BAB II
Single Linked List

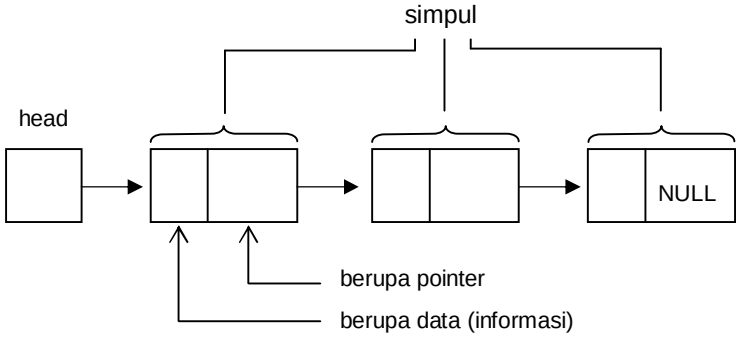
Tujuan

1. Memahami pengertian *linked list*, gunanya dan dapat mengimplementasikan dalam pemrograman
2. Dapat mengidentifikasi permasalahan-permasalahan pemrograman yang harus diselesaikan dengan menggunakan *linked list*, sekaligus menyelesaikannya

---

Dalam pemakaian sehari-hari istilah senarai berantai (*linked list*) adalah kumpulan linear sejumlah data. Contoh implementasi senarai berantai misalnya sebuah list yang berisi daftar belanjaan, yang berupa barang pertama, kedua, ketiga dan seterusnya. Untuk hari berikutnya, maka daftar tersebut bisa berubah sesuai dengan barang yang harus dibeli lagi atau barang yang tidak perlu dibeli lagi.

Contoh penerapan alokasi dinamis yaitu untuk membuat struktur data yang disebut senarai berantai *(linked list)*. Gambaran sebuah linked list ditunjukkan pada gambar 2.1 di bawah ini. Pada gambar tersebut terdapat 3 simpul (node). Namun dalam contoh nantinya akan terlihat bahwa jumlah simpul dalam daftar berantai dapat diperbanyak atau dikurangi.



Gambar 2.1 Bentuk sebuah daftar berantai *(linked list)*

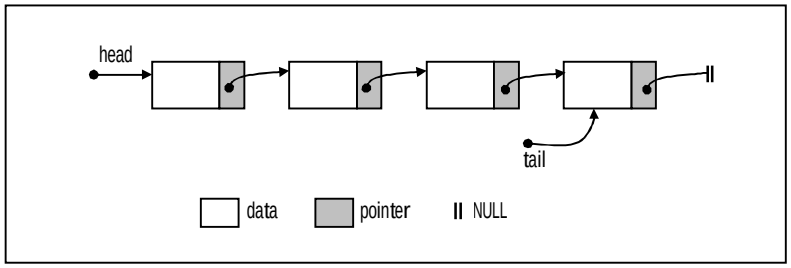
2.1 Definisi Linked List

Pengolahan data yang kita lakukan menggunakan komputer seringkali mirip dengan ilustrasi di atas, yang antara lain berupa penyimpanan data dan pengolahan lain dari sekelompok data yang telah terorganisir dalam sebuah urutan tertentu. Salah satu cara untuk menyimpan sekumpulan data yang kita miliki adalah menggunakan array. Telah kita bicarakan dalam bagian sebelumnya, keuntungan dan kerugian pemakaian array untuk menyimpan sekelompok data yang

banyaknya selalu berubah dan tidak diketahui dengan pasti kapan penambahan atau penghapusan akan berakhir.

2.2 Single Linked List

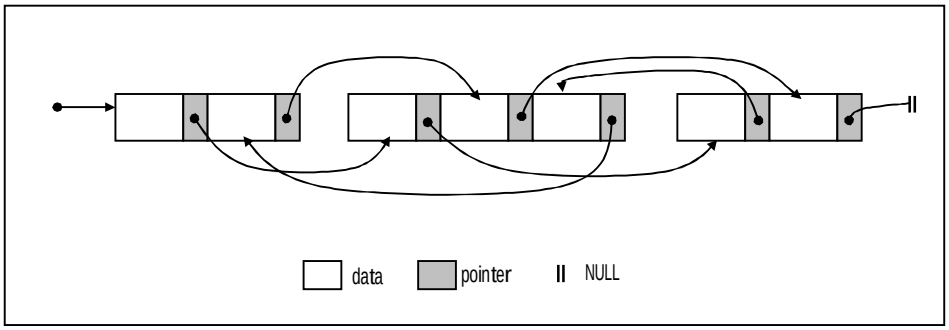
Single linked list atau biasa disebut linked list terdiri dari elemen-elemen individu, dimana masing-masing dihubungkan dengan pointer tunggal. Masing-masing elemen terdiri dari dua bagian, yaitu bagian data/informasi yang disimpan dan bagian pointer yang disebut dengan pointer *next*. Dengan menggunakan struktur *two-member* seperti ini, linked list dibentuk dengan cara mengarahkan pointer *next* dari suatu elemen ke elemen yang mengikutinya seperti gambar 2.2. Pointer *next* pada elemen terakhir merupakan NULL, yang menunjukkan akhir dari suatu list. Elemen pada awal suatu list disebut *head*, dan elemen terakhir dari suatu list disebut *tail*.



