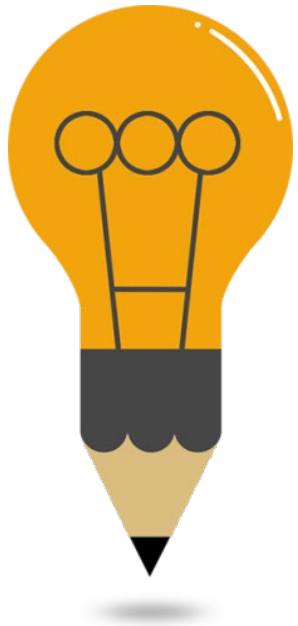
STRUKTUR DATA

Pertemuan 4

(Tim)

Agenda Pertemuan



1. Alokasi Memori (Lanjutan)
2. Single Linked List

Alokasi Memori

- **Dynamic Memory Allocation :** suatu prosedur dimana struktur data (seperti array) berubah selama program dieksekusi (*run time)* => ada 4 fungsi dari **<stdlib.h>** untuk melakukan alokasi memori:
  - Alokasi: **`malloc(), calloc(), realloc()`**
  - Dealokasi: **`free()`**

- Fasilitas ini memungkinkan user untuk membuat struktur data dengan ukuran dan panjang berapapun yang disesuaikan dengan kebutuhan di dalam program.

Fungsi `sizeof()`

- Output dari sizeof() adalah jumlah memori yang dialokasikan untuk tipe data tersebut (dalam byte)

```c
#include <stdio.h>
struct employee{
    char name[40];
    int id;
};
int main() {
   int myInt = 16;
   struct employee john;
   int arr[] = { 1, 2, 3, 4, 7 };
   printf("Size of variable myInt : %d\n",sizeof(myInt));
   printf("Size of variable john : %d\n",sizeof(john));
   printf("Size of variable arr : %d\n",sizeof(arr));
   printf("Size of int data type : %d\n",sizeof(int));
   printf("Size of char data type : %d\n",sizeof(char));
   printf("Size of float data type : %d\n",sizeof(float));
   printf("Size of double data type : %d\n",sizeof(double));
   return 0;
}
```

```
Size of variable myInt : 4
Size of variable john : 44
Size of variable arr : 20
Size of int data type : 4
Size of char data type : 1
Size of float data type : 4
Size of double data type : 8
```

Fungsi `malloc()`

- "malloc" atau "*memory allocation*" digunakan untuk mengalokasikan satu blok memory dengan ukuran tertentu secara dinamis
- Mengembalikan **`void*` (pointer bertipe void)**, perlu dikonversikan sesuai dengan data yang akan disimpan dalam blok memory tersebut

- **`ptr = (tipe_data_konversi*) malloc(jumlah_byte)`**
- Contoh:
  - **`int *ptr = (int*) malloc(100 * sizeof(int));`**

  => (ukuran dari int adalah 4 byte, fungsi malloc di sini akan mengalokasikan memori 400 bytes, dan pointer ptr akan menyimpan **alamat byte pertama dari memori yang dialokasikan**)

Penggunaan `malloc()`

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int i, n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    int* arr = (int*)malloc(n * sizeof(int));

    if (arr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
        printf("Memory successfully allocated using malloc.\n");
        for (i = 0; i < n; ++i) {
            arr[i] = i + 1;
        }
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", arr[i]);
        }
    }
    return 0;
}
```

Hasil:

Enter number of elements: 5

Memory successfully allocated using malloc.

The elements of the array are: 1, 2, 3, 4, 5,

Setelah memory dialokasikan, space tersebut dapat diakses sebagai array 1 dimensi.

Contoh:

```c
int *p = malloc(3* sizeof(int));
```

Inisialisasi nilai ke memory yg dialokasikan dengan cara:

```c
*p = 34;          atau     p[0] = 34;
*(p+1) = 23;               p[1] = 23;
*(p+2) = 10;               p[2] = 10;
```

Fungsi `free()`

- Jika bekerja dengan menggunakan memori yang dialokasikan secara dinamis, maka memori harus dibebaskan kembali setelah selesai digunakan untuk dikembalikan kepada sistem.
- Setelah suatu ruang memori dibebaskan, ruang tersebut bisa dipakai lagi untuk alokasi variabel dinamis lainnya.

Fungsi `free()`

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char *pblok;
    pblok = (char *) malloc(500 * sizeof(char));
    if (pblok == NULL)
        printf("Error on malloc");
    else {
        printf("OK, alokasi memori sudah dilakukan\n");
        printf("------\n");
        free(pblok);
        printf("Blok memori telah dibebaskan kembali\n");
    }
}
```

