Data Structure
- Linear Structure
  - Dense list(ordered/linear)
  - Linked list (SLL, DLL, CLL)
  - Stack, Queue
- Non-linear structure { Tree, Graph}
- File structure

목차:

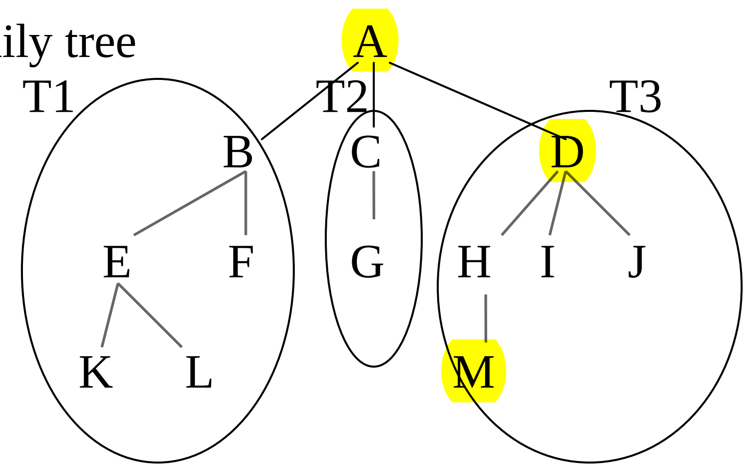
1) **트리 정의**
2) **이진트리 (Binary Tree: BT) : definition and Representation**
3) **BT algorithm:**
   - **Traversing (Inorder, Preorder, Postorder)**
   - **Tree Build (exercise)**
4) **Threaded Tree: definition**
5) **HEAP (Maxheap, Min Heap)**
6) **Binary Search Tree (BST)**
   - **Insert, delete, search algorithm.**

1. Tree Introduction

definition: A tree is finite set of one or more nodes

1) there is a special node called ROOT
2) the remaining nodes are partitioned into ($n \geq 0$ disjoint sets) T_1, ...T_n(subtrees of the root), where each of these sets is a tree

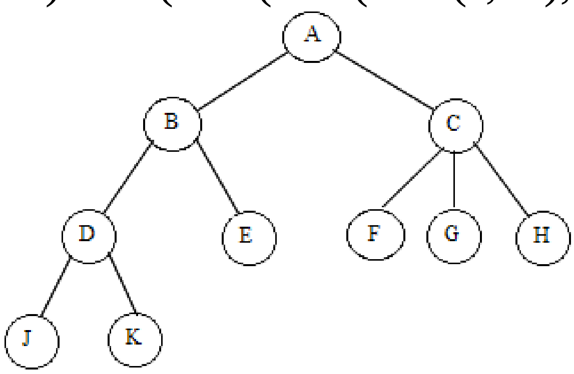
ex) family tree



- **Definitions/terminologies**
  1) **node** : item of information + branches to other nodes
  2) **degree**: number of subtrees of a node
     ex) degree of A =3, C=1, F=0
  3) **leaves**: nodes that have degree zero(leaf node, terminal node, K,L,F,G,M,I,J)
  4) **children**: nodes that are directly accessible from a given node in the lower level (children of B=> E,F)
  5) **parent**: A node that has children (D is parent of H)
  6) **siblings**: children of same parent
  7) **grandparents, grandchildren**
     (D is grandparent of M , A is grandparent of EFGHIJ)
  8) **path**: a sequence from a node Ni to Nk
     가장 중요
     (ex, ABEL -> a path from A to L)
  9) **ancestor**: all the nodes along the path from root to that node
     (ancestor of M-> A,D,H)
  10) **descendants**: all the nodes that are in its subtrees
      (E,F,K,L are descendants of B)
  11) **level**: let the root be at level 1 내가 레벨 2에 있으면 내 child는 3에 있다
      (if a node is at level I, then its children is level I+1)
  12) **height or depth**: maximum level of any node in the tree
      (ex. Depth of the figure is 4)

- **Representation of Trees**

  1) **List representation** (트리의 표현)
     (ex) (A (B (D (J, K), E), C (F,G, H)))



⇨ 임의의 노드는 varying number of fields, depending on the number of branches (예: 어떤 노드는 1 개의 child, 어떤 노드는 5 개의 child) So, the possible representation is:

data	$link_1$	$link_2$	....	$link_n$

2) Left Child-Right sibling representation

. Since it is easier to work with fixed size => require exactly <u>two link or pointer fields</u> per node

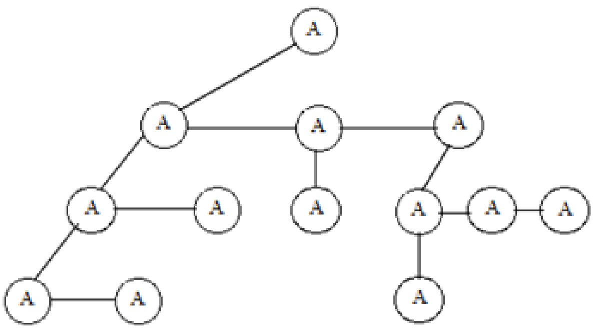
. condition:

every node has <u>one leftmost child</u> and <u>right siblings</u>
order of children in a tree is not important