Gambar 2.2 Elemen yang Dihubungkan Bersama Dalam Bentuk Linked List

Untuk mengakses elemen dalam linked list, dimulai dari **head** dan menggunakan pointer **next** dari elemen selanjutnya untuk berpindah dari elemen ke elemen berikutnya sampai elemen yang diminta dicapai. Dengan single linked list, list dapat dilintasi hanya satu arah dari head ke tail karena masing-masing elemen tidak terdapat link dengan elemen sebelumnya. Sehingga, apabila kita mulai dari head dan berpindah ke beberapa elemen dan berharap dapat mengakses elemen sebelumnya, kita harus mulai dari head.

Secara konseptual, linked list merupakan deretan elemen yang berdampingan. Akan tetapi, karena elemen-elemen tersebut dialokasikan secara dinamis (menggunakan *malloc*), sangat penting untuk diingat bahwa kenyataannya, linked list akan terpencar-pencar lokasinya di memori seperti Gambar 2.3. Pointer dari elemen ke elemen berarti sebagai penjamin bahwa semua elemen dapat diakses.



Gambar 2.3 Elemen Pada Linked List Dihubungkan Secara Terpencar-Pencar pada Alamat Memori

2.2.1 Representasi Simpul (*Node*)

Struktur node pada linked list merupakan suatu simpul(node) yang berisi pointer ke suatu data yang merupakan data dirinya sendiri. Model struktur dari linked list tersebut dalam C adalah sebagai berikut:

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

Dalam hal ini, tipe `Node` berisi :

- Informasi berupa `data`, serta
- Pointer bernama `next` yang menunjuk ke obyek bertipe `Node`

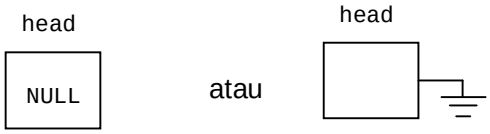
dilanjutkan dengan deklarasi global dari pointer ke struktur (pointer to struct) di atas sebagai berikut:

```
Node *head = NULL;
Node *p;
```

Dengan adanya pendeklarasian `head` sebagai sebuah variabel bertipe pointer yang menunjuk ke obyek bertipe `Node`) melalui pernyataan:

```
Node *head = NULL;
```

maka `head` akan berisi `NULL` (artinya linked list belum memiliki simpul)



Gambar 2.4 `head` yang berisi `NULL`

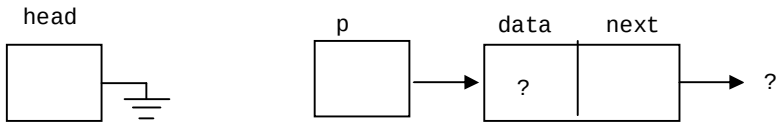
Catatan : NULL adalah konstanta yang didefinisikan pada file `stdio.h`

Selain itu juga dideklarasikan variabel `p` sebagai sebuah pointer to Node juga (tanpa inisialisasi) yang akan digunakan sebagai pointer yang menunjuk kepada simpul yang akan dibuat.

Pembentukan simpul pertama dilakukan melaui serangkaian pernyataan berikut :

```
1) p = (Node *) malloc(sizeof(Node));
2) p->data = 11511;
3) p->next = head;
4) head = p;
```

Sesudah pernyataan pertama (dengan anggapan alokasi memori berhasil dilakukan) maka terbentuk diagram sebagai berikut :



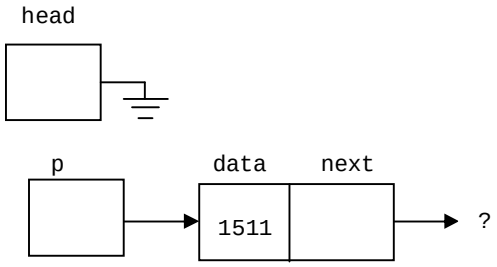
Gambar 2.5 Pembentukan simpul pertama

Pada gambar di atas tanda ? menyatakan isinya belum dispesifikasikan. Pernyataan

```
p->data = 11511;
```

akan menyebabkan :