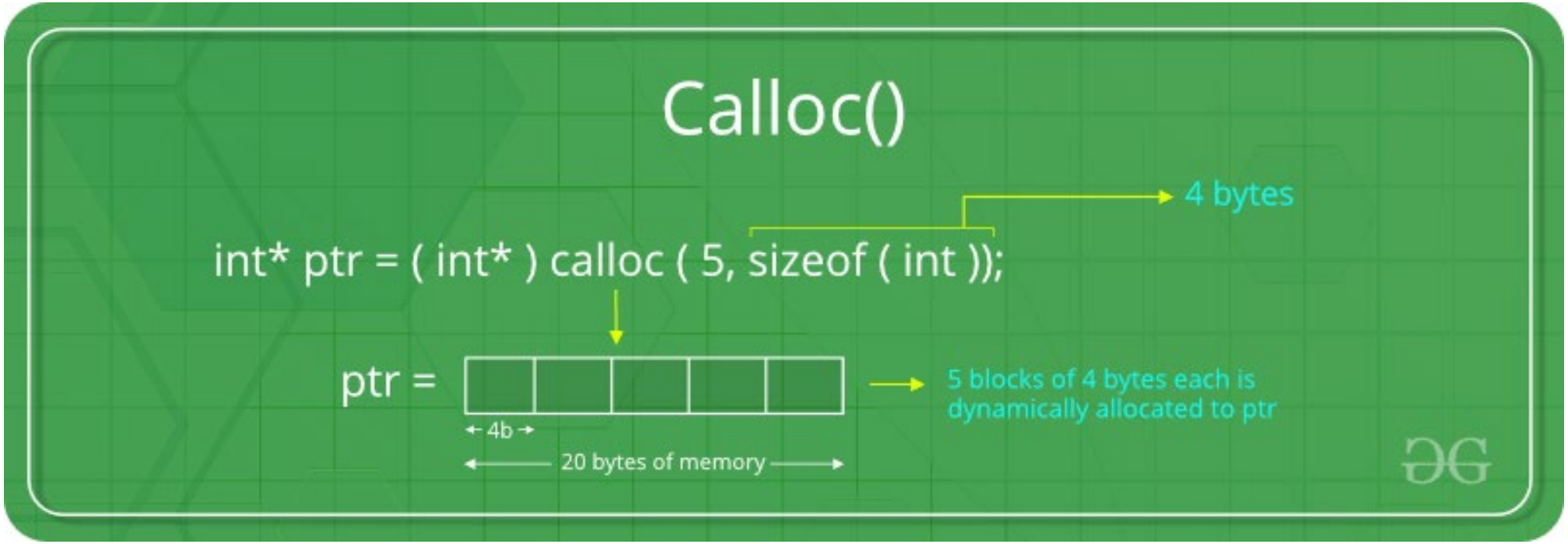
Fungsi `calloc()`

- **`calloc`** atau *"contiguous allocation"* digunakan untuk alokasi memory dinamis seperti `malloc`
- Sama seperti `malloc`, `calloc` juga return pointer bertipe void (**`void*`**)
- **Jika berhasil/sukses**, `calloc()` akan return sebuah pointer bertipe *void* yang dapat dikonversi ke pointer dengan tipe lain dan blok memory yang telah dialokasikan akan **terinisialisasi dengan nilai 0 (nol)**
- **Jika gagal**, fungsi akan return sebuah pointer NULL

- **`ptr = (tipe_data_konversi*) calloc(jumlah_blok, ukuran_masing2_blok)`**

- Contoh:
  - **`int *arr = (int *)calloc(5, sizeof(int));`**
    - **`calloc()`** akan mengalokasikan 5 blok integer dan menginisialisasi masing-masing blok dengan nilai 0

Fungsi `calloc()`



Fungsi **`calloc`** akan mengalokasi memory sebesar 5 blok integer masing-masing berukuran **`sizeof(int)`** yaitu 4 byte = total 20 byte, karena akan kita isi memory tersebut dengan integer, maka kita konversi dengan syntax **`(int*)`**. Kemudian, masing-masing blok tersebut akan otomatis terinisialisasi dengan nilai 0 (nol).

Perbedaan `malloc()` dan `calloc()`

- Tampilkan isi dari memory yang telah dialokasikan

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

int main()
{
    int n = 5;
    printf("Number of elements: %d\n", n);

    int *ptr_m = (int*)malloc(n * sizeof(int));

    printf("malloc :\n");
    for(int i = 0; i < n; i ++){
        printf("%d\n", ptr_m[i]);
    }

    int *ptr_c = (int*)calloc(n, sizeof(int));

    printf("\ncalloc :\n");
    for(int i = 0; i < n; i ++){
        printf("%d\n", ptr_c[i]);
    }

    free(ptr_m);
    free(ptr_c);

    return 0;
}
```

Output:

```
Number of elements: 5
malloc :
13057624
13053488
1868852841
1867543415
1400006007

calloc :
0
0
0
0
0
```

Blok memory yang dialokasikan masih berisi **nilai yang tersisa dari program dan operasi sebelumnya *(Garbage value)***

Blok memory dari `calloc()` telah terinisialisasi dengan 0

Perbedaan `malloc()` dan `calloc()`

`malloc()`	`calloc()`
1 parameter = ukuran	2 parameter = jumlah blok dan ukuran masing-masing blok
Isi/nilai dari blok memory yang dialokasikan belum terinisialisasi (belum ada nilainya)	Masing-masing blok memory telah terinisialisasi dengan 0 (nol)
malloc lebih cepat dibanding calloc (tentu saja karena selain mengalokasikan, calloc juga menginisialisasi nilai 0 ke setiap blok)	
	Mengapa calloc? menghindari buffer overflow (ketika kita alokasi memory, bisa saja alokasi memory kita sukses, tetapi sebenarnya memory fisik tidak cukup, seperti pada linux yang menerapkan Optimistic Memory Allocation), sehingga dengan kita inisialisasi 0 maka memastikan bahwa memory benar-benar tersedia. buffer overflow bisa menyebabkan crash program, karena ketika kita mau mengisi blok memory, jika tidak cukup, maka akan disimpan di memory yang berdekatan (meluap), bisa saja sedang dipakai program lain.

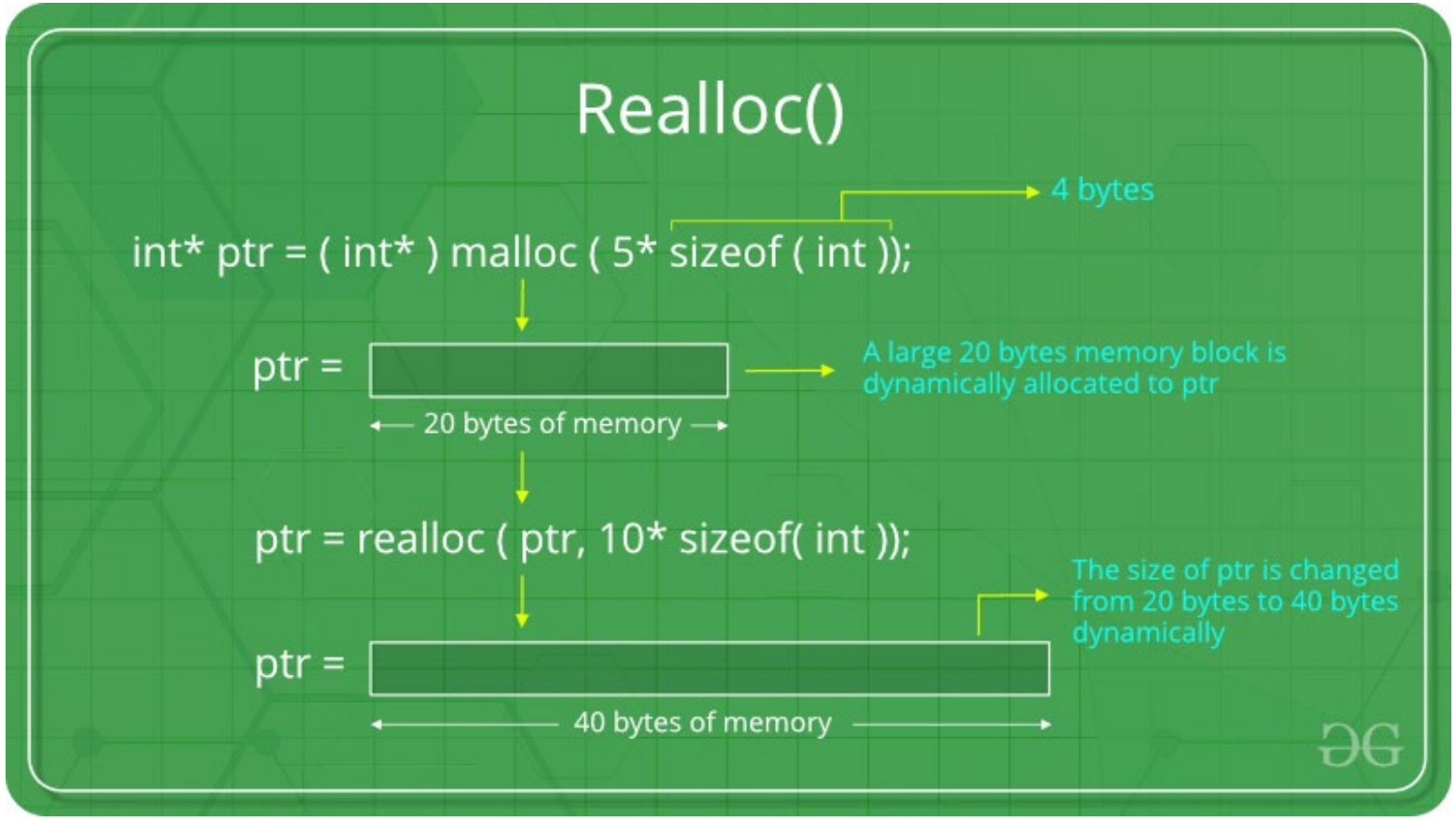
Fungsi `realloc()`

- **`realloc`** atau *"re-allocation"* digunakan untuk mengubah ukuran memori yang dialokasikan fungsi `malloc` dan `calloc`
- Jika memory yang sebelumnya dialokasikan tidak cukup/berlebih, `realloc` dapat digunakan untuk merealokasi memory secara dinamis
- **Jika berhasil**, `realloc` akan melakukan relokasi memory
- **Jika gagal**, fungsi akan return sebuah pointer NULL

- **`ptr = realloc(ptr, ukuran_baru)`**
  - Di mana **`ptr`** adalah pointer dari return fungsi **`malloc`** atau **`calloc`**

- Contoh:
- **`ptr = realloc(ptr, 10 * sizeof(int));`**
  - alokasi memory **`ptr`** dari **`malloc`** atau **`calloc`** akan diubah menjadi sebesar 40 byte

Fungsi `realloc()`



Misal kita punya pointer **`ptr`** hasil dari mengalokasikan memory menggunakan **`malloc`** sebesar 5 x 4 = 20 byte.

Ternyata alokasi tidak cukup, maka perlu realokasi memory (dalam hal ini menambahkan) secara dinamis. Kita ubah alokasi memory **`ptr`** dari 20 byte menjadi 10 x 4 = 40 byte.

Penggunaan fungsi `realloc()`

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

int main()
{
    int* ptr;
    int i, n = 5;
    ptr = (int*)calloc(n, sizeof(int));

    if (ptr == NULL) {
        printf("Out of memory.\n");
        exit(0);
    }
    else {
        printf("Alokasi memory sukses.\n");
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        printf("Elemen array adalah: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }

        n = 10;
        ptr = realloc(ptr, n * sizeof(int));
```

```
Alokasi memory sukses.
Elemen array adalah: 1, 2, 3, 4, 5,
Realokasi memory berhasil.
Elemen array adalah: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

Lanjutan:

```c
        n = 10;
        ptr = realloc(ptr, n * sizeof(int));

        printf("\nRealokasi memory berhasil.\n");

        for (i = 5; i < n; ++i) {
            ptr[i] = i + 1;
        }

        printf("Elemen array adalah: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }

        free(ptr);
    }
    return 0;
}
```

Penggunaan fungsi `realloc()`

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

int main()
{
   char *str;

   /* Inisialisasi alokasi memori */
   str = (char *) malloc(15);
   strcpy(str, "badanpusat");
   printf("String = %s\n", str);

   /* Re-alokasi memori */
   str = (char *) realloc(str, 35);
   strcat(str, "statistik.com");
   printf("String = %s\n", str);

   free(str);

   return(0);
}
```

```
String = badanpusat
String = badanpusatstatistik.com
```

`strcpy (var_dest, var_source)`:
mengcopy string `var_source` ke `var_dest`

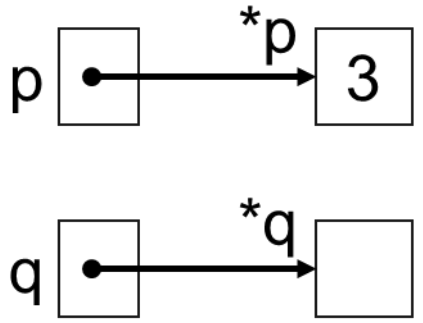
`strcat(var_dest, var_source)`:
menambahkan `var_source` di akhir `var_dest`

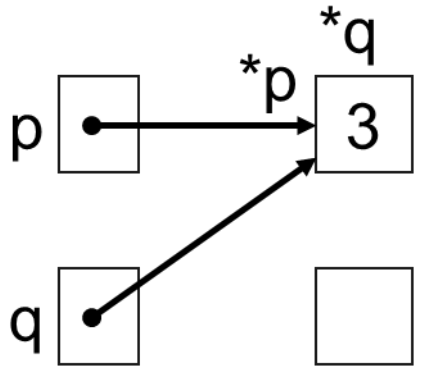
Latihan

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

int main()
{
    int *p, *q;
    p = (int*) malloc(sizeof(int));
    q = (int*) malloc(sizeof(int));

    *p = 3;

    free(q); // q di-free-kan dulu sebelum q = p
    q = p;

    printf("Nilai p = %d\n", *p);
    printf("Nilai q = %d\n", *q);

    printf("%d\n", p);
    printf("%d\n", q);

    free(p);
    return 0;
}
```

- Apakah output dari program tersebut?

