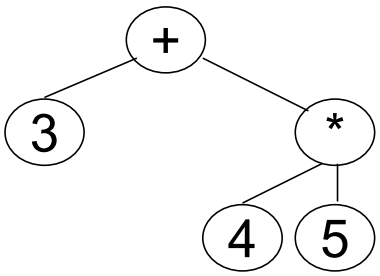
Pohon Biner (Bagian 2)

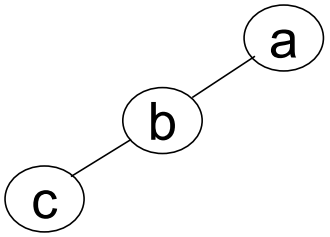
IF2110 – Algoritma dan Struktur Data
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung

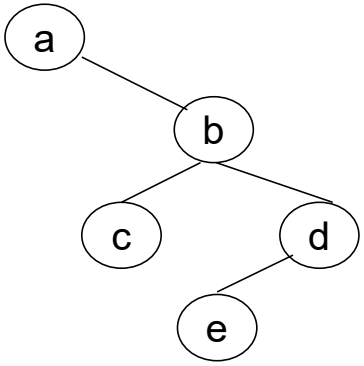
Pohon Biner

Pohon biner adalah himpunan terbatas yang

- mungkin **kosong**, atau
- terdiri atas sebuah simpul yang disebut **akar** dan dua buah himpunan lain yang *disjoint* yang merupakan pohon biner, yang disebut sebagai **sub pohon kiri** dan **sub pohon kanan** dari pohon biner tersebut







Pohon Seimbang

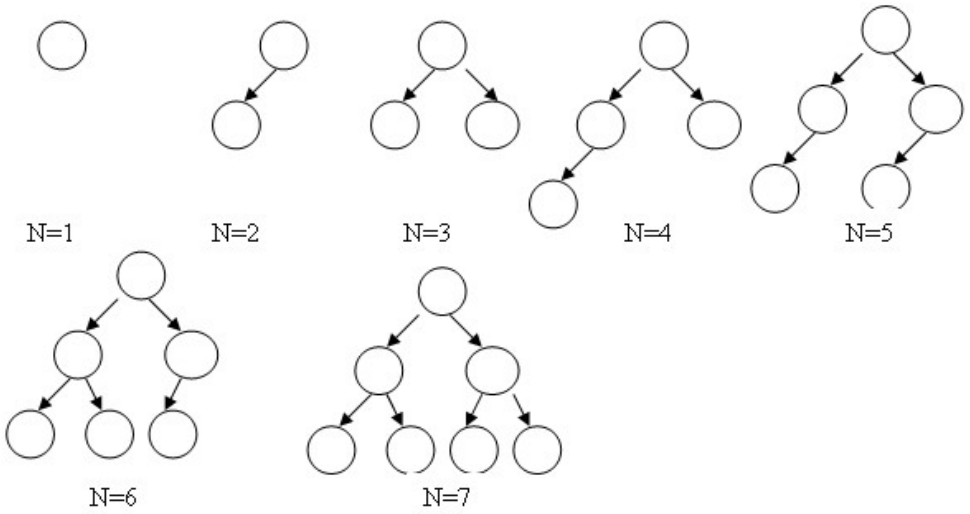
Pohon seimbang (*balanced tree/B-tree*) adalah pohon dengan:

- Perbedaan tinggi subpohon kiri dan subpohon kanan maksimum 1
- Perbedaan banyaknya simpul subpohon kiri dan subpohon kanan maksimum 1
- Subpohon kiri dan subpohon kanan adalah pohon seimbang

Aplikasi: pengelolaan indeks dalam file system dan database system

Yang akan dibahas adalah pohon biner seimbang (balanced binary tree)