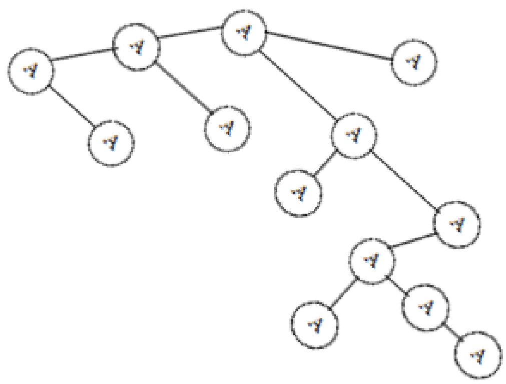
. data representation:

data	
left child	right siblings

. Left-child-right-sibling tree.



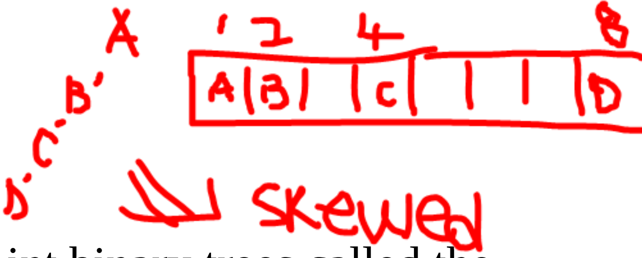
. Left child-right child tree



3) Representation As a Degree two tree

. to obtain degree two tree representation of a tree => rotate the left child-right-sibling tree clockwise by <u>45 degree</u>

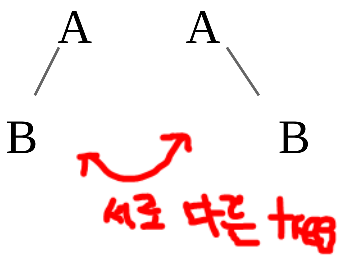
. This tree is also known as <u>BINARY TREE</u>



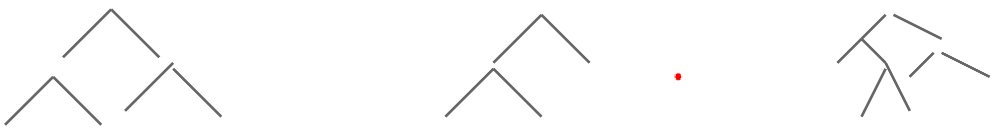
2. 이진 트리(Binary Tree: BT)

정의 : consists of a **root** and and two disjoint binary trees called the **left subtree** and the **right subtree.** (have a maximum of two children)

- difference with tree
  1) tree has no empty tree, but BT has empty
  2) tree has no order, but BT has order
     => two different BT

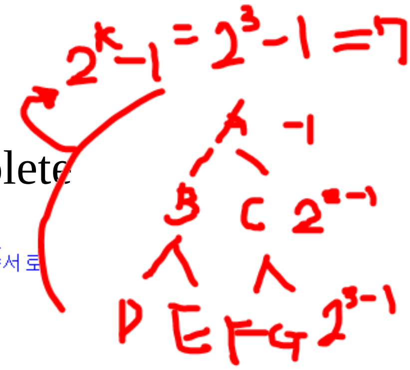


- Kinds of BT
  1) Skewed BT: node 들이 한쪽으로 치우친 형태
  2) Full BT: BT, in which all of the leaves are on the same level and every non-leaf node has two children
  3) Complete BT: BT that is either full or full through the next to last level, with the leaves on the last level as far as to the left as possible



Full and complete   complete   not complete

-> complete 조건:
노드가 위에서 아래로
왼쪽에서 오른쪽의 순서로
채워져야됨



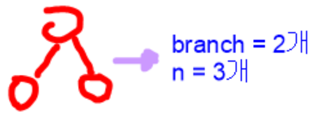
- **Properties of BT**

  we like to know maximum number of nodes in a BT of depth k, ...

1) The maximum number of nodes on level i of a BT is => 2^{i-1}, $i \geq 1$
2) The maximum number of nodes in a BT of depth k is => $2^k - 1$, $k \geq 1$

- **Relation between number of leaf nodes and nodes of degree 2**

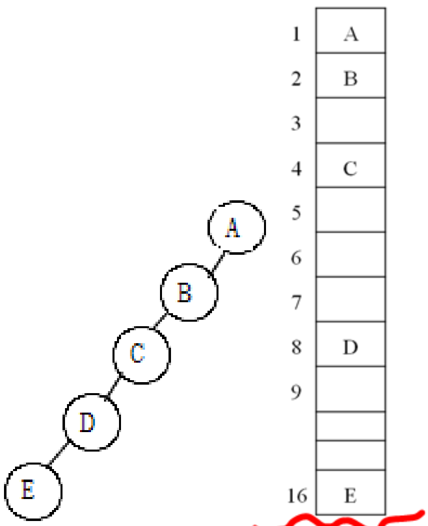
  "For any BT, T, if n0 is the number of leaf nodes and n2 is the number of nodes of degree 2, then n0 = n2+1"