```
Nilai p = 3
Nilai q = 3
p = 142896
q = 142896
```





Latihan

```c
#include <stdlib.h>

int main()
{
    int *ptr = (int *) malloc(sizeof(int));

    return 0;
}
```

```c
#include <stdlib.h>

int main()
{
    int *ptr = (int *) malloc(sizeof(int));
    free(ptr);
    return 0;
}
```

- Mana yang lebih tepat?

Memory leak : terjadi jika programmer mengalokasikan memory tetapi tidak mendealokasikannya.

Bukan masalah serius untuk aplikasi biasa jika exit() maka otomatis program akan membebaskan memory. Bagaimana dengan *service*, *daemon*, *server* yang selalu running?

Latihan

```c
#include <stdio.h>
#include <stdlib.h>
int main () {
  int *p, *q;
  p = (int*) malloc(sizeof(int));
  q = p;
  *q = 3;

  printf("%d %d", *p, *q);

  free(p);

  printf("%d %d", *p, *q);

  return 0;

}
```

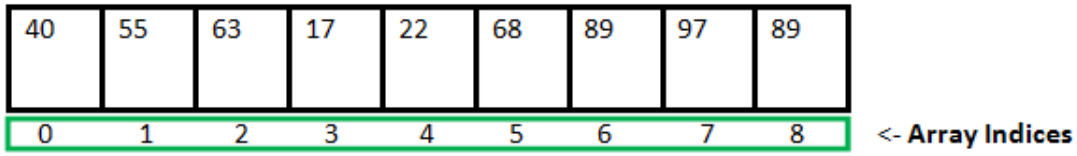
- Apa output dari program tersebut?

Jawab: Hanya akan menampilkan `printf()` pertama. Setelah `free()`, nilai *p (dan *q) tidak ada. Oleh sebab itu kita tidak dapat menggunakan *p atau *q

Hanya free() satu kali, pointer p atau q, tidak dua-duanya karena p dan q menyimpan alamat memory yang sama

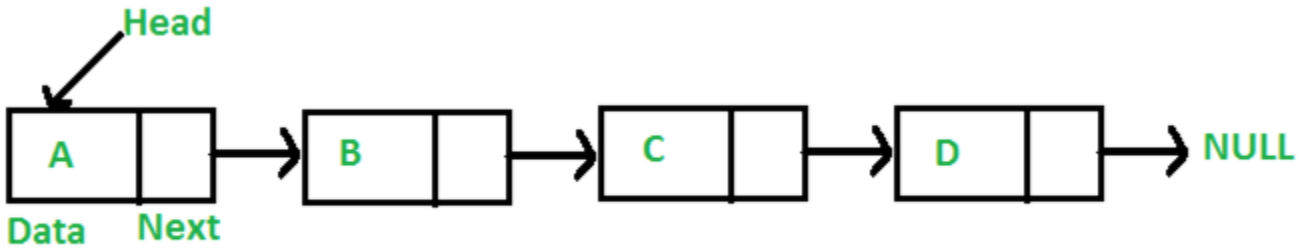
Linked List

- Sebuah struktur data seperti array yang berupa **sekumpulan node (simpul) yang saling terhubung secara linear dengan node lain melalui sebuah pointer**
- Node-node tersebut tidak disimpan secara berdampingan seperti array, tetapi terpencar-pencar di dalam memory membutuhkan pointer yang menghubungkan satu node ke node berikutnya (**pointer bertugas menyimpan address node selanjutnya**)



Array Length = 9
First Index = 0
Last Index = 8

(Array)



(Linked List)

<u>Representasi</u>

Elemen dalam array = node dalam linked list

Linked List vs Array

Mengapa menggunakan linked list daripada array untuk menyimpan data?

- Ukuran array adalah tetap (tidak dinamis). Alokasi memory terbuang jika array tidak diisi penuh, dan bermasalah ketika harus menambah ukuran yang ditetapkan di awal
  - linked list akan mengalokasikan memory untuk setiap elemennya secara terpisah dan hanya ketika dibutuhkan

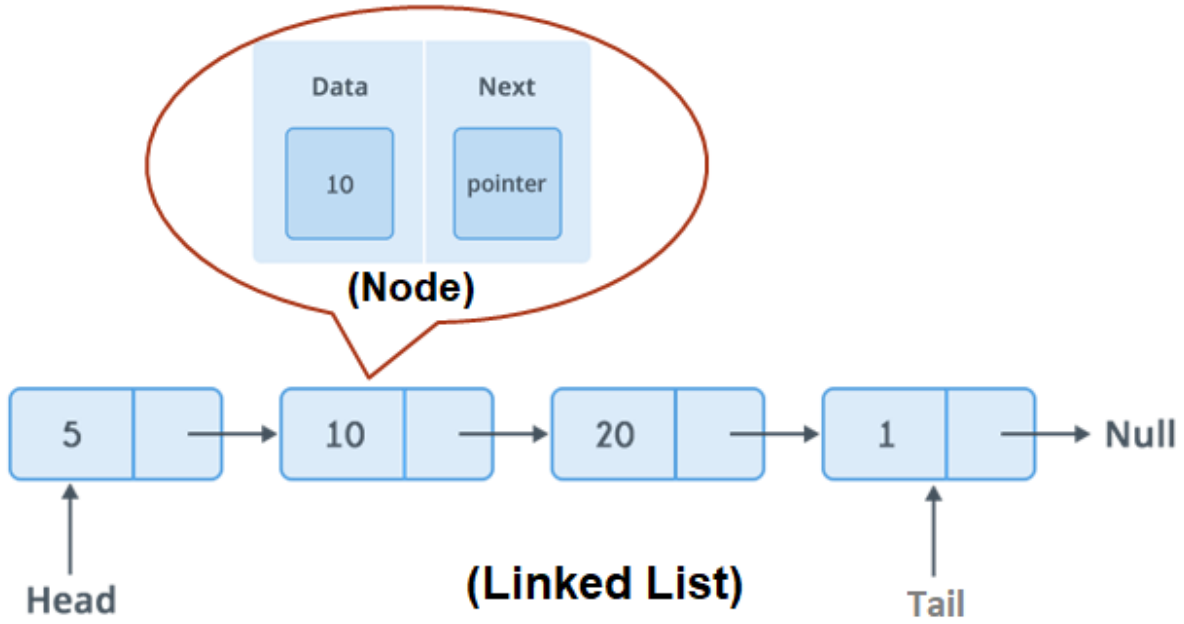
- Insert (sisip) elemen baru di awal ataupun tengah array membutuhkan usaha/komputasi yang besar (apalagi untuk array dengan jumlah elemen yang besar)
  - Misal: kita punya array nama mahasiswa berukuran 100, kita ingin sisipkan elemen di index ke 50, berarti kita harus menggeser satu per satu nilai di setiap index ke 51 sampai 100.

Linked List

- Setiap node terdiri dari 2 bagian:
  - **Data** berisi elemen data dalam node tersebut
  - **Pointer Next** berisi alamat memory node selanjutnya (untuk menghubungkan)

- Diawali dengan sebuah node **head** untuk menyimpan alamat awal dan diakhiri dengan node **tail** dengan pointer mengarah ke Null (menunjukan akhir dari sebuah list)

- Setiap node diimplementasikan secara dinamis (memory dialokasikan pada saat *runtime*)



Tipe Linked List

- **Single Linked List**
  - Pointer **Next** menyimpan alamat dari node berikutnya
- **Double Linked List**
  - Pointer **Next** menyimpan alamat dari node sebelumnya dan node berikutnya

Deklarasi Single Linked List

- Setiap node akan berbentuk **`struct`** dan memiliki satu buah field bertipe **`struct`** yang sama yang berfungsi sebagai pointer
- **Ingat**: cara mendeklarasikan **structure**

