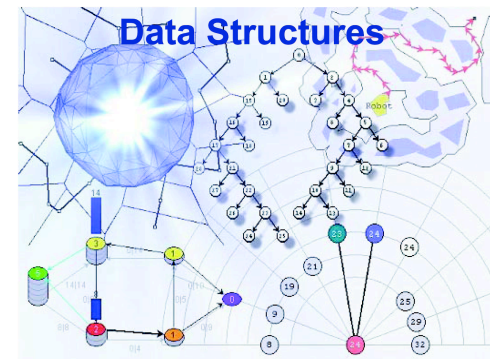
BBM 201
DATA STRUCTURES

Lecture 9:

Implementation of Linked Lists

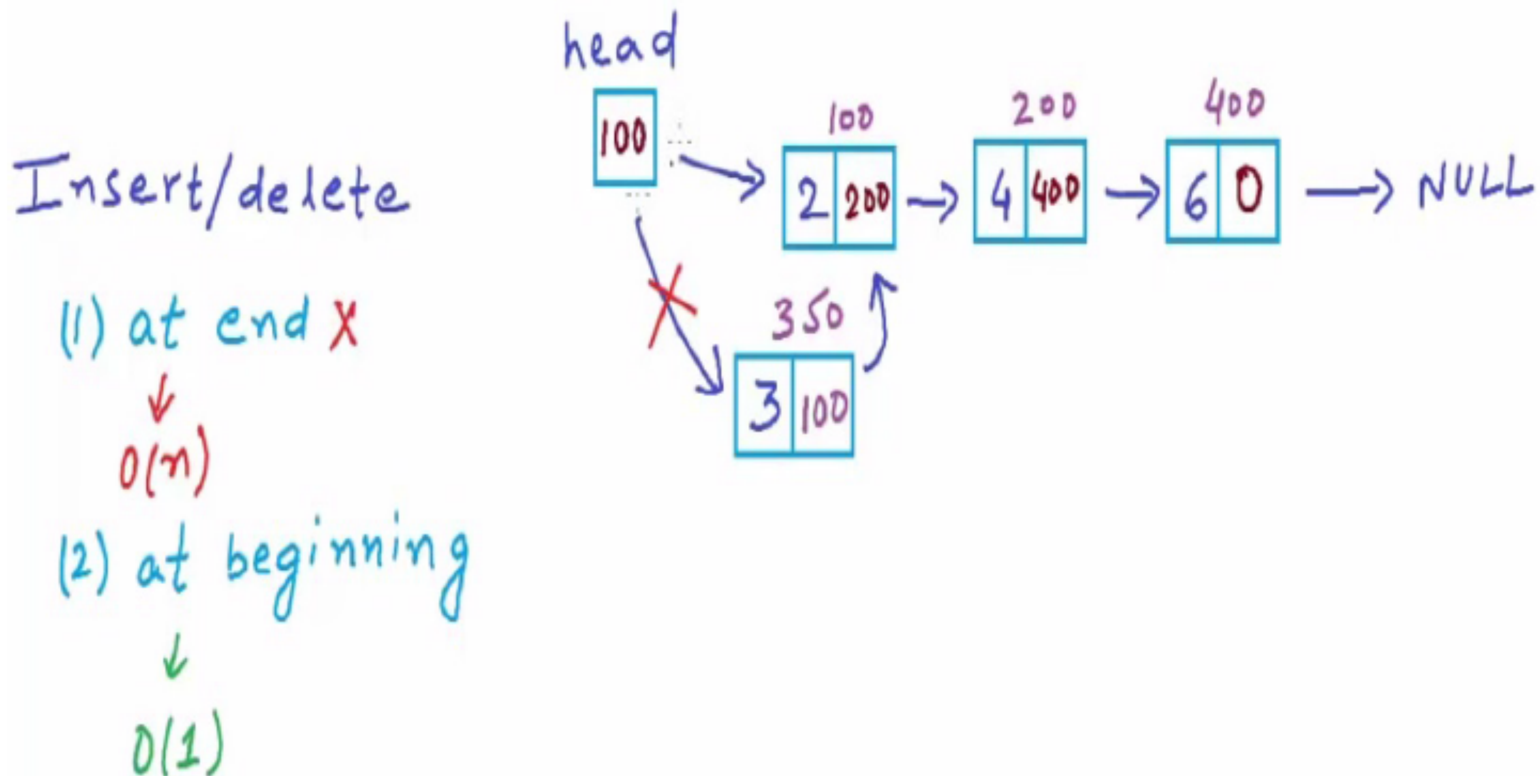
(Stacks, Queue, Hashtable)



2018-2019 Fall

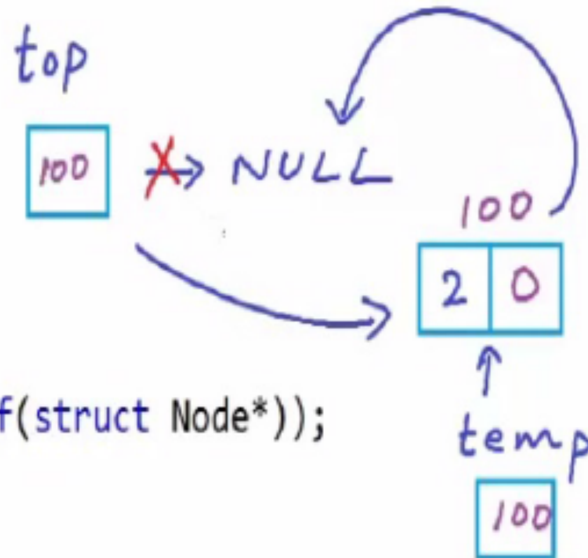
Linked list implementation of stacks

The cost of insert and delete at the beginning of a linked list is constant, that is O(1).
So, in the linked list implementation of stack, we insert and delete a node at the beginning so that time complexity of Push and Pop is O(1).



Instead of head, we use variable name top. When top is NULL, the stack is empty.
With dereferencing temp->data = x and temp->link = top, we fill the fields of the new node, called temp. Finally, with top=temp, top points to the new node.

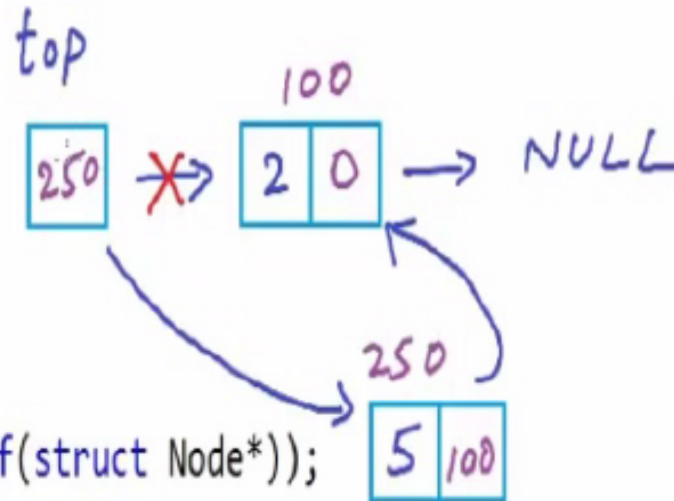
```
struct Node {
    int data;
    struct Node* link;
};
struct Node* top = NULL;
void Push(int x) {
    struct Node* temp =
      (struct Node*)malloc(sizeof(struct Node*));
    temp->data = x;
    temp->link = top;
    top = temp;
}
```



Push(2)

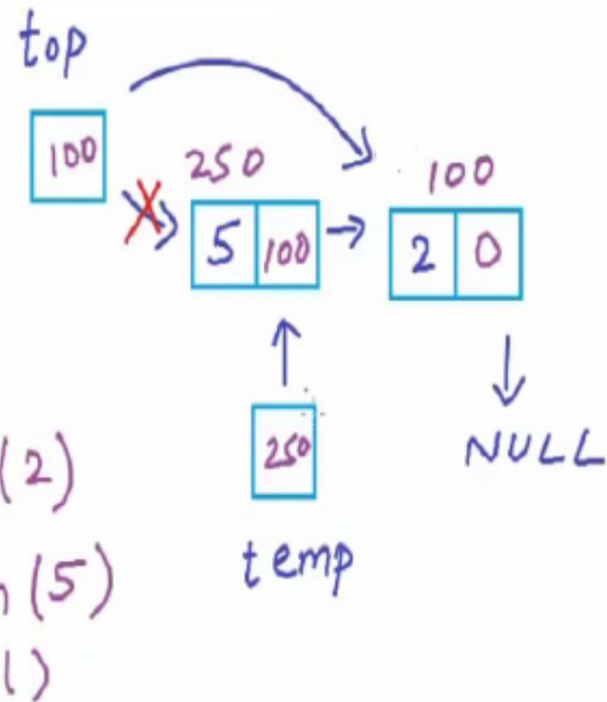
Stack data structure uses the memory efficiently by using push when something is needed and pop when not needed in an array or linked list.

```c
struct Node {
    int data;
    struct Node* link;
};
struct Node* top = NULL;
void Push(int x) {
    struct Node* temp =
      (struct Node*)malloc(sizeof(struct Node*));
    temp->data = x;
    temp->link = top;
    top = temp;
}
```



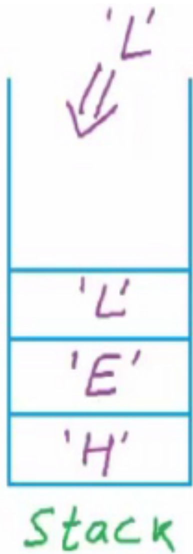
Push(2)
Push(5)

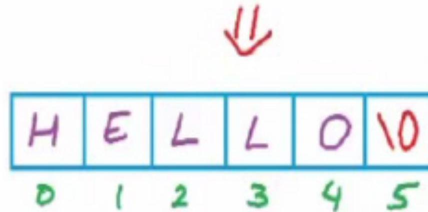
```c
struct Node* top = NULL;
void Push(int x) {
    struct Node* temp =
      (struct Node*)malloc(sizeof(struct Node*));
    temp->data = x;
    temp->link = top;
    top = temp;
}
void Pop() {
    struct Node *temp;
    if(top == NULL) return;
    temp = top;
    top = top->link;
    free(temp);
}
```

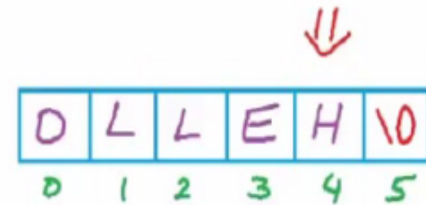


Reverse a string using stack

We can create a stack of characters by traversing the string from left to right and using the push operation.
Then, we can pop the elements from the stack and overwrite all the positions in the string.







We pass the string and the length of the string to the Reverse function.

```
// string reversal using stack
#include<iostream>
using namespace std;

void Reverse(char C[],int n)
{

}
int main() {
    char C[51];
    printf("Enter a string: ");
    gets(C);
    Reverse(C,strlen(C));
    printf("Output = %s",C);
}
```

In C++, we can use these operations available from the standard temporary library called stack.

```
// string reversal using stack
#include<iostream>
using namespace std;
class Stack
{
private:
    char A[101];
    int top;
public:
    void Push(int x);
    void Pop();
    int Top();
    bool IsEmpty();
};
```

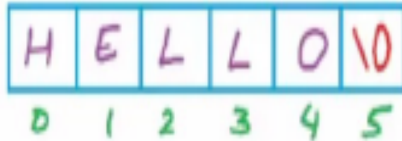
As we traverse the string from left to right, we use the push function. In the second loop, we use the top and pop functions for each character in the stack to reverse the string C. See next slide for output.

```
#include<iostream>
#include<stack> // stack from standard template library (STL)
using namespace std;
void Reverse(char *C,int n)
{
    stack<char> S;
    //loop for push
    for(int i=0;i<n;i++){
        S.push(C[i]);
    }
    //loop for pop
    for(int i =0;i<n;i++){
        C[i] = S.top(); //overwrite the character at index i.
        S.pop();  // perform pop.
    }
}
```

```
#include<stack> // stack from standard template library (STL)
using namespace std;
void Reverse(char *C,int n)
{
    stack<char> S;
    //loop for push
    for(int i=0;i<n;i++){
        S.push(C[i]);
    }
    //loop for pop
    for(int i =0;i<n;i++){
        C[i] = S.top(); //overwrite the character at index i.
        S.pop();  // perform pop.
    }
}
int main() {
    char C[51];
    printf("Enter a string: ");
    gets(C);
    Reverse(C,strlen(C));
    printf("Output = %s",C);
}
```

```
C:\Users\animesh\Documents\Visual Studio 2010\Projects\SampleApp2\Debug\S
Enter a string: HELLO
Output = OLLEH
```

Problem: Reverse a string



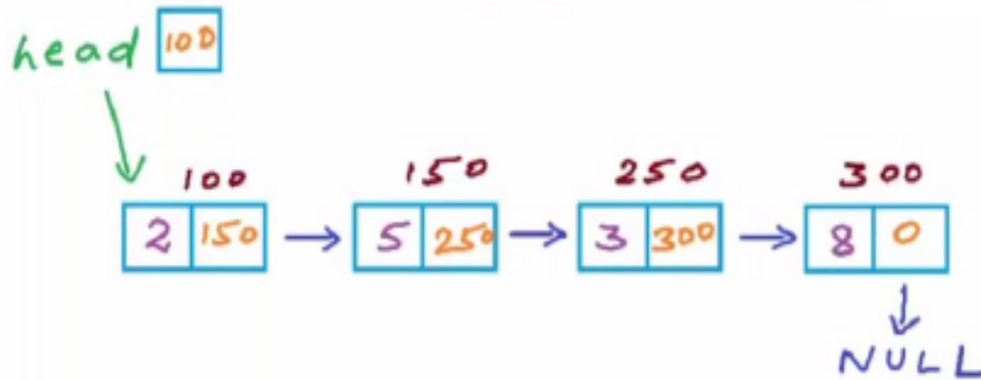

```
void Reverse(char *C,int n)
{
    stack<char> S;
    //loop for push
    for(int i=0;i<n;i++){
        S.push(C[i]);
    }
    //loop for pop
    for(int i =0;i<n;i++){
        C[i] = S.top();
        S.pop();
    }
}
```

O(n) (push loop), O(n) (pop loop)

- All operations on stack take constant time, O(1).
- Thus, for each loop, time cost is O(n).
- Space complexity is also O(n).

