



TREE

Tennov Simanjuntak

Topik hari ini (senin)

- Pengantar
- Definisi **tree**
- **Generic tree** (pohon umum)
- Implementasi **generic tree**
- **Binary tree** (pohon biner)
- **Binary search tree** (pohon pencarian biner)
- Inseri **binary search tree**

Topik hari Rabu

- Pencarian elemen (*search*) pada **binary search tree**
- Pencarian elemen maksimum dan minimum pada **binary search tree**
- Penghapusan elemen pada **binary search tree**

Pengantar

- Linked list telah memberikan solusi terhadap kekurangan array (sebuah primitif objek bahasa pemrograman C) untuk penyimpanan data.
- Linked list memberikan waktu linier untuk penyisipan elemen, bahkan linked list memberikan waktu konstan untuk penghapusan elemen.
- Akan tetapi, untuk data yang sangat besar sekali, ada jenis struktur data yang dapat meningkatkan efisiensi operasi-operasi struktur data.
- Struktur data tersebut adalah TREE (pohon).
- Tree dan variannya bersama-sama dengan GRAPH(graf) adalah struktur data non linier.

Struktur data linier vs non-linier

- Struktur data linier adalah struktur data yang mengorganisasi data secara linier dimana elemen-elemen disusun secara sekuensial, yang berarti jika diambil salah satu elemen pada struktur data, elemen tersebut memiliki **1** elemen pendahulu dan **0** atau **1** elemen pengikut. Array 1 dimensi, linked list, stack, dan queue adalah struktur data linier.
- Struktur data non-linier adalah struktur data yang tidak diorganisasi secara sekuensial. Sebuah elemen data tidak perlu bertaut pada 1 elemen saja, tetapi bisa beberapa elemen data. Array multi dimensi, tree, graph adalah struktur data non-linier.

Definisi pohon umum

- Pohon umum (**Generic tree**) merupakan kumpulan dari simpul (**node**) yang terhubung/terkait dengan pointer.
- Pohon berakar (**rooted tree**) memiliki sebuah simpul khusus yang merupakan simpul awal yang disebut dengan **root** r atau akar dan nol atau lebih sub-pohon (**sub-trees**) T_1, T_2, \dots, T_k dimana setiap root dari sub-tree terhubung ke r [2].
- Root dari setiap sub-tree merupakan anak (**child**) dari r dan r adalah induk (**parent**) dari setiap sub-tree.
- Setiap simpul pohon umum dapat memiliki anak berapa pun.
- Simpul tanpa anak disebut dengan daun (**leaf**).
- Simpul-simpul (nodes) dengan induk yang sama disebut saudara kandung (**siblings**).

Definisi pohon umum

- Sebuah lintasan dari n_1 ke n_k merupakan urutan dari n_1, n_2, \dots, n_k sedemikian hingga n_i adalah induk dari n_{i+1} untuk

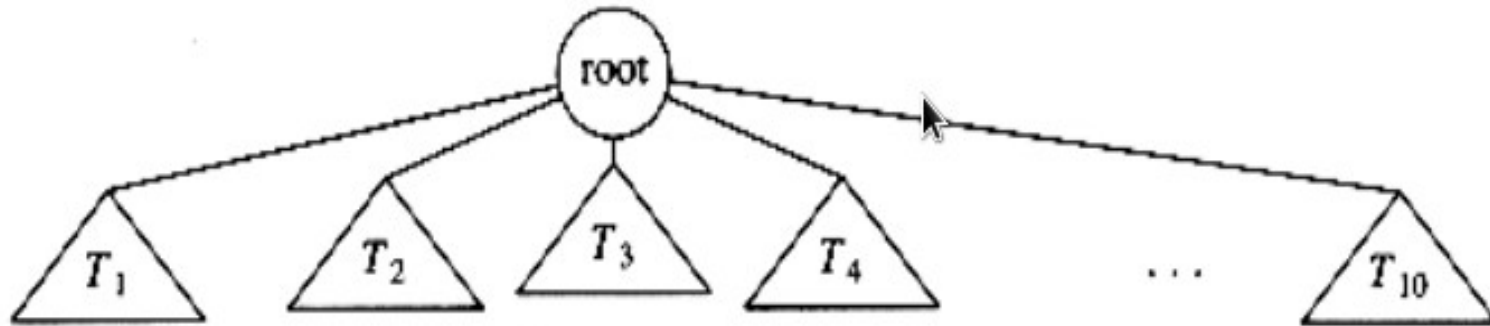
$$1 \leq i < k$$

- Untuk sembarang node n_i , kedalaman n_i adalah panjang lintasan unik dari root ke n_i . Dengan demikian, kedalaman root adalah 0 (nol).
- Untuk sembarang node n_i , tinggi n_i adalah panjang lintasan maksimum dari n_i ke **leaf**. Dengan demikian, tinggi daun (**leaf**) adalah 0 (nol).