- field `data` berisi 11511

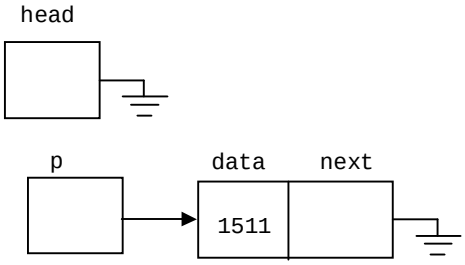


Gambar 2.6 Simpul pertama yang telah diisi field informasinya

Selanjutnya pernyataan

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

akan menyebabkan field `next` yang ditunjuk oleh `p` akan diisi dengan `head` (yaitu NULL). Hasilnya adalah seperti pada Gambar 12.6

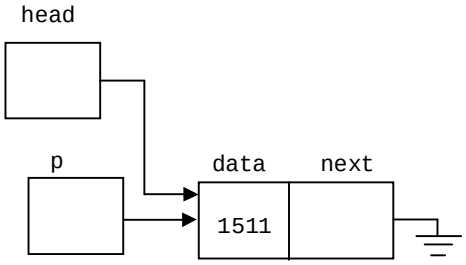


Gambar 2.7 Simpul pertama yang telah diisi field pointer next-nya

Pernyataan berikutnya yaitu

```
head = p;
```

dipakai untuk mengisikan nilai pada `p` ke `head`. Sebagai akibatnya, `head` akan menunjuk ke simpul yang ditunjuk `p`. Hasilnya ditunjukkan pada Gambar 2.8



Gambar 2.8 Mengarahkan `head` agar menunjuk ke simpul baru

2.2.2 Alokasi Simpul

Pengalokasian memory secara dinamis memungkinkan untuk melakukan pemesanan memory berulang-ulang sesuai dengan kebutuhan. Untuk mengurangi duplikasi kode, maka bagian dari program yang melakukan pemesanan memory dan memasukkan data ke simpul yang baru dialokasikan akan dibuat dalam sebuah fungsi. Sehingga, fungsi untuk mengalokasikan sebuah simpul baru, fungsi `allocate_node()` menggunakan `malloc()` untuk mendapatkan memori aktual, yang akan memasukkan data/informasi pada field `data` serta menginisialisasi `next` dengan `NULL`.

Untuk melihat kemungkinan alokasi memori gagal, maka fungsi `allocate_node` menghasilkan 0, bila berhasil maka menghasilkan 1. Fungsi dari alokasi node adalah sebagai berikut :

```
int allocate_node() {
      int nilai;

      printf("Nilai yang akan disimpan dalam node baru : ");
      scanf("%d", &nilai);
      p = (Node *) malloc(sizeof(Node));
      if(p == NULL)
            return 0;
      p->data = nilai;
      p->next=NULL;
      return 1;
}
```

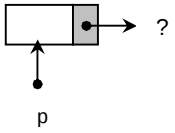
Untuk inisialisasi linked list (belum ada simpul sebelumnya), maka setelah alokasi untuk node pertama maka ditambahkan statement sebagai berikut:

```
head = p;
```

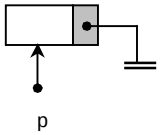
untuk mengarahkan pointer `head` agar menunjuk ke data pertama.

Ilustrasi dari fungsi i `allocate_node()` adalah sebagai berikut

```
p = (Node *) malloc(sizeof(Node));
p->data = nilai;
```

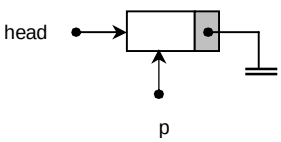



```
p->next = NULL;
```



Untuk menginisialisasi linked list setelah alokasi untuk node pertama ilustrasinya adalah sbb:

```
head = p;
```

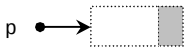


Sama dengan operasi pemesanan memori, operasi pembebasannya juga dilakukan berulang-ulang. Oleh karena itu kode-kode yang diperlukan untuk membebaskan memory dibuat dalam fungsi free_node(). Fungsi `free_node()` menggunakan fungsi `free()` untuk membebaskan lokasi memori yang sebelumnya dialokasikan (sehingga tempat yang ada dapat dipakai untuk alokasi lainnya). Akan tetapi, pointer yang menjadi parameter dalam fungsi `free()` tidak otomatis menjadi `NULL`, tetapi tetap akan menunjuk ke node yang sudah tidak ada. Oleh karena itu pointer tersebut secara eksplisit harus diassign dengan `NULL`.