Pohon Biner Seimbang



Algoritma untuk membuat pohon biner seimbang dari n buah node

```
function buildBalancedTree (n: integer) → BinTree
{ Menghasilkan sebuah balanced tree }
{ Basis: n = 0: Pohon kosong  }
{ Rekurens: n>0: partisi banyaknya node anak kiri dan kanan,
                 lakukan proses yang sama  }
KAMUS LOKAL
    p: Address; l, r: BinTree; x: ElType
    nL, nR: integer
ALGORITMA
    if (n = 0) then { Basis-0 }
       → NIL
    else { Rekurens }
       { bentuk akar }
       input(x) { mengisi nilai akar }
       p ← newTreeNode(x)
       if (p ≠ NIL) then
          { Partisi sisa node sebagai anak kiri dan anak kanan }
          nL ← n div 2; nR ← n – nL - 1
          l ← buildBalancedTree(nl); r ← buildBalancedTree(nr)
          p↑.left ← l; p↑.right ← r
       → p
```

Binary Search Tree - 1

Binary Search Tree (BST)/pohon biner terurut/pohon biner pencarian adalah pohon biner yang memenuhi sifat:

- Setiap simpul dalam BST mempunyai sebuah nilai
- Subpohon kiri dan subpohon kanan merupakan BST
- Jika p adalah sebuah BST:
  - semua simpul pada subpohon kiri < Akar p
  - semua simpul pada subpohon kanan >= Akar p

Aplikasi BST: algoritma searching dan sorting tingkat lanjut

Binary Search Tree - 2

Nilai simpul (key) dalam BST bisa unik bisa juga tidak.

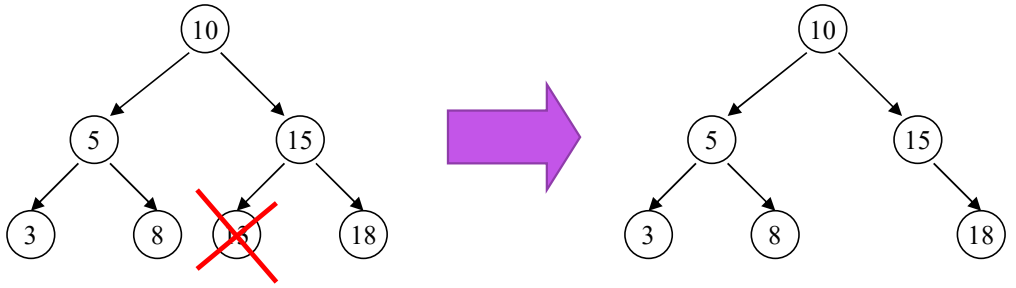
Pada pembahasan ini semua simpul BST (key) bernilai unik. Banyak kemunculan suatu nilai key disimpan dalam field “count”.

```
type ElType: < key: ..., { terdefinisi }
             count: integer >
type Node:  < info: ElType,
              left: BinTree,
             right: BinTree >
type BinTree: Address
```

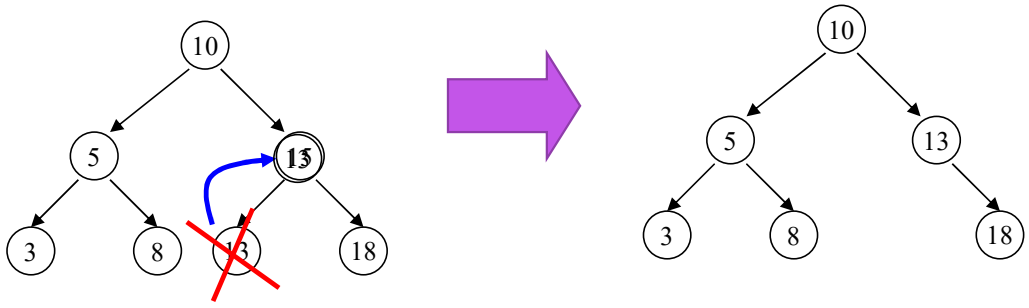
Insert Node dalam BST

```
procedure insSearchTree(input x: ElType, input/output p: BinTree)
{ Menambahkan sebuah node x ke pohon biner pencarian p }
{ ElType terdiri dari key dan count. Key menunjukkan nilai unik, dan count berapa kali muncul }
{ Basis: Pohon kosong  }
{ Rekurens: Jika pohon tidak kosong, insert ke anak kiri jika nilai < p↑.info.key }
{ Atau insert ke anak kanan jika nilai > p↑.info.key }
{ Perhatikan bahwa insert selalu menjadi daun terkiri/terkanan dari subpohon }
KAMUS LOKAL
   -
ALGORITMA
   if (isTreeEmpty(p)) then { Basis: buat hanya akar }
      CreateTree(x,NIL,NIL,p)
   else { Rekurens }
      depend on x, p↑.info.key
         x.key = p↑.info.key : p↑.info.count ← p↑.info.count + 1
         x.key < p↑.info.key : insSearchTree(x, p↑.left)
         x.key > p↑.info.key : insSearchTree(x, p↑.right)
```