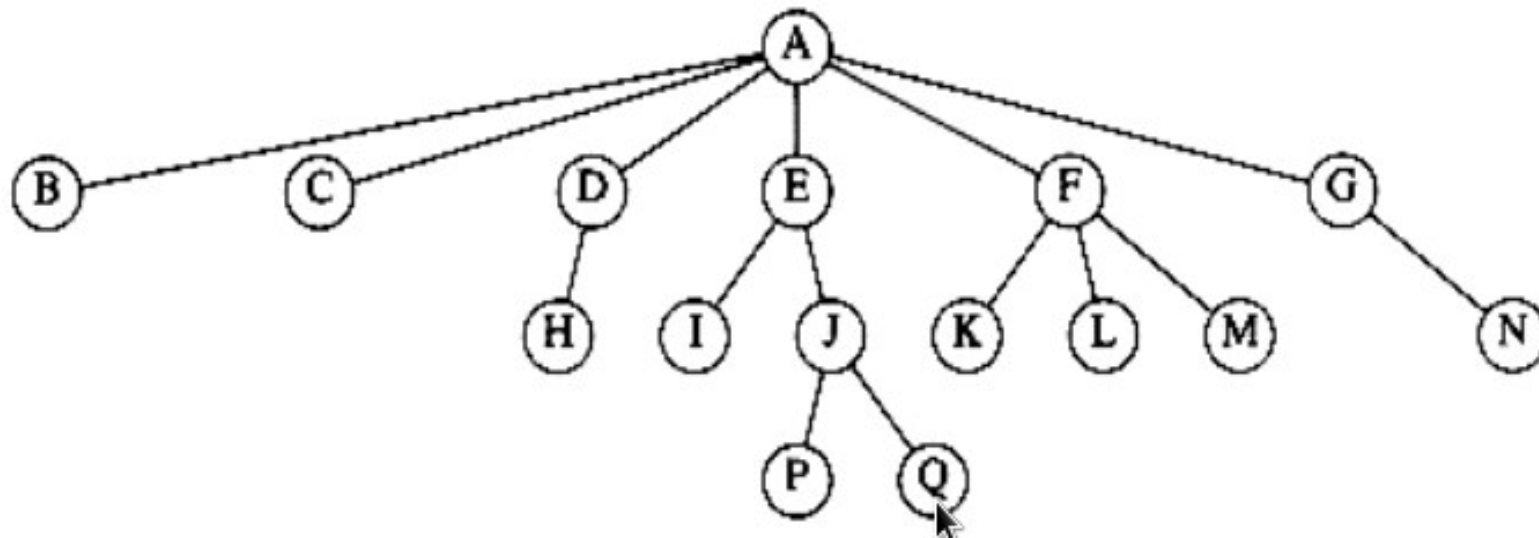
Definisi pohon umum

- Dengan definisi pada halaman sebelumnya, **tinggi pohon adalah sama dengan tinggi root**.
- Tinggi pohon selalu sama dengan kedalaman leaf terdalam sama pada pohon. Ini sama dengan kedalaman pohon.
- Diberikan 2 node n_1 dan n_2 , dimana n_1 lebih tinggi dari n_2 . Jika terdapat lintasan dari n_1 ke n_2 , maka setiap node n_i , $n_1 > n_i > n_2$ (*tanda > dibaca lebih tinggi*) yang ada pada lintasan merupakan leluhur atau nenek moyang (**ancestor**) dari n_2 dan n_2 merupakan turunan (**descendant**) dari setiap node pada lintasan.
- Setiap node merupakan ancestor dan descendant dari node itu sendiri. Dengan demikian jika $n_i \neq n_2$ maka n_i adalah benar-benar moyang (**proper ancestor**) dari n_2 dan n_2 benar-benar keturunan (**proper descendant**) dari n_i .

Generic tree



Sebuah pohon umum (1)



Sebuah pohon umum (2)

- Root \rightarrow node A
- Sub-tree \rightarrow D, E, F, G, J
- Leaves \rightarrow B, C, H, I, P, Q, K, L, M, N
- Tinggi pohon \rightarrow 3
- E berada pada kedalaman 1 dan tinggi 2

Implementasi tree generik

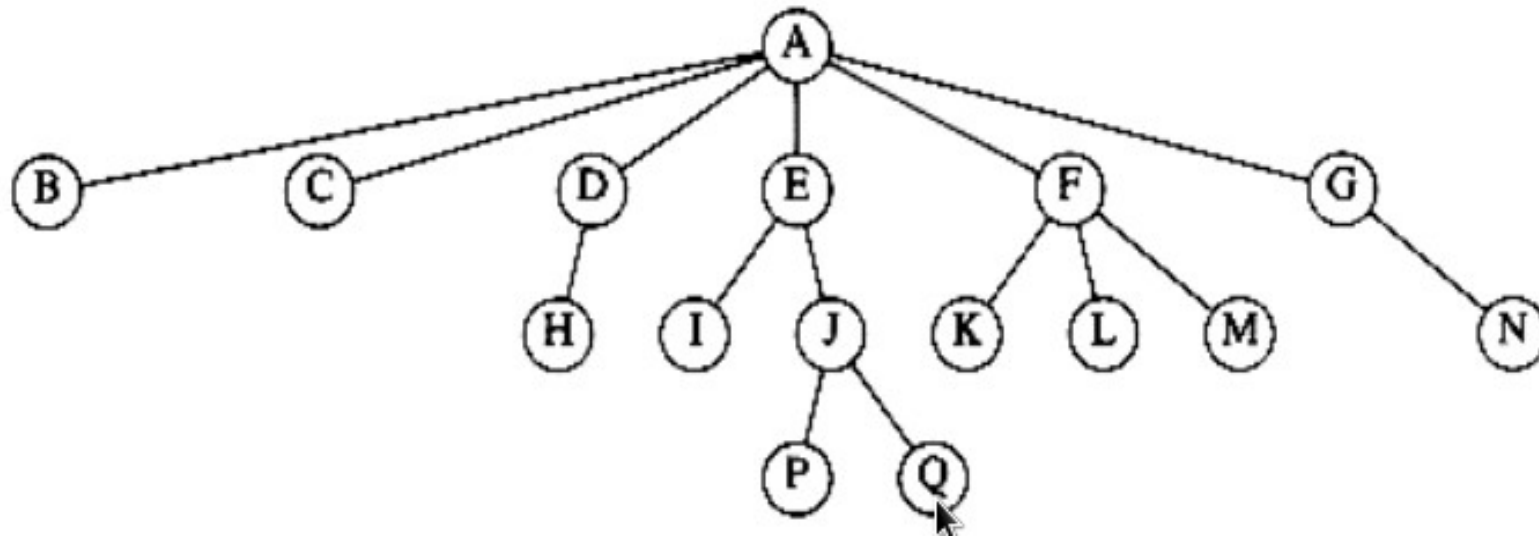
- Salah satu cara menerapkan pohon umum adalah dengan membuat node yang menyimpan elemen dan sebuah pointer ke setiap anak (**child**).
- Akan tetapi, karena jumlah anak (**children**) dapat bervariasi dan dengan demikian sulit untuk diketahui sebelumnya, maka cara ini tidak efisien. Tidak baik jika seorang programmer harus membuat jumlah pointer yang sangat banyak untuk dapat menampung anak dari node tersebut.
- Untuk mengatasinya diberikan solusi sederhana dimana, membuat semua anak dari node (root dari sub-tree) ke dalam linked list

Implementasi tree generik

```
1 typedef struct tree_node *tree_ptr;  
2  
3 struct tree_node{  
4     element_type element;  
5     tree_ptr first_child;  
6     tree_ptr next_sibling;  
7 };  
8
```

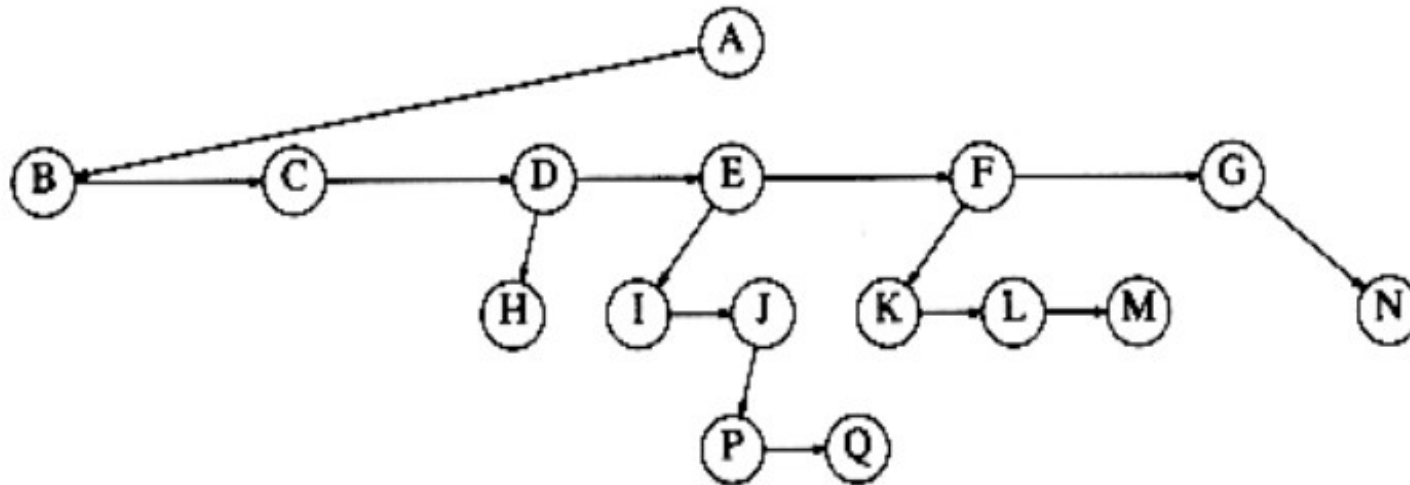
Contoh implementasi tree generik

Sebuah pohon umum:



Contoh implementasi generik tree

Implementasi pohon umum



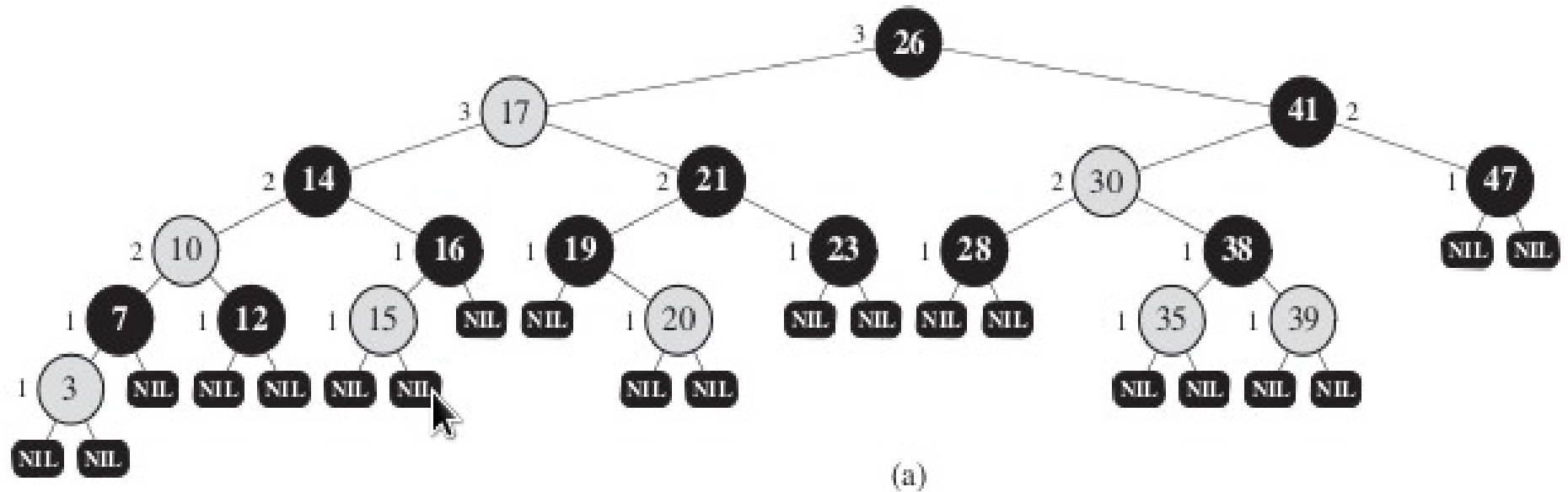
Varian tree

- Ada beberapa varian tree generik yang dibedakan oleh struktur pengorganisasian data pada tree.
- Beberapa varian tersebut adalah
 - **Binary tree**: tree dengan jumlah anak sebanyak 2 untuk setiap node.
 - **Binary search tree**: binary tree dengan properti tambahan yakni, untuk setiap node x pada, tree nilai kunci pada sub-tree sebelah kiri x selalu lebih kecil dari nilai kunci x , sedangkan nilai kunci pada sub-tree sebelah kanan x selalu lebih besar dari nilai kunci x .
 - **AVL tree**: Binary search tree yang seimbang (self-balancing binary search tree).

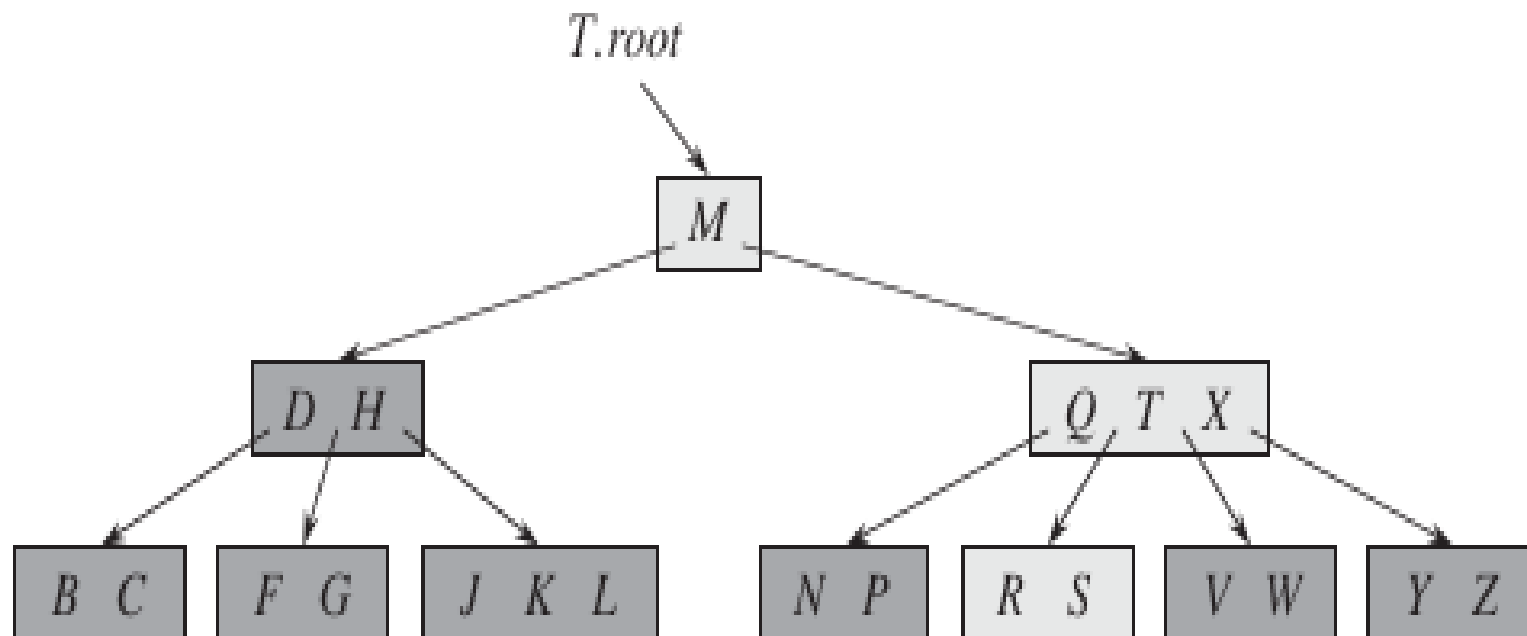
Varian tree

- **Red-black tree:** binary search tree yang seimbang dimana setiap node diberi warna merah (red) atau hitam sesuai dengan aturan red black tree.
- **B-tree:** suatu struktur tree yang menyimpan data secara terurut dimana anak (child) dari suatu node bisa lebih dari 2. Jadi b-tree bukanlah pohon biner. Properti yang penting adalah tree harus seimbang (balanced). B-tree lazim dipakai untuk menyimpan indeks tabel pada basis data.

