

Linked List pada C

Akan bekerja dengan *record* dalam jumlah besar? Barangkali Anda membutuhkan alokasi memory dinamis dengan penggunaan *linked list*. Alokasi memory konvensional tidak lagi bisa diandalkan.

Bekerja dengan data yang besar tidak dapat kita hindari. Dan tidak jarang pula, data besar tersebut memiliki hubungan yang erat. Sebagai contoh, kita akan bekerja dengan file yang menyimpan sangat banyak record, di mana setiap record juga memiliki banyak *field*. Tugas kita tidak hanya sekadar menampilkan setiap record-nya, melainkan harus pula menambahkan record, menghapus beberapa record sesuai keinginan *user*, sampai mengurutkan record! Kondisi tersebut mengharuskan kita memiliki satu rantai data yang panjang dan saling berhubungan. Rantai data tersebut harus mampu menampung semua data yang kita miliki. Penggunaan *array* saja jelas tidak bisa, karena kita bekerja dengan barisan data heterogen.

Kita perlu menggunakan *union* ataupun *struct*. Namun, kita juga tidak dapat semena-mena menggunakan *array of struct* dalam jumlah besar. Karena lokasi penampungan memory untuk alokasi konvensional tidak akan mampu mencukupi kebutuhan memory kita. Selain itu, kita mungkin akan melakukan alokasi dan dealokasi beberapa kali di dalam program untuk mengoptimasi penggunaan memory. Solusi yang lebih baik adalah menggunakan linked list, baik *singly (single) linked list* ataupun *doubly (double) linked list*.

Pembuatan struct

Apa yang harus kita lakukan kali pertama adalah mengerti penggunaan struct di C. Bagi pengguna Pascal, struct adalah istilah di C untuk record. Penggunaan struct memungkinkan kita bekerja dengan satu record yang memiliki banyak field. Kita akan mulai dengan penggunaan struct sederhana.

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

```
int main()
```

```
{
```

```
struct Test
```

```
{
```

```
int x;
```

```
char c;
```

```
};
```

```
struct Test test1;
```

```
test1.x = 10;
```

```
test1.c = 'A';
```

```
printf("Isi dari test1.x: %d\n",  
test1.x);
```

```
printf("Isi dari test1.c: %c\n",  
test1.c);
```

```
return 0;
```

```
}
```

Program sederhana tersebut akan membuat satu struct bernama Test, kemudian membuat variabel test1 yang bertipe Struct Test. Setelah itu, nilai 10 dan 'A' masing-masing akan diisikan ke field x dan c untuk test1 tersebut. Terakhir, program mencetak isi dari test1.x dan test1.c.

Perhatikanlah kode berikut ini:

```
1. struct Test
```

```
{
```

```
int x;
```

```
char c;
```

```
};
```

```
2. struct Test test1;
```

```
3. test1.x = 10;
```

```
test1.c = 'A';
```

Bagian (1) memperlihatkan bagaimana mendeklarasikan sebuah struct dengan nama Test. Sementara, bagian (2) memperlihatkan bagaimana mendeklarasikan variabel test1 yang bertipe *Struct Test*. Bagian terakhir, memperlihatkan bagaimana cara mengakses field di dalam test1.



Struct adalah salah satu dasar penggunaan linked list. Apabila pemahaman akan struct sudah baik, maka penggunaan linked list akan semakin mudah. Berikutnya, kita akan membahas lebih jauh tentang alokasi memory dinamis.

Alokasi memory dinamis

Setelah memahami penggunaan struct, kita akan bergerak ke pengalokasian memory secara dinamis. Setelah yang satu ini beres, maka kita akan melangkah dengan tenang ke linked list.

