NOMOR ISBN: 978 602 98513 3 5

# PETUNJUK PRAKTIKUM
# ALGORITMA & STRUKTUR DATA
## Edisi Ganjil 2009/2010

**Laboratorium Komputer Jurusan Teknik Elektro**

**FAKULTAS TEKNIK UNIVERSITAS TARUMANAGARA**

PETUNJUK PRAKTIKUM

# ALGORITMA & STRUKTUR DATA

Edisi ganjil 2009/2010

**Tim Penyusun:**

Didi Surian, ST., MT
Joni Fat, ST.

**LABORATORIUM KOMPUTER**
**JURUSAN TEKNIK ELEKTRO**
**FAKULTAS TEKNIK**
**UNIVERSITAS TARUMANAGARA**

# DAFTAR ISI

# PERCOBAAN 1
# REKURSI DAN ITERASI

## A. Tujuan Instruksional

1. Mampu memahami proses rekursi dan iterasi.
2. Mempelajari cara membuat program-program rekursi dan iterasi dengan C++.
3. Memahami keuntungan dan kerugian pemakaian rekursi dibandingkan dengan iterasi.

## B. Teori Percobaan

Proses rekursi di dalam dunia pemrograman adalah suatu proses yang memanggil dirinya sendiri. Proses pemanggilan diri sendiri ini diterapkan dalam pemanggilan suatu fungsi dari fungsi itu sendiri. Proses ini harus dikontrol karena bila tidak dikontrol pemanggilan ini dapat saja terjadi tanpa batas tertentu. Rekursi digunakan untuk melakukan suatu komputasi atau proses tertentu yang membutuhkan proses sama yang berulang-ulang. Setiap pemanggilan proses rekursi akan melahirkan *copy* dari *variable* fungsi baru, dimana hal ini akan menyebabkan penambahan penggunaan *processor* dan *memory*.

Pada intinya proses rekursi dan iterasi adalah sama karena kedua proses ini akan melakukan pengulangan terhadap suatu proses atau komputasi. Namun demikian baik proses rekursi maupun iterasi memiliki kelebihan dan kekurangannya sendiri-sendiri.

## C. Pelaksanaan Percobaan

1. Jalankan *compiler* C++ Anda dan buatlah program untuk menghitung nilai faktorial dengan menggunakan fungsi rekursi berikut ini:

```
unsigned long factorial (unsigned long n)
{
  if ( n <= 1)
     return 1;
  else
     return n * factorial ( n – 1);
}
```

Program dibuat dengan meminta input dari *user* nilai yang akan dihitung faktorialnya!

2. Buatlah program untuk menghitung nilai deret Fibonacci dengan menggunakan fungsi rekursi berikut ini:

```
unsigned long fibonacci (unsigned long n)
{
  if ( n == 0 || n == 1)
     return n;
  else
     return fibonacci (n – 1) + fibonacci (n – 2);
}
```

Program dibuat dengan meminta input dari *user* nilai yang akan dihitung deret Fibonaccinya!

3. Setelah Anda membuat program rekursi dari kedua program di atas, cobalah untuk membuat program iterasinya dan bandingkan hasil program iterasi dan rekursi dari program yang telah dibuat!

**D. Tugas**

Perhatikan keluaran program sebagai berikut:

Banyaknya karakter yang akan dipermutasikan : 3 ↵

Permutasi untuk 3 karakter adalah:

Permutasi ke 1 : A B C

Permutasi ke 2 : A C B

Permutasi ke 3 : B A C

Permutasi ke 4 : B C A

Permutasi ke 5 : C A B

Permutasi ke 6 : C B A

Jumlah permutasi keseluruhan : 6

1. Berdasarkan keluaran program di atas, buatlah *flowchart* algoritma program di atas!
2. Buatlah programnya dalam bentuk rekursif!
3. Buatlah programnya dalam bentuk iterasi!
4. Berikan kesimpulan Anda mengenai praktikum yang telah Anda lakukan!

**E. Daftar Acuan**

J. Hubbard. *Programming With C++*. McGraw-Hill Int. Ed., 1996

# PERCOBAAN 2

# TIPE DATA POINTER

## A. Tujuan Instruksional

1. Mampu memahami tipe data pointer.
2. Mempelajari cara membuat program dengan tipe data pointer.

## B. Teori Percobaan

Tipe data pointer dapat digunakan untuk pemrograman yang membutuhkan pengalokasian *memory* yang bersifat dinamis. Konsep pengalokasian *memory* secara dinamis sangat berguna sehingga lokasi *memory* dapat digunakan seefisien mungkin. Pemrograman dengan tipe data ini hanya akan mengalokasikan *memory* saat program dieksekusi.

Nama perubah yang digunakan untuk mewakili suatu nilai data sebenarnya merupakan suatu lokasi tertentu di *memory* komputer. Pada saat sebuah program dikompilasi, *compiler* akan melihat pada deklarasi perubah untuk mengetahui nama-nama perubah apa saja yang akan digunakan sekaligus mengalokasikan tempat dalam *memory* untuk menyimpan data tersebut. Hal ini menunjukkan bahwa sebelum program dieksekusi, lokasi-lokasi data dalam *memory* sudah ditentukan dan tidak dapat diubah selama program tersebut dieksekusi. Perubah-perubah jenis ini dinamakan perubah statis (*static variable*).

Perubah dinamis (*dynamic variable*) berlaku sebaliknya, yaitu lokasi *memory* hanya akan dialokasikan pada saat program dieksekusi, sehingga ukuran *memory* yang digunakan akan selalu berubah. Perubah dinamis diterapkan dengan menggunakan tipe data *pointer* dan struktur data *class*. Berikut adalah deklarasi sebuah perubah dinamis dengan menggunakan *class*.

```
class perubah {
    public:
        int nomor;
};
```

**C. Pelaksanaan Percobaan**

1. Jalankan *compiler* C++ Anda dan buatlah program berikut ini.

```
#include <iostream>
#include <string>
using namespace std;

class X {
    public:
        string data;
};

int main()
{
   X* namaA = new X;
   X* namaB = new X;
   namaA->data = "Adi";
   namaB->data = "Budi";
   cout << namaA->data << endl;
   cout << namaB->data << endl;
   system("pause");
   return 0;
}
```

2. Kemudian jalankanlah program tersebut! Apa tujuan dari program di atas?

3. Gantilah program tersebut menjadi seperti di bawah ini.

```
#include <iostream>
#include <string>
using namespace std;

class X {
    public:
        string data;
};

int main()
{
   X* namaA = new X;
   namaA->data = "Adi";
   cout << "Nama A adalah : " << namaA->data << endl;
   X* namaB = new X;
   X* namaC = new X;
   namaC->data = "Budi";
   namaA->data = namaC->data;
   namaB->data = namaC->data;
   cout <<  "Nama A adalah : " << namaA->data << endl;
```

```
    cout <<  "Nama B adalah : " << namaB->data << endl;
    system("pause");
    return 0;
}
```