Red-black tree: contoh [1]



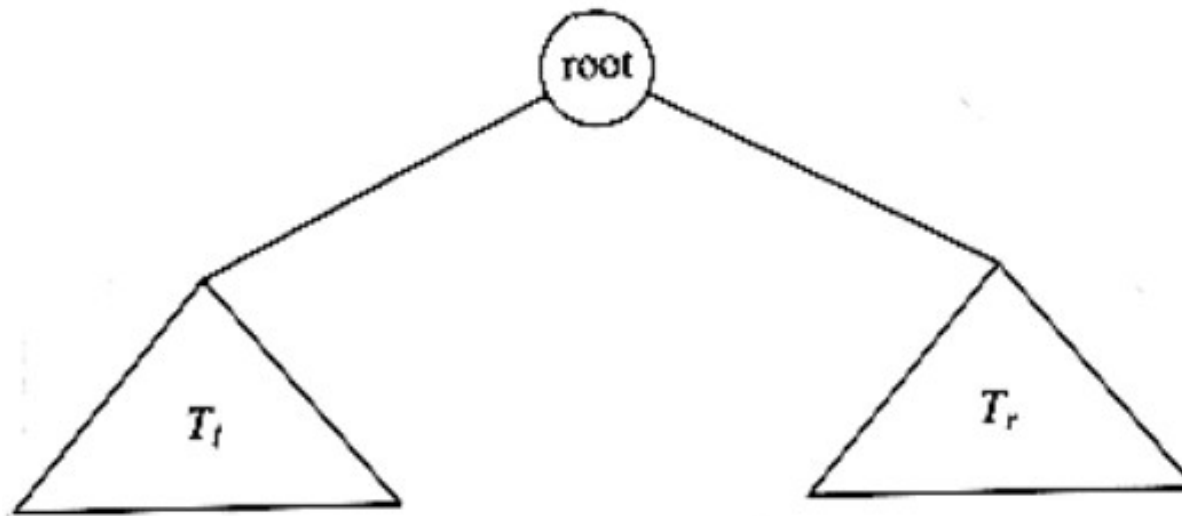
B-tree: contoh [1]



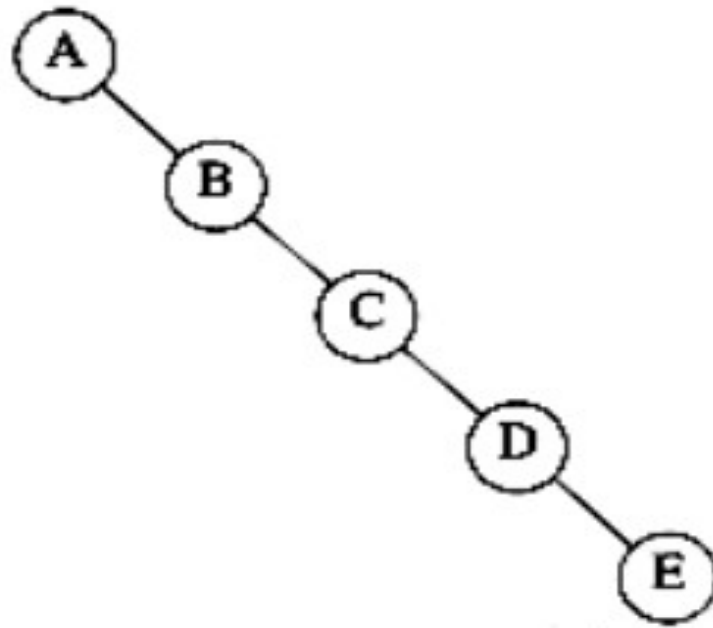
Pohon biner (*binary tree*)

- Fokus kita pada topik ini adalah pohon biner (**binary tree**), khususnya pada variannya yang disebut dengan pohon pencarian biner (**binary search tree**).
- Pada sesi praktikum, anda akan membuat struktur data untuk pohon pencarian biner (Binary Search Tree ADT). Dengan pemahaman teori dan keahlian pemrograman untuk pohon pencarian biner, anda dapat diharapkan mudah untuk mempelajari dan memprogram varian tree lainnya.
- Pohon biner adalah pohon dimana tidak ada node yang memiliki lebih dari 2 anak.

Pohon biner umum



Pohon biner terburuk (*worst-case binary tree*)



Implementasi pohon biner

- Karena setiap node pohon biner memiliki paling banyak 2 anak, kita dapat membuat 2 pointer untuk menunjuk anak tersebut.

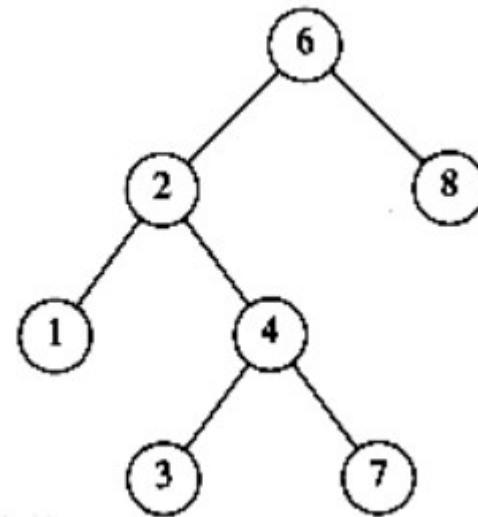
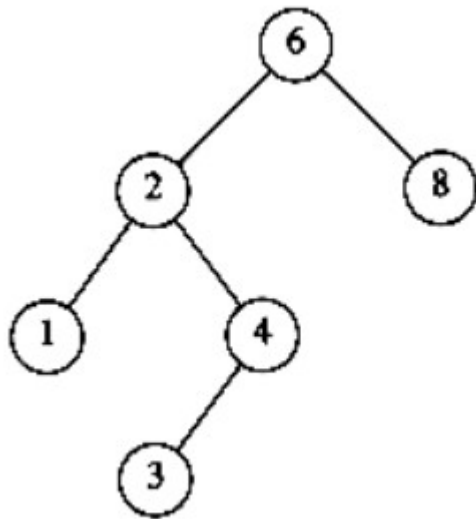
```
1 typedef struct tree_node *tree_ptr;
2
3 struct tree_node{
4     element_type element;
5     tree_ptr left;
6     tree_ptr right;
7 };
8
9 typedef tree_ptr BINARY_TREE;
```

Pohon pencarian biner (*binary search tree*)

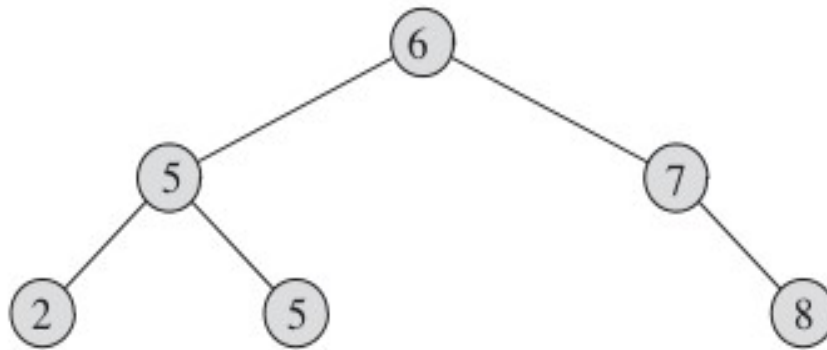
- **Pohon pencarian biner atau binary search tree** adalah pohon biner dengan properti tambahan yakni, untuk setiap node x pada pohon, nilai kunci pada sub-tree sebelah kiri x selalu lebih kecil dari nilai kunci x , sedangkan nilai kunci pada sub-tree sebelah kanan x selalu lebih besar dari nilai kunci x .