Anda membutuhkan sebuah *array of character* yang begitu panjang. Dan sayangnya, array tersebut mungkin tidak Anda perlukan sepanjang program berjalan. Memesan memory secara konvensional seperti `char s[10000]` mungkin mudah dipakai. Sayangnya, kita akan kehilangan kebebasan untuk mendealokasikan memory yang sudah tidak terpakai. Sementara, C bukanlah Java atau Python yang cukup pemaaf dan sangat mengerti kelalaian programmer dalam menggunakan memory. Apa yang akan terjadi apabila Anda memiliki banyak variabel sejenis `char s[10000]` tanpa pengalokasian dinamis?

Melangkah lebih jauh, apa yang akan terjadi dengan memory sistem apabila kita menggunakan *array of struct* besar dalam jumlah banyak? Kita menjadi begitu boros dengan sumber daya yang cukup terbatas! Kita butuh alokasi memory secara dinamis.

Pengalokasian memory secara dinamis dapat dilakukan dengan salah satu fungsi alokasi memory yaitu `malloc()`. Sementara, lawannya, dealokasi memory, dapat dilakukan dengan memanggil fungsi `free()`.

Fungsi malloc()

```
void *malloc(size_t size);
```

Fungsi ini dideklarasikan di dalam *header* `stdlib.h`. Dengan demikian, jangan lupa untuk meng-*include* header tersebut. Fungsi

ini akan meminta memory sebanyak *size bytes*, dan apabila sukses, pointer ke lokasi memory yang dipesan akan dikembalikan. Apabila gagal, fungsi ini akan mengembalikan NULL. Memory yang dialokasikan tidak akan dibersihkan. Hal ini berbeda dengan fungsi `calloc()`.

Linux mengikuti prinsip *optimistic memory allocation strategy*. Hal ini berarti, apabila `malloc()` sukses dan mengembalikan nilai bukan NULL, tidak akan ada jaminan bahwa memory selalu tersedia. Pustaka `libc` (di atas 5.4.23) dan GNU Libc (2.x) datang bersama `malloc()` yang dapat dipengaruhi oleh *variabel environment*.

Umumnya, kita juga melibatkan penggunaan *typecasting*.

Fungsi `free()`

```
void free(void *ptr);
```

Fungsi ini dideklarasikan di dalam header `stdlib.h`. Dengan demikian, jangan lupa untuk meng-include header tersebut. Fungsi ini akan membebaskan memory teralokasi yang ditunjuk oleh variabel `ptr`. Apabila `ptr` adalah NULL, maka tidak ada operasi yang akan dilakukan. Fungsi ini tidak mengembalikan nilai.

Contoh:

Sebagai contoh, kita akan mengalokasikan *array of character* sejumlah 50000 buah secara dinamis. Untuk itu, kita akan menggunakan kode berikut ini:

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

int main()
{
    char *s = (char *) malloc (50000 *
        sizeof (char));
    free(s);
    return 0;
}
```

Program ini tidak melakukan tindakan yang *visible* terhadap pengguna. Berikut ini adalah penjelasan untuk fungsi `malloc()` dan `free()` yang digunakan.

1. `char *s = (char *) malloc (50000 * sizeof (char));`
2. `free(s);`

Pada bagian (1), kita memesan memory sejumlah 50000 kali ukuran `char`. Setelah itu, kita melakukan *typecasting* ke bentuk `char *`. Sementara pada bagian (2), kita membahaskan memory teralokasi.

Contoh berikut mungkin akan sedikit lebih dekat kepada penggunaan linked list. Kita akan memesan sebuah struct secara dinamis.

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

int main()
{
    struct Test
    {
        int x;
        char c;
    };

    struct Test * test1 = (struct Test *) malloc (sizeof (struct Test));

    test1 -> x = 10;
    test1 -> c = 'A';
    printf("Isi dari test1->x: %d\n",
        test1->x);
    printf("Isi dari test1->c: %c\n",
        test1->c);

    free(test1);

    return 0;
}
```