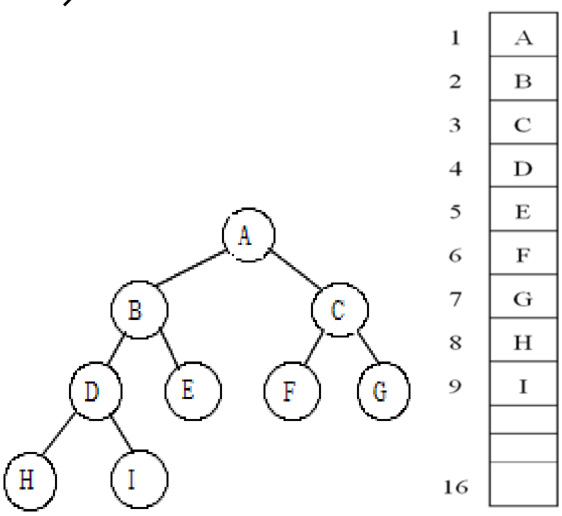


(Proof) n1 = number of nodes of degree 1.
n = total number of nodes , then
n = n0 + n1 + n2
If B is the branches, then n = B+1
And all nodes stem from a node of degree one or two,
then B=n1+2n2 So, we obtain n = (n1+2n2)+1 ,
n0+n1+n2=n1+2n2+1. Therefore, n0 = n2+1

- <u>Binary Tree Representation</u>

1) Array Representation : for any node with index i, $1 \leq i \leq n$





skewed

. parent(I) is at ($\lfloor i/2 \rfloor$, if $i \neq 1$),

. left-child(I) is at (**2i**, if $2i \leq n$), If 2i > n then i has no left child

. right-child(I) is at (**2i+1**, if $(2i+1) \leq n$),
=> If (2i+1) > n then I has no right child
공간의 크기를 넘어갈 경우

⇨ for full binary tree, this representation is efficient
⇨ But problem for skewed binary tree

Depth k 인 tree 에서 worst case 에는 2^k-1 space 필요, 실제로는 k space 만 사용. => waste of a lot of space)

- Array representation:

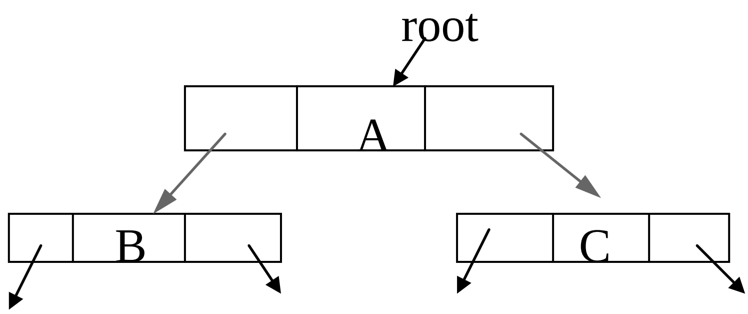
1) waste of space (skewed tree 의 경우)
2) waste of time (Insertion & deletion requires movement of many nodes)

- **Linked Representation**

```
Typedef struct node *tree-pointer;
Typedef struct node {
        int data;
        struct node *left;
        struct node *right;
        };
```

left-child	data	right-child

```
class Node {
private:
    int data;
    Node *left;
    Node *right;
};
```



<u>특성</u>

(1) array 로 표현하는 방법에 비해 메모리 절약 (필요한 node 의 숫자만큼의 memory 사용)

(2) array 로 표현은 static (고정적)방법이고 linked list 표현방법은 dynamic 방법이다.
array 로 표현하면 고정된 메모리 사용하게 되나 linked list 로 표현하면 run-time 에 노드 생성 시 필요한 메모리를 확보하기 때문에 가변적(동적)이다.

(3) insertion, deletion 이 빠르다(no data movement)

★ 암기

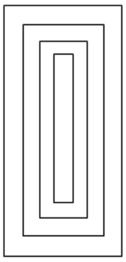
3. Binary Tree Traversal (이진 트리의 순회)

* traversing a tree: L- moving left, R- moving right, V- visit node

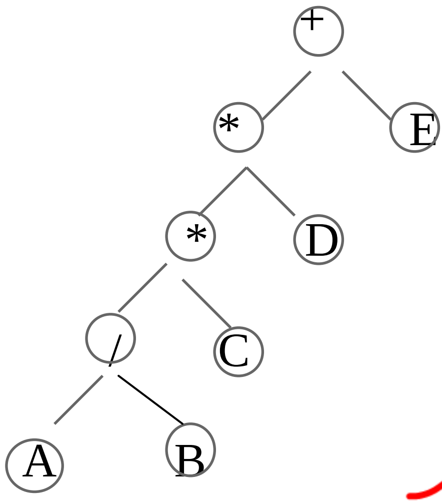
- Traversing method(순회방법):
  1) LPR(Inorder) : Visit left, Print current node, Visit right
  2) LRP(Post order): Visit left, Visit right, print current node
  3) PLR(Preorder): Print current node, visit left, visit right