Contoh pohon pencarian biner

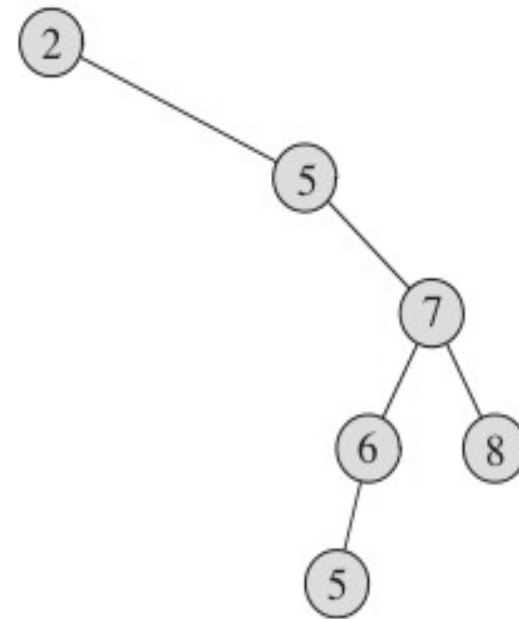
- Gambar sebelah kiri adalah pohon pencarian biner, sedangkan sebelah kanan



Pohon pencarian biner contoh efisien dan tidak efisien



(a)

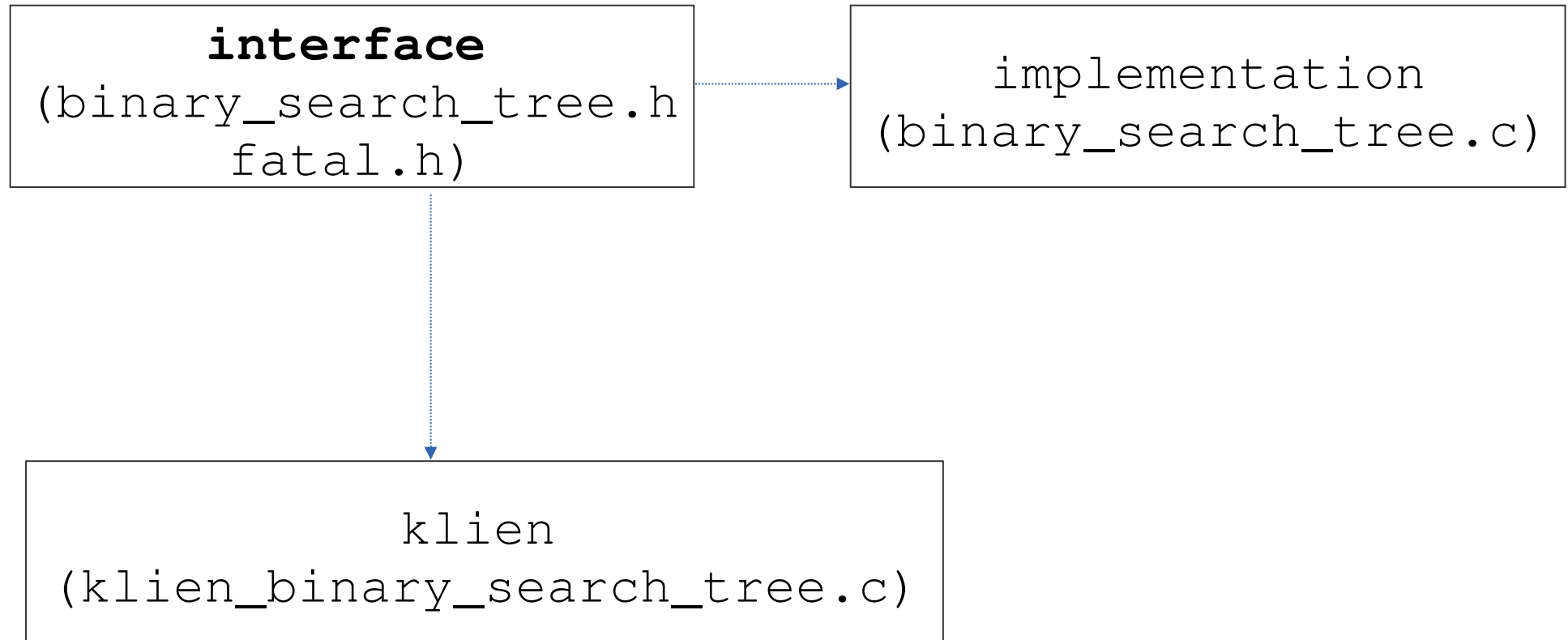


(b)

Operasi pada pohon pencarian biner

- Operasi-operasi minimum yang dapat dilakukan pada pohon pencarian biner adalah
 - INSERT: memasukkan elemen baru
 - FIND: mencari elemen tertentu
 - MINIMUM: mencari elemen minimum
 - MAXIMUM: mencari elemen maksimum
 - DELETE: menghapus elemen
- Operasi-operasi tersebut dapat ditambahkan sesuai dengan kebutuhan pembuat struktur data, contohnya dengan menambahkan operasi *MakeEmpty* untuk mengosongkan pohon pencarian biner.

