Programming and Data Structures - I

Lecture 13

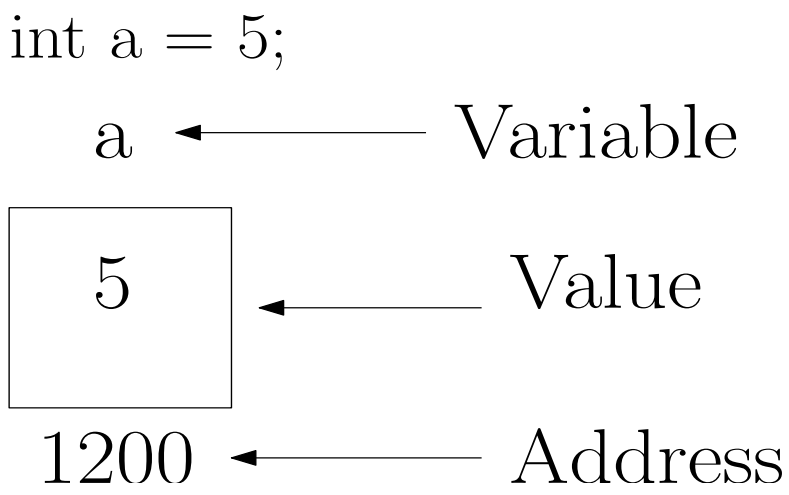
Kripabandhu Ghosh

CDS, IISER Kolkata

POINTERS

Variable

During the declaration of a variable (e.g. int a = 5), some location (address) is allocated in the memory to store the value of the variable

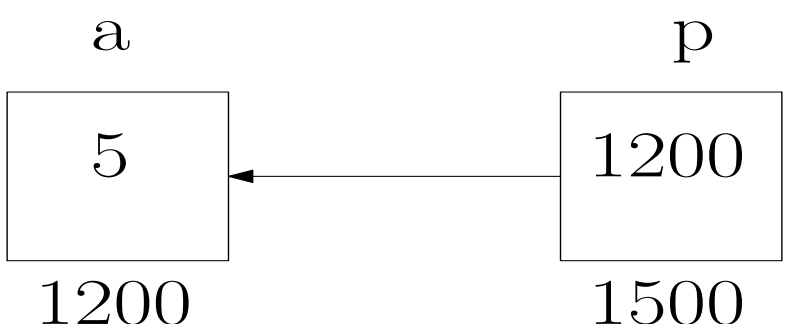


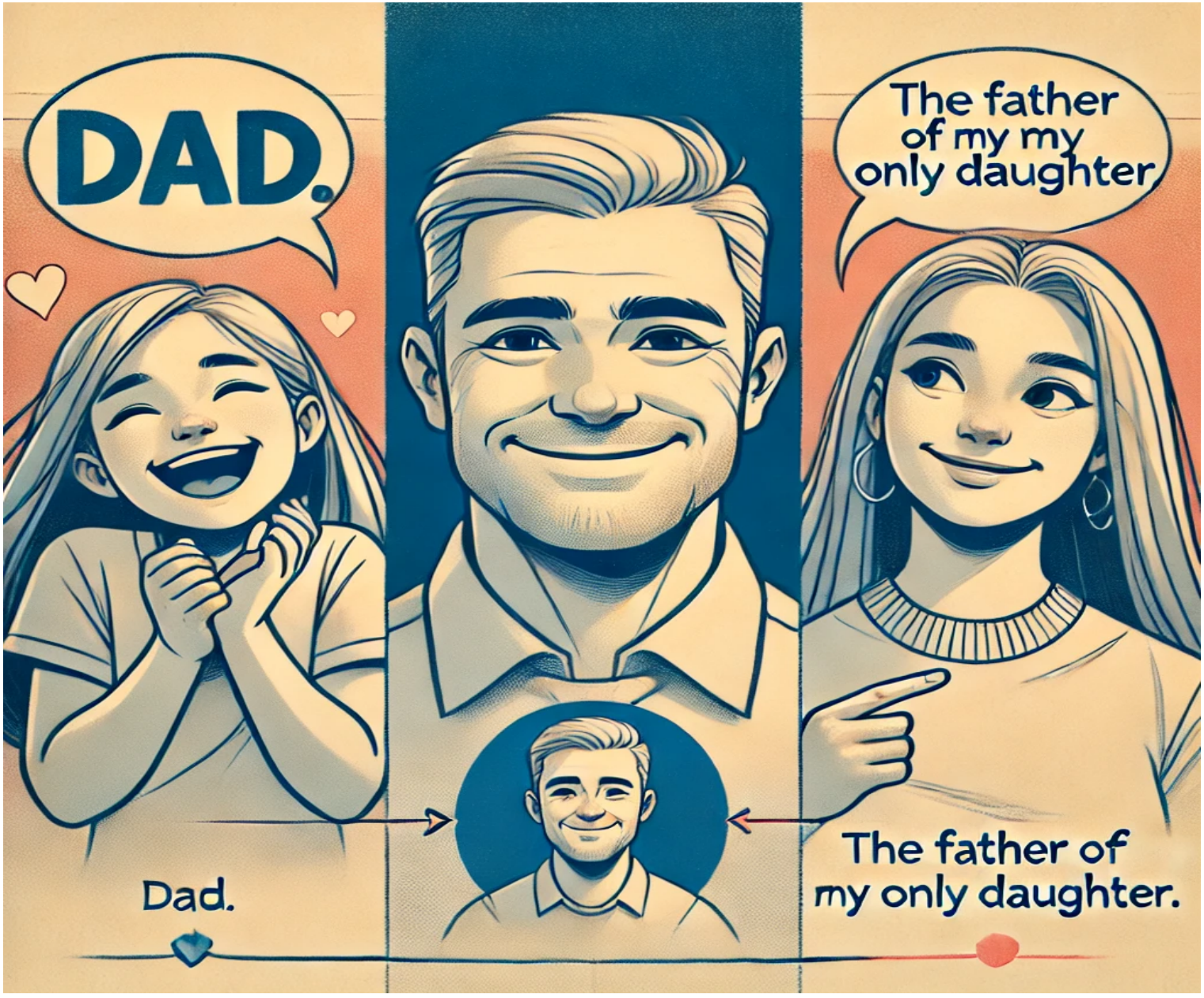
Pointers

Pointers are variables (e.g. p) which store the memory address of other variables (e.g. a)