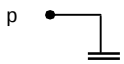
```
void free_node(Node *p) {
     free(p);
     p = NULL;
}
```

Ilustrasi dari fungsi `free_node()`dapat dilihat pada gambar berikut :

```
free(p);
```




```
p = NULL;
```



2.2.3 Operasi pada Linked List

Pada sub bab ini akan dijelaskan mengenai operasi yang terpenting pada linked list, yaitu menambahkan node (insert) dan menghapus node (delete).

2.2.3.1 Insert

Fungsi insert pada linked list meliputi :

- insert sebagai node awal (`head`) dari linked list
- insert setelah node tertentu
- insert sebelum node tertentu
- insert sebagai node akhir (`tail`) dari linked list

Secara umum algoritma untuk insert adalah :

1. Siapkan node baru yang akan disisipkan
2. Cari posisi tempat node baru akan disisipkan
3. Sambungkan dengan *existing linked list*

Insert sebagai node awal (head) dari linked list

Algoritma untuk insert node di awal adalah sbb :

1. Siapkan node yang akan disisipkan (panggil fungsi `allocate_node()`)
   a. Pesan memori dan assign ke p

b. Jika sukses, masukkan informasi ke bagian `p->data`

2. Arahkan pointer next dari p (`p->next`) kepada node yang ditunjuk oleh `head` (node yg semula jadi node pertama)

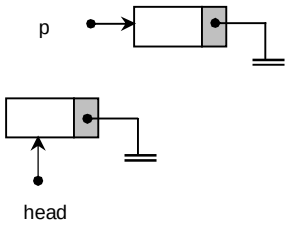
3. Arahkan pointer head agar menunjuk data baru (p) tsb.

Statement kelanjutan dengan deklarasi seperti di atas untuk insert sebagai node awal (`head`) dari linked list adalah sebagai berikut:

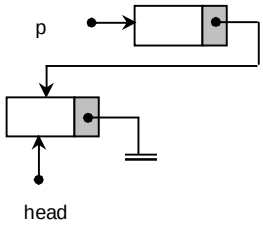
```
void sisip_awal() {
   p->next=head;
   head=p;
}
```

ilustrasi dari fungsi diatas adalah sebagai berikut:

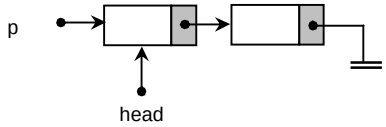
kondisi awal



`p->next=head;`



`head = p;`



Insert setelah node tertentu

Algoritma untuk insert setelah node tertentu adalah sbb :

1. Siapkan node yang akan disisipkan (panggil fungsi `allocate_node()`)
   a. Pesan memori dan assign ke p
   b. Jika sukses, masukkan informasi ke bagian `p->data`
2. Cari posisi node yang akan disisipi pada posisi setelahnya (gunakan pointer bantuan `after` untuk menandai)

a. Masukkan data yang akan disisipi posisi setelahnya (misalnya simpan di var x)

b. Baca mulai dari `head->data` sampai dengan ketemu data yang sama dengan `x.`

- Selama data yang dibaca belum habis, setiap kali data tidak ditemukan update posisi `after` `(after = after -> next).`
- Jika data telah habis terbaca dan tidak data yang dicari tidak ditemukan, maka tampilkan pesan kemudian exit.

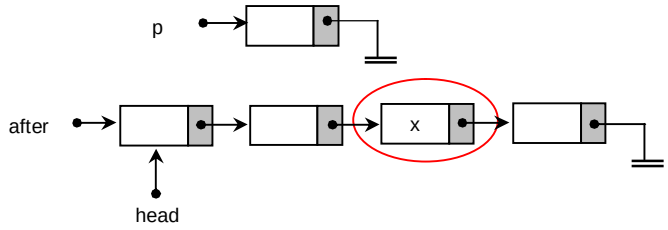
3. Jika data ketemu, arahkan p->next pada node sesudah `after` dan arahkan pointer `after->next` supaya menunjuk p.