Delete Node dalam BST

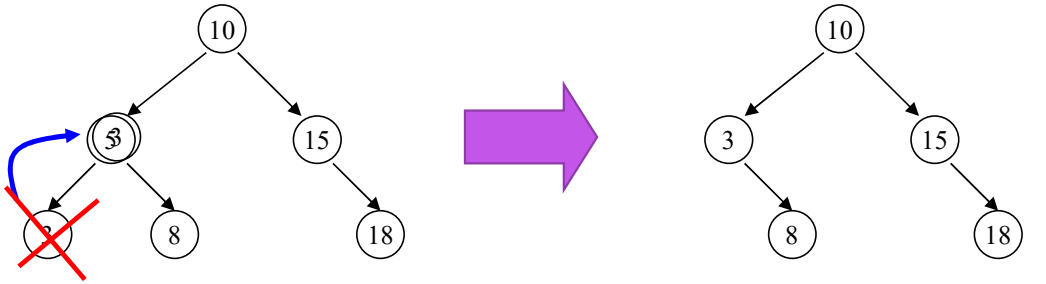


DelBTree(p,13)

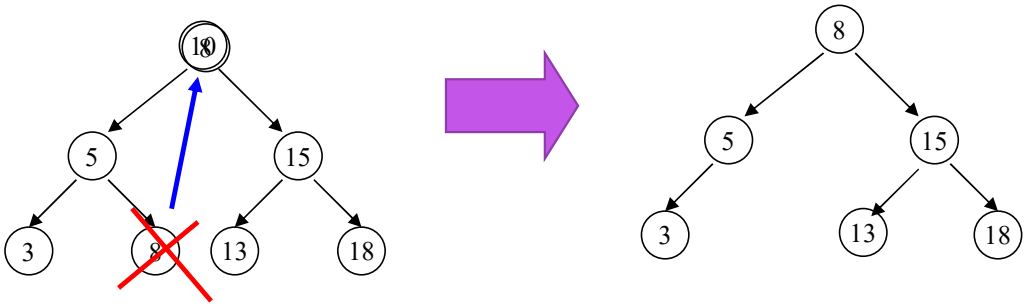


DelBTree(p,15)

Delete Node dalam BST



DelBTree(p,5)



DelBTree(p,10)

Delete Simpul dalam BST - 1

```
procedure delBTree (input/output p: BinTree, input x: ElType)
{ Menghapus simpul bernilai p↑.info.key = x.key, asumsi: x.key pasti ada di p}
{ ElType terdiri dari key dan count. Key menunjukkan nilai unik, dan count berapa kali
  muncul }
{ Basis: ? ; Rekurens: ? }
KAMUS LOKAL
   q: Address
   procedure delNode (input/output p: BinTree)
   { … }
ALGORITMA
   depend on x, p↑.info.key
     x.key < p↑.info.key : delBTree(p↑.left,  x)
     x.key > p↑.info.key : delBTree(p↑.right, x)
     x.key = p↑.info.key : { Delete simpul ini }
                      q ← p
                      depend on q
                         isOneElmt(q)  : p ← NIL
                         isUnerLeft(q) : p ← q↑.left
                         isUnerRight(q): p ← q↑.right
                         isBiner(q)    : delNode(q↑.left)
                      deallocTreeNode(q)
```