Binary Search Tree ADT



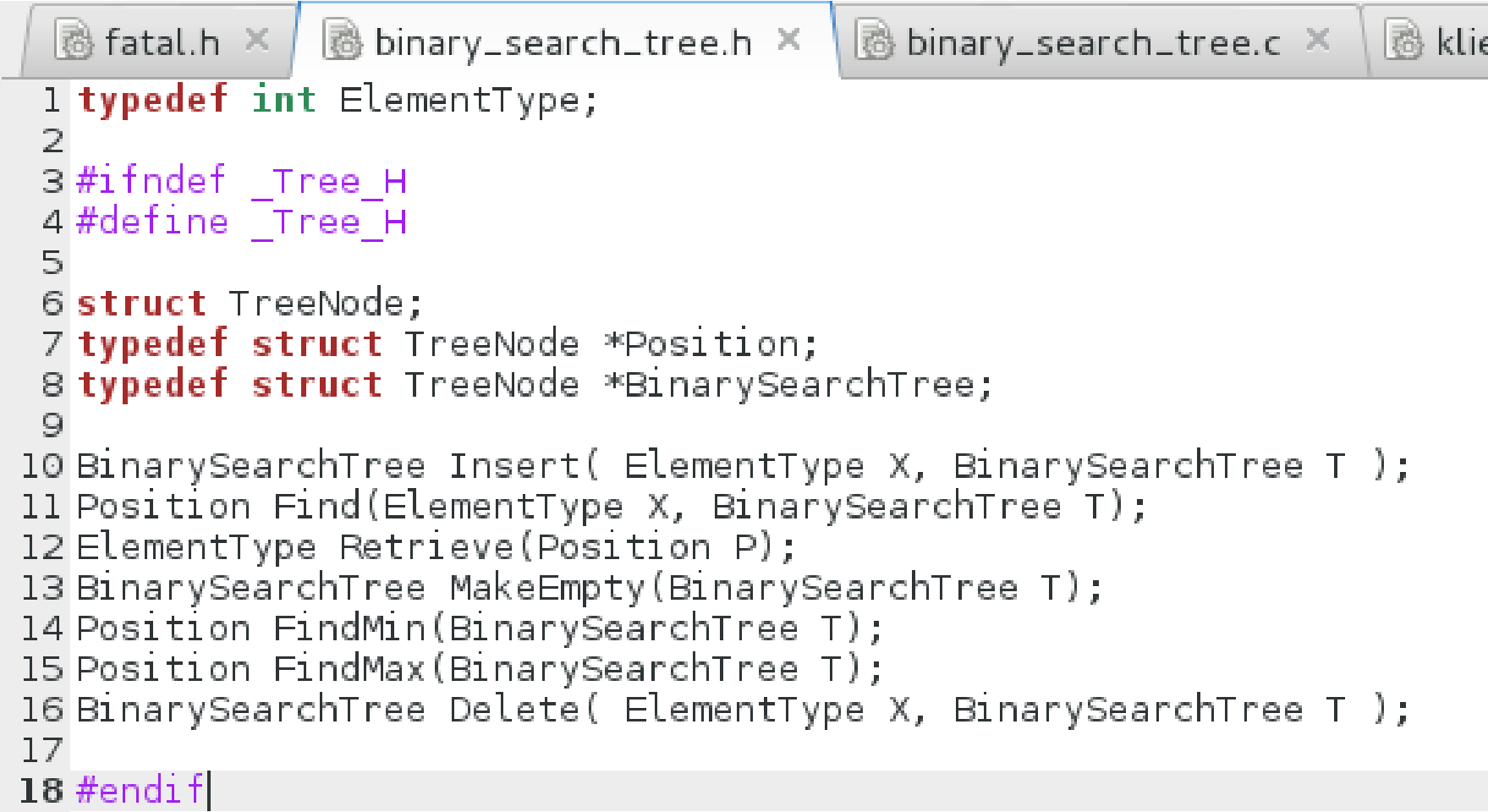
Antarmuka `fatal.h`

- Antarmuka `fatal.h` berguna untuk mencetak pesan error ke monitor. Ada 2 fungsi yang didefinisikan pada antarmuka tersebut menggunakan makro prapemrosesan: `#define`.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define Error(Str)      FatalError(Str)
5 #define FatalError(Str) fprintf(stderr, "%s\n", Str), exit(1)
```

Antarmuka

binary_search_tree.h



```
1 typedef int ElementType;
2
3 #ifndef _Tree_H
4 #define _Tree_H
5
6 struct TreeNode;
7 typedef struct TreeNode *Position;
8 typedef struct TreeNode *BinarySearchTree;
9
10 BinarySearchTree Insert( ElementType X, BinarySearchTree T );
11 Position Find( ElementType X, BinarySearchTree T );
12 ElementType Retrieve( Position P );
13 BinarySearchTree MakeEmpty( BinarySearchTree T );
14 Position FindMin( BinarySearchTree T );
15 Position FindMax( BinarySearchTree T );
16 BinarySearchTree Delete( ElementType X, BinarySearchTree T );
17
18 #endif
```

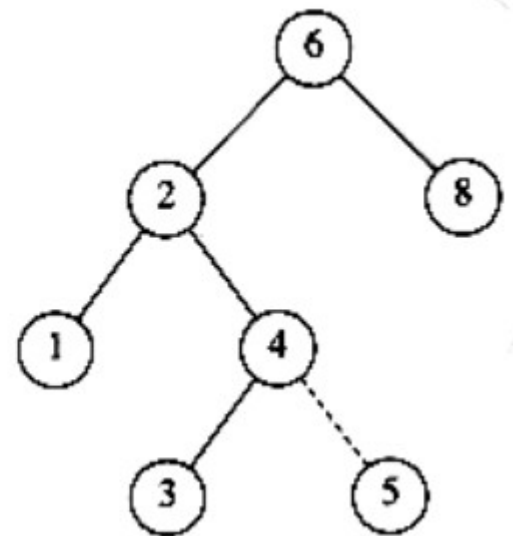
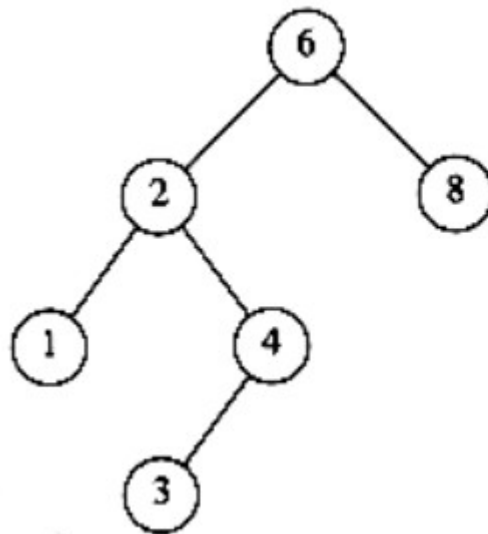
Definisi struct TreeNode

- Setiap node pada pohon pencarian biner direpresentasikan oleh struktur: `struct TreeNode` dengan definisi sebagai berikut.

```
5 struct TreeNode{  
6     ElementType Element;  
7     BinarySearchTree Left;  
8     BinarySearchTree Right;  
9 };
```

Operasi insert

- Kita asumsikan pohon pencarian biner memiliki nilai kunci yang berbeda untuk setiap node.
- Pencarian dilakukan secara rekursif dimulai dari root sampai ke simpul dimana elemen baru hendak dimasukkan.
- Ilustrasi:



Operasi insert

```
11 BinarySearchTree Insert(ElementType X, BinarySearchTree T){
12     if(T == NULL){
13         T = malloc(sizeof(struct TreeNode) );
14         if( T == NULL )
15             FatalError( "Out of space!!!" );
16         else{
17             T->Element = X;
18             T->Left = T->Right = NULL;
19         }
20     }else
21         if( X < T->Element )
22             T->Left = Insert(X, T->Left);
23         else if( X > T->Element )
24             T->Right = Insert(X, T->Right);
25
26     return T;
27 }
```

Pencarian elemen pada **binary search tree**

- Operasi ini direpresentasikan oleh fungsi Find yang telah dideklarasikan pada antarmuka sbb.:

```
Position Find(ElementType X, BinarySearchTree  
    T) ;
```

- Terlihat fungsi tersebut mengembalikan pointer ke simpul (node): `TreeNode` yang memiliki nilai kunci `x` dan mengembalikan `NULL` jika tidak ada node yang memiliki nilai `x` tersebut.
- Pencarian dilakukan dengan fungsi rekursif untuk mencari node yang memiliki nilai `x`. Jika belum didapat maka pencarian node selanjutnya dilanjutkan.

Definisi fungsi `find`

```
40  
47 Position Find( ElementType X, BinarySearchTree T ){  
48     if( T == NULL )  
49         return NULL;  
50     if( X < T->Element )  
51         return Find( X, T->Left );  
52     else if( X > T->Element )  
53         return Find( X, T->Right );  
54     else  
55         return T;  
56 }
```

Pencarian nilai minimum dan maksimum pada **Binary Search ADT**

- Pencarian nilai kunci minimum diwakili oleh fungsi `FindMin` yang dideklarasikan pada antar muka sbb.:
`Position FindMin(BinarySearchTree T);`
- Pencarian nilai kunci maksimum diwakili oleh fungsi `FindMax`:
`Position FindMax(BinarySearchTree T);`
- Pencarian nilai minimum selalu dimulai dari root dan selanjutnya melintasi sub-tree (node selanjutnya) yang merupakan anak pada simpul sebelah kiri.
- Sebaliknya, pencarian nilai maksimum dilakukan dengan melintasi anak yang berada pada cabang sebelah kanan.

Pencarian nilai minimum dan maksimum pada **Binary Search ADT**

- Pencarian nilai minimum dilakukan secara rekursif.
- Pencarian nilai maksimum dilakukan secara non-rekursif.

Definisi fungsi FindMin dan FindMax