4. Kemudian jalankanlah program tersebut! Apa tujuan dari program di atas?
5. Program-program di atas adalah program bertipe pointer sederhana, sekarang buatlah program berikut ini.

```
#include <iostream>
#include <string>
using namespace std;

class X {
    public:
        string nama;
        string alamat;
        int umur;
};

int main()
{
    X* Anggota1 = new X;
    X* Anggota2 = new X;
    Anggota1->nama = "Andrian";
    Anggota1->alamat = "Bandung";
    Anggota1->umur = 31;
    Anggota2 = Anggota1;
    cout << Anggota2->nama << endl;
    cout << Anggota2->alamat << endl;
    cout << Anggota2->umur << endl;
    system("pause");
    return 0;
}
```

6. Jalankan program tersebut, apa yang terjadi?
7. Buatlah sebuah program untuk menyimpan data-data mahasiswa yang berisi nama, NIM, fakultas, jurusan dan IPK. Data-data tersebut dimasukkan oleh *user* dari *keyboard* dan kemudian tampilkanlah!
8. Modifikasilah program tersebut sehingga dapat meminta input berkali-kali sesuai permintaan *user*!

**D. Tugas**

1. Buatlah sebuah program perparkiran yang berisi no polisi, tipe kendaraan (mobil atau motor), waktu masuk (boleh mengambil dari sistem atau dimasukkan manual). Input dari *user* dapat berkali-kali, terdapat menu untuk mengisi data, melihat data, dan menghapus data, melihat jumlah data yang telah dimasukkan!
2. Buatlah program untuk mensimulasikan pemesanan tempat duduk di bioskop. Data yang dimasukkan *user* berupa no studio, nama film (bisa berupa pilihan, minimal 3 pilihan), no tempat duduk, dan waktu film yang diputar. Input dari *user* dapat berkali-kali, terdapat menu untuk mengisi data, melihat data, dan menghapus data, melihat jumlah data yang telah dimasukkan!
3. Buatlah kesimpulan dari praktikum Anda!

**E. Daftar Acuan**

J. Hubbard. *Programming With C++*. McGraw-Hill Int. Ed., 1996

# PERCOBAAN 3

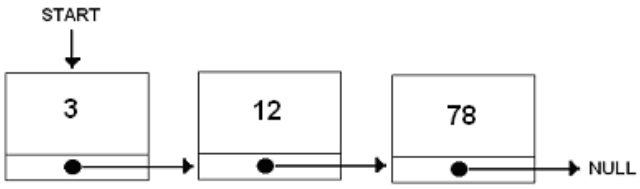
# SENARAI BERANTAI (*LINKED LISTS*) – BAGIAN 1

## A. Tujuan Instruksional

1. Mampu memahami konsep *linked lists.*
2. Mempelajari cara membuat program dengan *linked lists*.
3. Mempelajari operasi-operasi pada *linked lists*, seperti membaca isi, menambah dan menghapus simpul pada *linked lists.*

## B. Teori Percobaan

Penggunaan *linked lists* erat hubungannya dengan pemakaian *class* (ataupun *structure*) dan *pointer*. Melalui penggunaan *linked lists* pembentukkan struktur data dinamis dapat dilakukan dengan mudah. Pada dasarnya *linked lists* adalah sekumpulan komponen data yang disusun secara berurutan dan dihubungkan dengan menggunakan *pointer*. Masing-masing komponen data dalam suatu *linked lists* dinamakan dengan simpul (*node*). Sebuah *node* bilamana akan digunakan maka dapat baru dibuat sehingga menghemat penggunaan *memory*. Pada prinsipnya sebuah *node* terdiri dari 2 buah bagian, bagian pertama berisi informasi yang disimpan, sedangkan bagian kedua berisi *pointer* yang akan menunjuk ke *node* yang lain. Ilustrasi *linked lists* ditunjukkan pada Gambar 3.1.




Gambar 3.1 Ilustrasi *Linked Lists*

Dari Gambar 3.1 terlihat bahwa terdapat 3 buah *node* yang saling terhubung satu sama lain secara berurutan. *Node* terakhir memiliki *pointer* yang menunjuk ke nilai “NULL” yang menandakan bahwa *node* ini adalah *node* terakhir. Sedangkan *pointer* “START” menunjuk pada *node* pertama dari *linked lists* ini. *Node* pertama berisi data 3, *node* kedua berisi data 12, dan *node* ketiga berisi data 78.

Sebuah *linked lists* dapat direalisasikan dengan menggunakan *class* maupun *structure*. Deklarasi sebuah *linked lists* sederhana dapat dinyatakan pada cuplikan program sebagai berikut.

```
class Simpul {
    public:
        int data;
        Simpul* nextPtr;
};
```

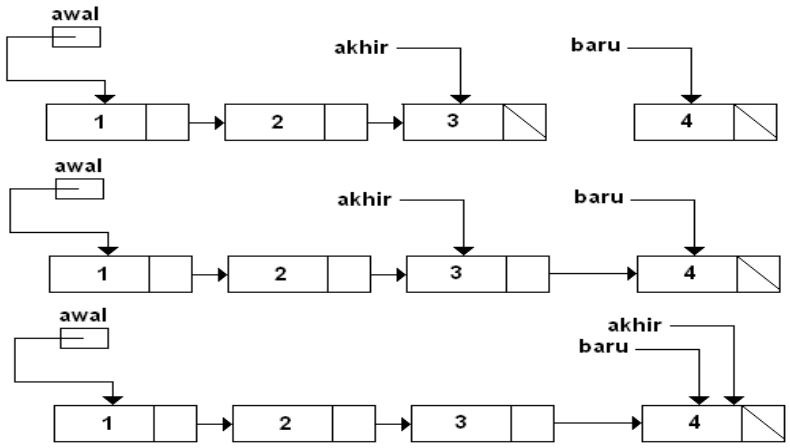
Pada deklarasi tersebut sebuah *linked lists* memiliki sebuah bagian yang menyimpan data tipe integer dan sebuah *pointer* yang akan menunjuk ke *linked lists* lainnya.

Ada beberapa operasi yang dapat dilakukan pada *linked lists*, yaitu operasi menambah *node* (di belakang, di tengah, di depan), menghapus *node* (di belakang, di tengah, di depan), dan membaca isi *node*.

## B.1 Menambah *Node*

### B.1.1 Menambah Di Belakang

Operasi penambahan *node* baru di belakang sebuah *linked lists* berarti bahwa *node* baru yang terbentuk akan selalu menjadi *node* terakhir dari suatu *linked lists*. Untuk lebih jelasnya, perhatikan ilustrasi Gambar 3.2.




Gambar 3.2 Operasi Penambahan *Node* Baru Di Belakang *Linked Lists*

Pada ilustrasi diperlihatkan bahwa *pointer* awal adalah *pointer* yang menunjuk ke *node* pertama, *pointer* akhir menunjuk ke *pointer* akhir, dan *pointer* baru menunjuk ke *node* baru yang akan ditambahkan. Proses yang terjadi adalah pertama-tama *pointer* akhir akan menunjuk ke *node* baru. Kemudian *pointer* akhir dibuat sama dengan *pointer* baru. *Function* penambahan *node* baru di belakang sebuah *linked lists* adalah sebagai berikut.