- **Declaration**: data_type *pointer_variable; (e.g. int *p;)
  - **data_type *** indicates that pointer_variable is a pointer variable pointing to a variable of type data_type
- **Assignment**: pointer_variable = &variable; (e.g. p = &a;)
- **Address of**: '&' operator (&variable ⇒ address of variable)

```
int a = 5;
int *p;
p = &a;
```




DAD.
The father of my my only daughter
Dad.
The father of my only daughter.

Accessing variable value using pointer

* operator

- **value at address**: '*' operator (also called the indirection operator)
  - int a = 5, y;
  - int *p;
  - p = &a;
  - y = *p; // y contains the value of a

Modes of variable access

- **By name**: using 'a'
- **By address**: using '*p'

Accessing variable value using pointer (contd.)

Program

```
#include<stdio.h>
#include<string.h>
int main()
{
   int x = 5, y;
   int *p;
   p = &x;
   y = *p;
   printf("x = %d is stored at address &x = %u\n", x, &x);
   printf("x = %d is stored at address &x = %u\n", *&x, &x);
   printf("Using p: x = %d is stored at address &x = %u\n", *p, p);
   printf("Using p, y: x = %d is stored at address &x = %u\n", y, &*p);
   printf("p = %u is stored at address &p = %u\n", p, &p);
   printf("y = %d is stored at address &y = %u\n", y, &y);
   printf("Changing the value of x via p...\n\n");
   *p = 10;
   printf("x = %d is stored at address x = %u\n", x, &x);
   return 0;
}
```

Accessing variable value using pointer (contd.)

Output

x = 5 is stored at address &x = 287414176
x = 5 is stored at address &x = 287414176
Using p: x = 5 is stored at address &x = 287414176
Using p, y: x = 5 is stored at address &x = 287414176
p = 287414176 is stored at address &p = 287414184
y = 5 is stored at address &y = 287414180
Changing the value of x via p...
x = 10 is stored at address &x = 287414176

Passing variables in function call

- **Call by value**:
  - The values of the local variables of one function are passed to another function
  - The local variables remain untouched
- **Call by reference/address**:
  - The addresses of the local variables of one function are passed to another function
  - The local variables of one function are modified in the scope of another function

Call by Value vs. Call by Reference

Program

```
#include<stdio.h>
#include<string.h>
void swap_value(int a, int b)
{
   int t;
   t = a;
   a = b;
   b = t;
   printf("In swap_value: address of a = %u, address of b = %u\n\n", &a, &b); }
void swap_ref(int* a, int* b)
{
   int t;
   t = *a;
   *a = *b;
   *b = t;
   printf("In swap_ref: address of a = %u, address of b = %u\n\n", a, b);
}
```

Call by Value vs. Call by Reference (contd.)

Program

```
int main()
{
   int a = 5, b = 10;
   printf("In main: address of a = %u, address of b = %u\n\n", &a, &b);
   printf("Call by value:\n\n");
   printf("Before swap: a = %d b = %d\n", a, b);
   swap_value(a, b);
   printf("After swap: a = %d b = %d\n", a, b);
   printf("\nCall by reference:\n\n");
   printf("Before swap: a = %d b = %d\n", a, b);
   swap_ref(&a, &b);
   printf("After swap: a = %d b = %d\n", a, b);
   return 0;
}
```

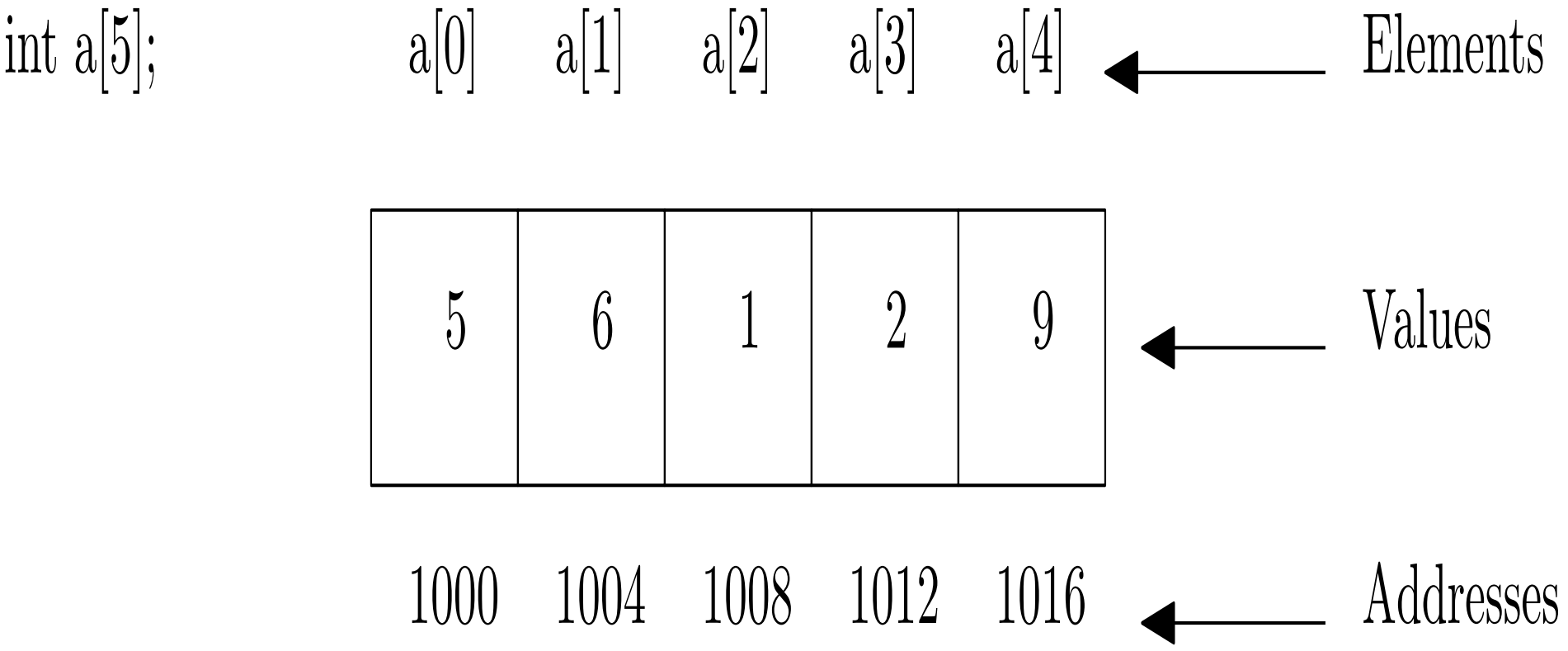
Call by Value vs. Call by Reference (contd.)