```
58 Position FindMin( BinarySearchTree T ){
59     if( T == NULL )
60         return NULL;
61     else
62         if( T->Left == NULL )
63             return T;
64         else
65             return FindMin( T->Left );
66 }
67 Position FindMax( BinarySearchTree T ){
68     if( T != NULL )
69         while( T->Right != NULL )
70             T = T->Right;
71
72     return T;
73 }
```

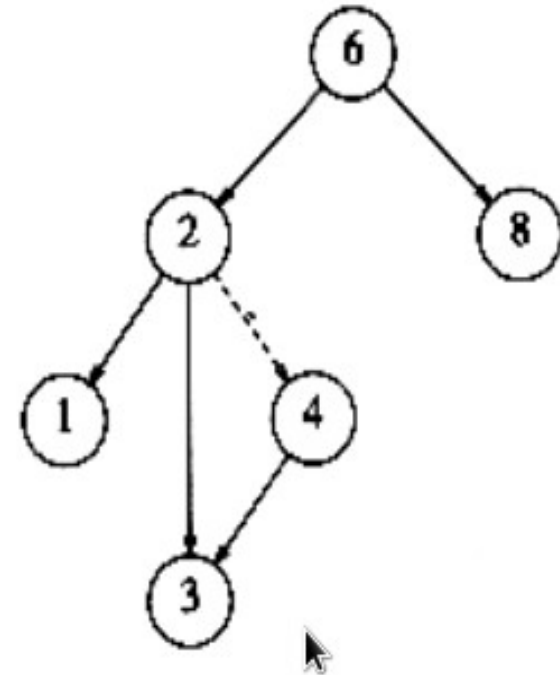
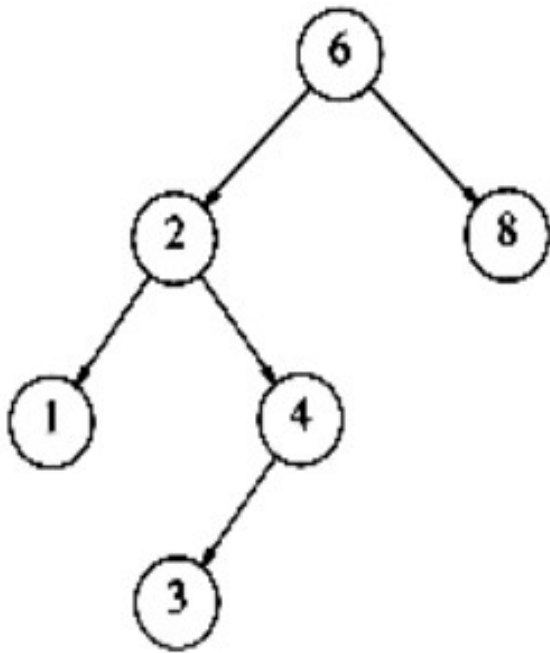
Penghapusan pada Binary Search Tree

- Penghapusan elemen diwakili oleh fungsi `Delete` yang dideklarasikan sbb. pada fail antar muka.

```
BinarySearchTree Delete(ElementType X, BinarySearchTree T);
```

- Operasi penghapusan elemen merupakan operasi yang paling sulit sebab ada *update*/ perubahan tree.
- Jika node merupakan leaf maka penghapusan dapat dilakukan secara langsung dan ini adalah proses sederhana.
- Akan tetapi, penghapusan node yang bukan leaf adalah proses yang lebih kompleks.

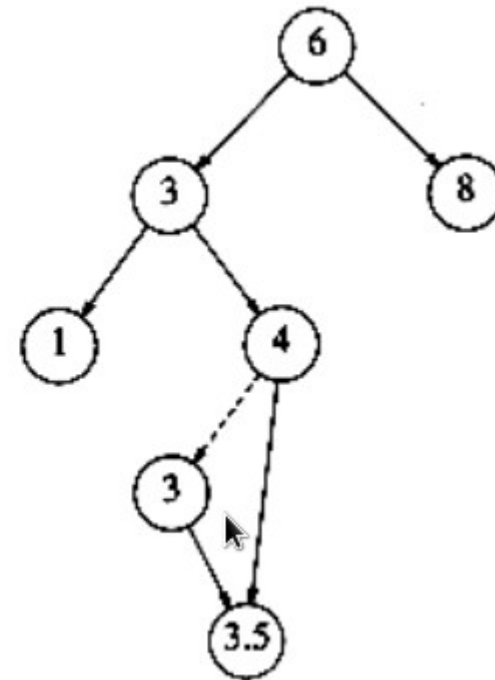
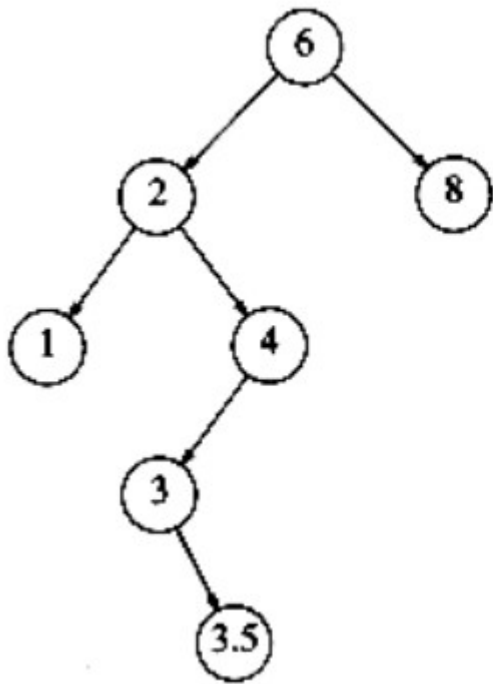
Penghapusan node dengan satu anak



Penghapusan node dengan 2 anak

- Penghapusan node dengan 2 anak dilakukan dengan mengganti nilai kunci (elemen) dari simpul yang mana elemennya hendak dihapus dengan nilai kunci terkecil pada sub-tree sebelah kanan. Selanjutnya menghapus (*free*) simpul dengan nilai kunci terkecil yang berada pada sub-tree sebelah kanan tersebut.

Penghapusan node dengan 2 anak



Definisi fungsi Delete