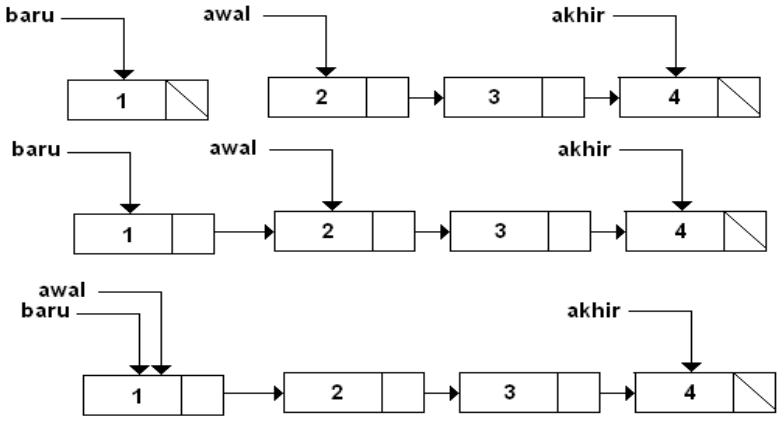
```
void Simpul::Tambah_Belakang(int nilai_baru)
{
```

```
    Simpul *baru = new Simpul;
    baru->data = nilai_baru;
    baru->nextPtr = NULL;

    if (awal == NULL){
      awal = akhir = baru;
    }
    else {
     akhir->nextPtr = baru;
     akhir = baru;
    }
}
```

### B.1.2 Menambah Di Depan

Operasi penambahan *node* baru di depan sebuah *linked lists* berarti bahwa *node* baru yang terbentuk akan selalu menjadi *node* pertama dari suatu *linked lists*. Untuk lebih jelasnya, perhatikan ilustrasi Gambar 3.3.




Gambar 3.3 Operasi Penambahan *Node* Baru Di Depan *Linked Lists*

Operasi penambahan *node* baru pada sebuah *linked lists* dimulai dengan menghubungkan *node* baru yang ditunjukkan oleh *pointer* baru ke *linked lists* yang telah ada. Kemudian *pointer* awal dibuat sama dengan *pointer* baru tersebut. *Function* penambahan *node* baru di depan sebuah *linked lists* adalah sebagai berikut.

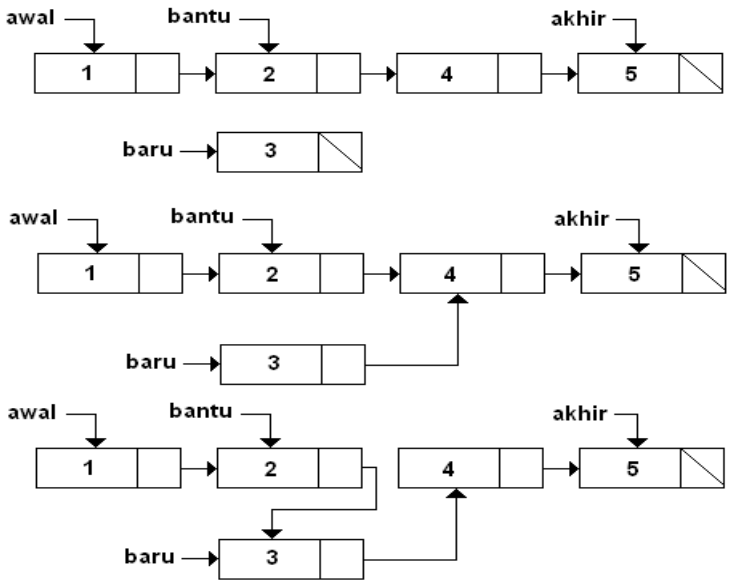
```
void Simpul::Tambah_Depan(int nilai_baru)
{
   Simpul *baru = new Simpul;
   baru->data = nilai_baru;

   if (awal == NULL){
     awal = akhir = baru;
```

```
    akhir->nextPtr = NULL;
  }
  else {
   baru->nextPtr = awal;
   awal = baru;
  }
}
```

### B.1.3 Menambah Di Tengah (Menyisipkan *Node*)

Operasi penambahan *node* baru di tengah sebuah *linked lists* membutuhkan sebuah *pointer* untuk menentukan letak dimana *node* baru akan ditambahkan. Operasi ini dapat dilihat pada ilustrasi Gambar 3.4.




Gambar 3.4 Operasi Penambahan *Node* Baru Di Tengah *Linked Lists*

*Function* yang menunjukkan proses penyisipan *node* baru ditunjukkan sebagai berikut ini.

```
void Simpul::Tambah_Tengah(int nilai_baru)
{
  bantu = awal;

  Simpul *baru = new Simpul;
  baru->data = nilai_baru;

  if (awal == NULL){
    awal = akhir = baru;
    akhir->nextPtr = NULL;
```

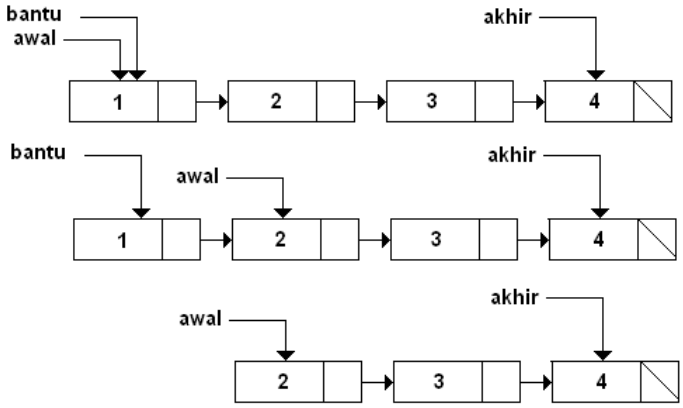
```
    }
    else {
      while(nilai_baru > bantu->nextPtr->data){
        bantu = bantu->nextPtr;
      }
      baru->nextPtr = bantu->nextPtr;
      bantu->nextPtr = baru;
    }
}
```

### B.2 Menghapus *Node*

Operasi penghapusan *node* dilakukan dengan menggunakan sebuah *pointer* bantu dan *node* yang dapat dihapus adalah *node* yang ditunjuk oleh *pointer* bantu tersebut.

### B.2.1 Menghapus *Node* Pertama

Operasi untuk menghapus *node* pertama dilakukan dengan menaruh *pointer* bantu pada *node* awal. Setelah itu *pointer* awal dipindahkan ke *node* yang ditunjukkan oleh *pointer* pada *node* yang ditunjuk oleh *pointer* bantu. Selanjutnya *node* yang ditunjuk oleh *pointer* bantu dihapus. Ilustrasi penghapusan *node* pertama diperlihatkan pada Gambar 3.5.