Output

In main: address of a = 2018057624, address of b = 2018057628
Call by value:
Before swap: a = 5 b = 10
In swap_value: address of a = 2018057580, address of b = 2018057576
After swap: a = 5 b = 10
Call by reference:
Before swap: a = 5 b = 10
In swap_ref: address of a = 2018057624, address of b = 2018057628
After swap: a = 10 b = 5

Pointers and Arrays: Memory allocation

Contiguous memory allocation



Pointers and Arrays: Access

Premise

```
int a[] = {5, 6, 1, 2, 9};
int *p;
p = a;
```

Pointer expression	Interpretation	Remarks
p = a;	p = &a[0]	Pointing to the starting address of the first element of array **a**
(p + i)	&a[i]	Accessing array element address via pointer
*(p + i)	a[i]	Accessing array element value via pointer
p++	a[++i]	Pointer shifts to the next array element (next element of the same data type)
&a[i]	base address + (i * sizeof(int))	base address: &a[0]
*(a + i)	a[i]	array as a pointer

Pointers and Arrays: Access

Program

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

int main()
{
  int a[] = {5, 6, 1, 2, 9}, *p, i;
  printf("Sizeof int = %zu\n", sizeof(int));
  printf("Accessing using array indices:\n\n");
  for(i = 0; i < 5 ; i++)
  {
    printf("Value = %d at address = %u\n", a[i], &a[i]);
  }
  p = &a[0]; //p = a also works
  i = 0;
  printf("Accessing using pointers:\n\n");
  while(i < 5)
  {
    printf("Array: Value = %d at address = %u\n", a[i], &a[i]);
    printf("Pointer *(p + i): Value = %d at address = %u\n", *(p + i), p);
    printf("Pointer *(i + p): Value = %d at address = %u\n", *(i + p), p);
    printf("Pointer *(a + i): Value = %d at address = %u\n", *(a + i), (a + i));
    printf("Pointer *(i + a): Value = %d at address = %u\n", *(i + a), (i + a));
    printf("Pointer i[a]: Value = %d at address = %u\n", i[a], &i[a]);
    printf("\n==========\n");
    i++;
    }
  return 0;
}
```

Pointers and Arrays: Access

Output

Sizeof int = 4
Accessing using array indices:
Value = 5 at address = 1513737520
Value = 6 at address = 1513737524
Value = 1 at address = 1513737528
Value = 2 at address = 1513737532
Value = 9 at address = 1513737536

Pointers and Arrays: Access

Output (contd.)

```
Accessing using pointers:
Array: Value = 5 at address = 1513737520
Pointer *(p + i): Value = 5 at address = 1513737520
Pointer *(i + p): Value = 5 at address = 1513737520
Pointer *(a + i): Value = 5 at address = 1513737520
Pointer *(i + a): Value = 5 at address = 1513737520
Pointer i[a]: Value = 5 at address = 1513737520
================================================================================
```

Pointers and Arrays: Access

Output (contd.)

Array: Value = 6 at address = 1513737524
Pointer *(p + i): Value = 6 at address = 1513737524
Pointer *(i + p): Value = 6 at address = 1513737524
Pointer *(a + i): Value = 6 at address = 1513737524
Pointer *(i + a): Value = 6 at address = 1513737524
Pointer i[a]: Value = 6 at address = 1513737524
==========
Array: Value = 1 at address = 1513737528
Pointer *(p + i): Value = 1 at address = 1513737528
Pointer *(i + p): Value = 1 at address = 1513737528
Pointer *(a + i): Value = 1 at address = 1513737528
Pointer *(i + a): Value = 1 at address = 1513737528
Pointer i[a]: Value = 1 at address = 1513737528
==========

Pointers and Arrays: Access

Output (contd.)

Array: Value = 2 at address = 1513737532
Pointer *(p + i): Value = 2 at address = 1513737532
Pointer *(i + p): Value = 2 at address = 1513737532
Pointer *(a + i): Value = 2 at address = 1513737532
Pointer *(i + a): Value = 2 at address = 1513737532
Pointer i[a]: Value = 2 at address = 1513737532
==========
Array: Value = 9 at address = 1513737536
Pointer *(p + i): Value = 9 at address = 1513737536
Pointer *(i + p): Value = 9 at address = 1513737536
Pointer *(a + i): Value = 9 at address = 1513737536
Pointer *(i + a): Value = 9 at address = 1513737536
Pointer i[a]: Value = 9 at address = 1513737536
==========

Pointers and Arrays: Access (shift)

Program

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

int main()
{
   int a[] = {5, 6, 1, 2, 9}, *p, i;
   p = &a[0]; //p = a also works
   i = 0;
   printf("Accessing using pointers:\n\n");
   while(i < 5)
   {
      printf("Pointer *p: Value = %d at address = %u\n", *p, p);
      p++;
      i++;
   }
   return 0;
}
```