* 산술식의 이진트리 표현 (A / B * C * D + E)

1) Inorder Traversal (LPR)



```
Void Tree::inorder(Node *ptr)
{
      if (ptr) {
          inorder(ptr->leftchild);
          cout << ptr->data;
          inorder(ptr->rightchild);
      }
}
```



해봐

1. 널 노드에 도달할 때까지 왼쪽 방향으로 이동
2. 널 노드에 도착하면 널 노드의 부모 방문
3. 순회는 오른쪽 방향으로 계속
4. 오른쪽으로 이동할 수 없을 때에는 바로 위 레벨의 방문하지 않은 노드에서 순회 계속

output : A / B * C * D + E

2) PostOrder Traversal (LRP)

```
void Tree:: postorder(Node *ptr)
{       if (ptr)
              {
                   postorder(ptr->leftchild);
                   postorder(ptr->rightchild);
                   cout << ptr->data;
              }
 }
```

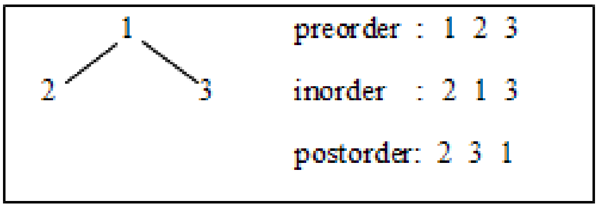
output: A B / C * D * E +

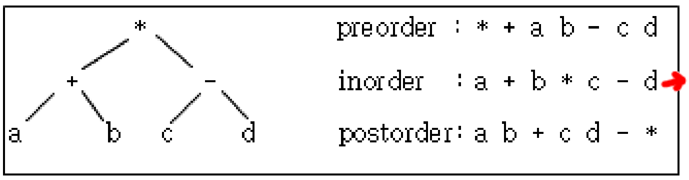
3) Preorder Traversal

```
void Tree::preorder(Node *ptr)
{
   if (ptr)
        {
            cout << ptr->data;
            preorder(ptr->leftchild);
            preorder(ptr->rightchild);
        }
 }
```

output: + * * / A B C D E

ex)





→ L→R

preorder : root left right
inorder : left root right
preorder : left right root

Tree Build   * Building tree for mathematical expression

```
Struct node {
   Char    data;          // one character input per node
   Int        prio;         // priority number from precedence table
   Struct node *left;        // left    link
   Struct node *right;      // right    link          }

class Node {
  private:
        int data;    Node *left;      Node *right;
        Node(int value) {data = value; left = 0; right = 0;}
  friend class Tree;          };

class Tree {
  private:
        Node *root;
  public:
        Tree() {root = 0;}
        ~Tree();
  ………….         };
```

*** Precedence Table**

char prec[5][2] = {'^' 3,   '*', 2,   '/', 2,   '+', 1,   '-', 1};

'^'	'*'	'/'	'+'	'-'
3	2	2	1	1

-Get expression: gets math expressions from KBD (ex. 8+9-2*3)

- Build Tree

while (input !=NULL)

{   . create new-node

  . assign DATA-INPUT into new-node's data field & default prio '4'

  . for i=0 to 4   (if new-node -> data == prec[i][0])

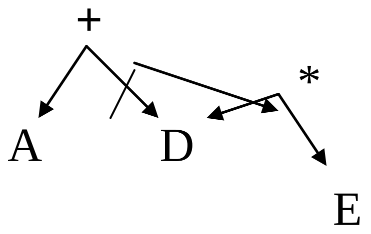
                 then new-node ->prio = prec[i][1]

```
. If (i==5)   then   call Operand(new-node)
              else   call operator(new-node)     }   }
```

*** Operand(new-node)**

```
If   HEAD==NULL   then    HEAD=new-node     return
P = Head
While (p->right !=NULL)    p=p->right
P->right = new-node
```

*** Operator (new-node)**




```
if (head->prio  ≥  new-node->prio)
     new-node->left = Head
     Head = new-node
Else
     New-node->left = Head->right
     Head -> right = new-node
```

*** Tree Evaluation**

```
   evalTree (p)          {
        if    p!=NULL      {
              if p->data in [0..9]    then    value = p->data-'0'
              else
                   left = evalTree(p->left)
                   right=evalTree(p->right)
                   switch (p->data)
                          case '+':    value=left+right;
                          case '-':    value=left-right;
                          case '*':    value=left*right;
                          case '/':    value=left/right;
               }endif
          }else    "Empty tree" }
       Return value
```

4. 스레드(threaded) 이진 트리

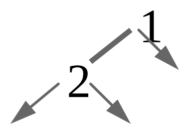
개념만 얘기, 코딩 X

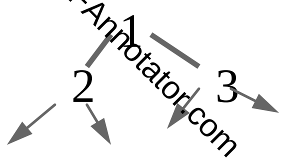
거의 안씀

정의: 스레드(*Thread*): [A.J. Perlis & C. Thronton]

=> null link를 다른 node를 가르키는 pointer로 변환한 것을 thread 라 하며 inorder 순회에 효과적으로 사용할 수 있다.