Sama seperti contoh struct sebelumnya, program ini mengisikan nilai 10 dan A kepada masing-masing field `x` dan `c` struct `test1`. Setelah itu, pencetakan dilakukan. Bedanya, contoh yang satu ini menggunakan alokasi memori secara dinamis yang ditandai dengan pemanggilan fungsi `malloc()`. Perhatikanlah bagian berikut ini:

1. `struct Test`

```
{
    int x;
    char c;
};
```
2. `struct Test * test1 = (struct Test *) malloc (sizeof (struct Test));`
3. `test1 -> x = 10;`
`test1 -> c = 'A';`

4. `free(test1);`

Pada bagian (1), seperti contoh terdahulu, kita mendeklarasikan Struct `Test`. Pada Bagian (2), kita memesan memori secara dinamis yang akan digunakan oleh struct `test1`. Kita memesan sejumlah ukuran Struct `Test`, dan hasilnya akan digunakan untuk menyimpan `test1`. Perhatikan bagian (3). Apabila sebelumnya kita menggunakan notasi titik untuk mengakses field, maka dalam pengalokasian dinamis ini, kita menggunakan notasi `->`. Bagian terakhir akan membebaskan memory teralokasi.

Saatnya untuk terbang ke *singly linked list*.

Singly linked list

Secara teori, linked list adalah sejumlah *node* yang dihubungkan secara linier dengan bantuan *pointer*. Dikatakan *singly* (*single*) linked apabila hanya ada satu pointer yang menghubungkan setiap node.

Setiap node akan berbentuk *struct* dan memiliki satu buah field bertipe struct yang sama, yang berfungsi sebagai pointer. Dalam menghubungkan setiap node, kita dapat menggunakan cara *first-create-first-access* ataupun *first-create-last-access*. Dalam artikel ini, kita hanya akan membahas cara *first-create-first-access*.

Kita akan mulai dengan deklarasi struct untuk node.

```
struct tnode
{
    int x;
    struct tnode *next;
}
```

Yang berbeda dengan deklarasi struct sebelumnya adalah satu field bernama *next*, yang bertipe struct `tnode`. Hal ini sekilas dapat membingungkan. Namun, satu hal yang jelas, variabel `next` ini akan menghubungkan kita dengan node di sebelah kita, yang juga bertipe struct `tnode`. Hal inilah yang menyebabkan `next` harus bertipe *struct tnode*.

Asumsikan kita memiliki sejumlah node yang selalu menoleh ke sebelah dalam arah yang sama. Atau, sebagai alat bantu, Anda bisa mengasumsikan beberapa orang yang bermain kereta api. Yang belakang akan

memegang yang depan, dan semuanya menghadap arah yang sama. Setiap pemain adalah node. Dan tangan pemain yang digunakan untuk memegang bahu pemain depan adalah variabel next. Sampai di sini, kita baru saja mendeklarasikan tipe data dasar sebuah node.

Selanjutnya, kita akan mendeklarasikan beberapa variabel pointer bertipe struct tnode. Beberapa variabel tersebut akan kita gunakan sebagai awal dari linked list, node aktif dalam linked list, dan node sementara yang kita gunakan dalam pembuatan node di linked list. Berikan nilai awal NULL kepada mereka.

Berikan deklarasi seperti berikut ini:

```
struct tnode *head=NULL, *curr=NULL,
*node=NULL;
```

Dengan demikian, sampai saat ini, kita telah memiliki tiga node. Satu sebagai kepala (*head*), satu sebagai node aktif dalam linked list (*curr*) dan satu lagi node sementara (*node*).

Kita telah siap untuk membuat singly linked list. Untuk itu, kita akan memasukkan nilai tertentu sebagai nilai untuk field x dengan cara melakukan perulangan sehingga campur tangan user tidak diperlukan.