```
#include <stdio.h>

struct mahasiswa {
 char nim[25];
 char nama[25];
 int usia;
};
```

Deklarasi dan Akses:

```
struct mahasiswa mhs1;
struct mahasiswa mhs1 = {100, "Adi", 18};
printf("%s", mhs1.nama);
```

Variabel pointer : (yang menyimpan alamat memory struct)

```
struct mahasiswa mhs1, *p_mhs1;
struct mahasiswa mhs1 = {100, "Adi", 18};

p_mhs1 = &mhs1;

printf("%s", p_mhs1->nim);
printf("%s", p_mhs1->nama);
```

Deklarasi Single Linked List

- Setiap node akan berbentuk **`struct`** dan memiliki satu buah field bertipe **`struct`** yang sama berfungsi sebagai pointer

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

menyimpan alamat node setelahnya yang juga bertipe **`struct node`**, maka pointer **`next`** juga harus bertipe sama

(**Ingat**: pointer harus bertipe sama dengan nilai yang disimpan dalam alamat yang ditunjuk)

Membuat Node yaitu menggunakan Alokasi Memory Dinamis

```
struct mynode{
    int data;
    struct mynode *next;
};
struct mynode* head = NULL;
struct mynode* second = NULL;

head = (struct mynode*)malloc(sizeof(struct mynode));
second = (struct mynode*)malloc(sizeof(struct mynode));

head->data = 1;
head->next = second;

second->data = 2;
second->next = NULL;
```

Nama structure suatu node. Structure ini bisa disimpan sebagai global atau local.

Node `head = NULL` menunjukan linked list masih kosong

Alokasikan memory secara dinamis

Isi elemen data dan pointer next di setiap node

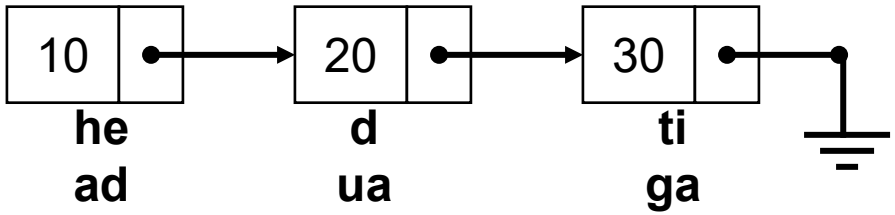
Akhir sebuah list (pointer node terakhir mengarah ke NULL)

Create Linked List

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

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

int main()
{
    struct Node* head = NULL;
    struct Node* dua = NULL;
    struct Node* tiga = NULL;
    head = (struct Node*)malloc(sizeof(struct Node));
    dua = (struct Node*)malloc(sizeof(struct Node));
    tiga = (struct Node*)malloc(sizeof(struct Node));

    head->data = 10;
    head->next = dua;

    dua->data = 20;
    dua->next = tiga;

    tiga->data = 30;
    tiga->next = NULL;

    printf("Isi dari linked list :\n");
    struct Node* n = head;
    while (n != NULL) {
        printf("%d\n", n->data);
        n = n->next;
    }
    return 0;
}
```

head, dua, tiga berisi alamat memory pertama node

```
Isi dari linked list :
10
20
30
```



Iterasi setiap node dalam sebuah linked list:

```c
node n;
n = head;
while(n != NULL){
    ....
    n = n->next;
}
```

node n adalah node bantuan. Lakukan **printf() field data** dari setiap node dari head, node berikutnya, dst sampai node tersebut NULL

Create Linked List

- Jika jumlah node dalam sebuah linked list ditentukan secara dinamis (misal dari input user), tidak ditentukan di awal. Bagaimana cara membentuk linked list?

```
struct Node* head = NULL;
struct Node* dua = NULL;
struct Node* tiga = NULL;

head = (struct Node*)malloc(sizeof(struct Node));
dua = (struct Node*)malloc(sizeof(struct Node));
tiga = (struct Node*)malloc(sizeof(struct Node));
```

Contoh statis (tidak dinamis):
Dideklarasikan secara statis 3 buah node dalam linked list tersebut

Hal-hal yang harus dilakukan:

1. Create node head
2. Insert node-node berikutnya sampai selesai
3. Delete node jika diperlukan

Create Linked List

1. Deklarasikan structure node yang berisi data dan pointer next

```
struct node{
    int value;
    struct node *next;
};

typedef struct node *mynode;
```

Untuk selanjutnya akan dipakai sampai slide terakhir sebagai global variable

`typedef`: untuk mendefiniskan tipe data baru atau memberi alias/nama baru suatu tipe data.

- Coding lebih rapi/bersih (menyederhanakan tipe data yang panjang dan complex)
- Tidak perlu menuliskan `struct` di semua tempat

Contoh lain penggunaan `typedef`:

```
typedef unsigned char HURUF;
HURUF b1, b2;

typedef long long int LLI;
int x = sizeof(LLI);
```

Create Linked List

2. Buat fungsi untuk membuat node (dibuat fungsi sendiri karena akan dipanggil berkali-kali)

```
mynode createNode(int nilai){
    mynode p;
    p = (mynode)malloc(sizeof(struct node));
    p->value = nilai;
    p->next = NULL ;
    return(p);
}
```

Bagaimana jika tidak mendeklarasikan **`typedef struct node *mynode`**?
Jawab:

```
struct node* createNode(int nilai){
    struct node* p;
    p = (struct node*)malloc(sizeof(struct node));
    p->value = nilai;
    p->next = NULL ;
    return(p);
}
```

Fungsi `free()` pada Linked List

- Membebaskan memory yang dialokasi untuk node tersebut

```
void free_node(mynode node){
     free(node);
}
```

Operasi Pada Linked List

1. **Menambahkan node (insert)**
   - Insert sebagai node awal (head) dari linked list
   - Insert sebagai node akhir (tail) dari linked list
   - Insert setelah node tertentu
   - Insert sebelum node tertentu
2. **Menghapus node (delete)**
   - Delete node pertama (head) dari linked list
   - Delete node terakhir (tail)
   - Delete pada node tertentu