```
74
75 BinarySearchTree Delete( ElementType X, BinarySearchTree T ){
76     Position TmpCell;
77
78     if( T == NULL )
79         Error( "Element not found" );
80     else if( X < T->Element ) /* Go left */
81         T->Left = Delete( X, T->Left );
82     else if( X > T->Element ) /* Go right */
83         T->Right = Delete( X, T->Right );
84     else /* Found element to be deleted */
85         if( T->Left && T->Right ){ /* Two children */
86             /* Replace with smallest in right subtree */
87             TmpCell = FindMin( T->Right );
88             T->Element = TmpCell->Element;
89             T->Right = Delete( T->Element, T->Right );
90         }else{ /* One or zero children */
91             TmpCell = T;
92             if( T->Left == NULL ) /* Also handles 0 children */
93                 T = T->Right;
94             else if( T->Right == NULL )
95                 T = T->Left;
96             free( TmpCell );
97         }
98     return T;
99 }
```

Kompleksitas operasi dasar **Binary Search Tree**

- Yang dimaksud dengan operasi dasar adalah operasi pemasukan (**insert**), pencarian (**search** atau **find**), dan penghapusan elemen (**delete**).
- Setelah mempelajari teori dan melakukan praktikum pembuatan Binary Search Tree ADT dapat anda pahami bahwa operasi dasar tersebut di atas dilakukan dengan melakukan pelintasan pohon (*tree traversal*) dari akar (**root**) sampai pada posisi simpul yang dituju.
- Dengan demikian pelintasan pohon tergantung kedalaman pohon. Semakin dalam, maka semakin lama eksekusi operasi dasar tersebut. Demikian juga sebaliknya.

Kompleksitas operasi dasar **Binary Search Tree**

- Jika variabel h disebut sebagai tinggi pohon (**tree height**), maka kompleksitas waktu operasi dasar adalah $O(h)$
- **Kondisi terburuk (*worst case*)**

Kondisi ini tercapai untuk pohon yang buruk (***degenerate tree***) dimana

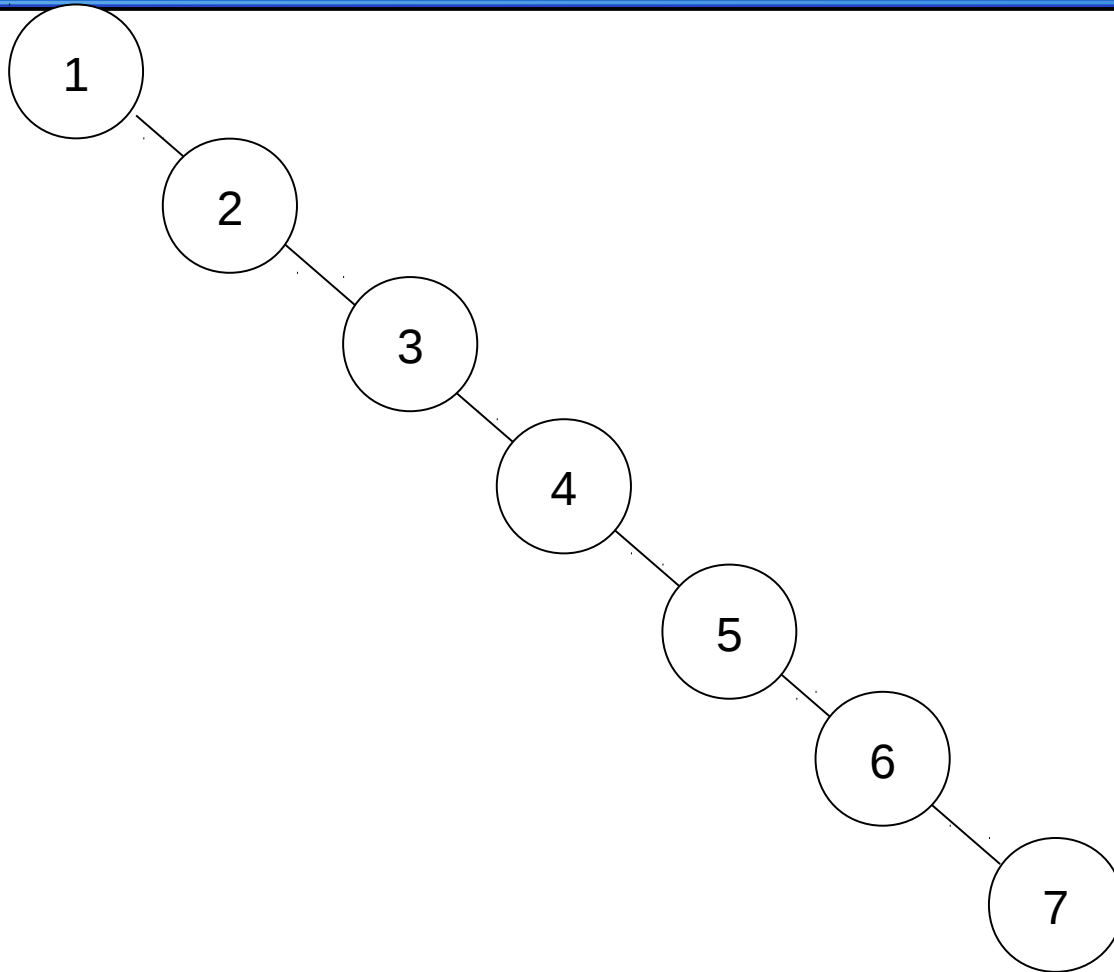
$$h = n - 1; n = \text{jumlah elemen (simpul)}$$

- **Kondisi terbaik (*best case*)**

Kondisi ini tercapai untuk pohon yang seimbang dimana

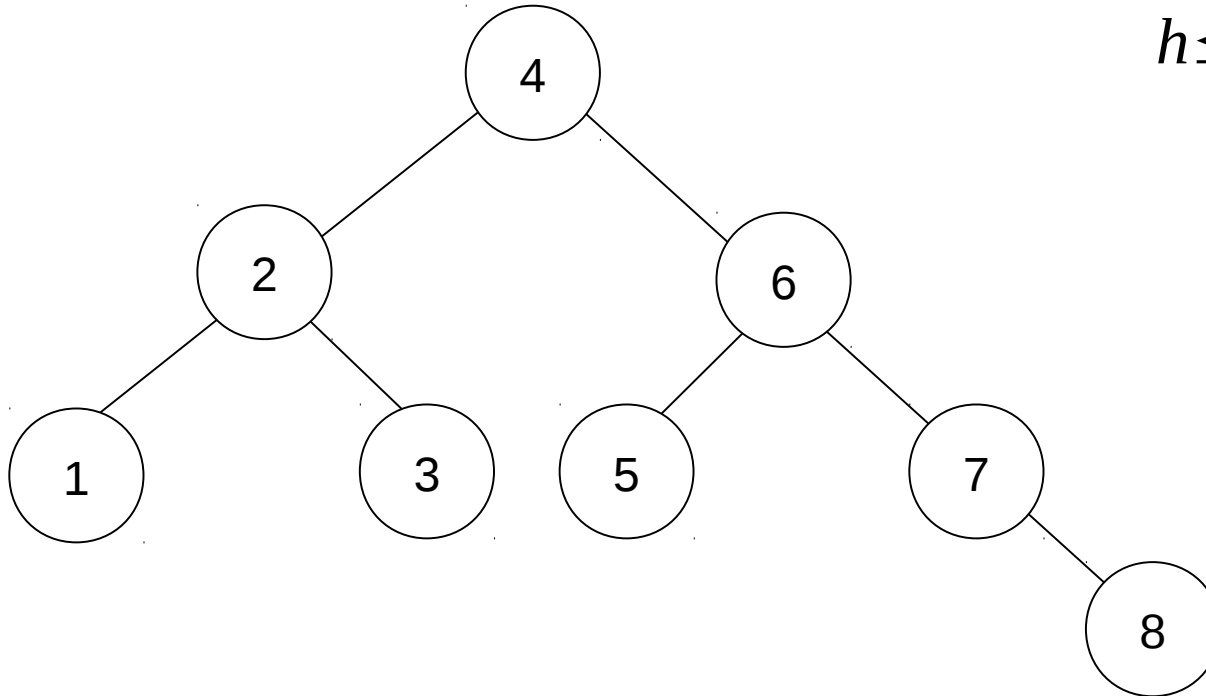
$$h = \lg(n); n = \text{jumlah elemen (simpul)}$$

Kompleksitas operasi dasar BST: *worst case*



Kompleksitas operasi dasar BST: *best case*

- Jumlah node setiap level = 2^h
- Untuk tinggi h , berlaku relasi berikut: $2^h \leq n$, sedemikian hingga $h \leq \lg(n)$



Referensi

1. T.H.Cormen, C.E.Leiserson, R.L.Rivest & C. Stein, *Introduction to Algorithms, 2nd eds.*, MIT Press 2001.
2. M.A.Weiss, *Data Structures and Algorithm Analysis in C, 2nd eds.*, Addison-Wesly, 1997.