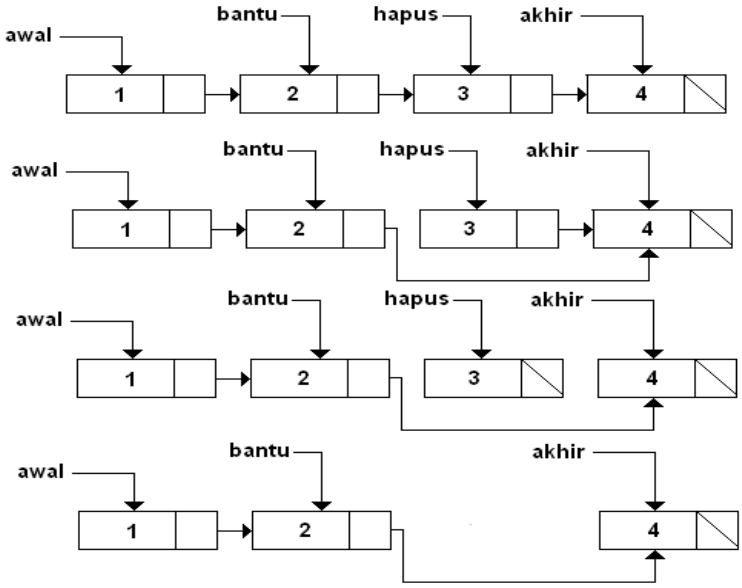

Gambar 3.5 Operasi Penghapusan *Node* Di Depan *Linked Lists*

*Function* yang menunjukkan proses penghapusan *node* di depan *linked lists* ditunjukkan sebagai berikut ini.

```
void Simpul::Hapus_Depan()
{
  bantu = awal;
  awal = bantu->nextPtr;
  bantu->nextPtr = NULL;
}
```

### B.2.2 Menghapus *Node* Di Tengah atau Terakhir

Operasi ini menyerupai proses penghapusan sebelumnya. Proses membutuhkan juga *pointer* bantu dan *node* yang akan dihapus adalah *node* yang ditunjukkan oleh *pointer* bantu. Ilustrasi penghapusan *node* pertama diperlihatkan pada Gambar 3.6.




Gambar 3.6 Operasi Penghapusan *Node* Di Depan *Linked Lists*

*Function* yang menunjukkan proses penghapusan *node* di tengah *linked lists* ditunjukkan sebagai berikut ini.

```
void Simpul::Hapus_Tengah(int nilai_hapus)
{
 bantu = awal;
 while(bantu->nextPtr->data != nilai_hapus)
 {
   bantu = bantu->nextPtr;
 }
 hapus = bantu->nextPtr;
 bantu->nextPtr = hapus->nextPtr;
 hapus->nextPtr = NULL;
}
```

Sedangkan *function* yang menunjukkan proses penghapusan *node* di akhir *linked lists* ditunjukkan sebagai berikut ini.

```
void Simpul::Hapus_Akhir()
{
 bantu = awal;
 while (bantu->nextPtr != akhir)
```

```
{
  bantu = bantu->nextPtr;
}
hapus = bantu->nextPtr;
akhir = bantu;
bantu->nextPtr = NULL;
}
```

**C. Pelaksanaan Percobaan**

1. Jalankan *compiler* C++ Anda dan buatlah program berikut ini.

```
#include <iostream>
using namespace std;
class Simpul {
    public:
        void Inisialisasi();
        void Tambah_Belakang(int n);
        void Tambah_Depan(int n);
        void Tambah_Tengah(int n);
        void Hapus_Depan();
        void Hapus_Tengah(int n);
        void Hapus_Akhir();
        void Baca_List();
    private:
        int data;
        Simpul* nextPtr;
        Simpul* awal;
        Simpul* akhir;
        Simpul* bantu;
        Simpul* hapus;
};

void Simpul::Inisialisasi()
{
  awal = NULL;
  akhir = awal;
}

void Simpul::Tambah_Belakang(int nilai_baru)
{
  Simpul *baru = new Simpul;
  baru->data = nilai_baru;
  baru->nextPtr = NULL;

  if (awal == NULL){
    awal = akhir = baru;
```

```
    }
    else {
      akhir->nextPtr = baru;
      akhir = baru;
    }
}

void Simpul::Tambah_Depan(int nilai_baru)
{
    Simpul *baru = new Simpul;
    baru->data = nilai_baru;

    if (awal == NULL){
       awal = akhir = baru;
       akhir->nextPtr = NULL;
    }
    else {
      baru->nextPtr = awal;
      awal = baru;
    }
}

void Simpul::Tambah_Tengah(int nilai_baru)
{
    bantu = awal;

    Simpul *baru = new Simpul;
    baru->data = nilai_baru;

    if (awal == NULL){
       awal = akhir = baru;
       akhir->nextPtr = NULL;
    }
    else {
       while(nilai_baru > bantu->nextPtr->data){
          bantu = bantu->nextPtr;
       }
       baru->nextPtr = bantu->nextPtr;
       bantu->nextPtr = baru;
    }
}

void Simpul::Hapus_Depan()
{
  bantu = awal;
  awal = bantu->nextPtr;
```

```
  bantu->nextPtr = NULL;
}

void Simpul::Hapus_Tengah(int nilai_hapus)
{
  bantu = awal;
  while(bantu->nextPtr->data != nilai_hapus)
  {
     bantu = bantu->nextPtr;
  }
  hapus = bantu->nextPtr;
  bantu->nextPtr = hapus->nextPtr;
  hapus->nextPtr = NULL;
}

void Simpul::Hapus_Akhir()
{
 bantu = awal;
 while (bantu->nextPtr != akhir)
 {
    bantu = bantu->nextPtr;
 }
 hapus = bantu->nextPtr;
 akhir = bantu;
 bantu->nextPtr = NULL;
}

void Simpul::Baca_List()
{
  bantu = awal;

  while(bantu != NULL)
  { cout << bantu->data;
    bantu = bantu->nextPtr; }
}

int main()
{
    int data_depan,data_belakang,data_tengah;
    Simpul X;
    X.Inisialisasi();

    cout << "Enter the data belakang : ";
    cin >> data_belakang;
    X.Tambah_Belakang(data_belakang);
```

```
    cout << "Enter the data depan : ";
    cin >> data_depan;
    X.Tambah_Depan(data_depan);

    cout << "Enter the data tengah 1 : ";
    cin >> data_tengah;
    X.Tambah_Tengah(data_tengah);

    cout << "Enter the data tengah 2 : ";
    cin >> data_tengah;
    X.Tambah_Tengah(data_tengah);

    cout << endl;
    X.Baca_List();
    cout << endl;

    cout << "Setelah hapus data pertama : ";
    X.Hapus_Depan();
    X.Baca_List();
    cout << endl;

    cout << "Setelah hapus data " << data_tengah << " : ";
    X.Hapus_Tengah(data_tengah);
    X.Baca_List();
    cout << endl;

    cout << "Setelah hapus data " << data_belakang << " : ";
    X.Hapus_Akhir();
    X.Baca_List();
    cout << endl;

    system("pause");
    return 0;
}
```

Tulislah keluaran program tersebut!