Reverse a linked list using stack

Time complexity of reversing a linked list



Iterative Solution

Time - O(n)

Space - O(1)

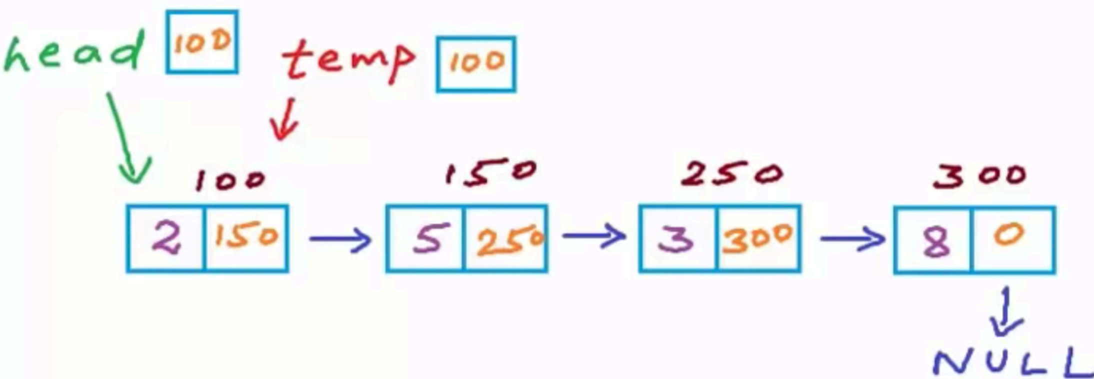
Recursive Solution

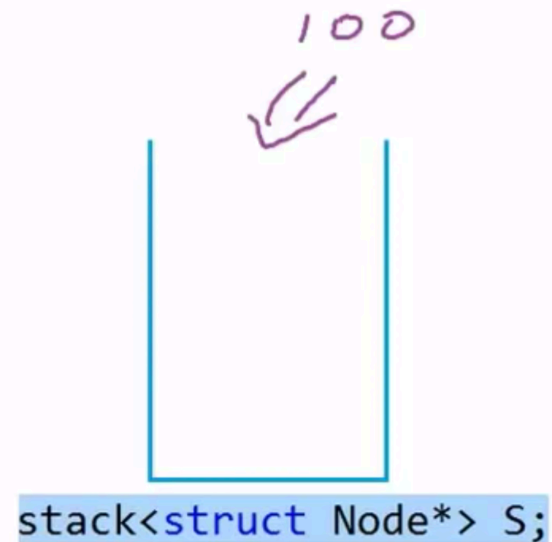
(Implicit Stack)

Time - O(n)

Space - O(n)

Time complexity of reversing a linked list



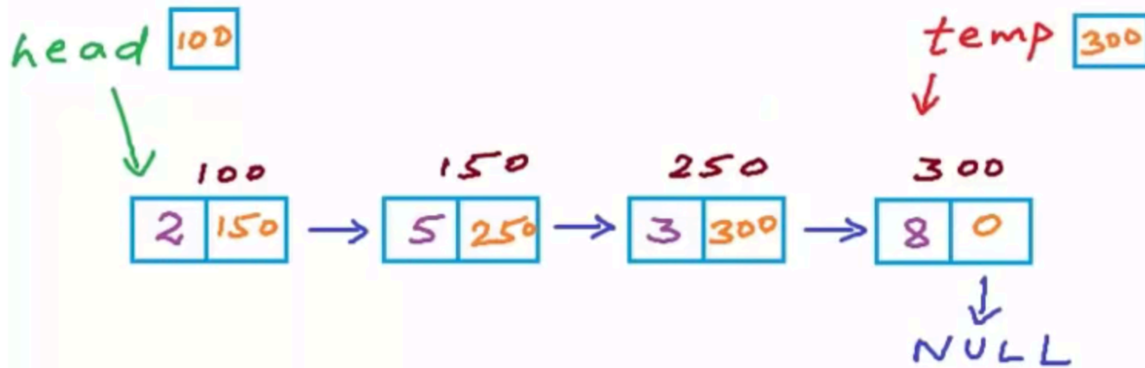


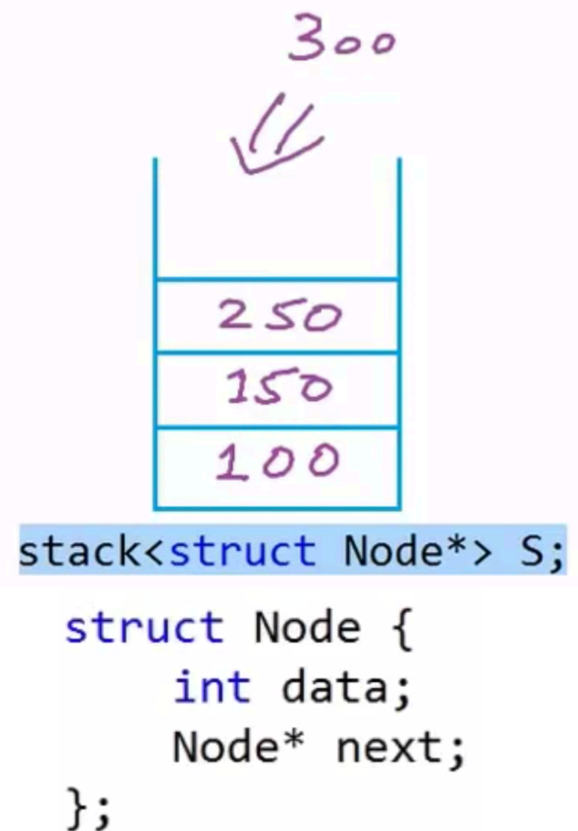

```
struct Node {
    int data;
    Node* next;
};
```

```
Node* temp = head;
while(temp != NULL) {
    S.push(temp);
    temp= temp->next;
}
```

The values we are pushing into the stack are the pointers (references) to the node.

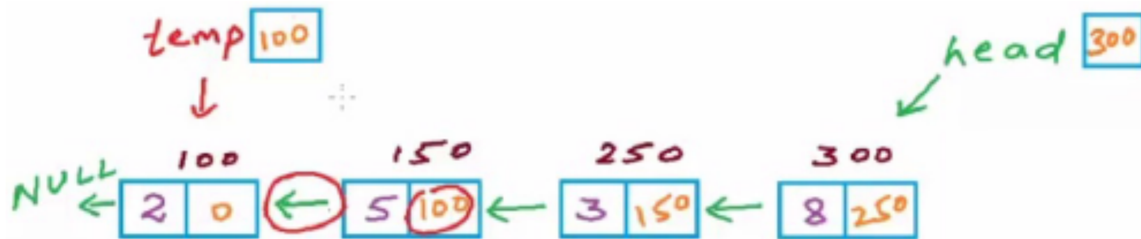
Time complexity of reversing a linked list



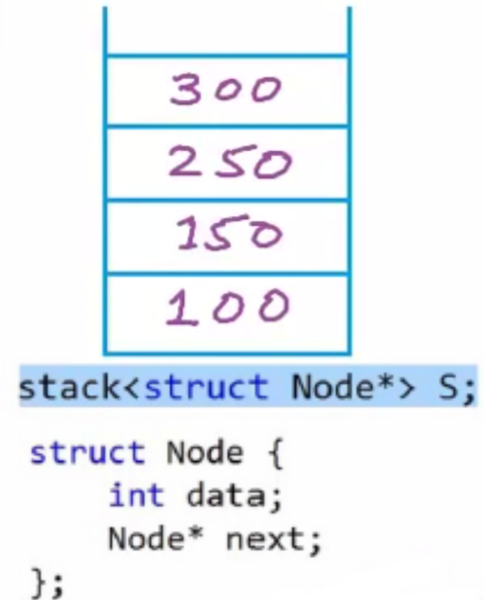



```
Node* temp = head;
while(temp != NULL) {
    S.push(temp);
    temp= temp->next;
}
```

Reverse a linked list using stack

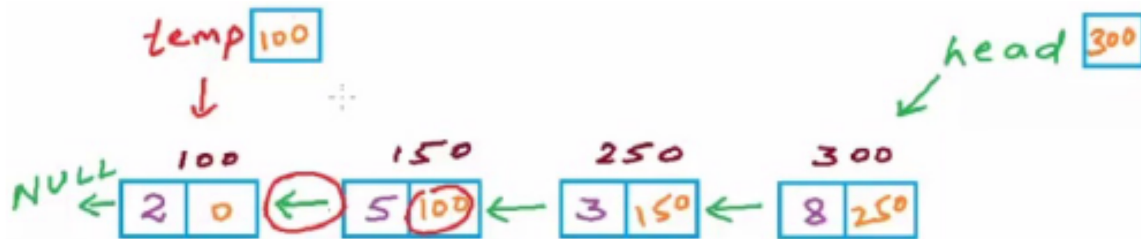



```
Node *temp = S.top();
head = temp;
S.pop();
while(!S.empty()){
    temp->next = S.top();
    S.pop();
    temp = temp->next;
}
temp->next = NULL;
```



Once all the references are pushed onto the stack, we can start popping them. As we pop them, we will get references to nodes in reverse order. While traversing the list in reverse order, we can build reversed links.

Reverse a linked list using stack




```
Node *temp = S.top();
head = temp;
S.pop();
while(!S.empty()){
    temp->next = S.top();
    S.pop();
    temp = temp->next;
}
temp->next = NULL;
```

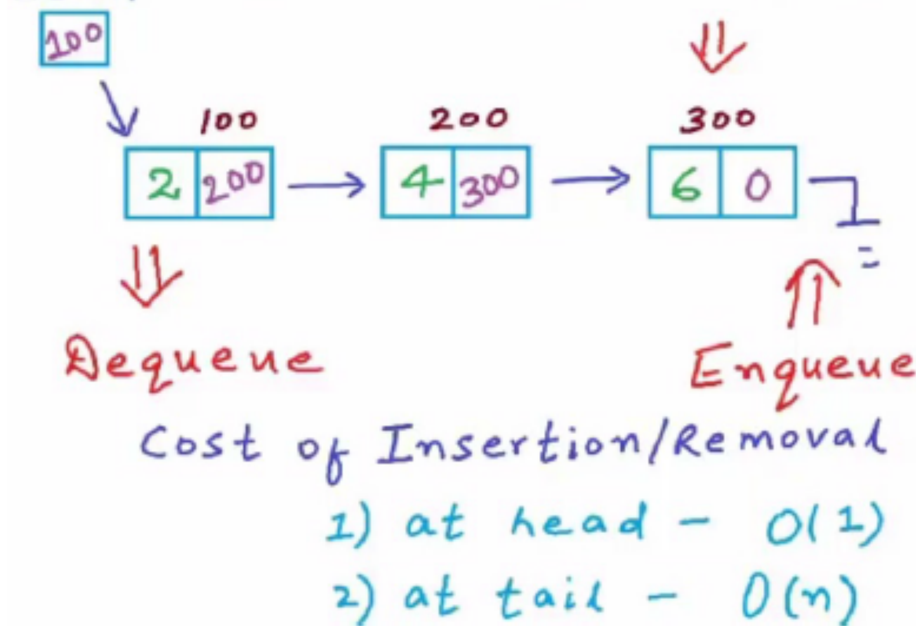
300
250
150
100

```
stack<struct Node*> S;

struct Node {
    int data;
    Node* next;
};
```

Initially, S.top() returns 300. We store head as this address, so head stores 300.
In the loop, while stack is not empty, set temp->next to the address at the top of the stack, which is 250 at first.
Finally, temp points to the node at address 100.
The last line is needed so that the end of the reversed linked list points to NULL and not to 150.