Delete Simpul dalam BST - 2

```
procedure delNode (input/output p: BinTree)
{ I.S. p adalah pohon biner tidak kosong }
{ F.S. q berisi salinan nilai daun terkanan }
{ Proses: }
{ Memakai nilai q yang global}
{ Traversal sampai NODE terkanan, copy nilai NODE terkanan p,
  salin nilai ke q semula }
{ q adalah NODE TERKANAN yang akan dihapus }
KAMUS LOKAL
   -
ALGORITMA
   depend on p
      p↑.right ≠ NIL: delNode(p↑.right)
      p↑.right = NIL:       { p anak terkanan }
         q↑.info.key ← p↑.info.key    { salin info p ke q }
         q↑.info.count ← p↑.info.count
         q ← p              { q: yang akan dihapus }
         p ← p↑.left        { pastikan anak kiri "naik", jika ada }
```

Membentuk Pohon Biner dari Pita Karakter

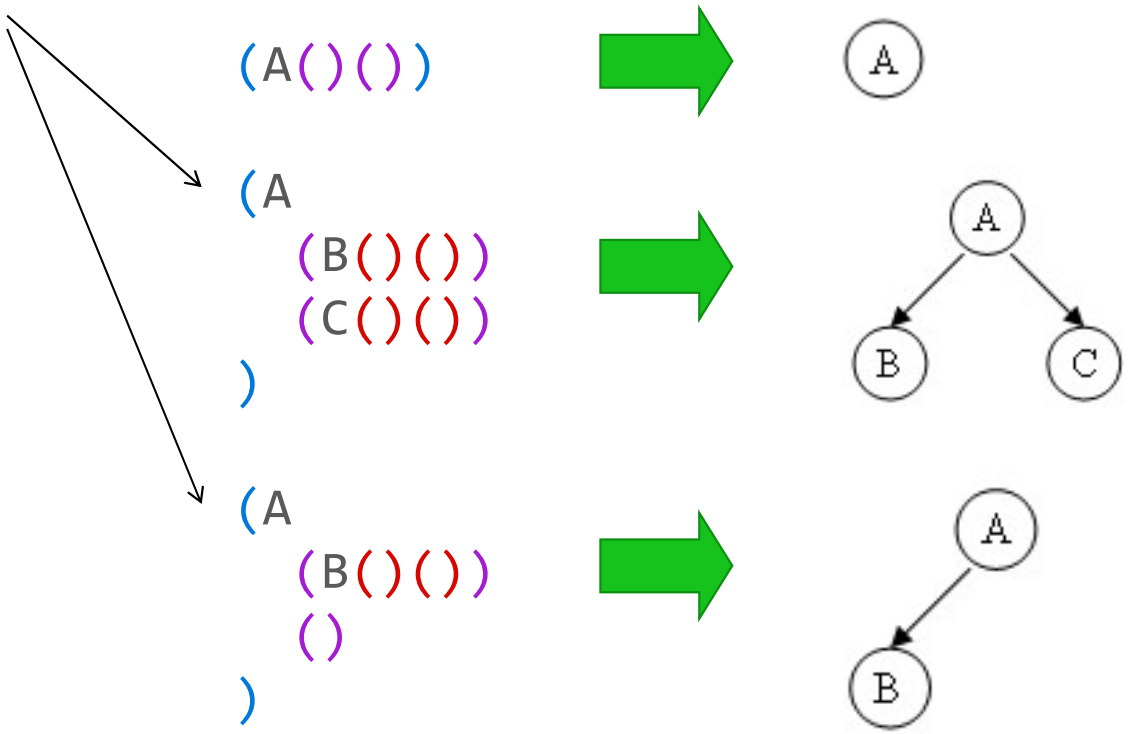
Ekspresi pohon dalam bentuk linier (list) dapat dituliskan dalam sebuah pita karakter

Ada 2 ide:

- Membangun pohon secara iteratif
- Membangun pohon secara rekursif

Contoh - 1

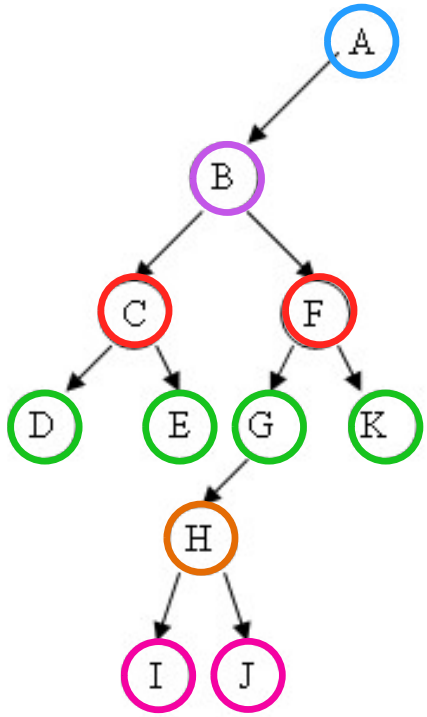
Ditulis dengan ganti baris, berwarna, dan indentasi untuk memudahkan pembacaan



Contoh - 2

Ditulis dengan ganti baris, berwarna, dan indentasi untuk memudahkan pembacaan

```
(A
  (B
    (C
      (D()())
      (E()()))
    (F
      (G
        (H
          (I()())
          (J()()))
        ())
      (K()())))
  ()
)
```



Ide 1: Membangun Pohon secara Iteratif

Karena pembacaan pita dilakukan secara sekuensial, pembentukan pohon selalu dimulai dari akar

Pembacaan karakter demi karakter dilakukan secara iteratif, untuk membentuk sebuah pohon, selalu dilakukan insert terhadap daun

Struktur data memerlukan pointer ke “Bapak”, dengan demikian yang dipakai adalah:

```
type Node: < parent: Address,
             left: Address,
             info: character,
            right: Address >
```

Ide Algoritma Membangun Pohon

Ada tiga kelompok karakter:

- Karakter berupa abjad, menandakan bahwa sebuah node harus dibentuk, entah sebagai anak kiri atau anak kanan.
- Karakter berupa ‘(‘ menandakan suatu sub pohon baru.
  - Jika karakter sebelumnya adalah ‘)’ maka siap untuk melakukan insert sub pohon kanan.
  - Jika karakter sebelumnya adalah abjad, maka siap untuk melakukan insert sub pohon kiri.
- Karakter berupa ‘)’ adalah penutup sebuah pohon, untuk kembali ke “Bapaknya”, berarti naik levelnya dan tidak melakukan apa-apa, tetapi menentukan proses karakter berikutnya.

Tidak cukup dengan mesin karakter (hanya CC), sebab untuk memproses sebuah karakter, dibutuhkan informasi karakter sebelumnya → karena itu digunakan mesin couple (C1, CC)

Implementasi dalam Bahasa C
File: tree.h

```
#include <stdlib.h>
#include "boolean.h"
#include "mesincouple.h"

typedef char ElType;
#define NIL NULL

/*** Selektor ****/
#define   INFO(p) (p)->info
#define   LEFT(p) (p)->left
#define  RIGHT(p) (p)->right
#define PARENT(p) (p)->parent

/*** Type Tree ***/
typedef struct tNode* Address;
typedef struct tNode {
   ElType  info;
   Address left;
   Address right;
   Address parent;
} Node;
typedef Address Tree;
```

Implementasi dalam Bahasa C
File: tree.h

```
Address newTreeNode(ElType x);
/* Alokasi sebuah address p, bernilai tidak NIL jika berhasil */

void CreateTree(Tree *t);
/* I.S. Sembarang */
/* F.S. t terdefinisi */
/* Proses: Membaca isi pita karakter dan membangun pohon dilakukan secara iteratif */

void displayTree (Tree t);
/* I.S. t terdefinisi */
/* F.S. t tertulis di layar */
```

```
void CreateTree (Tree *t) {
  /* Kamus Lokal */
  Address currParent;
  Address ptr;
  int level = 0;
  boolean insKi;
  /* Algoritma */
  startCouple();
  currParent = NIL;
  while (!EOP()) {
    switch (cc) {
      case '(': level++;
                insKi = ( c1 != ')'); //siap sisip kiri
                break;
      case ')': level--;
                if (c1 != '(') {
                  currParent = PARENT(CurrParent);
                }
                break;
      default : ptr = newTreeNode(cc);  /* CC adalah abjad */
                if (currParent != NIL) {
                  if (insKi) { LEFT(currParent) = ptr; }
                  else { RIGHT(currParent) = ptr; }
                } else { *t = ptr; }
                PARENT(ptr) = currParent;
                currParent = ptr;
                break;
    }
    advCouple();
  }
}
```