2. Modifikasilah program tersebut sehingga program dapat menentukan sendiri di mana letak data yang akan dihapus (hanya ada 1 buah *function* untuk menghapus)!
3. Modifikasilah program tersebut sehingga tidak mengijinkan *user* untuk mengisi *linked lists* tersebut dengan 2 buah data yang sama!
4. Setelah *pointer* tidak digunakan lagi ada baiknya dihapus dengan perintah ***delete*** *nama_pointer*, modifikasilah program tersebut agar setiap kali *function* telah selesai dilakukan maka *pointer* yang tidak digunakan lagi dihapus!

**D. Tugas**

1. Buatlah *flowchart* beserta program untuk membuat *linked lists* berisi daftar nama, NIM, dan nilai mahasiswa untuk mata kuliah Algoritma & Struktur Data. Hal-hal yang perlu diperhatikan adalah sebagai berikut:
   - Tidak boleh ada NIM yang sama, bila sama akan muncul peringatan dan kemudian *user* dapat memasukkan kembali
   - Proses menambahan data baru selalu di belakang *linked lists*
   - Terdapat menu untuk menghapus data *linked list* berdasarkan NIM-nya
2. Buatlah kesimpulan dari praktikum Anda!

**E. Daftar Acuan**

J. Hubbard. *Programming With C++*. McGraw-Hill Int. Ed., 1996

# PERCOBAAN 4

# SENARAI BERANTAI (*LINKED LISTS*) – BAGIAN 2

## A. Tujuan Instruksional

1. Mampu memahami konsep *linked lists.*
2. Mempelajari operasi pada *linked lists*, seperti membaca mundur dan mencari data pada *linked lists.*

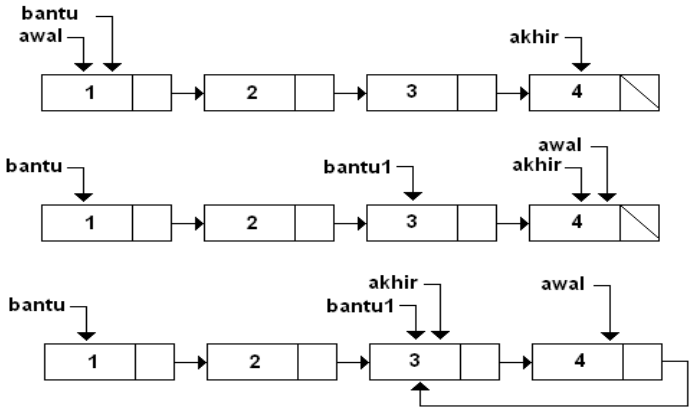
## B. Teori Percobaan

### B.1 Membaca Isi *Linked Lists*

Isi sebuah *linked lists* dapat dibaca maju maupun mundur. Pada prinsipnya kedua operasi ini adalah sama namun hanya arah *pointer* yang digunakan untuk membaca isi dibuat sebaliknya. Pembacaan isi secara maju sebuah *linked lists* telah dipelajari pada praktikum III dan *function* yang dapat digunakan adalah sebagai berikut.

```
void Simpul::Baca ()
{
 bantu = awal;
 while(bantu != NULL)
 { cout << bantu->data;
   bantu = bantu->nextPtr; }
}
```

Pembacaan *linked lists* secara mundur pada prinsipnya adalah sama, hanya saja arah *pointer* yang digunakan untuk membaca dibuat sebaliknya. Langkah pembalikkan arah *pointer* diperlihatkan pada Gambar 4.1.




Gambar 4.1 Ilustrasi Membalik *Pointer*

Langkah-langkah membalik *pointer* dapat dijelaskan sebagai berikut. Pertama dibuat *pointer* bantu sama dengan *pointer* awal. Kemudian *pointer* awal dibuat sama dengan *pointer* akhir dan dibuat *pointer* bantu1 untuk bergerak dari posisi *pointer* bantu sampai sebelum *pointer* akhir. Langkah berikutnya adalah *pointer* akhir dibuat sama dengan *pointer* bantu1. Langkah ini diulang hingga *pointer* akhir berhimpit dengan *pointer* bantu pada posisi pertama. *Function* untuk membalik *pointer* ini diperlihatkan sebagai berikut.

```
void Simpul::Balik_Pointer()
{  bantu = awal;
   awal = akhir;
   do
  { bantu1 = bantu;
     while(bantu1->nextPtr != akhir)
     { bantu1 = bantu1->nextPtr; }
     akhir->nextPtr = bantu1;
     akhir = bantu1;
   }
  while(akhir != bantu);
  akhir->nextPtr = NULL;
}
```

Setelah *pointer* dibalik dengan *function* tersebut, maka *linked lists* dapat dicetak dengan menggunakan *function* Baca yang telah dijabarkan.

### B.2 Mencari Data Tertentu Pada *Linked Lists*

Secara garis besar, proses untuk mencari data tertentu pada sebuah *linked lists* hampir sama dengan proses membaca isi *linked lists*, namun perlu ditambahkan dengan tes untuk menentukan apakah data yang dicari ada pada *linked lists*. Berikut adalah *function* untuk mencari data pada sebuah *linked lists*, apabila data yang dicari ada pada *linked lists* maka *function* akan mengembalikan nilai *boolean true*.

```
bool Simpul::Cari_Data(int data_cari)
{
    bool ketemu = false;
    bantu = awal;
    do
    {
      if(bantu->data == data_cari)
      {
         ketemu = true;
      }
      else
```

```
        {
          bantu = bantu->nextPtr;
        }
      }
      while( (ketemu != true) && (bantu->nextPtr != NULL));
      return ketemu;
  }
```

**B.3 *Circular Linked Lists***

*Circular linked lists* atau senarai melingkar adalah senarai berantai biasa dimana *pointer* pada *node* terakhir diarahkan kembali ke *node* pertama. Pada prinsipnya semua operasi yang ada pada *linked lists* biasa dapat pula diterapkan pada *circular linked lists* hanya saja diperlukan beberapa perubahan karena *pointer* akhir tidak menunjuk ke nilai NULL namun ke *node* yang awal.

**C. Pelaksanaan Percobaan**

1. Salinlah kembali *listing program* pada praktikum III dan modifikasilah sehingga menjadi program yang :
   - Dapat menerima input dari *user* berupa nama buah dan harga
   - Terdapat menu untuk membaca isi dari *linked lists* maju maupun mundur
   - Terdapat menu untuk mencari nama suatu buah dan harganya
2. Modifikasilah program Anda sehingga menjadi *circular linked lists*!

**D. Tugas**

1. Buatlah *flowchart* beserta program untuk membuat *linked lists* berisi daftar nama, umur, jenis kelamin (laki-laki atau wanita) dengan ketentuan sebagai berikut:
   - Data baru yang dimasukkan oleh *user* secara otomatis diurutkan berdasarkan umur
   - Tidak boleh ada data baru dengan nama yang sama
   - Terdapat menu untuk membaca isi *linked lists* secara maju maupun mundur
2. Buatlah kesimpulan dari praktikum Anda!