- Binary tree has: total 2n links, (그중 n+1 null links (or empty subtrees)) => more null links than actual pointers



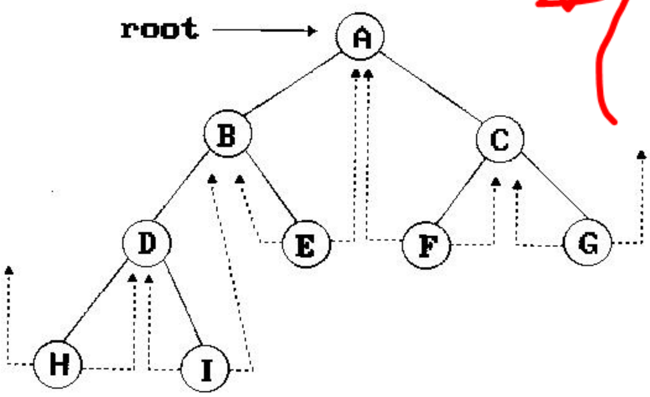


[n->2, Null->3] [n=3, null link=4]

- Threads 연결 규칙 (PTR 이 현재노드라 가정하면)

1) If ptr->left is null : ptr 의 inorder predecessor 를 가리키게함. 즉, inorder 순회시 ptr 앞에 방문하는 노드를 가리키게함

2) If ptr->right is NULL : ptr 의 inorder successor 를 가리키게함. 즉, inorder 순회시 ptr 의 다음에 오는 node 를 가리키게함.



inorder : hdibeafcg
pre: ??
post: ??

. E 의 leftchlid is null : E 의 left 는 inorder predecessor B 에연결
. E 의 rightchild is null: E 의 right 는 inorder successor A 에 연결

- Representation of Threaded Tree

```
Struct Node{
        int     left_thread;
        Node   *left-child;
        char    data;
        Node   *right-child;
        int     right_thread;
 }
```

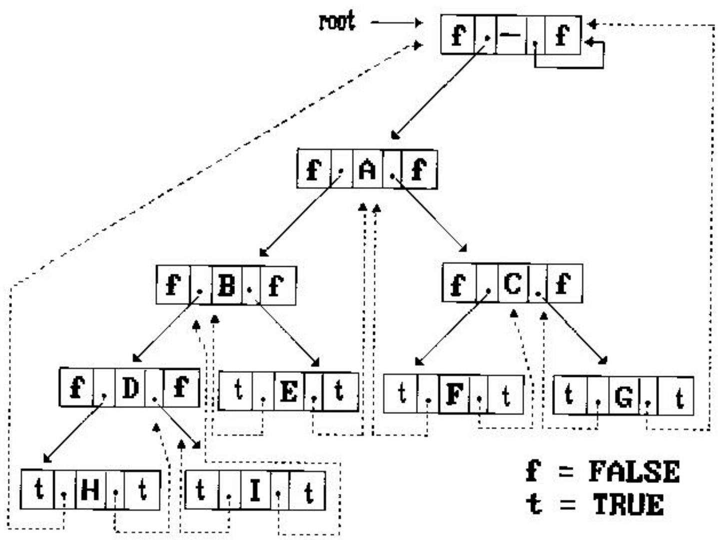
leftThread	leftChild	data	rightChild	rightThread
TRUE		'A'		FALSE

점선　　　　　　　　　　실선

thread 는 점선으로 표기, normal pointer 는 실선으로 표기

if thread field == <u>true</u> then contains a <u>thread</u>
if thread field == <u>false</u> then contains a <u>pointer</u> to a child

* 문제는 2 개의 thread 가 dangling 되어 있다는 점이다.
(1) H 의 inorder predecessor 가 없음(H 가 inorder 의 맨처음 node 임)
(2) G 의 inorder successor 가 없음 (G 가 inorder 의 맨 나중 node 임)
(3) head node 를 만들어 연결시킴으로 해결하며 그 결과는 아래 그림임



● 활용: inorder traversal 에 활용함

recursive inorder traversal 을 간단한 non-recursive version 으로 구현할 수 있다.