Untuk melakukan insert setelah node tertentu dari linked list diperlukan sebuah pointer bantuan berupa pointer `after` sebagai penanda node yang akan disisipi setelahnya. Statemennya adalah sebagai berikut:

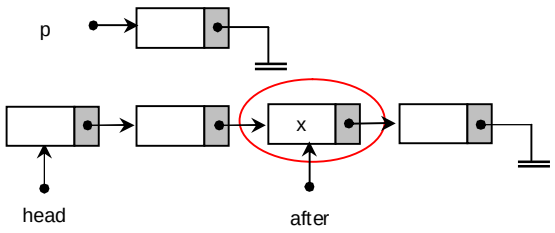
```
void sisip_after(int x) {
  Node *after;

  after = head;
  while(after->data != x)
  {
    if(after->next == NULL) {
      printf("Nilai %d tdk ada dlm list\n", x);
      exit(0);
    }
    else
      after = after->next;
  }
  p->next = after->next;
  after->next = p;
}
```

ilustrasi dari fungsi di atas adalah sebagai berikut:

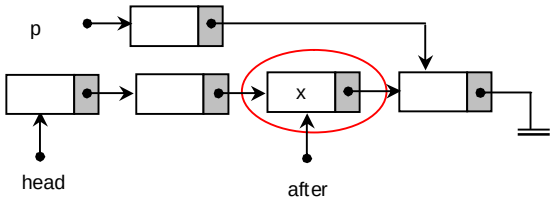
```
after = head;
```



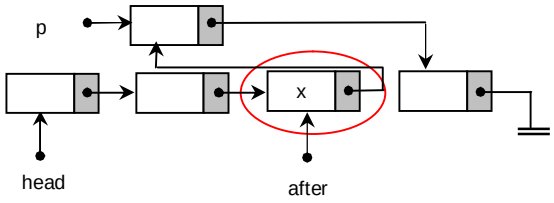

```
while(after->data != x) {
    if(after->next == NULL) {
      printf("Nilai %d tdk ada dlm list\n", x);
      exit(0);
    }
    else
      after = after->next;
}
```




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




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



Langkah-langkah untuk proses di atas adalah sebagai berikut:

1. Pointer `next` dari simpul baru (p) diarahkan menunjuk ke simpul setelah simpul tertentu (yang berisi data yang dicari)
2. Pointer `next` dari simpul yang telah ditandai dengan pointer `after` diarahkan menunjuk ke simpul baru (p)

Insert sebelum node tertentu

Algoritma untuk insert setelah node tertentu adalah sbb :

1. Siapkan node yang akan disisipkan (panggil fungsi `allocate_node()`)
   a. Pesan memori dan assign ke p
   b. Jika sukses, masukkan informasi ke bagian `p->data`
2. Cari posisi node yang akan disisipi pada posisi sebelumnya (gunakan pointer bantuan `before` dan `prevbefore` untuk menandai)
   a. Masukkan data yang akan disisipi posisi sebelumnya (misalnya simpan di var x)
   b. Baca mulai dari `head->data` sampai dengan ketemu data yang sama dengan `x` atau data tidak ditemukan & `exit()`. Selama data belum habis, setiap kali data tidak ditemukan update posisi `before` dan `prevbefore` (`prevbefore = before; before = before -> next`)

3. Sambungkan node baru dengan existing linked list

a. Jika data ketemu, arahkan p->next pada node yang ditendai `before`

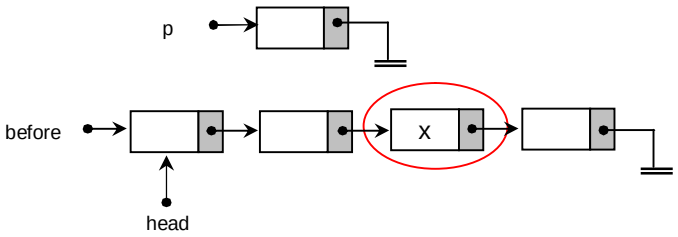
b. Arahkan pointer `prevbefore->next` supaya menunjuk p.

Untuk melakukan insert sebelum node tertentu dari linked list diperlukan dua buah pointer bantuan berupa pointer `before` sebagai penanda node yang akan disisipi sebelumnya dan `prevbefore` sebagai penanda node yang berada sebelum node yang ditunjuk oleh pointer `before`. Statemennya adalah sebagai berikut:

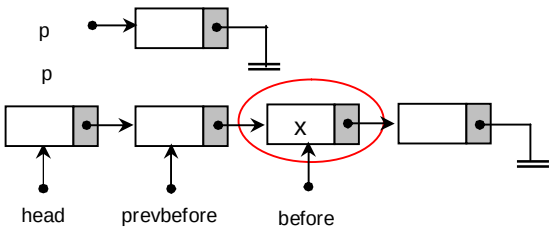
```
   void sisip_before(int x) {
      Node *before, *prevbefore;

      if(head->data == x)
         sisip_awal();
      else {
         before = head;
         do {
            prevbefore = before;
            if(before->next == NULL) {
               printf("Nilai %d tdk ada dlm list\n", x);
               exit(1);
            }
            else
               before = before->next;
         } while(before->data != x);
         p->next = before;
         prevbefore->next = p;
      }
}
```

ilustrasi dari fungsi diatas adalah sebagai berikut:

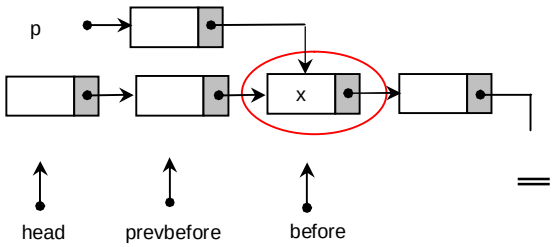
`before = head;`



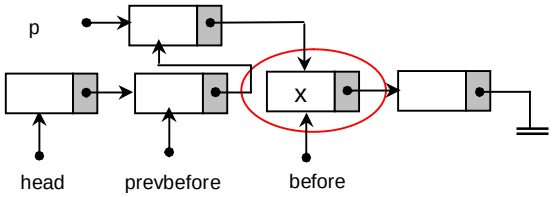

```
do {
   prevbefore = before;
   before = before->next;
} while(before->data != x);
```



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




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



Langkah-langkah untuk proses di atas adalah sebagai berikut:

1. Telusuri list sampai simpul yang dicari (yang akan disisipi sebelumnya) ketemu, tandai dengan pointer `before`. Tandai juga simpul sebelumnya dengan pointer `prevbefore`.
2. Lakukan penyisipan sebelum simpul yang ditunjuk oleh `before` tersebut

Insert di akhir (tail) dari linked list

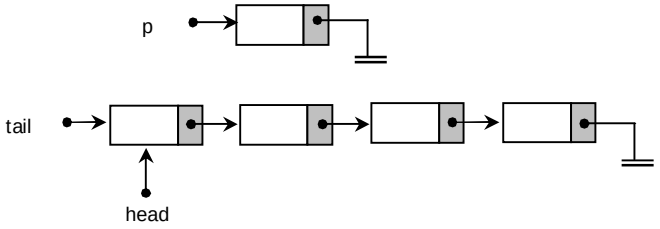
1. Siapkan node yang akan disisipkan (panggil fungsi `allocate_node()`)
   a. Pesan memori dan assign ke p
   b. Jika sukses, masukkan informasi ke bagian `p->data`
   c. Set pointer `p->next = NULL` (karena akan menjadi data terakhir)
2. Cari posisi node terakhir dan tandai pointer `tail`
   a. Baca mulai dari `head` sampai dengan ketemu node yang pointer next-nya = NULL, tandai dengan `tail` Setiap kali node belum berhasil ditemukan update posisi `tail`
3. Sambungkan node baru dengan existing linked list
   a. Jika data ketemu, arahkan `tail->next` pada p
   b. Update posisi `tail` agar menunjuk ke data terakhir.

Untuk melakukan insert di akhir dari linked list diperlukan sebuah pointer bantuan berupa pointer `tail` sebagai penanda node yang terakhir. Statemennya adalah adalah sebagai berikut:

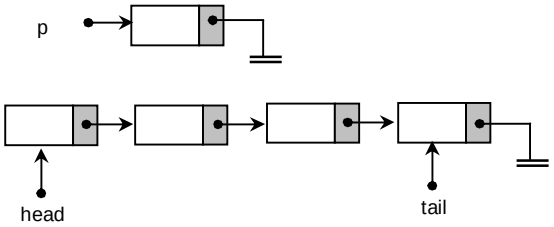
```
  void sisip_akhir(){
     Node *tail;

     tail = head
     while(tail->next != NULL)        //blm sampe list terakhir
        tail = tail->next;
     tail->next = p;
     tail = tail->next;
     }
  }
```

ilustrasi dari fungsi diatas adalah sebagai berikut:

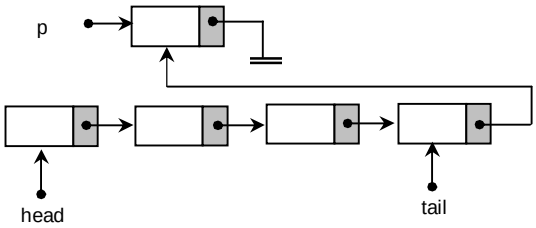
```
tail = head;
```



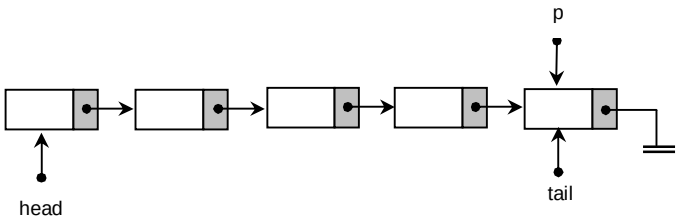

```
while (tail->next != NULL)
      tail = tail->next;
```




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




```
tail = tail->next;
```



Langkah-langkah untuk proses di atas adalah sebagai berikut:

1. Telusuri list sampai elemen terakhir (tail->next == NULL)
2. Lakukan penyisipan setelah elemen terakhir

2.2.3.2 Delete

Fungsi delete pada linked list meliputi :

- delete simpul pertama (head) dari linked list
- delete setelah simpul tertentu
- delete simpul terakhir

Secara umum, algoritma untuk melakukan operasi delete adalah ;

1. Cari posisi node yang akan di-delete
2. Sambungkan node agar linked list tidak terputus
3. Bebaskan lokasi dan NULL-kan pointernya

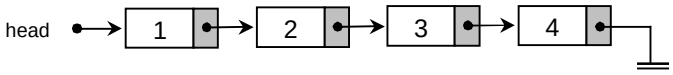
Penghapusan Simpul Pertama:

Algoritmanya sbb :

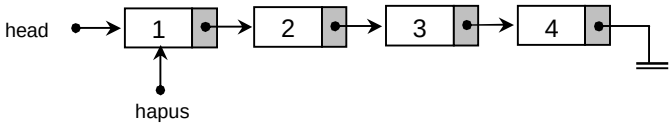
1. Posisi node yang akan ditunjuk adalah posisi yang disimpan oleh head, tandai dengan pointer hapus
2. Arahkan head agar menunjuk ke node sesudah node yang ditunjuk oleh hapus
3. Bebaskan lokasi yang ditunjuk oleh hapus dan NULL-kan pointer tsb

Untuk melakukan delete simpul pertama dari linked list diperlukan sebuah pointer bantuan berupa pointer hapus sebagai penanda node yang akan dihapus.

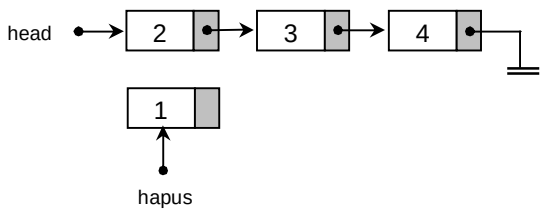
kondisi awal



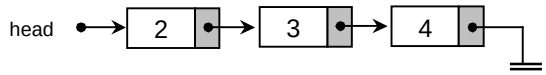

```
hapus = head;
```




```
head = hapus->next;
free_node(hapus);
```



kondisi akhir setelah penghapusan satu simpul di awal



Langkah-langkah untuk proses di atas adalah sebagai berikut:

1. Pointer `hapus` diarahkan pada data ke-1
2. Pointer `head` diarahkan pada data ke-2
3. Panggil fungsi `free_node(hapus)` untuk membebaskan lokasi yang ditunjuk oleh pointer `hapus` (data ke-1 terhapus) sekaligus mengarahkan pointer `hapus` ke `NULL`

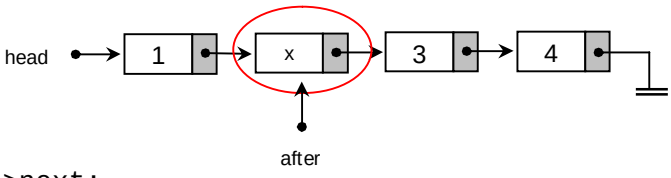
Penghapusan Setelah Simpul Tertentu

Algoritmanya sbb :

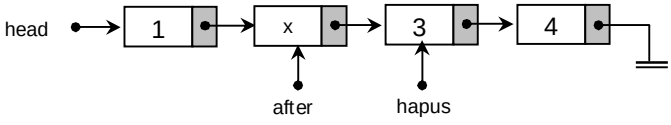
1. - Dimulai dari node pertama, cari posisi node yang berisi data yang akan dihapus node sesudahnya, tandai dengan pointer `after`.
   - Posisikan pointer `hapus` pada node yang terletak sesudah node yang ditunjuk pointer `after`
2. Arahkan `after->next` agar menunjuk ke node sesudah node yang ditunjuk oleh `hapus`
3. Bebaskan lokasi yang ditunjuk oleh `hapus` dan NULL-kan pointer tsb