**E. Daftar Acuan**

J. Hubbard. *Programming With C++*. McGraw-Hill Int. Ed., 1996

# PERCOBAAN 5

## *DOUBLE LINKED LISTS*

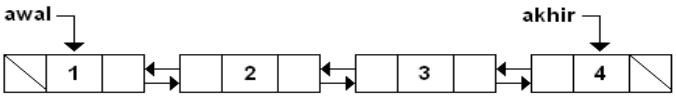
### A. Tujuan Instruksional

1. Mampu memahami konsep *double linked lists.*
2. Mempelajari cara membuat program dengan *double linked lists*.

### B. Teori Percobaan

Pada praktikum III dan IV telah dipelajari konsep, cara membuat, berbagai operasi pada *linked lists*. Namun demikian yang telah dipelajari hanyalah *linked lists* dengan sebuah *pointer* pada setiap *node*. Hal ini memiliki kelemahan, yaitu *linked lists* tersebut hanya dapat dibaca dalam satu arah, atau untuk membaca pada arah sebaliknya harus membalik *pointer* yang ada dahulu.

Praktikum V akan mempelajari mengenai *double linked lists*, yaitu *linked lists* yang memiliki *pointer* ganda. Kedua *pointer* ini menunjuk ke *node* yang berlawanan, yaitu *pointer* pertama akan menunjuk pada *node* sebelumnya, sedangkan *pointer* yang satunya lagi akan menunjuk *node* berikutnya. *Double linked lists* dapat pula disebut senarai dua arah (*two-ways lists*). *Double linked lists* dapat diilustrasikan sebagai berikut.




Deklarasi *double linked lists* dapat dinyatakan sebagai berikut.

```
class Simpul {
    public:
        int data;
        Simpul* kiriPtr;
        Simpul* kananPtr;
};
```

Pada deklarasi di atas *pointeri* kiriPtr hanyalah nama untuk *pointer* yang menunjuk ke *node* yang terletak di sebelah kiri dari *node* dan juga sebaliknya.

**C. Pelaksanaan Percobaan**

1. Buatlah program pos akses suatu perumahan dengan menggunakan *double linked lists* untuk menyimpan masukan *user* berupa nomor kendaraan, nama pengemudi, alamat rumah!
2. Buatlah *function* untuk membaca, menambahkan, dan menghapus *node*!
3. Modifikasilah program Anda sehingga menjadi *circular double linked lists*, yaitu *pointer node* terakhir menunjuk *node* awal dan sebaliknya!

**D. Tugas**

1. Buatlah *flowchart* dan program sederhana yang mensimulasikan cara kerja organizer dengan spesifikasi sebagai berikut:
   - Organizer akan menerima masukan *user* berupa: waktu kegiatan, tanggal kegiatan, deskripsi singkat kegiatan.
   - Terdapat menu untuk menambah, menghapus, dan membaca isi organizer.
   - Jumlah *node* yang diijinkan hanya sebanyak 10 buah.
   - Diimplementasikan dengan menggunakan *circular double linked lists*.
2. Buatlah kesimpulan dari praktikum Anda!

**E. Daftar Acuan**

J. Hubbard. *Programming With C++*. McGraw-Hill Int. Ed., 1996

# PERCOBAAN 6

# TUMPUKAN (*STACK*)

## A. Tujuan Instruksional

1. Mampu memahami struktur data tentang tumpukan (*stack*)
2. Mempelajari cara membuat program *stack* dengan *array*
3. Mempelajari cara membuat program *stack* dengan *pointer*

## B. Teori Percobaan

Sebuah *stack* dapat diartikan sebagai suatu kumpulan data yang saling diletakkan satu di atas yang lain. *Stack* memiliki sifat yaitu penambahan dan pengambilan data dilakukan pada ujung yang sama dari kumpulan data tersebut (*top of the stack*). Cara kerja ini disebut *Last In First Out* (LIFO). Sebuah tumpukan dapat diilustrasikan sebagai berikut.


Gambar 6.1 Ilustrasi Sebuah *Stack*

*Stack* dapat dilakukan dengan menggunakan 2 buah cara, yaitu dengan menggunakan struktur data *array* dan dengan menggunakan *pointer*. Bila disajikan dalam bentuk *array*, maka jumlah elemen dari *stack* tidak dapat diubah-ubah atau sesuai dengan ukuran *array* yang dibuat. Namun bilamana dilakukan dengan menggunakan *pointer* maka dapat lebih dinamis.

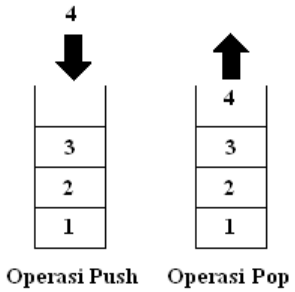
Penyajian stuktur data *stack* dengan menggunakan *array* dapat dilakukan seperti di bawah ini.

```
const int STACKSIZE = 5;
int stack[ STACKSIZE ];
```

Sedangkan penyajiaan dengan menggunakan *pointer* adalah sama seperti deklarasi *linked lists*.

```
class mystack {
    public:
        int data;
        Simpul* nextPtr;
};
```

Operasi dasar pada struktur data *stack* adalah operasi penambahan data dan pengambilan data dari *stack*. Operasi penambahan data disebut dengan operasi *push*, sedangkan operasi pengambilan data disebut operasi *pop*. Operasi-operasi tersebut dapat diilustrasikan sebagai berikut.




Gambar 6.2 Ilustrasi Operasi *Push* dan *Pop*

Berikut akan dibahas penerapan operasi tersebut baik pada *array* maupun dengan menggunakan *pointer*.

Berikut adalah penerapan *array* untuk operasi *pop* dan *push* pada sebuah *stack*.

```
void push(int mystack[],int n)
{
    index++;
    stack[index] = n;
}

void pop()
{
   if (index < 0)
     cout << endl << "Stack sudah kosong";
   else
   {
     cout << stack[index];
     index--;
   }
}
```

Penerapan *pointer* untuk operasi *pop* dan *push* pada sebuah *stack* pada dasarnya menyerupai operasi pada *linked lists*. Operasi *push* menyerupai operasi penambahan *node* pada awal *linked lists* dan operasi *pop* menyerupai penghapusan *node* pada awal *linked lists* namun nilainya diambil terlebih dahulu.

**C. Pelaksanaan Percobaan**

1. Dengan menggunakan struktur data *stack*, buatlah sebuah program untuk membalik kalimat berikut ini dengan menggunakan *array*.

   **BELAJAR ALGORITMA PEMROGRAMAN SANGAT MENYENANGKAN**

2. Modifikasilah program tersebut sehingga kalimat yang dibalik berasal dari masukan *user*!
3. Lakukan langkah 1 dan 2 kembali namun dengan menggunakan *pointer*!
4. Modifikasilah *function* yang telah dipelajari sehingga ukuran *stack* dengan menggunakan *array* dapat ditentukan oleh *user* dan bukan ditentukan oleh program!
5. Modifikasilah program no 2 sehingga proses pembalikan dilakukan terhadap masing-masing kata bukan keseluruhan kalimat!