Seperti yang dikemukakan sebelumnya, kita akan membuat singly linked list dengan cara *first-create-first-access*. Dengan cara ini, node yang dibuat pertama akan menjadi head. Node-node yang dibuat setelahnya akan menjadi node-node pengikut. Untuk mempermudah pengingatan, ingatlah gambar anak panah yang mengarah ke kanan. Head akan berada pada pangkal anak panah, dan node-node berikutnya akan berbaris ke arah bagian anak panah yang tajam.

```
[head]->[node1]->[node2]->[node3]...
[noden]->NULL
```

```
----->
```

Apabila kita perhatikan, setiap node memiliki petunjuk untuk node sebelahnyanya. Bagaimana dengan node terakhir? Kita berikan nilai NULL. Dengan demikian, setiap node kebagian jatah.

```
int i;
```

```
for (i=0; i<5; i++)
```

```
{
    node = (struct tnode *)
    malloc (sizeof(struct tnode));
    node -> x = i;

    if (head == NULL)
    {
        head = node;
        curr = node;
    }else
    {
        curr -> next = node;
        curr = node;
    }
}

curr -> next = NULL;
```

Berikut adalah penjelasan kode-kode pembuatan singly linked list tersebut. Pertama-tama, kita membuat perulangan dari 0 sampai 4, yang dimaksudkan untuk membuat lima buah node yang masing-masing field x nya berisikan nilai dari 0 sampai 4. Pembuatan node dilakukan dengan fungsi malloc().

```
for (i=0; i<5; i++)
{
    node = (struct tnode *)
    malloc (sizeof(struct tnode));
    node -> x = i;
```

```
...
...
}
```

Setelah itu, kita perlu membuat node dan penghubung. Pertama-tama, kita akan menguji apakah head bernilai NULL. Kondisi head bernilai NULL hanya terjadi apabila kita belum memiliki satu node pun. Dengan demikian, node tersebut akan kita jadikan sebagai head. Node aktif (*curr*), juga kita dapat dari node tersebut.

Sekarang, bagaimana kalau head tidak bernilai NULL alias kita telah memiliki satu atau lebih node? Yang pertama kita lakukan adalah menghubungkan pointer next dari node aktif (*curr*) ke node yang baru saja kita buat. Dengan demikian, kita baru saja membuat penghubung antara rantai lama dengan mata rantai baru. Atau, dalam permainan kereta api, pemain paling depan dalam barisan lama akan menempelkan tangannya pada bahu pemain yang baru bergabung. Node aktif (*curr*) kemudian kita pindahkan ke node yang baru dibuat.

```
if (head == NULL)
{
    head = node;
    curr = node;
}else
{
```

IKLAN

```
curr -> next = node;
curr = node;
}
```

Setelah semuanya dibuat, sudah saatnya bagi kita untuk menghubungkan pointer next untuk mata rantai terakhir ke NULL.

```
curr -> next = NULL;
```

Dan *bravo*! Singly linked list baru saja kita ciptakan. Ada yang terlupa? Ada. Bagaimana kita tahu bahwa linked list yang kita ciptakan adalah linked list yang benar? Salah satu cara untuk mengetahuinya adalah dengan mencetak field x untuk semua node. Dari head sampai node terakhir.

```
curr = head;
while (curr != NULL)
{
    printf("%d ", curr -> x);
    curr = curr -> next;
}
printf("\n");
```

Pertama-tama, kita meletakkan node aktif (curr) ke posisi head. Setelah itu, kita akan pindahkan kunjungi satu per satu node dengan memindahkan node aktif (curr) ke posisi sebelahnyanya. Semua kunjungan tersebut akan kita lakukan apabila node aktif (curr) tidak menemui NULL. Anda mungkin ingin menambahkan pemanggilan free() untuk melakukan dealokasi.