Linked list implementation of queue

Queue - Linked List implementation



Operations

(1) EnQueue(x)
(2) Dequeue()
(3) front()
(4) IsEmpty()

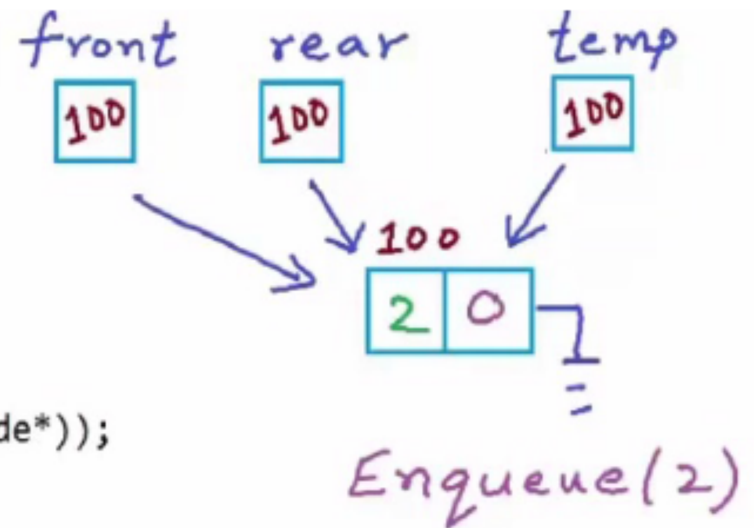
constant time or O(1)

In this case above, Dequeue will take constant time, but Enqueue will take O(n) time.

As before, the traversal of the linked list to learn the address of the last node takes O(n) time.

Solution: Keep a pointer called rear that stores the address of the node at the rear position.

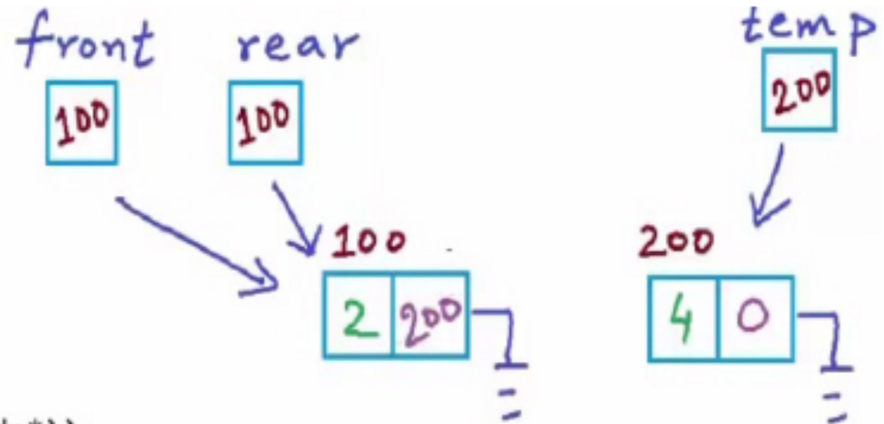
```
struct Node {
    int data;
    struct Node* next;
};
struct Node* front = NULL;
struct Node* rear = NULL;
void Enqueue(int x) {
    struct Node* temp =
        (struct Node*)malloc(sizeof(struct Node*));
    temp->data =x;
    temp->next = NULL;
    if(front == NULL && rear == NULL){
        front = rear = temp;
        return;
    }
    rear->next = temp;
    rear = temp;
}
```



Instead of declaring a pointer variable pointing to head, we declare two pointer variables one pointing to the front node and another pointing to the rear node.

For the Enqueue, we again use a pointer called temp to point to the new node created dynamically.
If the list is empty, both front and rear point to the new node.

```
struct Node {
    int data;
    struct Node* next;
};
struct Node* front = NULL;
struct Node* rear = NULL;
void Enqueue(int x) {
    struct Node* temp =
        (struct Node*)malloc(sizeof(struct Node*));
    temp->data =x;
    temp->next = NULL;
    if(front == NULL && rear == NULL){
        front = rear = temp;
        return;
    }
    rear->next = temp;
    rear = temp;
}
```



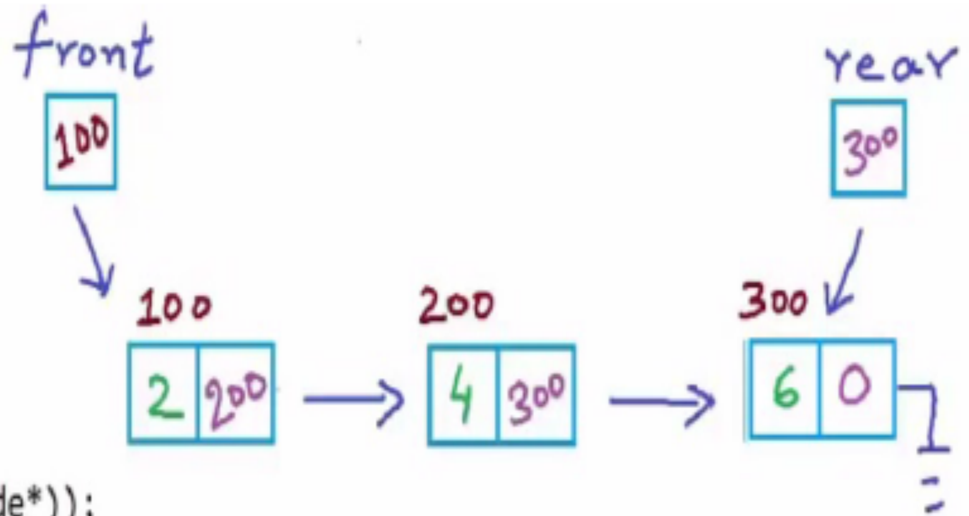
Enqueue(2)

Enqueue(4)

If the list is not empty, first, set the address part of the current rear node to the address of the new node.

Then let the rear pointer point to the new node.

```
struct Node {
    int data;
    struct Node* next;
};
struct Node* front = NULL;
struct Node* rear = NULL;
void Enqueue(int x) {
    struct Node* temp =
        (struct Node*)malloc(sizeof(struct Node*));
    temp->data =x;
    temp->next = NULL;
    if(front == NULL && rear == NULL){
        front = rear = temp;
        return;
    }
    rear->next = temp;
    rear = temp;
}
```



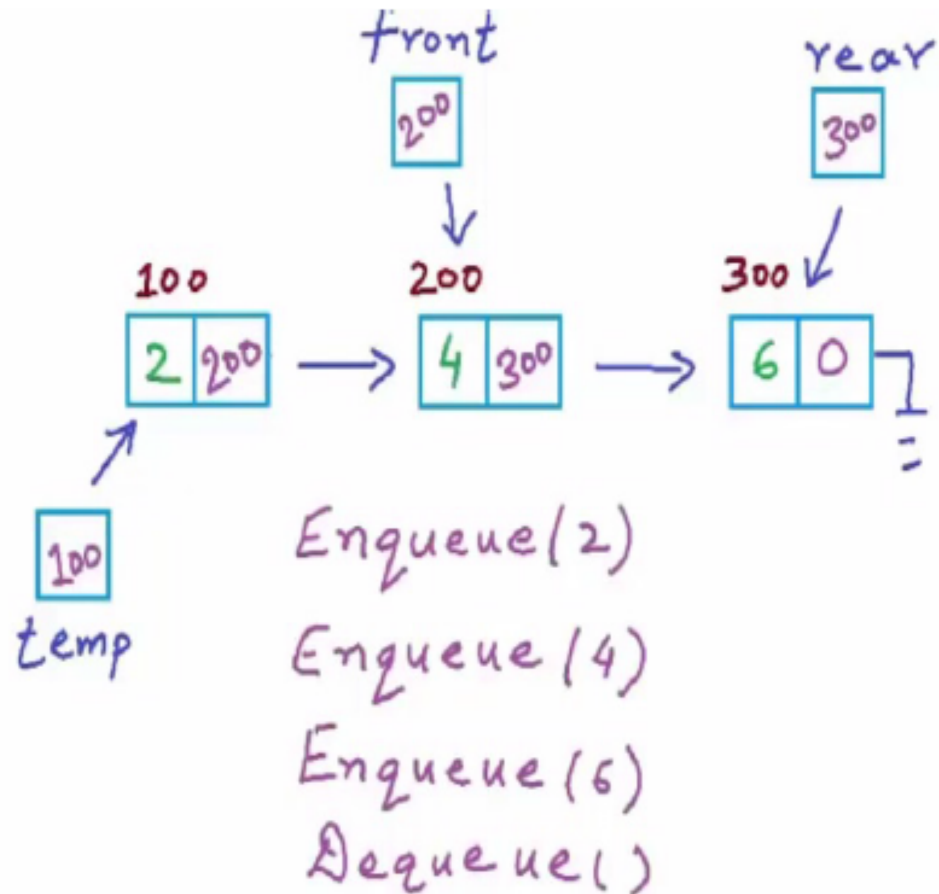
Enqueue(2)

Enqueue(4)

Enqueue(6)

Implementation of Dequeue

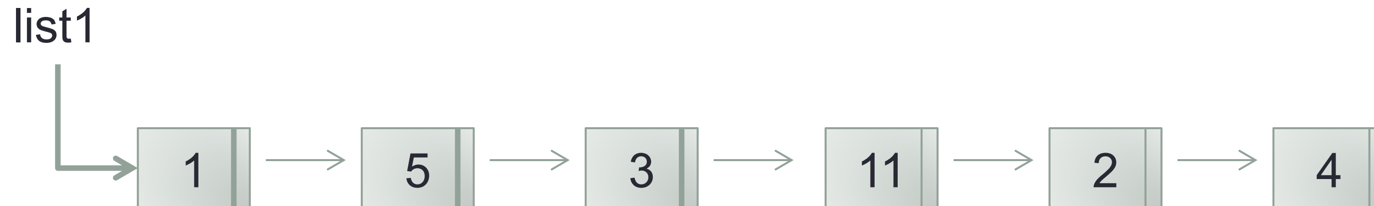
```
void Dequeue() {
    struct Node* temp = front;
    if(front == NULL) return;
    if(front == rear) {
        front = rear = NULL;
    }
    else {
        front = front->next;
    }
    free(temp);
}
```



Multiple Stack Implementation Using Linked List

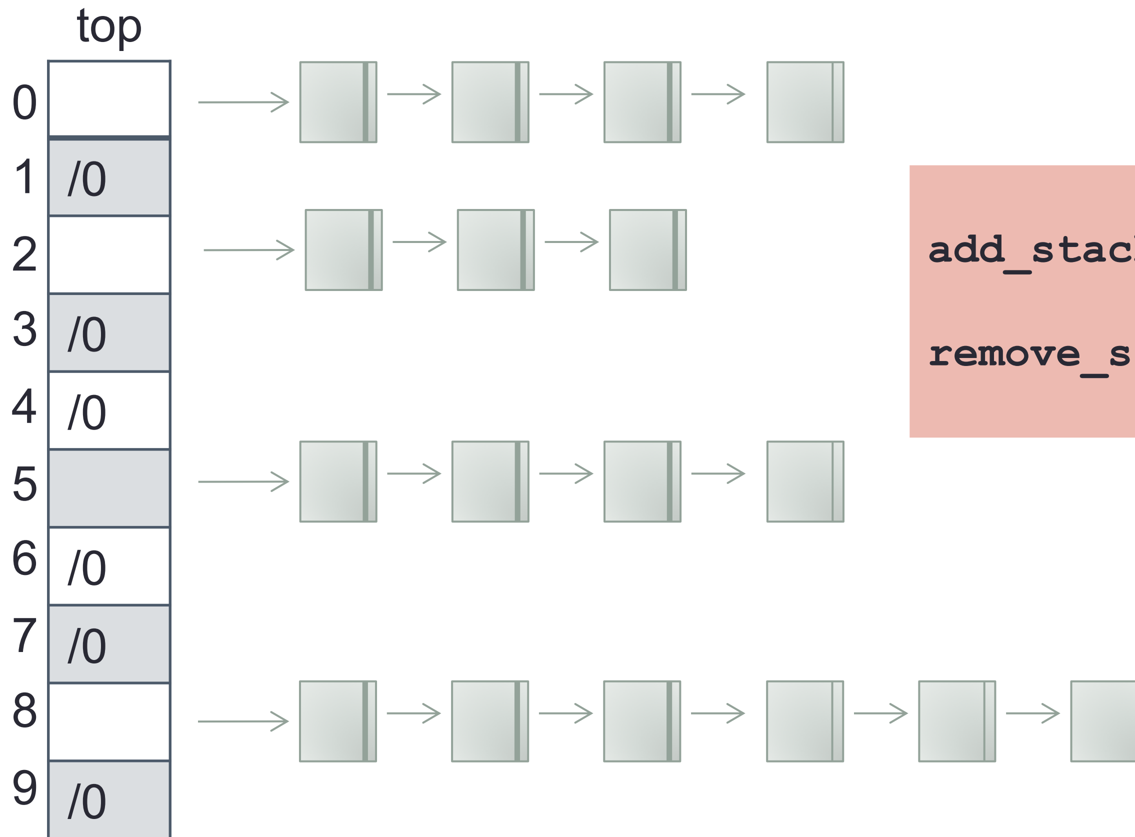
Multiple Stack Implementation Using Linked List