**D. Tugas**

1. Buatlah flowchart dan program dengan menggunakan struktur data *stack* yang mensimulasikan proses pembacaan isi *inbox* SMS. Penerimaan SMS berupa masukan dari *user*!
2. Ubahlah program no 1 dengan menggunakan *pointer*!
3. Berilah kesimpulan praktikum Anda!

**E. Daftar Acuan**

J. Hubbard. *Programming With C++*. McGraw-Hill Int. Ed., 1996

# PERCOBAAN 7

# ANTRIAN (*QUEUE*)

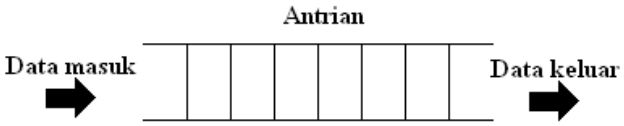
## A. Tujuan Instruksional

1. Mampu memahami struktur data tentang antrian (queue).
2. Mempelajari cara membuat program antrian dengan *array*
3. Mempelajari cara membuat program antrian dengan *pointer*

## B. Teori Percobaan

Struktur data antrian (*queue*) adalah suatu struktur data yang hanya memperbolehkan penambahan data pada satu ujung kumpulan data tersebut dan penghapusan/pengambilan data yang ada pada ujung yang lain. Prinsip kerja ini disebut juga *First In First Out* (FIFO). Dengan kata lain dalam suatu antrian urutan keluar sebuah elemen akan sama dengan urutan masuknya. Penyajian suatu antrian dapat direalisasikan dengan menggunakan *array* maupun *pointer*. Deklarasi ini sama dengan pada tumpukan yang telah dipelajari.

Operasi yang ada pada struktur data antrian sama seperti pada tumpukan, yaitu operasi penyimpan data (*push*) dan operasi pengambilan data (*pop*). Kedua operasi ini sama seperti pada tumpukan, hanya saja perbedaan terletak pada urutan pemanggilan *index* (pada *array*) atau *node* (pada *linked lists*). Operasi pada antrian dapat diilustrasikan sebagai berikut.




## C. Pelaksanaan Percobaan

1. Buatlah sebuah program untuk mensimulasikan deretan mobil yang mengantri pada pintu tol. Data mobil yang akan melewati pintu tol dimasukkan oleh *user*. Kemudian mobil yang sudah mengambil karcis akan dihapus dari daftar antrian! Program dibuat dengan menggunakan *pointer*.
2. Buatlah program antrian untuk mensimulasikan pendaftaran mahasiswa baru. Data yang dicatat adalah nama dan jurusan. Tersedia untuk dua buah jurusan, Teknik Elektro dan Teknik Mesin. Teknik Elektro dibuat dengan menggunakan *pointer,* dan Teknik Mesin dibuat dalam bentuk *array*. Pengalokasian NIM dilakukan secara otomatis!

**D. Tugas**

1. Buatlah *flowchart* dan program dengan menggunakan *array* untuk struktur data *array* yang mensimulasikan pembayaran di kasir pada sebuah supermarket. Tersedia 3 kasir dan masing-masing kasir menggunakan *array* yang berbeda!
2. Modifikasilah program no 1 dengan menggunakan *pointer*!
3. Berilah kesimpulan praktikum Anda!

**E. Daftar Acuan**

J. Hubbard. *Programming With C++*. McGraw-Hill Int. Ed., 1996

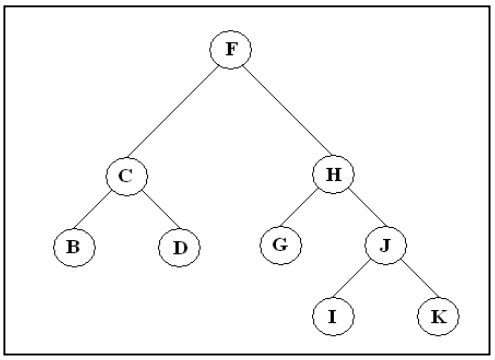
# PERCOBAAN 8
# POHON BINER

## A. Tujuan Instruksional

1. Mampu memahami struktur dari pohon biner.
2. Mampu membuat program untuk mengimplementasikan pohon biner.

## B. Teori Percobaan

Pohon biner terdiri dari elemen yang disebut simpul (*node*) dan penghubung (*link*) yang menghubungkan *node-node* yang ada. Setiap *node* dapat dihubungkan paling banyak ke dua buah *node* yang lain (disebut juga dengan keturunan/anak – *descendant*). Satu *node* yang berada pada awal struktur didesain sebagai akar (*root*), yang dianalogiskan dengan kepala (*head*) pada *linked lists*, atau dengan *top* seperti pada *stack*. *Node* yang tidak lagi memiliki anak disebut dengan daun (*leaf*). Ilustrasi pohon biner dapat dilihat pada Gambar 8.1.




Gambar 8.1 Ilustrasi Pohon Biner

Setiap *node* dalam pohon biner dapat diperlakukan sebagai *root*. Sebab setiap simpul merupakan orang tua (*parent*) dari dua cabang, yaitu bagian pohon sebelah kiri dan bagian pohon sebelah kanan yang dibuat pada setiap simpul. Satu atau bagian pohon yang lain mungkin tidak mempunyai elemen dan disebut sebagai bagian pohon kosong. Kedua bagian pohon pada daun adalah kosong.

Bentuk penyajian pohon biner dalam program menggunakan data berbentuk *pointer*. Oleh karena itu, pada pohon biner juga dapat ditambahkan simpul baru atau dihapus. Struktur pada pohon biner ada yang merupakan struktur pohon seimbang, dinamakan

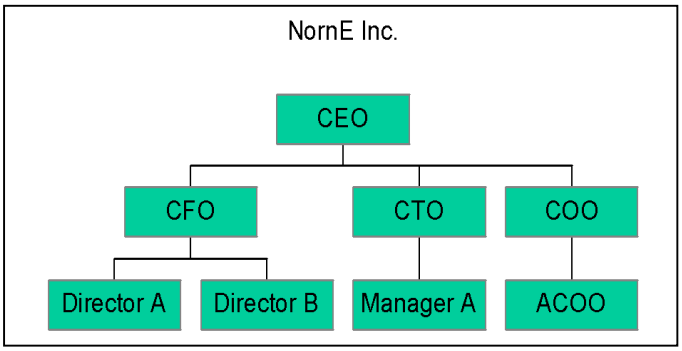
demikian karena setiap simpul selalu mempunyai dua cabang dengan kedudukan yang sama antara kiri dan kanannya.

Ada tiga macam operasi dasar dari penelusuran pohon biner, yaitu :

1. Penelusuran secara *preorder*.
2. Penelusuran secara *inorder*.
3. Penelusuran secara *postorder*.

## C. Pelaksanaan Percobaan

1. Buatlah sebuah program yang mengimplementasikan sebuah pohon biner yang merupakan struktur suatu organisasi. Anda dapat menggunakan struktur organisasi pada Gambar 8.2 atau dapat Anda buat sendiri.