Kode selengkapnya:

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

int main()
{
    struct tnode
    {
        int x;
        struct tnode *next;
    };

    struct tnode *head=NULL,
    *curr=NULL, *node=NULL;

    int i;

    for (i=0; i<5; i++)
    {
        node = (struct tnode *)
```

```
malloc (sizeof(struct tnode));
node -> x = i;
```

```
if (head == NULL)
```

```
{
    head = node;
    curr = node;
```

```
}else
```

```
{
    curr -> next = node;
    curr = node;
```

```
}
```

```
}
```

```
curr -> next = NULL;
```

```
curr = head;
while (curr != NULL)
```

```
{
    printf("%d ", curr -> x);
    curr = curr -> next;
```

```
}
printf("\n");
```

```
return 0;
```

```
}
```

Singly linked list pun selesai. Siap terbang lebih tinggi untuk doubly (double) linked list?

Doubly linked list

Apabila dalam singly linked list setiap node memiliki satu pointer yang menunjuk ke node sebelahnyanya, maka doubly linked list lebih serakah dengan membuat setiap node memiliki dua buah pointer: ke sebelah kiri (sebelum) dan ke sebelah kanan (setelah).

Gambarkan sekali lagi dalam anak panah berikut ini:

```
NULL <- [head] -> <- [node1] -> <- [node2] -> ... <- [tail] -> NULL
----->
```

Bertambah lagi komponen yang akan kita gunakan. Apabila dalam singly linked list kita hanya memiliki head, curr dan node, maka untuk doubly linked list, kita menambahkan satu penunjuk yang berfungsi sebagai akhir dari list: tail. Bagian kiri dari head akan menunjuk ke NULL. Demikian pula dengan bagian kanan dari tail. Setiap node saling terhubung dengan pointer kanan dan kiri.

Bagaimana dengan permainan kereta api kita yang populer tersebut? Sayang sekali. Kita tidak dapat memainkannya lagi. Sebagai gantinya, kita akan memainkan sesuatu yang lebih menarik. Saling berpegangan tangan, kemudian membuat barisan. Setiap pemain memegang teman di sebelah kirinya dengan tangan kiri, dan memegang teman di sebelah kanannya dengan tangan kanan. Bagaimana dengan pemain-pemain paling ujung yang kesepian? Pemain paling kiri akan memegang udara dengan tangan kirinya, dan pemain paling kanan akan memegang udara dengan tangan kanannya.

Seperti biasa, kita mulai dengan pembuatan struct tnode. Harap diperhatikan bahwa kini kita juga memiliki pointer prev selain next.

```
struct tnode
{
    int x;
    struct tnode *prev;
    struct tnode *next;
};
```

Kemudian, tiba saatnya bagi kita untuk mendeklarasikan beberapa node yang akan kita gunakan sebagai head, tail, node aktif (curr) dan node sementara (node).

```
struct tnode *head=NULL,
*curr=NULL, *node=NULL,
*tail=NULL;
```

Sama seperti pada pembuatan singly linked list, dalam pembuatan doubly linked list ini, kita akan membuat sebuah perulangan sebanyak 5 kali untuk mengisikan nilai 0 sampai 4 ke dalam field x untuk masing-masing node.

```
int i;
for (i=0; i<5; i++)
{
    node = (struct tnode *)
    malloc (sizeof(struct tnode));
    node -> x = i;
    if (head == NULL)
    {
        head = node;
        head -> prev = NULL;
        curr = node;
    }else
    {
        curr -> next = node;
```

```
node -> prev = curr;
curr = node;
}
}
curr -> next = NULL;
tail = curr;
```

Secara umum, kode yang kita buat hampir sama dengan pembuatan singly linked list. Hanya bedanya, pada doubly linked list, kita menghubungkan pointer kiri dan kanan suatu node.