A single stack:



Multiple Stack Implementation Using Linked List

Multiple stacks:




```
add_stack(&top[stack_i], x);

remove_stack(&top[stack_i]);
```

Linked List

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_STACK 10 // number of max stacks

typedef struct{
       int  value;
       //other fields
}item;

typedef struct stack{
       item x;
       stack * next;
};

typedef struct stack *stack_pointer;

stack_pointer top [MAX_STACK];
```

Linked List

```
#define ISFULL(p) (!p)
#define ISEMPTY(p) (!p)

int main()
{
    for(int i=0;i<MAX_STACK;i++)
        top[i] = NULL;

    return 0;
}
```

Linked List

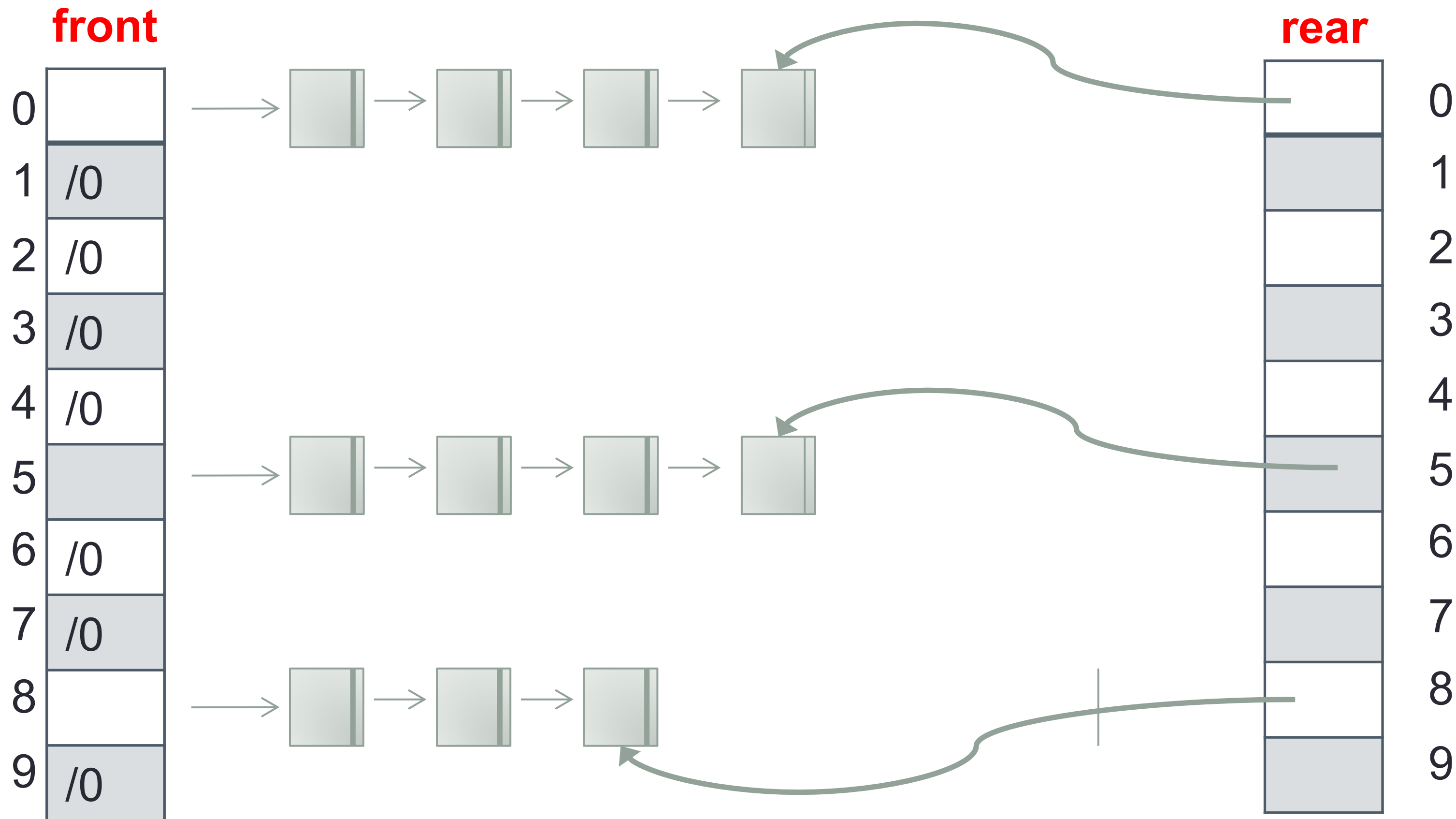
```
void add_stack(stack_pointer *top, item x)
{
        stack_pointer p = new stack;
        if(ISFULL(p))
        {
                printf("\n Memory allocation failed");
                exit(1);
        }
        p->x = x;
        p->next = *top;
        *top = p;
}
```

Linked List

```
item remove_stack(stack_pointer *top)
{
        stack_pointer p = *top;
        item x;

        if(ISEMPTY(p))
        {
                printf("\n Stack is empty!");
                exit(1);
        }
        x = p->x;
        *top = p->next;
        free(p);
        return(x);
}
```

Multiple Queue Implementation Using Linked List

Multiple Queue Implementation Using Linked List



Linked List

```
#define MAX_QUEUE 10 // number of max queues

typedef struct{
       int  value;
       //other fields
}item;

typedef struct queue{
       item  x;
       queue * link;
};

typedef struct queue *queue_pointer;

queue_pointer front[MAX_QUEUE],rear[MAX_QUEUE];

#define ISFULL(p) (!p)
#define ISEMPTY(p) (!p)

for(int i=0;i<MAX_QUEUE;i++)
      front[i]=NULL;
```

Linked List

```
void add_queue(queue_pointer *front, queue_pointer* rear, item x)
{
     queue_pointer p = new queue;
     if(ISFULL(p)){
         printf("\n Memory allocation failed!!!\n");
         exit(1);
     }
  p->x = x;
  p->link = NULL;
  if(*front)
     (*rear)->link = p;
  else
      *front = p;
  *rear = p;
}
```

Linked List

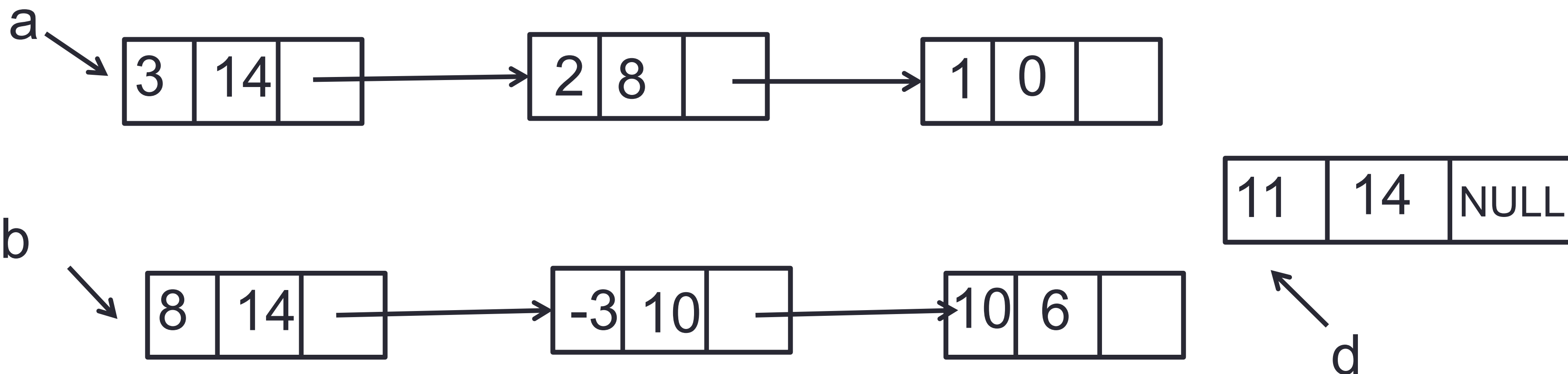
```
item remove_queue(queue_pointer *front)
{
     queue_pointer p = *front;
     item x;
     if(ISEMPTY(*front)){
          printf("\nQueue is empty!!!");
          exit(1);
     }