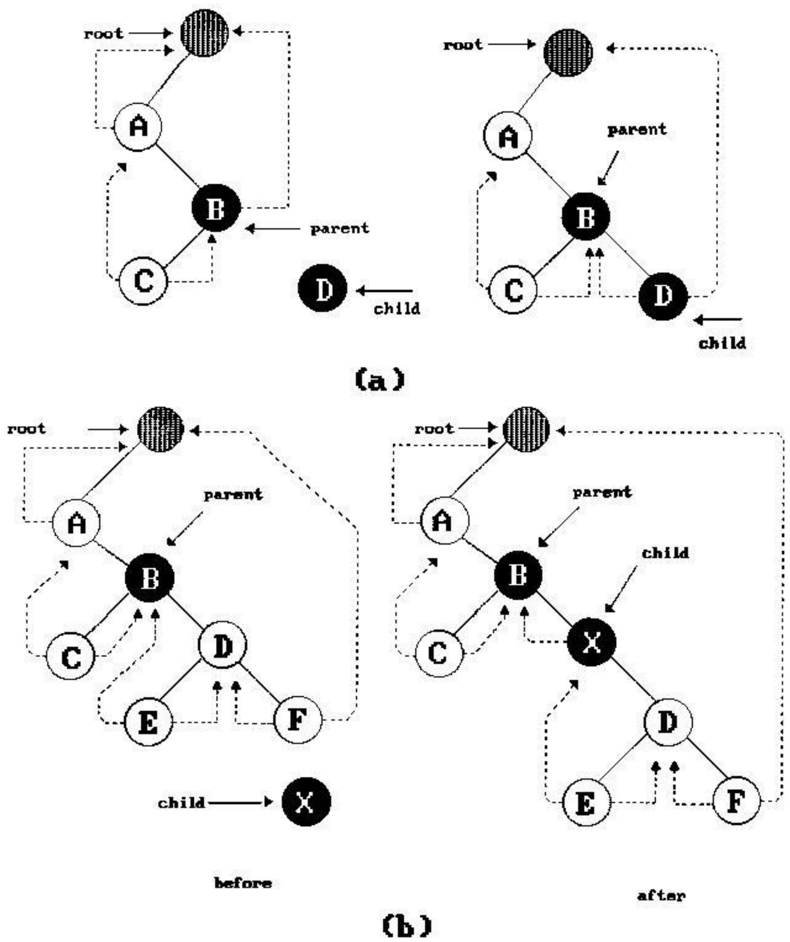
computing time은 마찬가지로 O(n)이지만 recursive call 에 따른 overhead는 없어짐

- Inorder: H D I B E A F C G

예) 1) Node E 의 right_thread is TRUE, successor of E is => A

2) Node A 의 right_thread is False, C 부터 시작하여, C 의 leftchild link 를 따라서 F 까지간다. => F is A 의 후속자

예)



(a)

(b)

5. HEAP

중요 열혈 340pg

정의: HEAP is a special form of FULL binary tree that is used in many applications

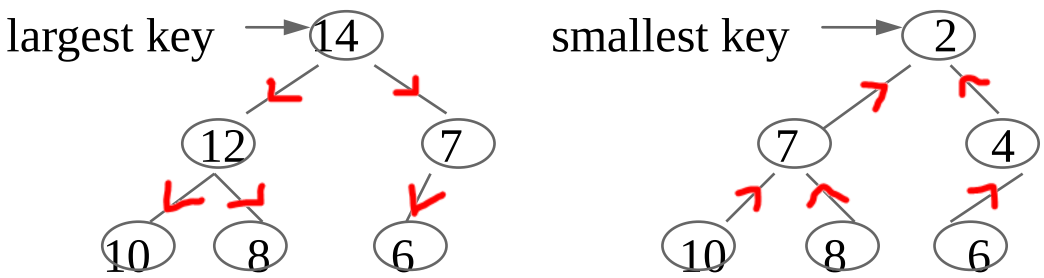
1) MAX TREE: is a tree in which the key value in each node is larger than the key values in its children (if any).

MAX HEAP => is a complete binary tree that is also a max tree

2) MIN TREE: is a tree in which the key value in each node is smaller than the key values in its children (if any).

MIN HEAP => is a complete binary tree that is also a min tree

ex) MAX HEAPS MIN HEAPS



- Represetation - Use Arrays, (same as tree for array representation scheme) 배열로하면 가장 쉬움

- *MaxHeap ADT*

----

structure *MaxHeap*

objects : 각 노드의 값이 그 자식들의 것보다 작지 않도록 조직된 n>0 원소의 완전 이진 트리

functions :

∀ heap∈MaxHeap, item∈Element, n, max-size∈integer

MaxHeap Create(max_size)::= 최대 max_size개의 원소를 가질 수 있는 공백 Heap 생성

Boolean HeapFull(heap, n) ::= if (n == max_size) return TRUE
else return FALSE

MaxHeap Insert(heap, item, n)::= if (!HeapFull(heap, n)) item을 heap에 삽입하고 그 heap을 반환 else return 에러