Pointers and Arrays: Access (shift)

Output

Accessing using pointers:
Pointer *p: Value = 5 address = 2520572032
Pointer *p: Value = 6 address = 2520572036
Pointer *p: Value = 1 address = 2520572040
Pointer *p: Value = 2 address = 2520572044
Pointer *p: Value = 9 address = 2520572048

Valid/Invalid Pointer expressions

Premise

int a[] = {5, 6, 1, 2, 9}, *p, *q;
p = a;

Expression	Valid?
p = p + 3;	✓
p = p - 1;	✓
q = q + p;	x
q = q / p;	x
q = 3*p;	x
q = p/2;	x
q = &2;	x

Pointers and Arrays: Passing arrays via functions

Program

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

int findMaxArray(int *array, int size)//Function that takes an array as an argument and returns its maximum element
{
   int i, max = array[0];
   for(i = 0; i < size; i++)
   {
      if(array[i] > max)
      {
         max = array[i];
      }
   }
   return max;
}

int main()
   {
   int a[] = {5, 6, 1, 2, 9};
   printf("Max = %d\n", findMaxArray(a, sizeof(a)/sizeof(int)));
   return 0;
}
```

Pointers and Matrices

Premise

```
int a[2][3] = {{5, 6, 1}, {2, 9, 7}}, *p, (*q)[3], *r;
p = (int*)a;
q = a;
r = q + i;
```

Pointer expression	Interpretation
a + i	Pointer to the ith row
*(a + i)	Pointer to the first element of the ith row
*(a + i) + j	Pointer to the jth element of the ith row
((a + i) + j)	a[i][j]
*(p + i*noOfCols + j)	a[i][j]
*(r + j)	a[i][j]
(*q)[3]	Pointer to an array of 3 integers

Pointers and Matrices: implementation

Program

```
#include<stdio.h>
#include<string.h>
int main()
{
  int a[2][3] = {{5, 6, 1}, {2, 9, 7}}, *p, (*q)[3], *r, i , j, noOfRows = 2, noOfCols = 3;
  printf("Printing the element-wise addresses:\n");
  for(i = 0; i < noOfRows; i++)
  {
    for(j = 0; j < noOfCols; j++)
    {
      printf("%u ", &a[i][j]);
    }
    printf("\n");
  }
  printf("\n\n");
  printf("Pointers to 0th row, 1st row: %u %u\n", &a[0][0], &a[1][0]);
  printf("Pointers to 0th row, 1st row: %u %u\n", *a, *(a + 1));
  printf("Pointers to the (0,1)th, (1,1)th elements: %u %u\n, &a[0][1], &a[1][1]);
  printf("Pointers to the (0,1)th, (1,1)th elements: %u %u\n", *(a + 0) + 1, *(a + 1) + 1);
  printf("\n\n");
  p = (int*)a;
  printf("Using a in pointer form...\n\n");
  for(i = 0; i < noOfRows; i++)
  {
    for(j = 0; j < noOfCols; j++)
    {
      printf("%d ", *(*(a + i) + j));
    }
    printf("\n");
  }
```

Pointers and Matrices: implementation (contd.)

Program (contd.)

```
  printf("Using p...\n\n");
  for(i = 0; i < noOfRows; i++)
  {
    for(j = 0; j < noOfCols; j++)
    {
      printf("%d ", *(p + i*noOfCols + j));
    }
    printf("\n");
  }
  printf("\n\n");
  printf("Using q..\n\n");
  q = a;
  for(i = 0; i < noOfRows; i++)
  {
    r = q + i;
    for(j = 0; j < noOfCols; j++)
    {
      printf("%d ", *(r + j));
    }
    printf("\n");
  }
  printf("\n\n");
  return 0;
}
```

Pointers and Matrices: implementation (contd.)

Output

Printing the element-wise addresses:
1843915904 1843915908 1843915912
1843915916 1843915920 1843915924

Pointers to 0th row, 1st row: 1843915904 1843915916
Pointers to 0th row, 1st row: 1843915904 1843915916
Pointers to the (0,1)th, (1,1)th elements: 1843915908 1843915920
Pointers to the (0,1)th, (1,1)th elements: 1843915908 1843915920

Pointers and Matrices: implementation (contd.)

Output (contd.)

Using a in pointer form...

5 6 1
2 9 7

Using p...

5 6 1
2 9 7

Using q..

5 6 1
2 9 7

Pointers and Strings in function calls

Program

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

char* reverseString(char* string) //Function that returns the reversed version of string
{
   char *reversed;
   int i = 0, len;
   len = strlen(string);
   reversed = (char*)malloc((len + 1)*sizeof(char)); //Allocated memory for the reversed string
   while(string[i] != '\0')
   {
      reversed[len - 1 - i] = string[i];
      i++;
   }
   reversed[i] = '\0';
   return reversed;
}

int main()
{
   char* name = "Kripa";
   printf("Reversed version of %s is %s\n", name, reverseString(name));
   return 0;
}
```