```
if (head == NULL)
{
    head = node;
    head -> prev = NULL;
    curr = node;
} else
{
    curr -> next = node;
    node -> prev = curr;
    curr = node;
}
```

Pertama-tama, tentunya kita perlu menguji apakah head bernilai NULL yang artinya belum ada satu node pun yang tercipta. Apabila demikian, maka node yang kita buat akan menjadi head. Node aktif (curr) pun di set sesuai node yang dibuat. Dan sebagai konsekuensi dari doubly linked list, kita mengatur pointer prev pada head menunjuk ke NULL. Pada permainan gandeng menggandeng tangan kita, kita meminta satu orang untuk mulai membuat barisan, dengan tangan kirinya memegang udara dan tangan kanannya siap untuk menggandeng pemain berikut.

Bagaimana kalau head tidak NULL? Hal tersebut berarti kita telah memiliki satu atau lebih node yang terhubung secara double. Yang perlu kita lakukan adalah membuat pointer next pada node aktif (curr) untuk menunjuk ke node yang baru saja kita buat. Dan tidak lupa, sebagai konsekuensi dari doubly linked list, kita juga membuat pointer prev pada node yang baru saja kita buat untuk menunjuk kepada node aktif (curr). Setelah keduanya saling tahu kiri dan kanan masing-masing, kita memindahkan node aktif (curr) ke node yang baru saja dibuat. Pada permainan kita, satu atau lebih pemain telah membentuk barisan. Lalu datang pemain baru. Pemain

kesepian di barisan akan menggandeng pemain yang baru bergabung tersebut dengan tangan kanannya. Sementara, pemain baru akan menyambut dengan tangan kirinya. Maka pemain baru telah bergabung ke dalam barisan. Dan tidak lupa, pemain aktif (yang akan menggandeng) kita pindahkan ke pemain baru tersebut.

Setelah semuanya terbentuk, saatnya untuk menutup doubly linked list. Pemain baru dilarang masuk dulu. Ekor (tail) akan kita pindahkan ke node terakhir.

```
curr -> next = NULL;
tail = curr;
```

Untuk menguji keberhasilan doubly linked kita, kita akan mencetak dari awal list sampai akhir list:

```
curr = head;
while (curr != NULL)
{
    printf("%d ", curr -> x);
    curr = curr -> next;
}
printf("\n");
```

Dan karena apa yang kita buat adalah doubly linked list, maka kita juga dapat mencetak dari tail sampai head. Berikut ini adalah caranya:

```
curr = tail;
while (curr != NULL)
{
    printf("%d ", curr -> x);
    curr = curr -> prev;
}
printf("\n");
```

Anda mungkin ingin membebaskan memori teralokasi dengan pemanggilan fungsi free().

Kode selengkapnya:


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

int main()
{
    struct tnode
    {
        int x;
        struct tnode *prev;
        struct tnode *next;
    };
```

```
struct tnode *head=NULL,
*curr=NULL, *node=NULL,
*tail=NULL;
int i;

for (i=0;i<5;i++)
{
    node = (struct tnode *)
        malloc (sizeof(struct tnode));
    node -> x = i;
    if (head == NULL)
    {
        head = node;
        head -> prev = NULL;
        curr = node;
    } else
    {
        curr -> next = node;
        node -> prev = curr;
        curr = node;
    }
    curr -> next = NULL;
    tail = curr;

    curr = head;
    while (curr != NULL)
    {
        printf("%d ", curr -> x);
        curr = curr -> next;
    }
    printf("\n");
    curr = tail;
    while (curr != NULL)
    {
        printf("%d ", curr -> x);
        curr = curr -> prev;
    }
    printf("\n");
    return 0;
}
```

Operasi pada linked list tidak hanya pembuatan dan pencetakan. Suatu saat, Anda mungkin perlu untuk menghapus node yang terletak di tengah-tengah list. Atau bahkan Anda mungkin perlu menyisipkan node di tengah-tengah node. Sama seperti halnya permainan gandeng-menggandeng tangan tersebut, sangat mungkin apabila tiba-tiba ada pemain baru yang ingin menyisipkan dirinya di tengah-tengah barisan. Tertantang? Selamat mencoba! 

Noprianto (noprianto@infolinux.co.id)