   x = p->x;
   *front = p->link;
   free(p);
   return(x);
}
```

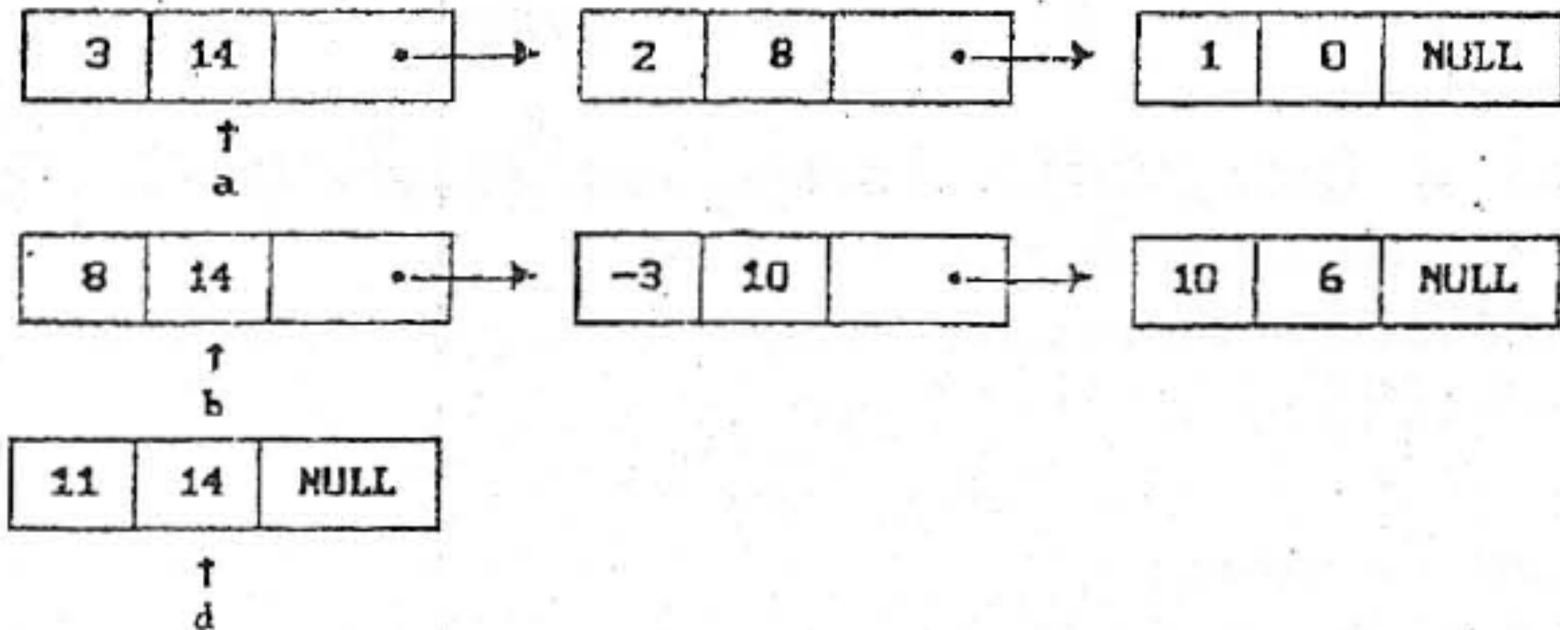
Addition of polynomials by using singly linked lists

Adding Polynomials using linked list

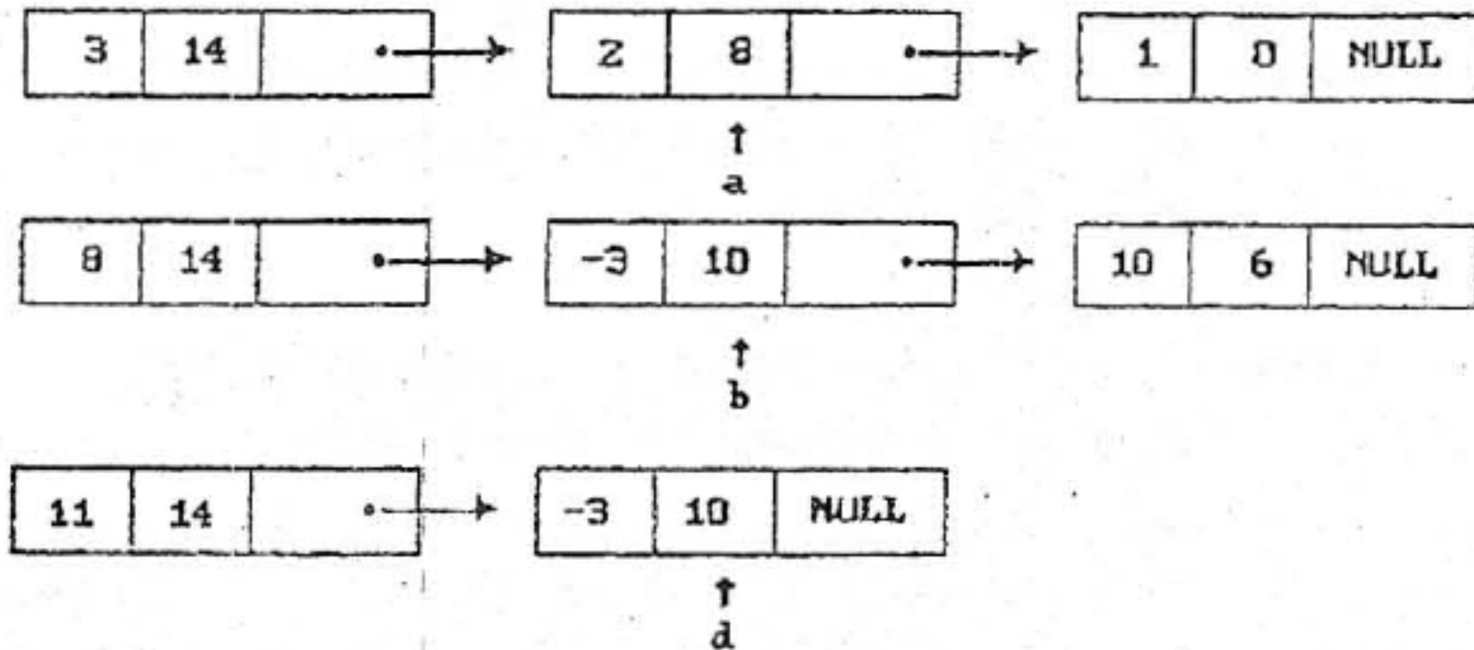
```
typedef struct poly_node* poly_pointer;
typedef struct poly_node{
    int coef; // coefficient
    int expon; // exponent
    poly_pointer link;
};
poly_pointer a,b,d;
```

$a(x) = 3x^{14} + 2x^{8} + 1$
$b(x) = 8x^{14} - 3x^{10} + 10x^{6}$





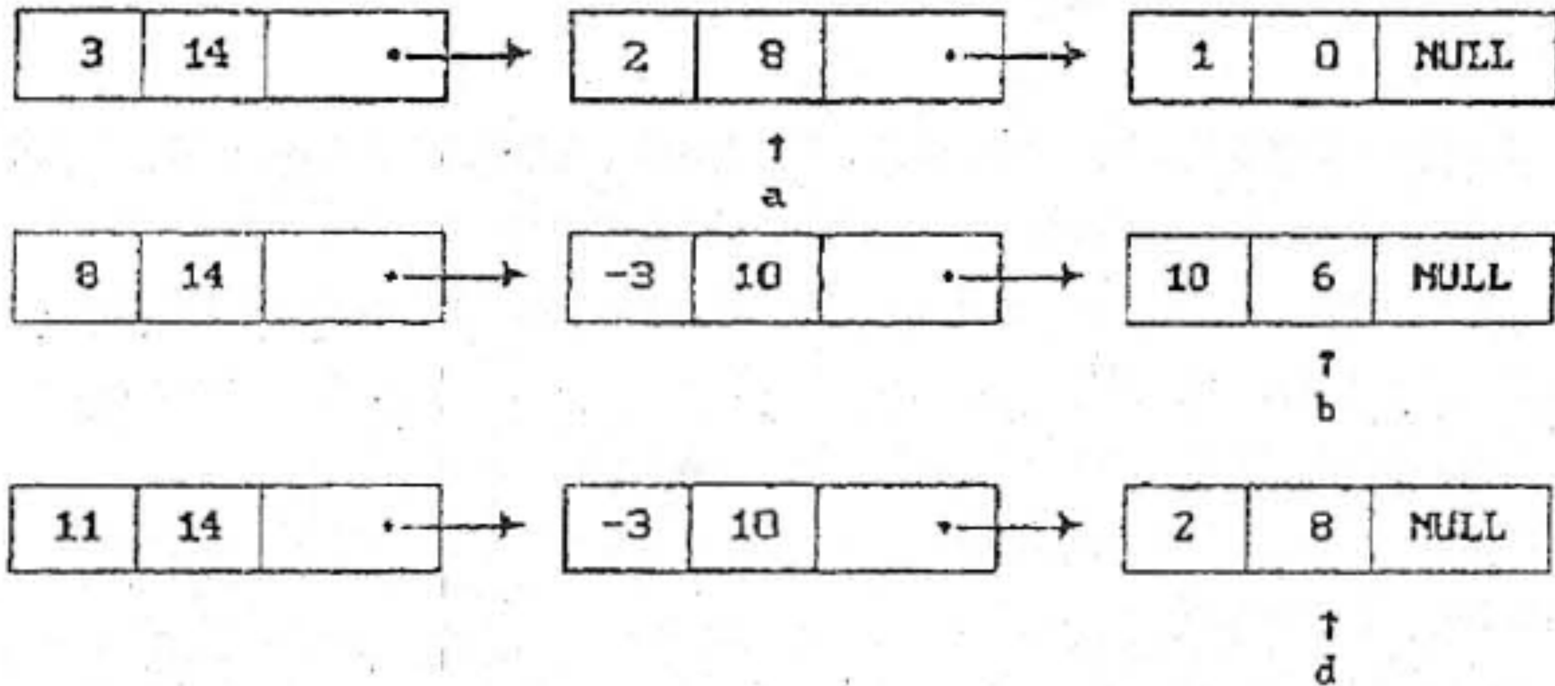
If the exponents of two terms are equal, we add the two coefficients and create a new term, also move the pointers to the next nodes in a and b.



(b) a -> expon < b -> expon

If the exponent of the current term in a is less than the exponent of the current term in b, then we create a duplicate term of b and attach this term to the result, called d. Then move the pointer to the next term in b.

(Similarly, if the exponent of the current term in a is larger than the exponent of the current term in b.)



To avoid having to search for the last node in d, each time we add a new node, we keep a pointer, “rear”, which points to the current last node in d.

Complexity of p.add: O(m+n).,
m and n, the number of terms in the polynomials a and b. Why?

```c
void attach(float coefficient,int exponent,poly_pointer *rear){
/* create a new node with coef = coefficient and expon =
exponent, attach it to the node pointed to by rear, current
last node in d. rear is updated to point to this new node */

    poly_pointer temp;
    temp = (poly_pointer)malloc(sizeof(poly_node));
    if (IS_FULL(temp)){
      fprintf(stderr, "The memory is full\n");
      exit(1);}
    temp->coef = coefficient;
    temp->expon = exponent;
    (*rear)->link = temp;
    *rear = temp;
}
```

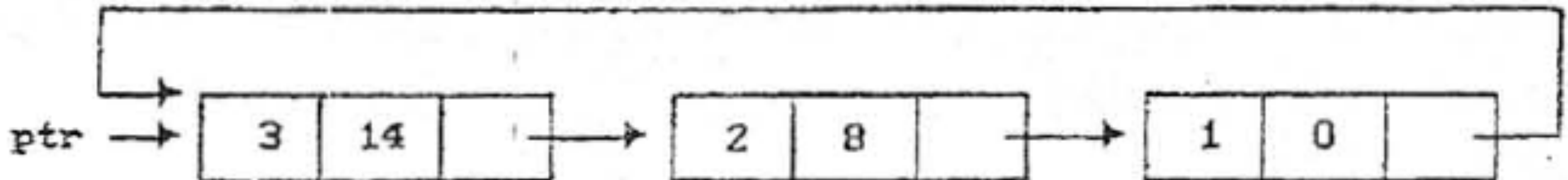
```
poly_pointer padd(poly_pointer a, poly_pointer b){
/* return a polynomial which is the sum of a and b */
    poly_pointer front, rear, temp;
    int sum;
    rear = (poly_pointer)malloc(sizeof(poly_node));
    if (IS_FULL(rear)){
        fprintf(stderr, "The memory is full\n");
        exit(1);
    }
    front = rear;
    while (a && b)
      switch (COMPARE(a->expon,b->expon)){
        case -1: /* a->expon < b->expon */
            attach(b->coef,b->expon, &rear);
            b = b->link; break;
        case 0: /* a->expon == b->expon */
            sum = a->coef + b->coef;
            if(sum) attach(sum,a->expon,&rear);
            a = a->link; b = b->link; break;
        case 1: /* a->expon > b->expon */
            attach(a->coef,a->expon,&rear);
            a = a->link; break;
      }
```

```
    /* copy rest of list a and then list b */
    for (; a; a=a->link) attach(a->coef,a->expon,&rear);
    for (; b; b=b->link) attach(b->coef,b->expon,&rear);
    rear->link = NULL;

    /* delete extra initial node */
    temp = front; front = front->link; free(temp);

    return front;
}
```

Singly-linked circular lists

Representing polynomials as circularly linked List

We can free the nodes of a polynomial more efficiently if we have a circular list:
In a circular list, the last node is linked to the first node.



$ptr(x) = 3x^{14} + 2x^{8} + 1$

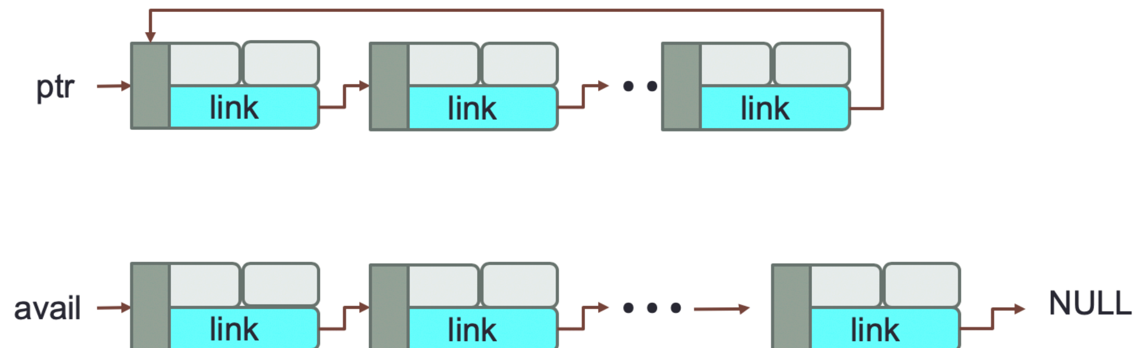
Methodology:

We free nodes that are no longer in use so that we may reuse these nodes later as follows:

- Obtain an efficient erase algorithm for circular lists
- Maintain our own list of nodes that have been "freed" as a chain. Let's call this list of free nodes as the available list.
- When we need a new node, examine the available list and use one of the free nodes if the available list is not empty
- Use "malloc" only if the available list is empty

- Let avail be a variable of type poly_pointer that points to the first node in the available list, our list of free nodes.
- Instead of using "malloc" and "free", we use get_node and ret_node.

(See next slide)



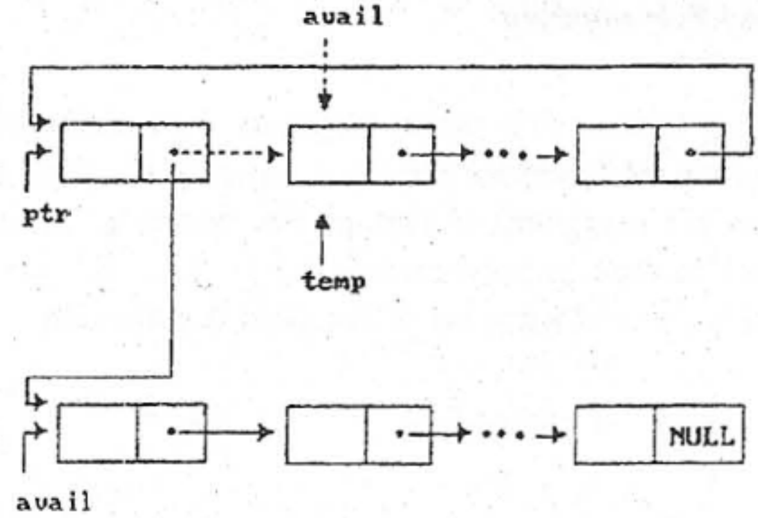
```
poly_pointer get_node(){
/*provide a node for use*/
    poly_pointer node; /*free node available, use that*/
    if (avail){
        node = avail;
        avail = avail->link;
    }
    else{/*there is no free node available, create one*/
        node = (poly_pointer) malloc(sizeof(poly_node));
        if(IS_FULL(node)){
            fprintf(stderr, The memory is full\n);
            exit(1);
        }
    }
    return node;
}
```

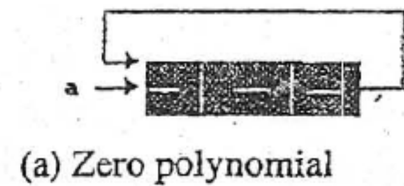
```
void ret_node(poly_pointer ptr){
/*return a node to the available list*/
    ptr->link = avail;
    avail = ptr;
}
```

Erasing in a circular list:

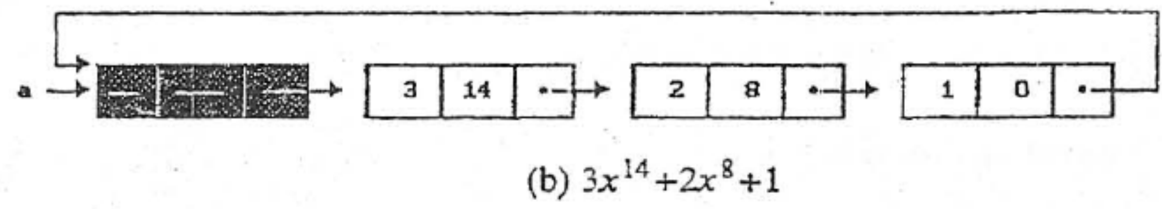
- We may erase a circular list in a fixed amount of time independent of the number of nodes in the list using *cerase.*
- To handle the special case of zero polynomial, a head node is introduced for each polynomial. Therefore, also the zero polynomial contains a node.