Output

```
Reversed version of Kripa is apirK
```

Pointers and Strings in function calls: const

Program

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

void xstrcpy(char* copy, const char* string) //Function that copies 'string' to 'copy'
{
  while(*string != '\0')
  {
    *copy = *string;
    copy++;
    string++;
  }
  *copy = '';
  //string[0] = 'I'; //Not allowed
}

int main()
{
  char* name = "Kripa", copy[20];
  xstrcpy(copy, name);
  printf("The copy of %s is %s\", name, copy);
  return 0;
}
```

Output

The copy of Kripa is Kripa

Note

The *const* (constant) qualifier prohibits changes in the value of the associated variable

More on *const*

Premise

```
//Pointer to a constant
const int *ptr;
int const *p;

int i = 5, j = 10;
const int k = 7;
```

Operation	Validity
ptr = &i;	✓
*ptr = 25;	x
ptr = &j;	✓
p = &k;	✓
*p = 100;	x
p = &i;	✓

Even more on *const*

Premise

```
//Constant pointer
int* const q = &i;
```

Operation	Validity
*q = 99;	✓
ptr = &j;	x

And even more on *const*

Premise

```
//Constant pointer to a constant
const int* const r = &k;
```

Operation	Validity
r = &j;	x
*r = 55;	x

Pointer to function

Format

return_type (*ptr_to_function)();

Example

int (*f)();

Pointer to function: example

Program

```
#include<stdio.h>

int add(int a, int b)
{
  return (a + b);
}

int printSum(int (*f)(), int a, int b)
{
  return ((*f)(a,b));
}

int main()
{
  int (*f)();
  f = add;
  printf("Sum = %d\n", (*f)(4, 5));
  printf("Sum = %d\n", printSum(add, 4, 5));
  return 0;
}
```

Output

```
Sum = 9
Sum = 9
```

Pointer to structures

Program

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

typedef struct book
{
  char title[50];
  char id[10];
  int no;
}BOOK;

void display(BOOK* b)
{
  printf("Book title: %s\n", b->title);
  printf("Book id: %s\n", b->id);
  printf("Copies in stock: %d\n\n", b->no);
}

void update(BOOK* b)
{
  b->no++;
}

int main(void)
{
  BOOK book1 = {"Let us C", "B123", 5}, book2 = {"The Art of Computer Programming", "A1245", 3};
  display(&book1);
  display(&book2);
  update(&book1);
  display(&book1);
  return 0;
}
```

Pointer to structures (contd.)

Output

```
Book title: Let us C
Book id: B123
Copies in stock: 5

Book title: The Art of Computer Programming
Book id: A1245
Copies in stock: 3

Book title: Let us C
Book id: B123
Copies in stock: 6
```

Memory allocation using pointers

- Explicit allocation of memory space
- Space allocation in *heap*

Function	**Task**
malloc	Allocates the requested number of bytes returns the pointer (void) to the first byte of the alllocated space
calloc	Allocates space for an array of elements, initializes them to zero and returns the pointer to the memory
free	Frees allocated space]
realloc	Modifies the size of allocated space

Memory allocation using pointers (contd.)

Program

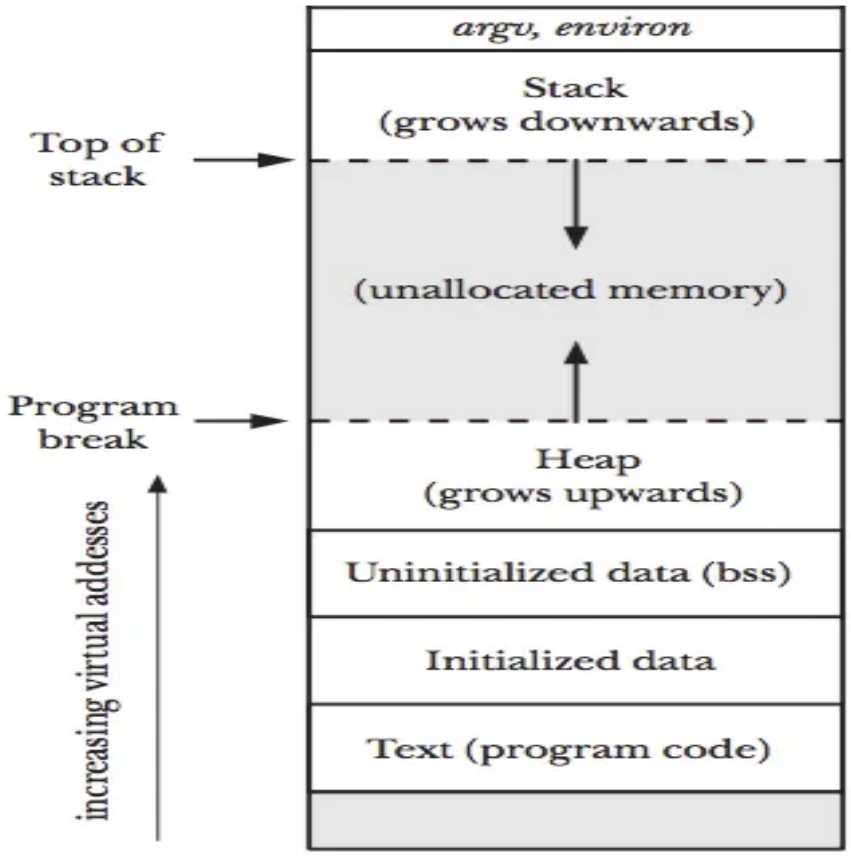
```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

int main(void)
{
  char *buffer;
  //Memory allocation
  buffer = (char*)malloc(20*sizeof(char));
  if(!buffer)
  {
    printf("Allocation failed!\n");
    exit(0);
  }
  strcpy(buffer, "Using Malloc");
  printf("Stored value: %s\n", buffer);
  //Memory reallocation
  buffer = (char*)realloc(buffer, 40*sizeof(char));
  if(!buffer)
  {
    printf("Reallocation failed!\n");
    exit(0);
  }
  strcpy(buffer, "Using Malloc and Realloc");
  printf("Stored value: %s\n", buffer);
  //Memory freed
  free(buffer);
  return 0;
}
```

Memory allocation using pointers (contd.)