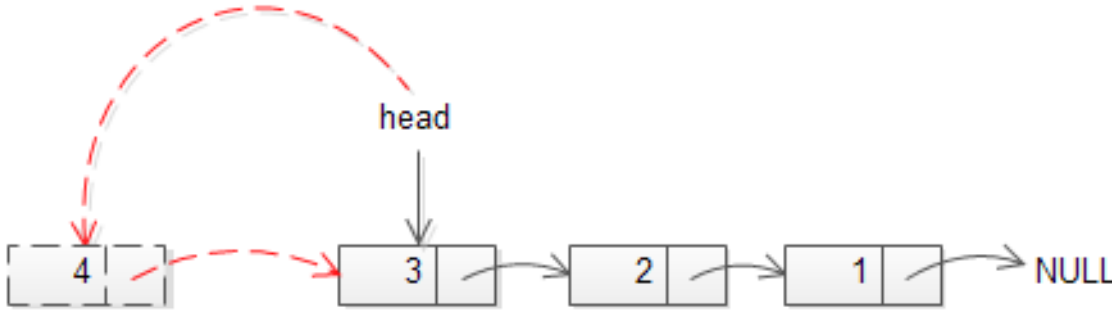
Ingat!!

Untuk mengakses array, kita memakai nama variabel array dan indexnya

Untuk mengakses linked list (node-node di dalamnya) yang diketahui adalah **node/pointer head**
(karena dari head kita bisa baca seluruh elemen dalam linked list)

Insert sebagai node awal (*head*) dari linked list

Contoh: Insert node dengan data = 4 sebagai head linked list 3->2->1 sehingga menjadi 4->3->2->1

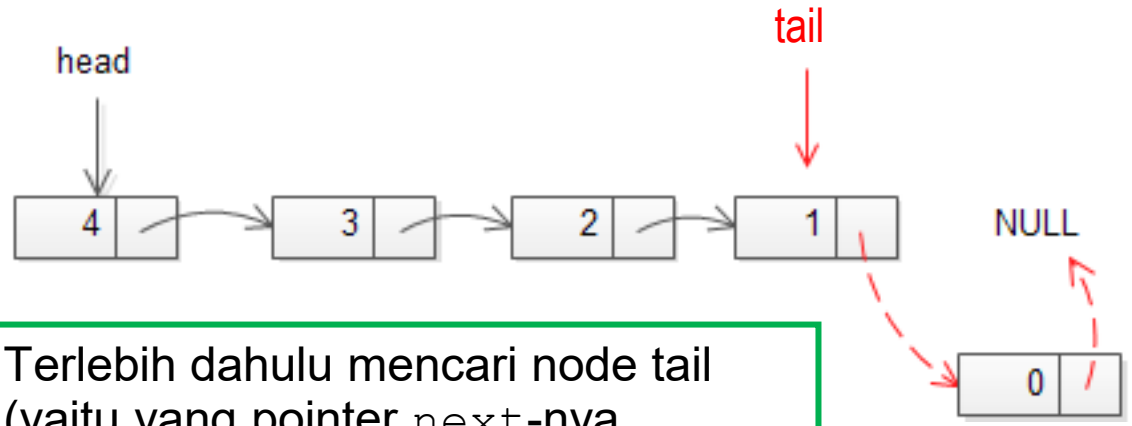



```
mynode insert_head(mynode head, int nilai){
    mynode new_node = createNode(nilai);
    new_node->next = head;
    head = new_node;

    return(head);
}
```

- Membuat node baru `new_node`
- Mengarahkan pointer `next` dalam `new_node` ke `head`, sehingga head yang baru adalah `new_node`
- Karena linked list sudah berubah, maka return-kan head yang baru

Insert sebagai node akhir (*tail*) dari linked list (Append)

Contoh: Insert node dengan data = 0 sebagai tail linked list **3->2->1** sehingga menjadi **3->2->1->0**

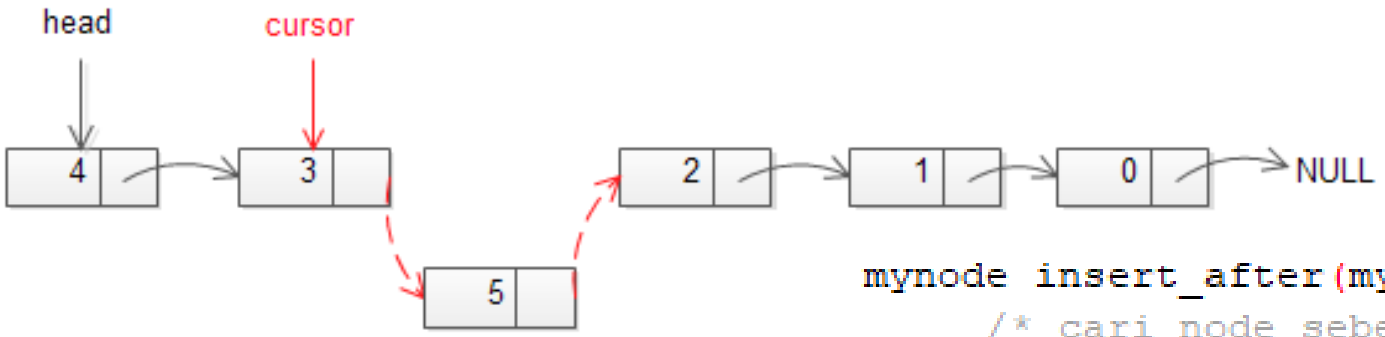


- Terlebih dahulu mencari node tail (yaitu yang pointer `next`-nya mengarah ke NULL) dengan melakukan iterasi dari head.
- Setelah ditemukan node tail, buat node baru `new_node`.
- Mengarahkan pointer `next` dari tail sebelumnya ke `new_node`, sehingga `new_node` menjadi tail.

```
mynode insert_tail(mynode head, int nilai){
    /* iterasi mencari node terakhir*/
    mynode tail = head;
    while(tail->next != NULL)
        tail = tail->next;

    /* buat node baru */
    mynode new_node = createNode(nilai);
    tail->next = new_node;

    return(head);
}
```

Insert setelah node tertentu (misal node dengan nilai tertentu)

Contoh: Insert node dengan data = 5 setelah node yang ditandai dengan "**`cursor`**" (data = 3)