Boolean HeapEmpty(heap, n) ::= if (n==0) return TRUE
else return FALSE

Element Delete(heap, n) ::= if (!HeapEmpty(heap, n))
Heap에서 가장 큰 원소를 제거하고 그 원소를 반환 else return 에러

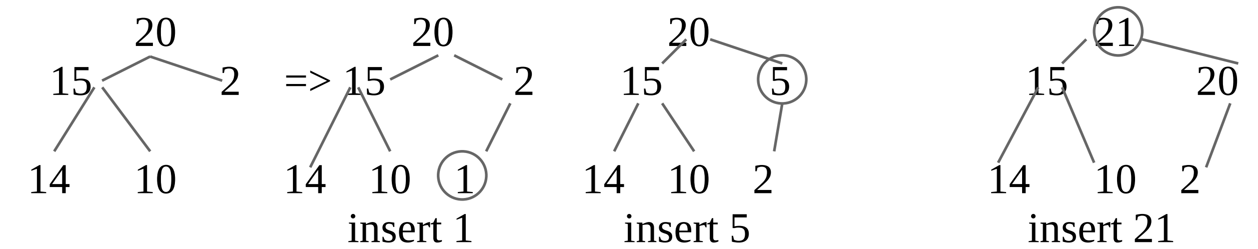
```
#define MAX_ELEMENTS 200 /* 최대 heap 크기 + 1 */
#define HEAP_FULL(n)   (n == MAX_ELEMENTS-1)
#define HEAP_EMPTY(n)   (!n)

typedef struct {
        int key;
        /* 다른 필드들 */
        } element;
element heap[MAX_ELEMENTS];
int n = 0;
```

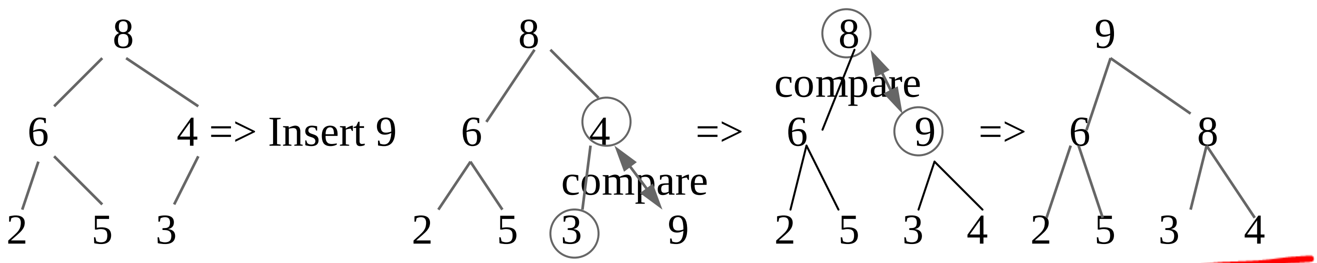
- PRIORITY QUEUE

. HEAPs are used to implement PRIORITY QUEUE
- ⇨ the element to be deleted is the one with **highest(lowest) priority**
- ⇨ For example, Job scheduler use the priority with the **shortest run time**, implement the priority queue that holds the jobs as a **min heap**
- ⇨ MAX(MIN) HEAP may be used

- ⇨ **ARRAY** is a simple representation of a priority queue (easily add to P.Q. by placing the new item at the current end of array) => insertion complexity O(1)

- **Insertion into a MAX HEAP**





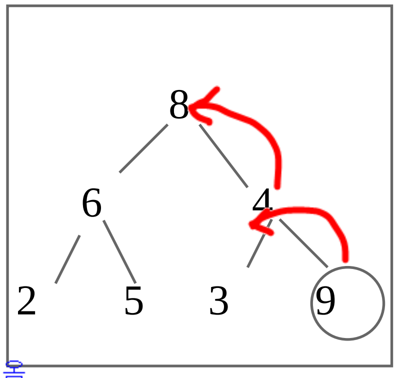
tree가 좋은점이 자신의 위치 /2 하면 부모의 위치가 나옴.
위에서 9의 위치가 7인데 부모의 위치는 3(4의 위치)이고 할아버지의 위치는 7/2/2 하면 1 8의 위치이다

```
void insert_maxheap (element item, int *n)
 {
   int i
   if (HEAP_FULL(*n)) {
          printf("Heap is full....\n"); exit(1);
   }
   I = ++(*n);
   while ((I!=1) && (item    > heap[i/2]       ))
     {       heap[I] = heap[i/2];
             I = | i/2 |;
     }
   Heap.[I] = item;
 }
```

② ③ 몫 3

9를 1에다 넣어라

if item 이 1이면 while문 바로 빠져나옴
if item 이 7이면 while문 한번 돌고 빠져나옴



1	2	3	4	5	6	7
8	6	4	2	5	3	9

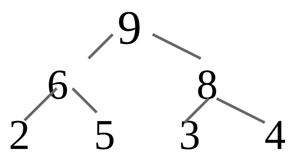
for I = 7,
since (item = 9) > (heap[7/2] = 4), => (heap[7] = heap[3])