Output

Stored value: Using Malloc
Stored value: Using Malloc and Realloc

Memory allocation using `malloc()`[1]



- Allocates memory from the heap, and adjusts the size of the heap as required, including expansion
- By default, Linux follows an optimistic memory allocation strategy; when malloc() returns non-NULL, there is no guarantee that the memory really is available; if not, one or more processes will be killed

[1] Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

All Storage Classes[2]

Storage Class	Location	Lifetime	Initialization
`auto` (local)	Stack	Function scope	Garbage
`static` (local)	Data/BSS	Program duration	Zero (if uninit)
`global`	Data/BSS	Program duration	Zero (if uninit)
`malloc`	Heap	Until `free()`	Garbage

[2]Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition; "The C Programming Language" by Brian Kernighan and Dennis Ritchie

Accessing with pointers

Expression	Interpretation
*p++	Increments the pointer and not the value
(*p)++ or ++*p	Increments the value pointed at by the pointer
*ptr->p	Fetches the value p points at
*ptr->p++	Increments p after accessing whatever p points at
(*ptr->p)++	Increments the value that p points at
*ptr++->p	Increments ptr after accessing whatever it points at

Priority of operators

- **Higher**: ->, ., (), []
- **Lower**: ++, *

Accessing with pointers (examples)

Program

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

typedef struct book
{
  char title[50];
  char id[10];
  int no;
  int *ptr;
}BOOK;

void display(BOOK* b)
{
  printf("Book title: %s\n", b->title);
  printf("Book id: %s\n", b->id);
  printf("Copies in stock: %d\n\n", b->no);
{

void initialize(BOOK *b, char* title, char* id, int no)
{
  strcpy(b->title, title);
  strcpy(b->id, id);
  b->no = no;
  b->ptr = &b->no;
}

int main(void)
{
  BOOK books[2], *p;
  int n, array[] = {1, 5, 15}, *q;
  p = books;
  initialize(books, "Let us C", "B123", 5);
  initialize(books + 1, "The Art of Computer Programming", "A1245", 3);
```

Accessing with pointers (examples)

Program (contd.)

```
    display(&books[0]);
    display(&books[1]);
    printf("\nPointers to structures:\n");
    printf("%d\n", p->no);
    printf("%d\n", *p->ptr);
    //printf("%d\n", *p->ptr++);
    //printf("%d\n", *p->ptr);
    (*p->ptr)++;
    printf("%d\n", *p->ptr);
    printf("%d\n", ++p->no);
    printf("%d\n", p->no);
    n = p++->no;
    printf("%d %d\n", n, p->no);
    q = array;
    printf("\nPointers in general:\n");
    printf("%d\n", *q++);
    printf("%d\n", *q);
    printf("%d\n", (*q)++);
    printf("%d\n", *q);
    printf("%d\n", ++*q);
    printf("%d\n", *q);
    return 0;
}
```

Accessing with pointers (examples)

Output