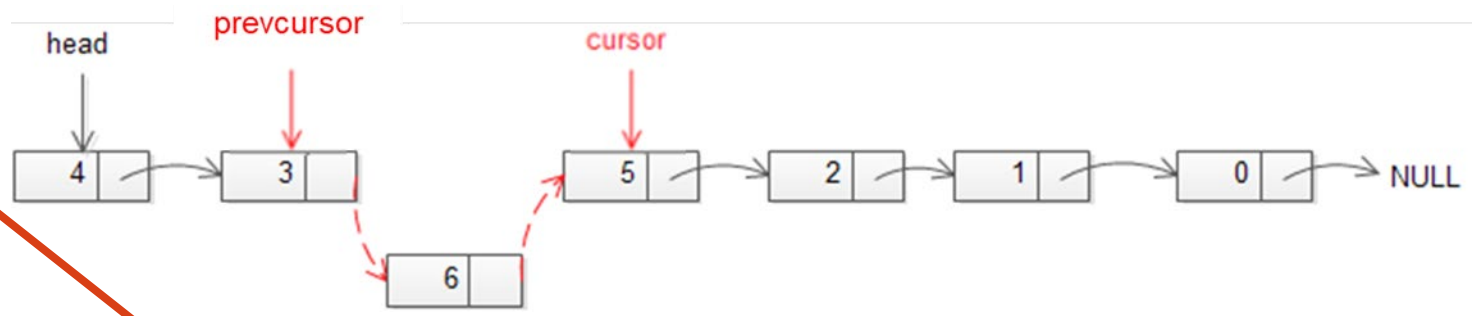
- Terlebih dahulu mencari node cursor (yaitu node yang mempunyai nilai `prev_nilai`) dengan iterasi dari node head.
- Jika sudah ditemukan, buat node baru `new_node`. Arahkan pointer next `new_node` ke alamat yang ditunjuk pointer next cursor. Dan arahkan pointer next cursor ke new_node.

```
mynode insert_after(mynode head, int nilai, int prev_nilai){
     /* cari node sebelumnya, starting from the first node*/
    mynode cursor = head;
    while(cursor->value != prev_nilai)
        cursor = cursor->next;

    mynode new_node = createNode(nilai);
    new_node->next = cursor->next;
    cursor->next = new_node;

    return(head);
}
```

Insert sebelum node tertentu (misal node dengan nilai tertentu)

Contoh: Insert node dengan data = 6 sebelum node yang ditandai dengan "`cursor`" (data = 5)

```
mynode insert_before(mynode head, int nilai, int next_nilai){
    if (head->value == next_nilai)
        head = insert_head(head, nilai);
    else
    {
        mynode cursor, prevcursor;
        cursor = head;
        do
        {   prevcursor = cursor;
            cursor = cursor->next;
        }
        while (cursor->value != next_nilai);

        mynode new_node = createNode(nilai);
        new_node->next = cursor;
        prevcursor->next = new_node;
    }

    return(head);
}
```

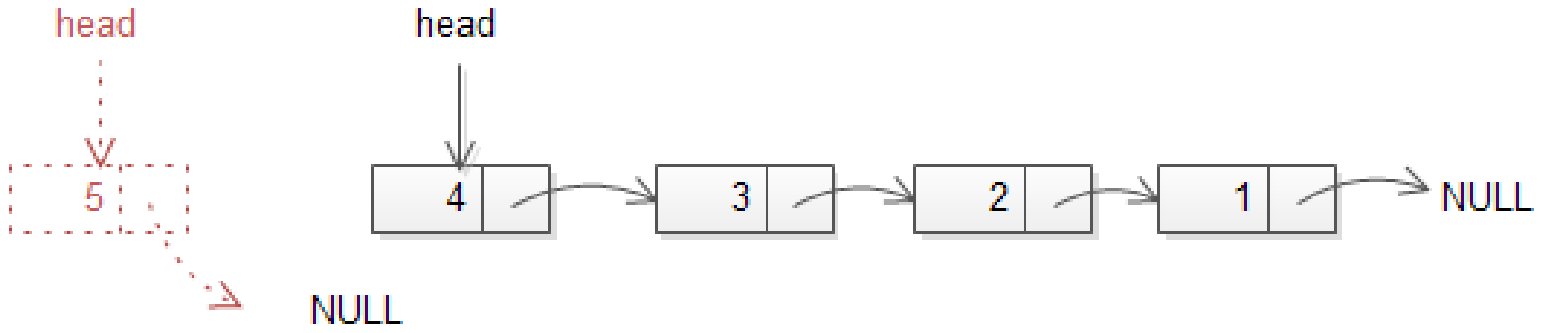


If: Jika data yang dicari berada pada awal Linked List atau head berarti gunakan fungsi sebelumnya

Else: Jika data yang dicari tidak berada pada awal Linked List

Menggunakan node bantuan `prevcursor` untuk menyimpan node sebelumnya, agar bisa dihubungkan dengan node baru

Delete node pertama (head) dari linked list



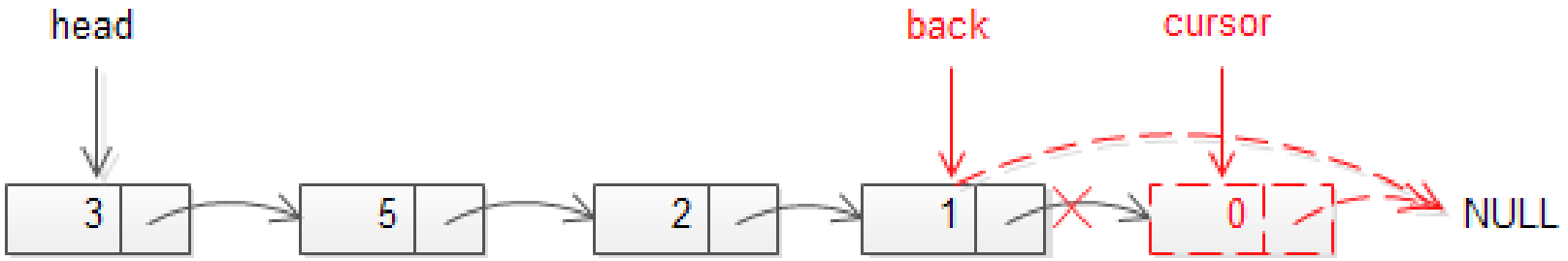

```
mynode remove_first(mynode head){
    if(head == NULL)
        return;

    mynode first = head;
    head = head->next;
    first->next = NULL;

    free(first);

    return(head);
}
```

Jika linked list empty (head == null) maka keluar dari fungsi.