Implementasi dalam Bahasa C
File: tree.c
Prosedur MakeTree

Ide 2: Membangun Pohon Secara Rekursif - 1

Struktur data yang digunakan adalah tree biasa (tidak memerlukan pointer ke Bapak)

```
typedef struct tNode* Address;
typedef struct tNode {
   ElType  info;
   Address left;
   Address right;
} Node;
typedef Address Tree;
```

Hanya memerlukan modul **mesin karakter** untuk membaca pita karakter

```
void BuildTree(Tree *t)
/* Dipakai jika input dari pita karakter */
/* I.S.: Sembarang */
/* F.S.: T terdefinisi */
/* Proses: Membaca isi pita karakter dan membangun pohon secara
           rekursif */
```

Ide 2: Membangun Pohon secara Rekursif - 2

```
void BuildTree(Tree *t)
/* Dipakai jika input dari pita karakter */
/* I.S. cc berisi '(' */
/* F.S. t terdefinisi */
/* Proses: Membaca isi pita karakter dan membangun pohon secara rekursif, hanya
           membutuhkan mesin karakter */
{   /* Kamus Lokal */

    /* Algoritma */
    adv();                /* advance */
    if (cc==')')          /* Basis: pohon kosong */
        (*t)=NIL;
    else {                /* Rekurens */
        t = newTreeNode(cc);
        adv();            /* advance */
        BuildTree(&(LEFT(*t)));
        BuildTree(&(RIGHT(*t)));
    }
    adv();                /* advance */
}
```

Ide 2: Membangun Pohon secara Rekursif - 3

Contoh pemanggilan di program utama:

```
#include "tree.h"

int main () {
    /* KAMUS */
    Tree t;

    /* ALGORITMA */
    start();
    BuildTree(&t);
    displayTree(t); /* mencetak pohon */
    return 0;
}
```

Membangun Pohon dari String - 1

Menggunakan ide pembangunan pohon dari pita karakter secara rekursif

Struktur data yang digunakan adalah struktur data pohon biasa (tidak perlu pointer ke Bapak)

```
void BuildTreeFromString (Tree *t, char *st, int *idx);
/* Input dari string st */
/* I.S. Sembarang */
/* F.S. t terdefinisi */
/* Proses: Membaca string st dan membangun pohon secara rekursif */
```

Membangun Pohon dari String - 2

```
void BuildTreeFromString (Tree *T, char *st, int *idx)
/* Input dari string st */
/* I.S. st[*idx]=='(' */
/* F.S. T terdefinisi */
/* Proses: Membaca string st dan membangun pohon secara rekursif */
{  /* Kamus Lokal */
   /* Algoritma */
   (*idx)++;                          /* advance */
   if (st[*idx]==')')  /* Basis: pohon kosong */
      (*T)=NIL;
   else {              /* Rekurens */
      t = newTreeNode(st[*idx]);
      (*idx)++;                        /* advance */
      BuildTreeFromString(&LEFT(*t),st,idx);
      BuildTreeFromString(&RIGHT(*t),st,idx);
   }
   (*idx)++;                          /* advance */
}
```

Membangun Pohon dari String - 3

Contoh pemanggilan di program utama:

```
#include "tree.h"

int main () {
    /* KAMUS */
    Tree t;
    char *s = "(A()())";
    int idx = 0;

    /* ALGORITMA */
    BuildTreeFromString(&t,s,&idx);
    displayTree(T); /* mencetak pohon */
    return 0;
}
```