```
Book title: Let us C
Book id: B123
Copies in stock: 5

Book title: The Art of Computer Programming
Book id: A1245
Copies in stock: 3

Pointers to structures:
5
5
6
7
7
7 3

Pointers in general:
1
5
5
6
7
7
```

LINKED LISTS

Linked lists

Features

- Dynamic data structure (array is a static data structure, size to be specified precisely at the beginning)
  - Size can grow or shrink throughout the program
- No wastage of memory space – insertion when spece required, deletion when space no longer required
- Insertion and deletion efficient – no moving of data

Disadvantages

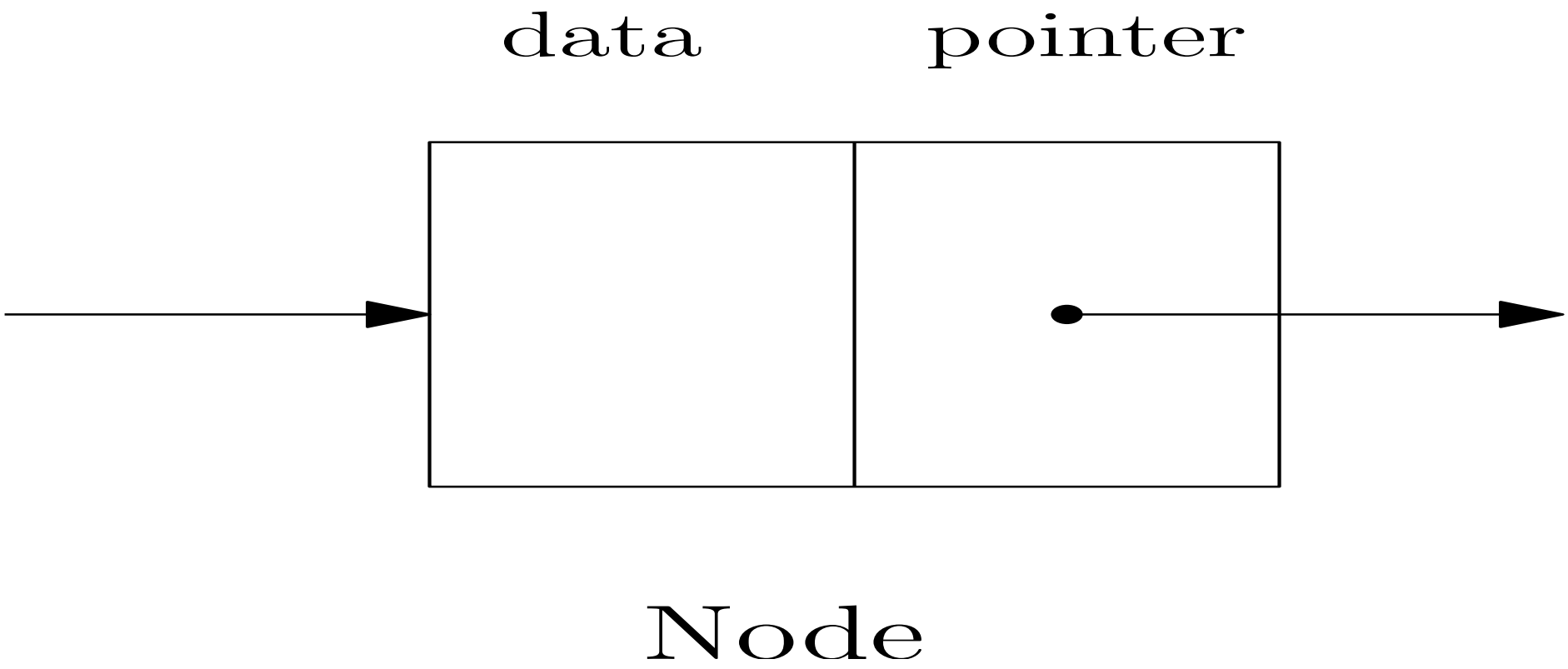
- Linear search to access a data item (array can access directly)
- More storage than an array of the same number of elements (data and pointer)

Linked lists: node

Format

- **Data**: information stored
- **Pointer**: address of the next node

Linked lists: node



Linked lists: node (implementation)

Code

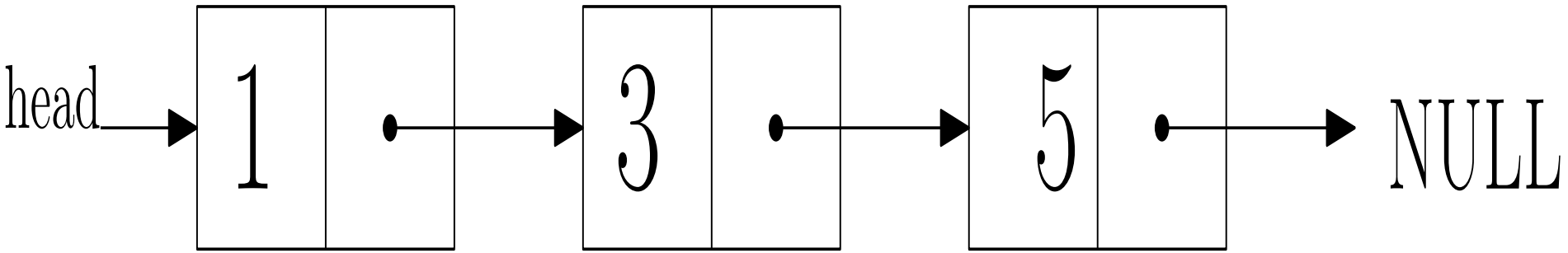
```
struct node
{
   int info;
   struct node *link;
};
```

Linked lists

Organization

- **head/start**: Pointer to the start of the linked list
- The first node is pointed at by **head/start**
- Each node (except the last one) points to the next node
- The last node points to NULL

Linked lists

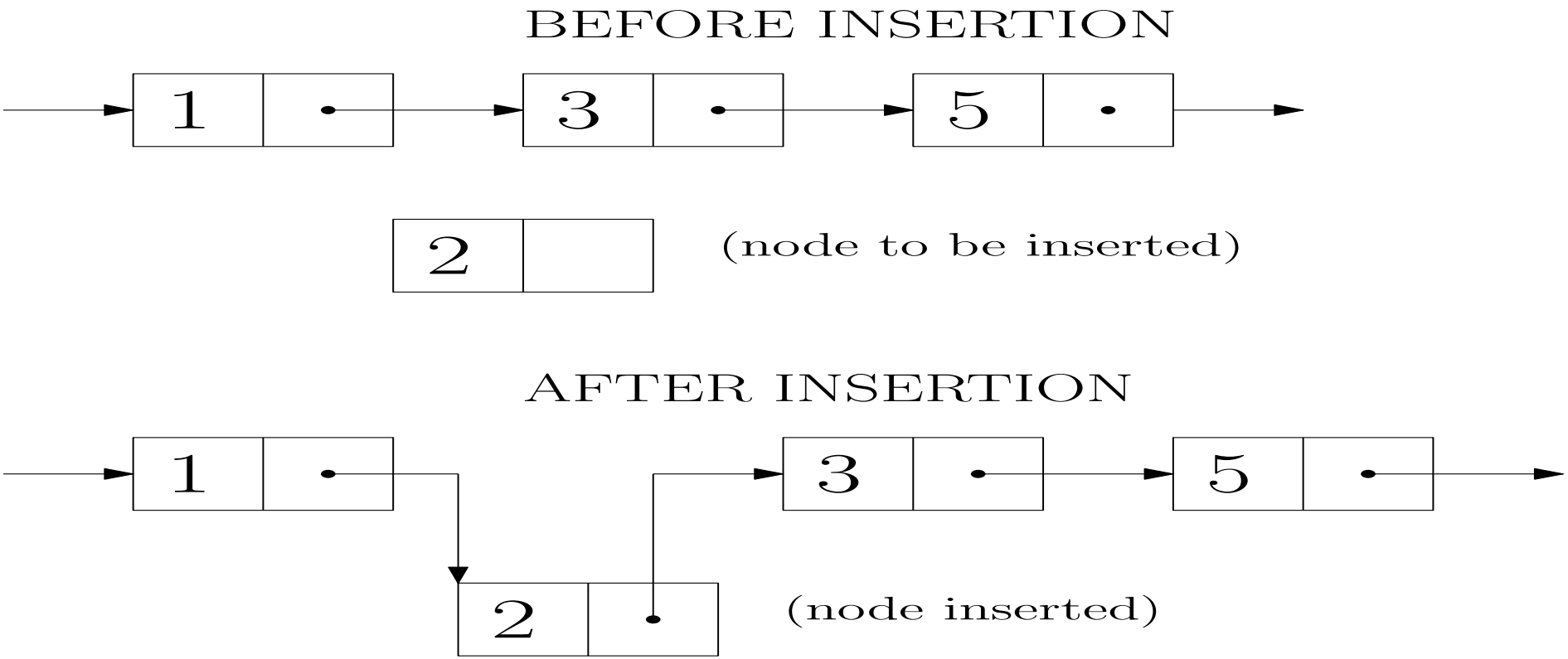


Linked lists: insertion (implementation)

Code

