Expressions

X = a / b - c + d * e - a * c

a = 4, b = c = 2, d = e = 3

Interpretation 1:
((4/2)-2)+(3*3)-(4*2)=0 + 8+9=1

Interpretation 2:
(4/(2-2+3))*(3-4)*2=(4/3)*(-1)*2=-2.66666…

How to generate the machine instructions corresponding to a given expression?

precedence rule + associative rule

Token	Operator	Precedence[1]	Associativity
( ) [ ] -> .	function call array element struct or union member	17	left-to-right
-- ++	increment, decrement[2]	16	left-to-right
-- ++ ! - - + & * sizeof	decrement, increment[3] logical not one’s complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	mutiplicative	13	Left-to-right

+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
|	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
||	logical or	4	left-to-right

?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^= ☒	assignment	2	right-to-left
,	comma	1	left-to-right

1.The precedence column is taken from Harbison and Steele.
2.Postfix form
3.prefix form

Figure: Precedence hierarchy for C

user **compiler**

Infix	Postfix
2+3*4	234*+
a*b+5	ab*5+
(1+2)*7	12+7*
a*b/c	ab*c/
(a/(b-c+d))*(e-a)*c	abc-d+/ea-*c*
a/b-c+d*e-a*c	ab/c-de*ac*-

Figure: Infix and postfix notation

Postfix: no parentheses, no precedence

Token	Stack [0]	[1]	[2]	Top
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
-	6/2-3			0
4	6/2-3	4		1
2	6/2-3	4	2	2
*	6/2-3	4*2		1
+	6/2-3+4*2			0

Figure: Postfix evaluation

Goal: infix --> postfix

Assumptions:

operators: +, -, *, /, %

operands: single digit integer

```
#define MAX_STACK_SIZE 100 /* maximum stack size */
#define MAX_EXPR_SIZE 100 /* max size of expression */
typedef enum{1paran, rparen, plus, minus, times, divide,
                        mod, eos, operand} precedence;
int stack[MAX_STACK_SIZE]; /* global stack */
char expr[MAX_EXPR_SIZE]; /* input string */
```

```
int eval(void)
{
/* evaluate a postfix expression, expr, maintained as a
   global variable, '\0' is the the end of the expression.
   The stack and top of the stack are global variables.
   get_token is used to return the token type and
   the character symbol. Operands are assumed to be single
   character digits */
  precedence token;
  char symbol;
  int op1, op2;
  int n = 0;  /* counter for the expression string */
  int top = -1;
  token = get_token(&symbol, &n);
  while (token != eos)  {
    if (token == operand)
        add(&top, symbol-'0');   /* stack insert */
```

exp: character array

```
else {
     /* remove two operands, perform operation, and
        return result to the stack */
  op2 = delete(&top);  /* stack delete */
  op1 = delete(&top);
  switch(token) {
     case plus: add(&top, op1+op2); break;
     case minus: add(&top, op1-op2); break;
     case times: add(&top, op1*op2); break;
     case divide: add(&top, op1/op2); break;
     case mod: add(&top, op1%op2);
  }
}
token = get_token (&symbol, &n);
}
return delete(&top); /* return result */
}
```

***Program 3.9:** Function to evaluate a postfix expression (p.122)

```
precedence get_token(char *symbol, int *n)
{
/* get the next token, symbol is the character
   representation, which is returned, the token is
   represented by its enumerated value, which
   is returned in the function name */

  *symbol =expr[(*n)++];
  switch (*symbol)  {
    case '(' : return lparen;
    case ')' : return rparen;
    case '+': return plus;
    case '-' : return minus;
```

```
    case '/' :  return divide;
    case '*' : return times;
    case '%' : return mod;
    case '\0' : return eos;
    default  : return operand;
               /* no error checking, default is operand */
    }
}
```

***Program 3.10:** Function to get a token from the input string (p.123)

Infix to Postfix Conversion
(Intuitive Algorithm)

(1) Fully parenthesize expression

a / b - c + d * e - a * c -->

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

(2) All operators replace their corresponding right parentheses.

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

/ - * + * -

(3) Delete all parentheses.

ab/c-de*+ac*-

two passes

The orders of operands in infix and postfix are the same.

a + b * c, * > +

Token	Stack			Top	Output
	[0]	[1]	[2]		
a				-1	a
+	+			0	a
b	+			0	ab
*	+	*		1	ab
c	+	*		1	abc
eos				-1	abc*=

Figure: Translation of a+b*c to postfix

$a *_1 (b + c) *_2 d$

Token	Stack [0]	Stack [1]	Stack [2]	Top	Output
a				-1	a
$*_1$	$*_1$			0	a
(	$*_1$	(		1	a
b	$*_1$	(		1	ab
+	$*_1$	(	+	2	ab
c	$*_1$	(	+	2	abc
)	$*_1$	match)		0	abc+
$*_2$	$*_2$	$*_1 = *_2$		0	abc+$*_1$
d	$*_2$			0	abc+$*_1$d
eos	$*_2$			0	abc+$*_1$d$*_2$

Figure: Translation of a*(b+c)*d to postfix

Rules

(1) Operators are taken out of the stack as long as their in-stack precedence is higher than or equal to the incoming precedence of the new operator.

(2) (has low in-stack precedence, and high incoming precedence.

	(	)	+	-	*	/	%	eos
isp	0	19	12	12	13	13	13	0
icp	20	19	12	12	13	13	13	0

```
precedence stack[MAX_STACK_SIZE];
/* isp and icp arrays -- index is value of precedence
lparen, rparen, plus, minus, times, divide, mod, eos */
static int isp [ ] = {0, 19, 12, 12, 13, 13, 13, 0};
static int icp [ ] = {20, 19, 12, 12, 13, 13, 13, 0};
```

isp: in-stack precedence
icp: incoming precedence

```
void postfix(void)
{
/* output the postfix of the expression. The expression
    string, the stack, and top are global */
   char symbol;
   precedence token;
   int n = 0;
   int top = 0; /* place eos on stack */
   stack[0] = eos;
   for (token = get _token(&symbol, &n); token != eos;
                     token = get_token(&symbol, &n)) {
     if (token == operand)
        printf ("%c", symbol);
     else if (token == rparen ){
```

```
        /*unstack tokens until left parenthesis */
        while (stack[top] != lparen)
            print_token(delete(&top));
        delete(&top); /*discard the left parenthesis */
       }
       else{
        /* remove and print symbols whose isp is greater
            than or equal to the current token's icp */
        while(isp[stack[top]] >= icp[token] )
            print_token(delete(&top));
        add(&top, token);
       }
   }
   while ((token = delete(&top)) != eos)
       print_token(token);
   print("\n");
}
```

$\theta(n)$

f(n)=θ(g(n)) iff there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) \le f(n) \le c_2 g(n)$ for all n, $n \ge n_0$.

f(n)=θ(g(n)) iff g(n) is both an upper and lower bound on f(n).

Function to convert from infix to postfix

Infix and postfix expressions

Infix	Prefix
a*b/c a/b-c+d*e-a*c a*(b+c)/d-g	/*abc -+-/abc*de*ac -/*a+bcdg

(1) evaluation

(2) transformation