```
void cerase(poly_pointer *ptr){
/*erase the circular list ptr*/
    poly_pointer temp;
    if(*ptr){
        temp = (*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
    }
}
```





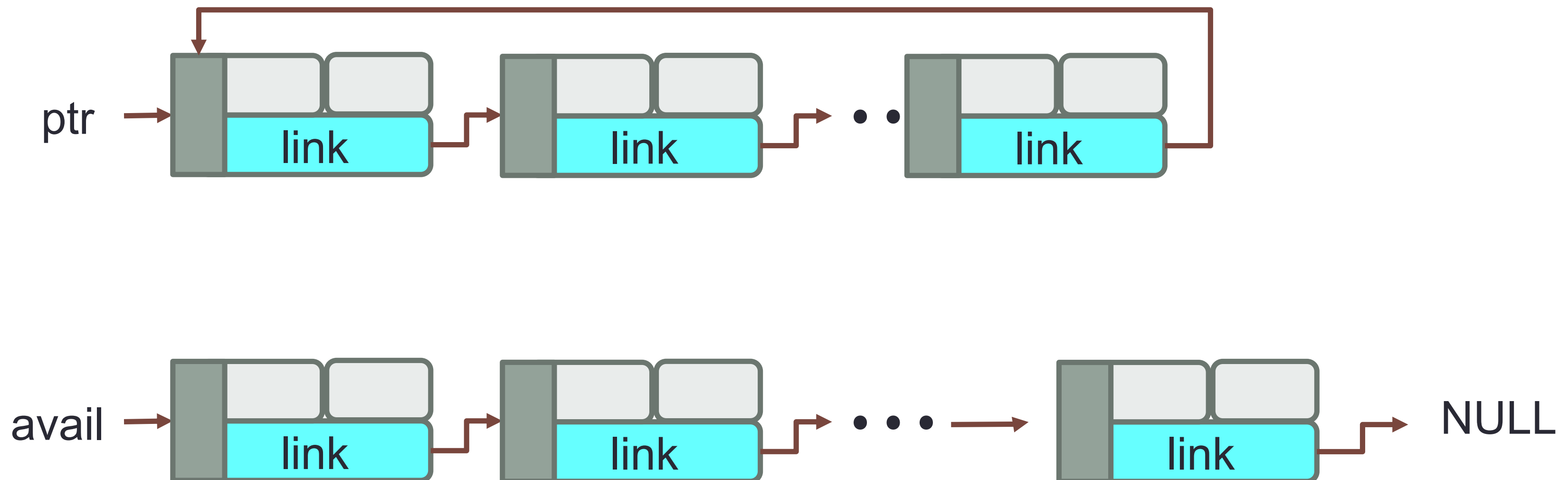
(a) Zero polynomial



(b) $3x^{14}+2x^{8}+1$

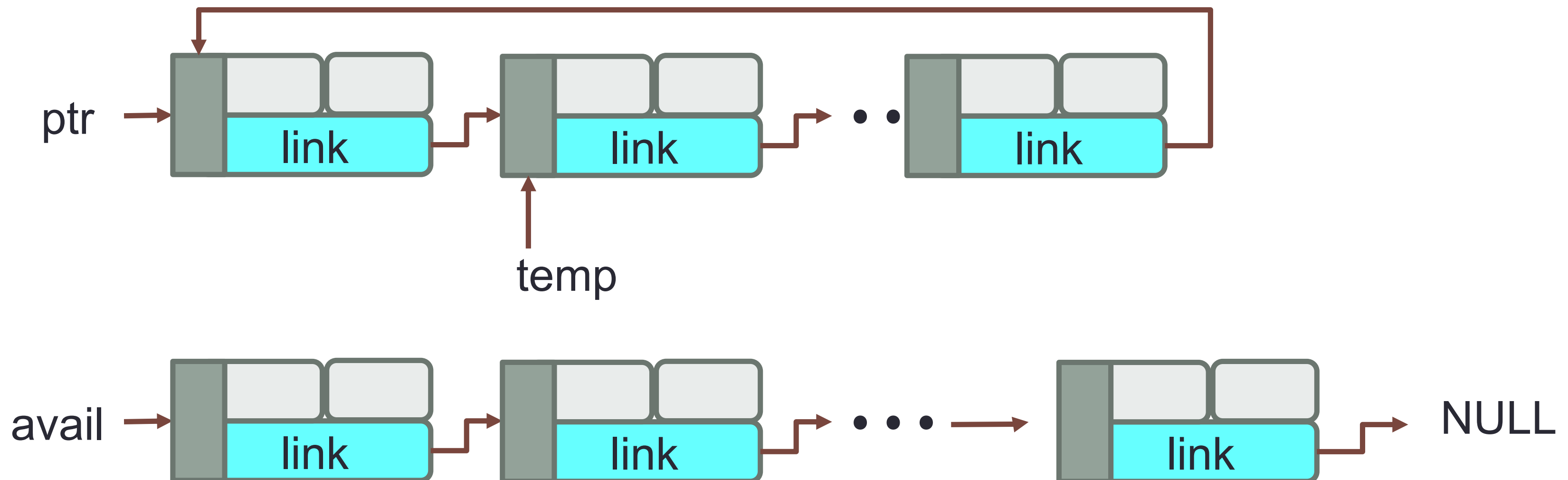
Erasing in a circular list:

```
void cerase(poly_pointer *ptr){
/*erase the circular list ptr*/
    poly_pointer temp;
    if(*ptr){
        temp = (*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
    }
}
```



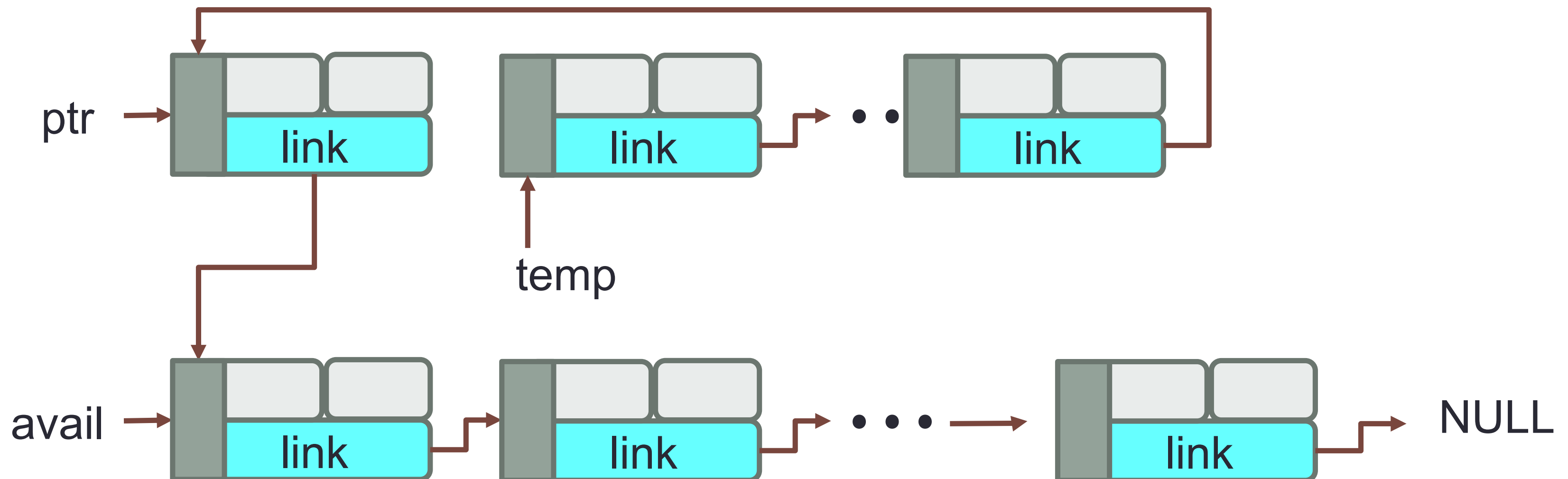
Erasing in a circular list:

```
void cerase(poly_pointer *ptr){
/*erase the circular list ptr*/
    poly_pointer temp;
    if(*ptr){
        temp = (*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
    }
}
```



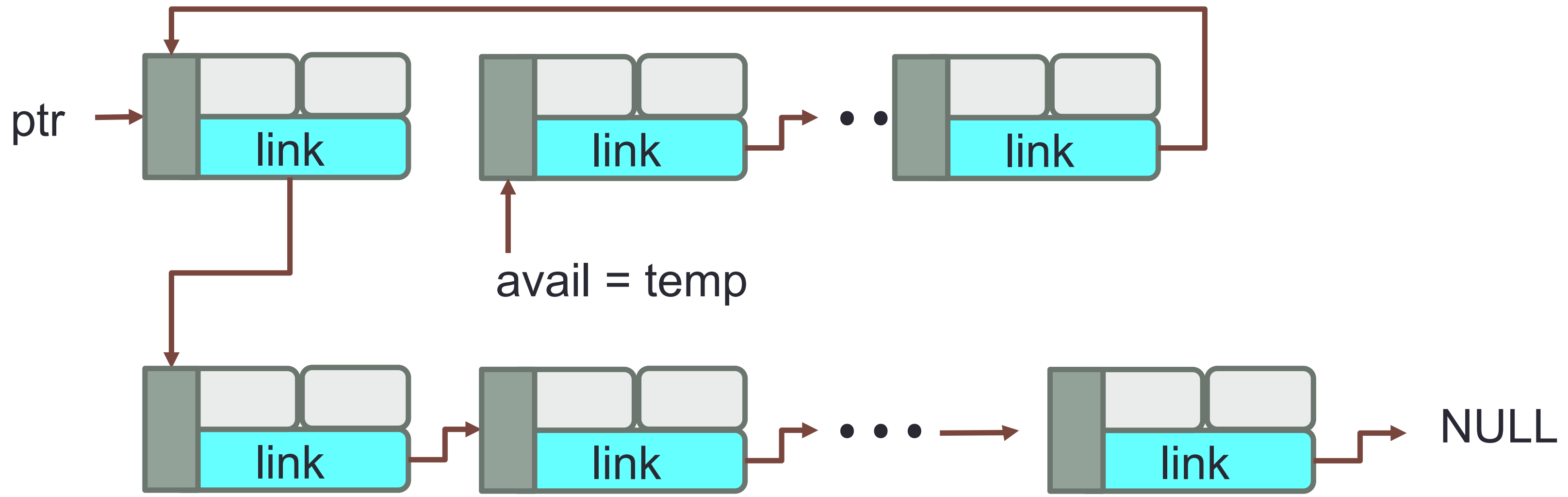
Erasing in a circular list:

```
void cerase(poly_pointer *ptr){
/*erase the circular list ptr*/
    poly_pointer temp;
    if(*ptr){
        temp = (*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
    }
}
```



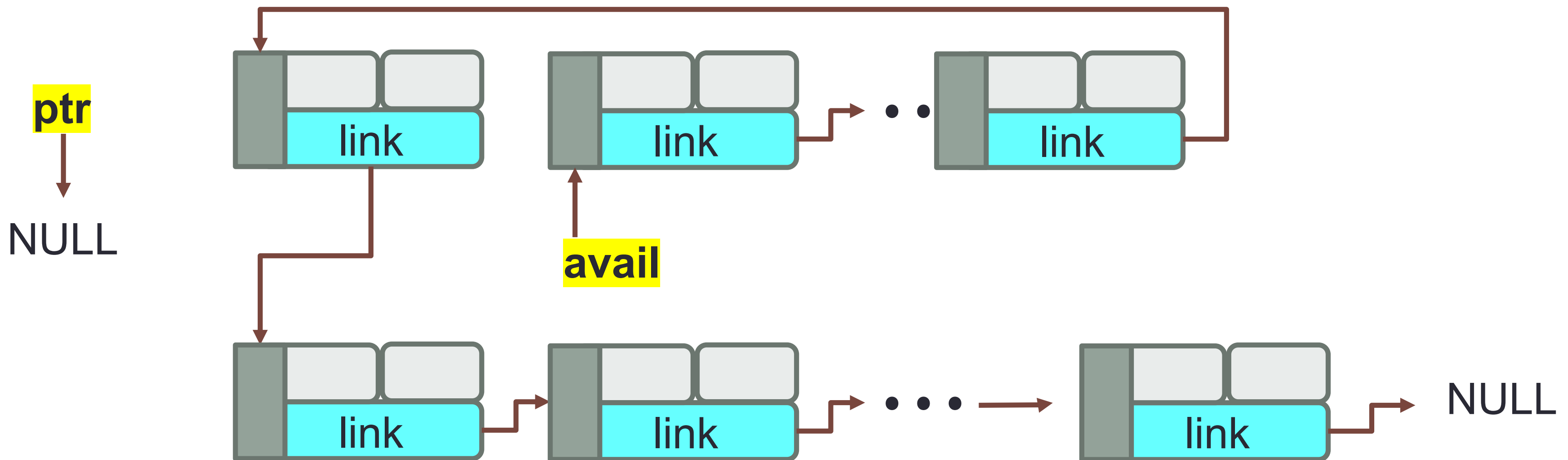
Erasing in a circular list:

```
void cerase(poly_pointer *ptr){
/*erase the circular list ptr*/
    poly_pointer temp;
    if(*ptr){
        temp = (*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
    }
}
```



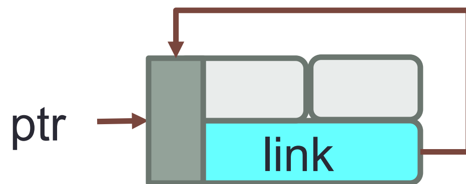
Erasing in a circular list:

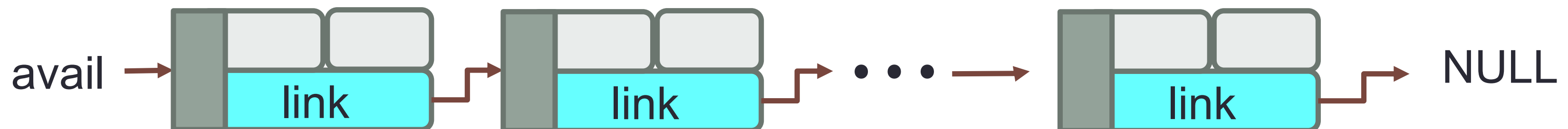
```
void cerase(poly_pointer *ptr){
/*erase the circular list ptr*/
    poly_pointer temp;
    if(*ptr){
        temp = (*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
    }
}
```



Erasing in a circular list: **Special case of zero polynomial only**

```
void cerase(poly_pointer *ptr){
/*erase the circular list ptr*/
    poly_pointer temp;
    if(*ptr){
        temp = (*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
    }
}
```

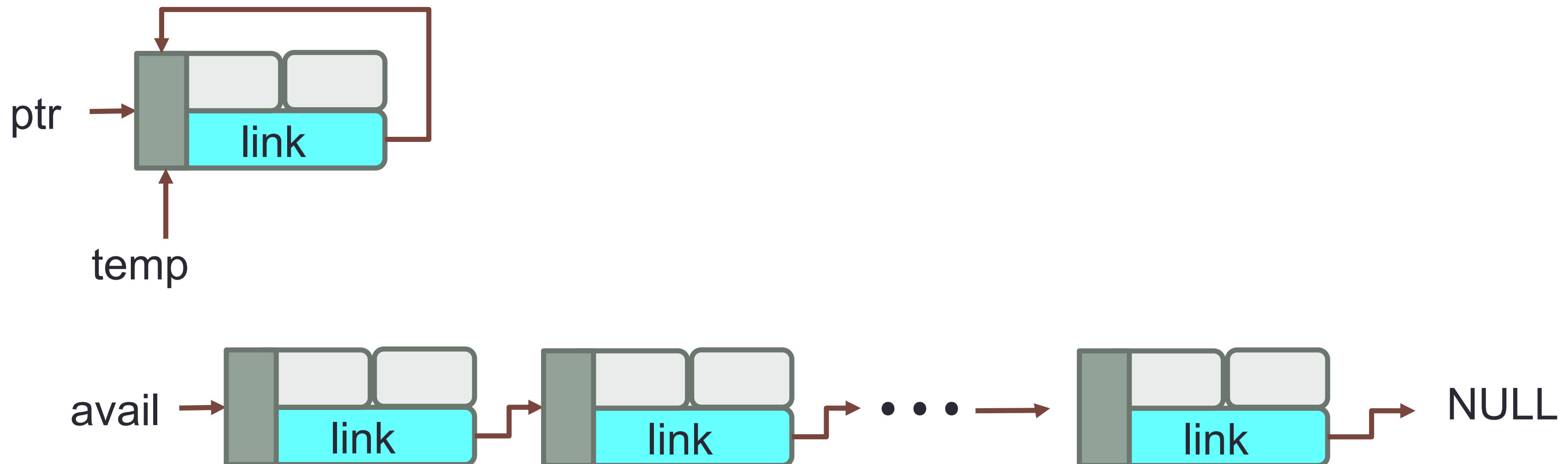




Erasing in a circular list:

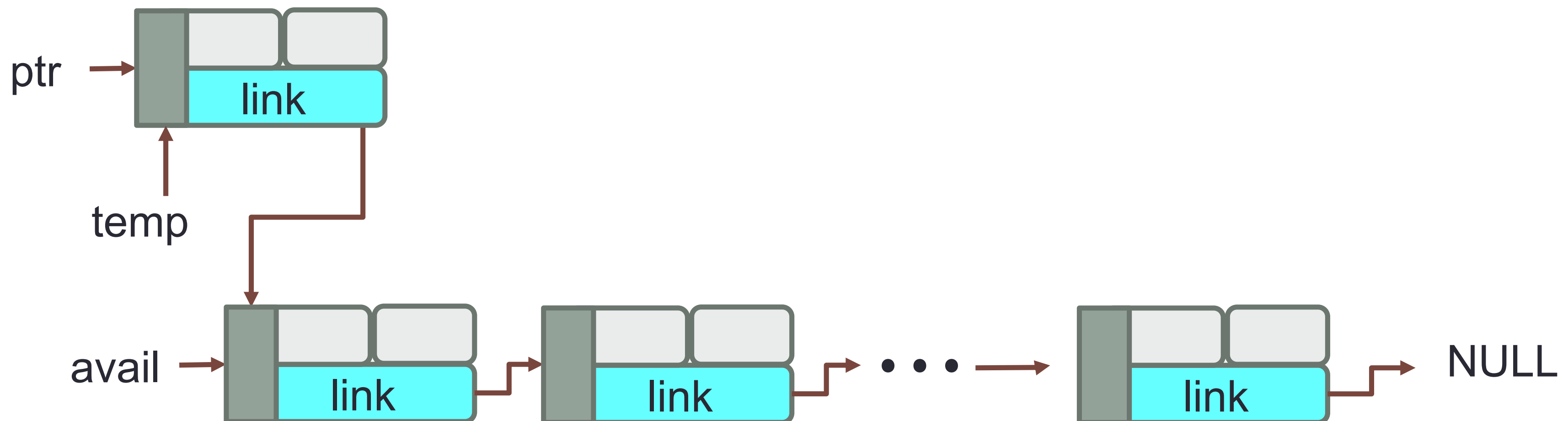
Special case of zero polynomial only

```
void cerase(poly_pointer *ptr){
/*erase the circular list ptr*/
    poly_pointer temp;
    if(*ptr){
       temp = (*ptr)->link;
       (*ptr)->link = avail;
       avail = temp;
       *ptr = NULL;
    }
}
```



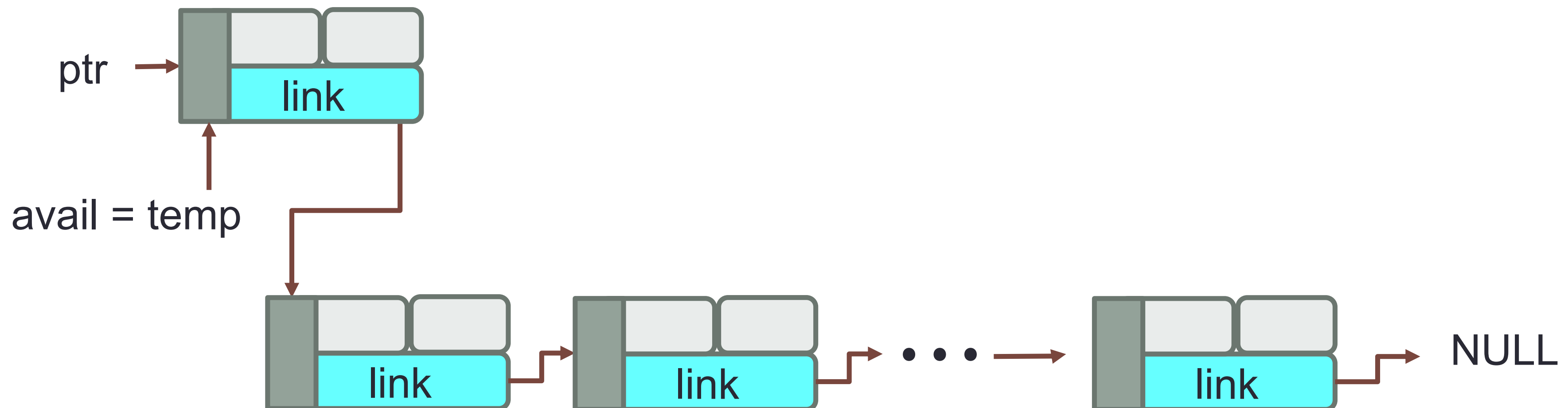
Erasing in a circular list: **Special case of zero polynomial only**

```
void cerase(poly_pointer *ptr){
/*erase the circular list ptr*/
    poly_pointer temp;
    if(*ptr){
        temp = (*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
    }
}
```



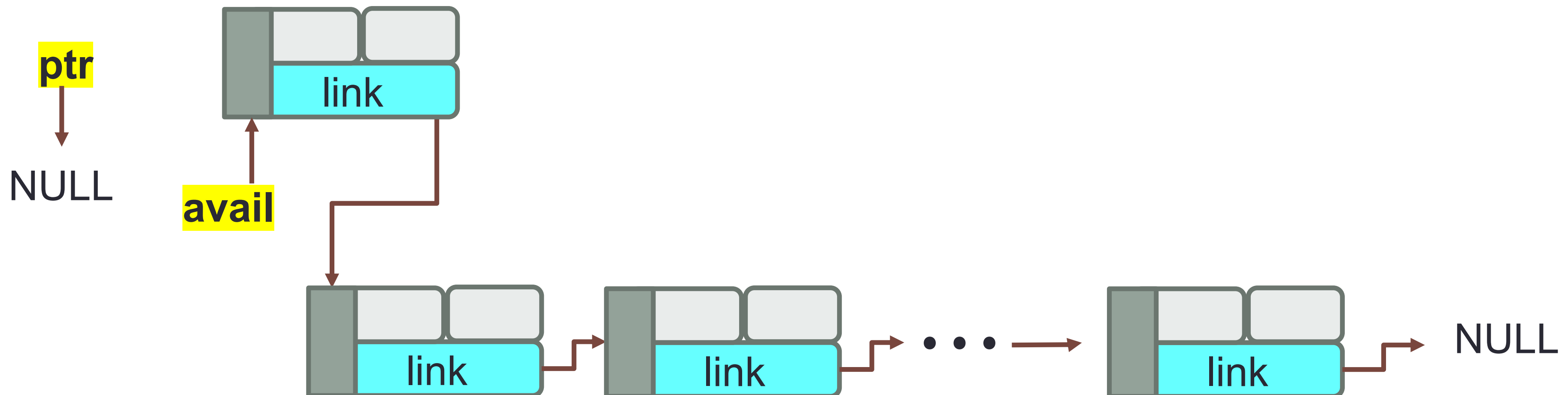
Erasing in a circular list: **Special case of zero polynomial only**