```
struct node *addafter(struct node *start, int data, int item)//To insert 'data' after 'item'
{
   struct node *tmp,*p;
   p=start;
   while(p!=NULL)
   {
      if(p->info==item)
      {
         tmp=(struct node *)malloc(sizeof(struct node));
         tmp->info=data;
         tmp->link=p->link;
         p->link=tmp;
         return start;
      }
      p=p->link;
   }
   printf("%d not present in the list\n",item);
   return start;
}
```

Linked lists: insertion (visualization)

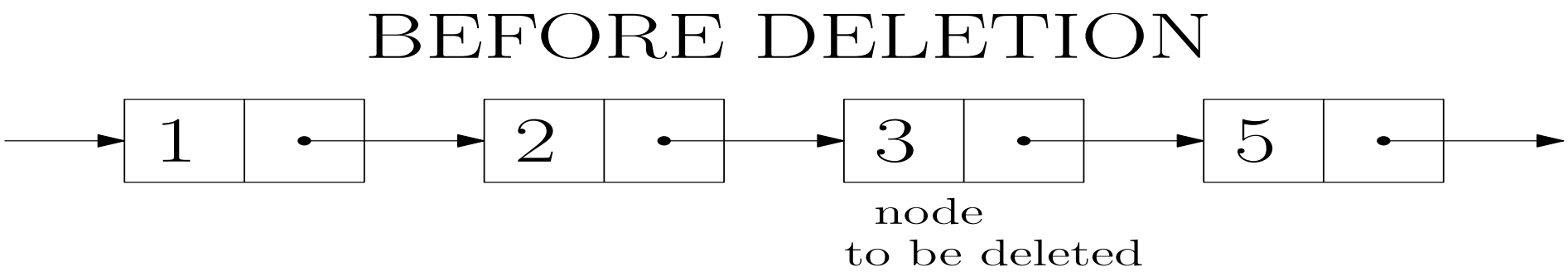


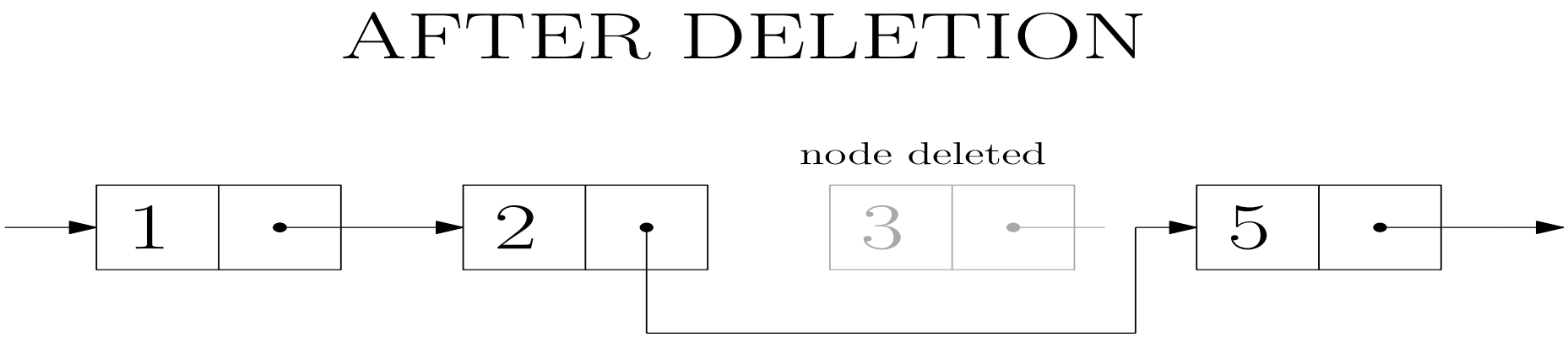
Linked lists: deletion (implementation)

Code

```
sruct node *del(struct node *start,int data)
{
  struct node *tmp,*p;
  if(start==NULL)
  {
    printf("List is empty\n");
    return start;
  }
  /*Deletion of first node*/
  if(start->info==data)
  {
    tmp=start;
    start=start->link;
    free(tmp);
    return start;
  }
  /*Deletion in between or at the end*/
  p=start;
  while(p->link!=NULL)
  {
    if(p->link->info==data)
    {
      tmp=p->link;
      p->link=tmp->link;
      free(tmp);
      return start;
    }
    p=p->link;
  }
  printf("Element %d not found\n",data);
  return start;
}
```

Linked lists: deletion (visualization)






THANK YOU