Jika tidak:

- Node first diarahkan pada node head
- Node head diarahkan pada node setelah head
- Bebaskan node first (secara otomatis data pada node pertama terhapus)

Delete node terakhir (tail)



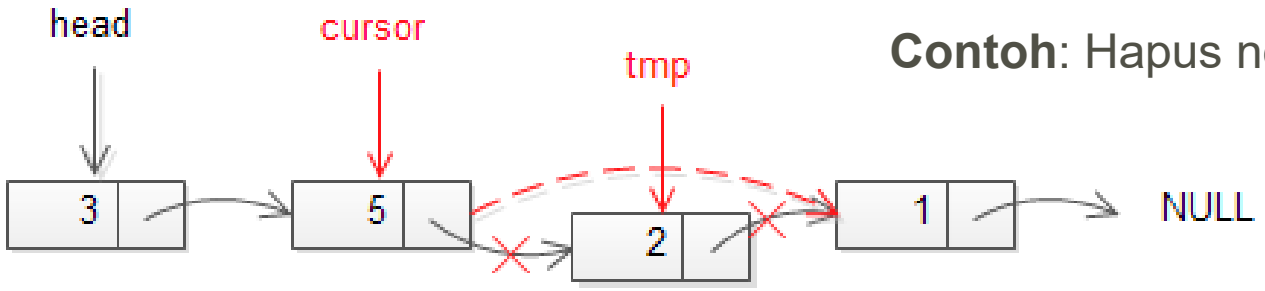

```
mynode remove_last(mynode head){
    if(head == NULL)
        return;

    mynode cursor = head;
    mynode back = NULL;
    while(cursor->next != NULL)
    {
        back = cursor;
        cursor = cursor->next;
    }
    if(back != NULL)
        back->next = NULL;

    free(cursor);

    return(head);
}
```

Jika linked list empty keluar dari fungsi.

Jika tidak:

- Iterasi dari head untuk mencari node terakhir (cursor) dan node sebelum terakhir (back)
- Setelah ditemukan, arahkan pointer next dari node back ke NULL dan bebaskan node terakhir (cursor).

Delete pada node tertentu (node dengan nilai tertentu)



Contoh: Hapus node dengan data = 2

Iterasi dari node head ke node terakhir (menggunakan cursor).

Sebelum *free*/membebaskan memory node yang ditunjuk cursor, buat dulu temporary node **tmp** untuk menyimpan alamat node selanjutnya (karena jika node yang ditunjuk cursor sudah di-*free*-kan, alamat next-nya sudah tidak ada)

```
mynode remove_middle(mynode head, int nilai){
    mynode cursor = head;
    while(cursor != NULL)
    {
        if(cursor->next->value == nilai)
            break; //keluar dari iterasi
        cursor = cursor->next;
    }

    if(cursor != NULL)
    {
        mynode tmp = cursor->next;
        cursor->next = tmp->next;
        tmp->next = NULL;
        free(tmp);
    }

    return(head);
}
```

Delete Linked List (Dispose)

- Penting untuk menghapus seluruh memory yang digunakan node-node pada linked list ketika sudah tidak digunakan/diperlukan

```
mynode dispose(mynode head)
{
    mynode cursor, tmp;
    if(head != NULL)
    {
        cursor = head->next;
        head->next = NULL;
        while(cursor != NULL)
        {
            tmp = cursor->next;
            free(cursor);
            cursor = tmp;
        }
    }
    head = NULL;

    return(head);
}
```

- Satu per satu menghapus node dari head ke terakhir. Iterasi dari node head ke node terakhir (menggunakan cursor).
- Sebelum *free*/membebaskan memory node yang ditunjuk cursor, buat dulu temporary node tmp untuk menyimpan alamat node selanjutnya (karena jika node yang ditunjuk cursor sudah di-*free*-kan, alamat next-nya sudah tidak ada).
- Setelah selesai iterasi, free-kan node head karena cursor dimulai dari head->next jadi head belum terbebaskan.

Ringkasan

So far, kita sudah mempunyai fungsi-fungsi:

- `createNode()`
- `insert_head()`
- `insert_tail()`
- `insert_after()`
- `insert_before()`
- `remove_first()`
- `remove_last()`
- `remove_middle()`
- `dispose()`

Fungsi-fungsi di atas bukan satu-satunya solusi, **algoritma bisa berbeda-beda** tetapi fungsi/tujuannya sama.

Implementasi di program C (coding)

- **Jangan lupa**: misalnya jika kita ingin memakai **void()** akan tetapi kita ingin mengubah nilai aslinya maka harus ***pass by reference*** (buka kembali slide pertemuan 3)
- Contoh: fungsi dalam `insert_head` harus diubah menjadi:

```
mynode insert_head(mynode head, int nilai){
    mynode new_node = createNode(nilai);
    new_node->next = head;
    head = new_node;

    return(head);
}
```

```
void insert_head(mynode *head, int nilai){
    mynode new_node = createNode(nilai);
    new_node->next = *head;
    *head = new_node;
}
int main(){
    mynode head = NULL;
    mynode dua = NULL;
    head = (mynode)malloc(sizeof(struct node));
    dua = (mynode)malloc(sizeof(struct node));

    head->value = 10;
    head->next = dua;
    dua->value = 20;
    dua->next = tiga;

    insert_head(&head, 99);

    return 0;
}
```

Bedakan jika fungsi yang kita buat bukan return value tapi void()!

Latihan

1. Buat fungsi untuk menampilkan nilai dari linked list! (tadi sudah dicontohkan)
2. Buat fungsi untuk menghitung jumlah node dalam sebuah linked list! (looping sama dengan no. 1)
3. Buat program untuk mengkonversi dari array 1D ke linked list!
   - user input ukuran array dan isinya, buat linked list dimulai dengan node head, kemudian loop tambahkan node-node baru ke node head tersebut
4. Buat fungsi untuk membalik nilai dari head ke tail! Contoh: 5->4->3->2->1 menjadi 1->2->3->4->5
   - hanya nilai saja, memory address (pointer node) tetap sama
   - buat temporary pointer node sebagai bantuan: prev, current, next dan loop dari head ke tail
5. Dari 3 fungsi delete (`remove_first, remove_last, remove_middle`), buat sebuah fungsi untuk menghapus node secara umum bisa di awal, di tengah, atau di akhir! (gabungkan 3 fungsi tersebut)
   - **`void remove_node(mynode head, int nilai)`** asumsi: nilai di dalam linked list berbeda (tidak ada yang sama)
6. Buat program untuk menyimpan data students berisi **`int nim, char nama[50]`** secara dinamis!

TERIMA KASIH