Gambar 8.2 Struktur Organisasi *NornE Inc.*

2. Buatlah sebuah program yang mengimplementasikan sebuah pohon biner dengan struktur yang seimbang.
3. Telusuri dua program di atas dengan menggunakan tipe penelusuran *preorder*.
4. Buatlah sebuah program yang mengimplementasikan pohon biner, kemudian terapkan prosedur hapus dan tambah pada pohon tersebut.

## D. Tugas

1. Buatlah program untuk menelusuri dua program (Percobaan No.1 dan No. 2) di atas dengan menggunakan tipe penelusuran inorder dan postorder.
2. Buatlah *flowchart* dan program pohon biner untuk Nomor Induk Mahasiswa (NIM) mahasiswa elektro. NIM di-*input* oleh *user*, kemudian dibuat dalam bentuk pohon.
3. Berikan kesimpulan praktikum Anda.

## E. Daftar Acuan

J. Hubbard. *Programming With C++*. McGraw-Hill Int. Ed., 1996

# PERCOBAAN 9

# *SEARCHING* (PENCARIAN)

## A. Tujuan Instruksional

1. Mempelajari berbagai macam tipe pencarian data.
2. Mempelajari cara membuat program untuk mencari data berdasarkan tipe-tipe tersebut.

## B. Teori Dasar

Pencarian data yang sering disebut juga dengan *table look-up* atau *storage and retrieval information*, adalah suatu proses untuk mengumpulkan sejumlah informasi di dalam memori komputer dan kemudian mencari kembali informasi yang dibutuhkan secepat mungkin. Ada dua macam teknik pencarian yang akan dibahas dalam praktikum kali ini, yaitu pencarian sekuensial dan pencarian biner.

Perbedaan dari kedua teknik ini terletak pada keadaan data. Pencarian sekuensial (*sequential search*) digunakan apabila data dalam keadaan acak atau tidak terurut, sedangkan pencarian biner (*binary search*) digunakan pada data yang sudah dalam keadaan terurut.

Pada pencarian sekuensial, pencarian dilakukan dengan pengulangan dari 1 sampai dengan jumlah data. Pada setiap pengulangan dilakukan perbandingan antara data ke-*i* dengan data yang dicari. Apabila sama, berarti data telah ditemukan. Sebaliknya jika sampai akhir pengulangan tidak ada yang sama berarti data yang dicari tidak ada.

Pada pencarian biner, pencarian dilakukan apabila data dalam keadaan terurut. Jadi apa bila data belum terurut, pencarian tidak dapat dilakukan dengan metoda ini. Pencarian dimulai dengan mencari data yang terletak di tengah. Kemudian data yang dicari dibandingkan dengan data yang berada di tengah. Apabila data yang dicari lebih kecil dari data tengah, maka dilakukan pencarian data tengah kembali dengan catatan data terakhirnya adalah data tengah yang sebelumnya. Demikian terus sampai data tengah sama dengan data yang dicari.

## C. Pelaksanaan Percobaan

1. Buatlah sebuah program untuk mencari data 26 dari urutan data berikut ini : 23 56 37 5 67 89 87 26 34 54 22 45 12 78 10 19 16 15 dengan menggunakan metoda *sequential search*.

2. Buatlah sebuah program untuk mencari data 56 dari kumpulan data berikut dengan menggunakan metoda *binary search*: 2 4 6 12 13 15 15 17 25 26 28 39 56 59 60 65 68 77 79 89 100.
3. Buatlah program untuk mencari data tertentu dengan menggunakan metode *sequential* dan *binary search*, dengan kumpulan data yang ada yang dicari berasal dari *input user*.

**D. Tugas**

1. Pelajari metoda pencarian *hashing*, buat algoritma dan prosedur pencariannya.
2. Buatlah program pencarian data dengan metoda *hashing* tersebut.
3. Buatlah program pencarian data dengan menggunakan metode *sequential search* pada sekumpulan data yang telah terurut.
4. Berikan kesimpulan praktikum Anda.

**E. Daftar Acuan**

J. Hubbard. *Programming With C++*. McGraw-Hill Int. Ed., 1996

# PERCOBAAN 10

# *SORTING* (PENGURUTAN)

## A. Tujuan Instruksional

1. Memahami beberapa cara atau metoda pengurutan data.
2. Mempelajari cara membuat program mengurutkan data dengan berbagai metoda tersebut.

## B. Teori Dasar

Pengurutan data (*sorting*) secara umum dapat didefinisikan sebagai suatu proses untuk menyusun kembali himpunan objek menggunakan aturan tertentu. Secara umum ada dua jenis pengurutan data, yaitu pengurutan secara urut naik (*ascending*), yaitu dari data yang nilainya paling kecil ke data yang nilainya paling besar, dan pengurutan data secara urut turun (*descending*) yang merupakan kebalikan dari urut naik. Dalam hal pengurutan data yang bertipe *string* atau *char*, nilai data dikatakan lebih kecil atau sebaliknya didasarkan pada urutan relatif (*collating sequence*) seperti dinyatakan dalam tabel ASCII. Tujuan pengurutan data adalah untuk lebih mempermudah pencarian data.

Pengurutan data menjadi satu bagian yang penting dalam ilmu komputer karena waktu yang diperlukan untuk melakukan proses pengurutan perlu dipertimbangkan. Dengan demikian pemilihan algoritma atau metoda yang digunakan dalam proses pengurutan akan sangat berpengaruh terhadap waktu dari pengurutan tersebut. Ada berbagai jenis metoda pengurutan data, yaitu: *selection sort, exchage sort/bubble sort, shell sort, quick sort, radix sort* dan *merge sort*. Masing-masing penjelasan dan algoritma dari tiap metoda di atas dapat dilihat pada buku kuliah Anda.

## C. Pelaksanaan Percobaan

- Buatlah masing-masing prosedur dari algoritma jenis pengurutan data yang telah disebutkan di atas dengan menggunakan C++.
- Setelah membuat masing-masing procedur, simpan dengan nama tipe dari masing-masing metoda tersebut.
- Kemudian urutkanlah data-data berikut dengan menggunakan masing-masing metoda: 21-20-67-34-12-13-45-68-98-4-6-9-79-35-15.

- Buatlah sebuah program yang berisi daftar nama mahasiswa. Nama-nama mahasiswa tersebut didapatkan dari *input user*. Buat program tersebut menjadi berbentuk *linked list*. Kemudian lakukan pengurutan dengan metoda *bubble sort* berdasarkan huruf awal dari nama tiap mahasiswa tersebut, dan tampilkan hasilnya.

## D. Tugas

1. Buatlah *flowchart* dari setiap metode *sorting* yang sudah Anda pelajari!
2. Lalu buatlah program yang mengurutkan data-data yang diperoleh dari komputer secara acak dengan menggunakan tiap metoda *sorting* tersebut.
3. Berikan kesimpulan praktikum Anda.

## E. Daftar Acuan

J. Hubbard. *Programming With C++*. McGraw-Hill Int. Ed., 1996