Untuk melakukan delete setelah node tertentu dari linked list diperlukan dua buah pointer bantuan berupa pointer `after` sebagai penanda node yang akan dihapus setelahnya dan pointer `hapus` untuk menandai simpul yang akan dihapus

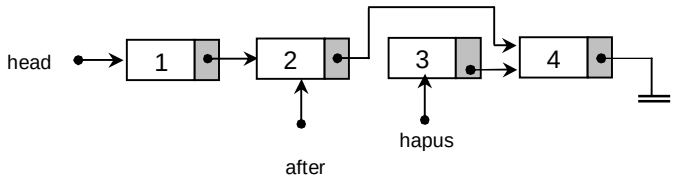
```
after = head;
while(after->data != x){
  if(after->next == NULL)
    exit(0);
  else
      after = after->next;
}
```



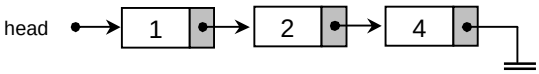

```
hapus = after->next;
```



```
after->next = hapus->next;
```




```
free_node(hapus);
```



Langkah-langkah untuk proses di atas adalah sebagai berikut:

1. Arahkan pointer `after` s/d simpul yang berisi data yang dicari, yang akan dihapus simpul sesudahnya
2. Pointer `hapus` diarahkan pada `after->next`
3. Arahkan pointer `after->next` pada `hapus->next`
4. Panggil fungsi `free_node(hapus)` untuk membebaskan lokasi yang ditunjuk oleh pointer `hapus` sekaligus mengarahkan pointer `hapus` ke `NULL`

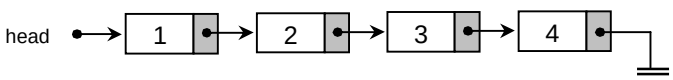
Penghapusan Simpul Terakhir

Untuk melakukan delete node terakhir dari linked list diperlukan dua buah pointer bantuan berupa pointer `prevhapus` sebagai penanda node yang akan dihapus setelahnya dan pointer `hapus` untuk menandai simpul yang akan dihapus

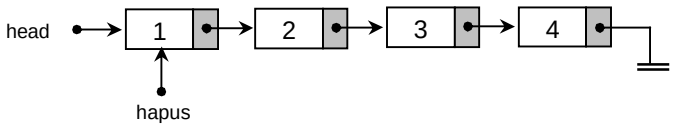
Algoritmanya sbb :

1. cari posisi node yang akan dihapus dimulai dari node pertama, tandai dengan pointer hapus. Setiap kali hendak mengupdate posisi `hapus` simpan posisi tsb ke pointer `prevhapus`, barulah posisi `hapus` dipindah ke node sesudahnya
2. Arahkan `prevhapus->next` agar menunjuk ke `NULL` → karena akan menjadi data yang terakhir
3. Bebaskan lokasi yang ditunjuk oleh `hapus` dan `NULL`-kan pointer tsb

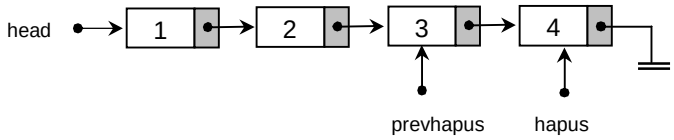
kondisi awal



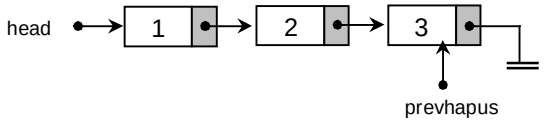
```
hapus = head;
```




```
while (hapus->next != NULL)  {
  prevhapus = hapus;
  hapus = hapus->next;
}
```




```
prevhapus->next = NULL;
free_node(hapus);
```



Langkah-langkah untuk proses di atas adalah sebagai berikut:

1. Arahkan pointer `hapus` ke `head`
2. Telusuri simpul s/d `hapus->next == NULL`, tandai simpul sebelumnya dengan pointer `prevhapus`
3. Arahkan pointer prevhapus->next ke NULL
4. Panggil fungsi `free_node(hapus)` untuk membebaskan lokasi yang ditunjuk oleh pointer `hapus` sekaligus mengarahkan pointer `hapus` ke `NULL`