```
void cerase(poly_pointer *ptr){
/*erase the circular list ptr*/
    poly_pointer temp;
    if(*ptr){
        temp = (*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
    }
}
```



Erasing in a circular list: **Special case of zero polynomial only**

```
void cerase(poly_pointer *ptr){
/*erase the circular list ptr*/
    poly_pointer temp;
    if(*ptr){
        temp = (*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
    }
}
```



```c
poly_pointer cpadd(poly_pointer a, poly_pointer b){
/* polynomials a and b are singly linked circular lists with a head node.
Return a polynomial which is the sum of a and b */
    poly_pointer start_a, d, last_d;
    int sum, done = FALSE;
    start_a = a;          /*record start of a*/
    a = a->link;          /*skip headnode for a and b*/
    b = b->link;
    d = get_node();       /*get a head node for sum*/
    d->expon = -1; last_d = d;
    do{
      switch (COMPARE(a->expon,b->expon)){
        case -1: /* a->expon < b->expon */
            attach(b->coef, b->expon, &last_d);
            b = b->link; break;
        case 0: /* a->expon == b->expon */
            if(start_a == a) done = TRUE;
            else{
                sum = a->coef + b->coef;
                if(sum) attach(sum,a->expon,&last_d);
                a = a->link; b = b->link; }
            break;
        case 1: /* a->expon > b->expon */
            attach(a->coef,a->expon,&last_d);
            a = a->link;
      }
    }while(!done);
    last_d->link = d;
    return d;
}
```

Hashtable

Hashing

- An important and widely useful technique for implementing dictionaries
- Constant time per operation (on the average)
- Worst case time proportional to the size of the set for each operation (like linked list and arrays).

Basic Idea

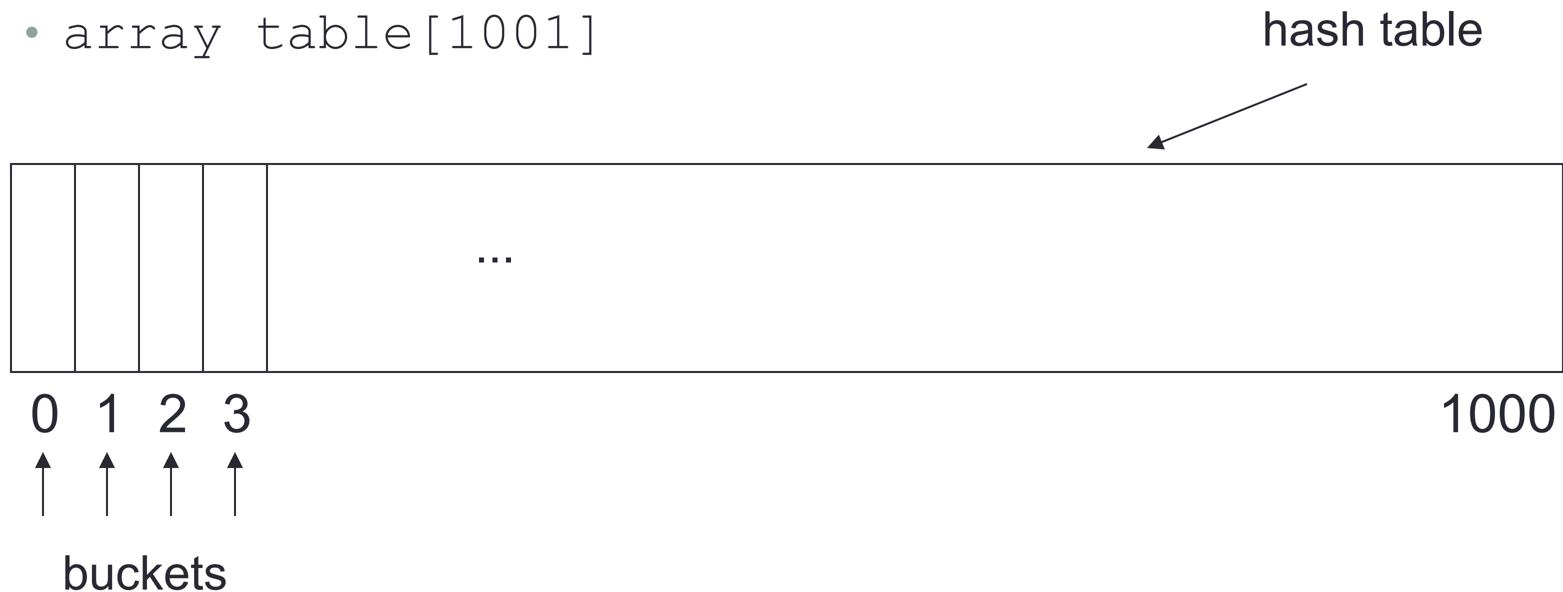
- Use *hash function* to map keys into positions in a *hash table.*

Ideally

- If element *e* has key *k* and *h* is hash function, then *e* is stored in position *h(k)* of table
- To search for *e*, compute *h(k)* to locate position. If no element, dictionary does not contain *e*.

Example

- Dictionary Student Records
  - Keys are ID numbers (951000 - 952000), no more than 1001 students
  - Hash function: h(k) = k-951000 maps ID into distinct table positions 0-1000
  - `array table[1001]`



Analysis (Ideal Case)

- O(b) time to initialize hash table (b number of positions or buckets in hash table)
- O(1) time to perform *insert, remove, search*

Ideal Case is Unrealistic

- Works for implementing dictionaries, but many applications have key ranges that are too large to have 1-1 mapping between buckets and keys!

Example:

- Suppose key can take on values from 0 .. 65,535 (2 byte unsigned int)
- Expect ≈ 1,000 records at any given time
- Impractical to use hash table with 65,536 slots!

Hash Functions

- If key range too large, use hash table with fewer buckets and a hash function which maps multiple keys to same bucket:

  $h(k_1) = \beta = h(k_2)$: k_1 and k_2 have collision at slot β
- Popular hash functions: hashing by division

  $h(k) = k\%D$, where D number of buckets in hash table
- Example: hash table with 11 buckets

  $h(k) = k\%11$

  $80 \rightarrow 3$ ($80\%11= 3$), $40 \rightarrow 7$, $65 \rightarrow 10$

  $58 \rightarrow 3$ collision!

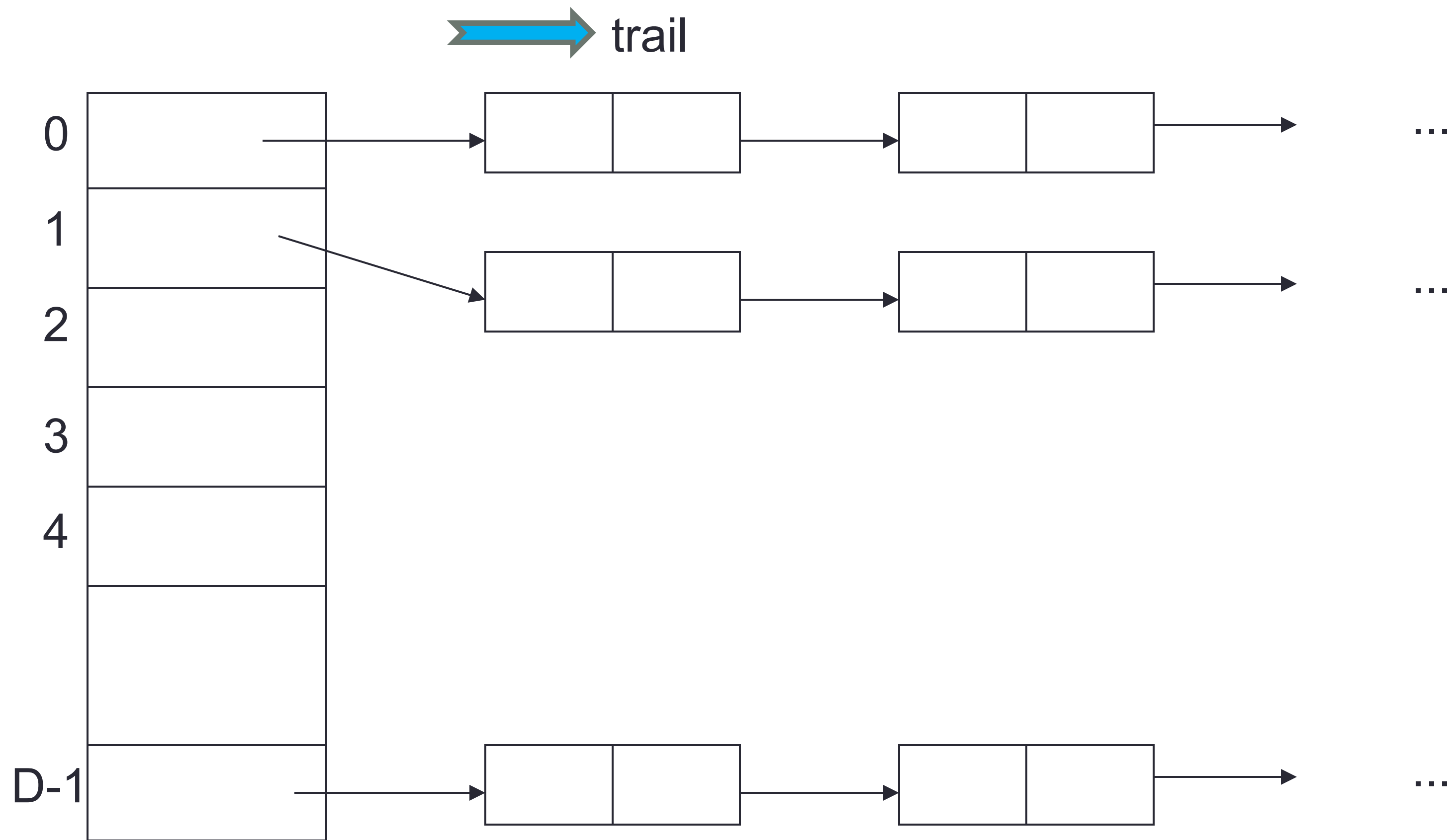
Collision Resolution Policies

- Two classes:
  - (1) Open hashing, a.k.a. separate chaining
  - (2) Closed hashing, a.k.a. open addressing
- Difference between the two has to do with
  - whether collisions are stored *outside the table* (open hashing)
  - or whether collisions result in storing one of the records at *another slot in the table* (closed hashing)

Open Hashing

- Each bucket in the hash table is the head of a linked list
- All elements that hash to a particular bucket are placed on that bucket's linked list
- Records within a bucket can be ordered in several ways
  - by order of insertion, by key value order, or by frequency of access order

Open Hashing Data Organization



Analysis

- Open hashing is a collision avoidence method which uses array of linked list to resolve the collision.
  - It is also known as the separate chaining method**: each linked list is considered as a chain.**
  - It is most appropriate when the hash table is kept in main memory, implemented with a standard in-memory linked list.

- We hope that number of elements per bucket roughly equal in size, so that the lists will be short
- If there are *n* elements in set, then each bucket will have roughly *n/D*
- If we can estimate *n* and choose *D* to be roughly as large, then the average bucket will have only one or two members

Analysis Cont'd

Average time per dictionary operation:

- *D* buckets, *n* elements in dictionary $\Rightarrow$ average *n/D* elements per bucket
- *insert, search, remove* operation take O(1+n/D) time each
- If we can choose D to be about n, constant time
- Assuming each element is likely to be hashed to any bucket, running time is constant, independent of n

Data Structure for Chaining

```
#define MAX_CHAR 10
#define TABLE_SIZE 13
#define IS_FULL(ptr) (!(ptr))

typedef struct {
    char key[MAX_CHAR];
    /* other fields */
} element;

typedef struct list *list_pointer;
typedef struct list {
    element item;
    list_pointer link;
};

list_pointer hash_table[TABLE_SIZE];
```

Chain Insert

```
void chain_insert(element item, list_pointer ht[])
{
  int hash_value = hash(item.key);
  list_pointer ptr, trail=NULL, lead=ht[hash_value];
  for (; lead; trail=lead, lead=lead->link)
    if (!strcmp(lead->item.key, item.key)) {
      fprintf(stderr, "The key is in the table\n");
      exit(1);
  }
  ptr = (list_pointer) malloc(sizeof(list));
  if (IS_FULL(ptr)) {
    fprintf(stderr, "The memory is full\n");
    exit(1);
  }
  ptr->item = item;
  ptr->link = NULL;
  if (trail) trail->link = ptr;
  else ht[hash_value] = ptr;
}
```

Results of Hash Chaining

acos, atoi, char, define, exp, ceil, cos, float, atol, floor, ctime
hash(x)= (x[0] - 'a') //first character of x