1	2	3	4	5	6	7
8	6	4	2	5	3	4

for I = i/2 = 3,
since (9 > heap[3/2] = 8) => (heap[3] = heap[1])

1	2	3	4	5	6	7
8	6	8	2	5	3	4

i/2 = 1, so exit while loop
heap[1] = item => (heap[1] = 9)

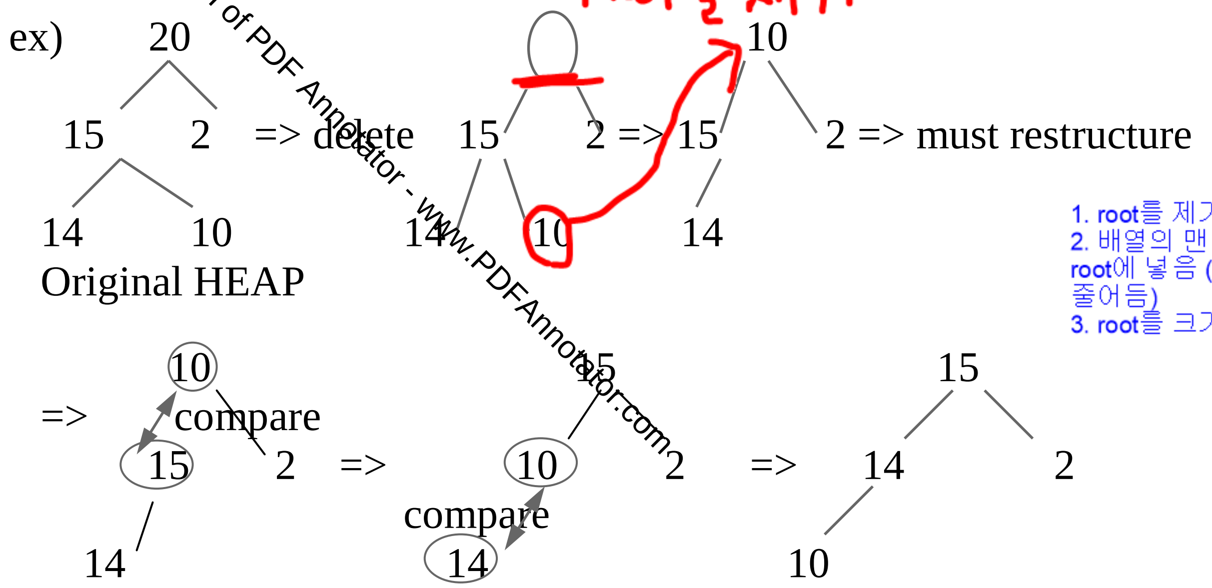


1	2	3	4	5	6	7
9	6	8	2	5	3	4

- **Deletion from a MAX HEAP**

. Always take it from the ROOT of the HEAP. => must restructure the HEAP (complexity is O(logn))

Root를 제거

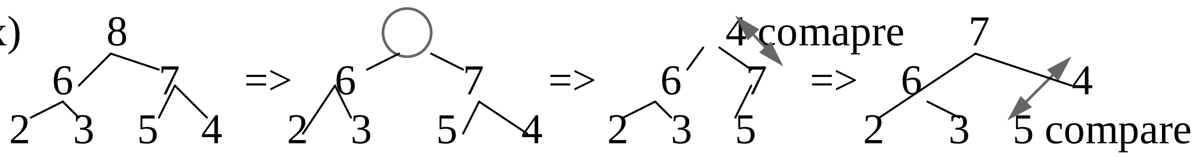


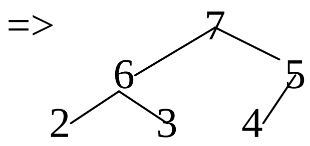
1. root를 제거하고
2. 배열의 맨끝에 있는 걸 root에 넣음 (n사이즈 하나 줄어듬)
3. root를 크기비교하며 정렬

```
element delete-maxheap (int *n)
{    int parent, child;
   element item, temp;
      .......................
20= item = heap[1];              /* save the highest key*/
  temp = heap[(*n)—];      /* use the last element */
  parent = 1;
  child = 2;

  while (child <= *n) {
       if (child < *n) && (heap[child] <heap[child+1]))
             child++;                    /* find largest child */
       if (temp >= heap[child]) break;
       heap[parent] = heap[child];
       parent = child;
       child = child * 2;
   }
   heap[parent] = temp;
   return item;    } =20
```

delete니까 item(root) 돌려줄라고(= pop이랑 같음)

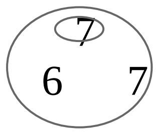
ex)





1	2	3	4	5	6	7
8	6	7	2	3	5	4

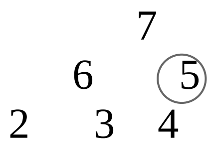
1) n = 7, item = 8, temp = 4, n = 6, child=2, parent =1
2) While (child <= 6)
   . child <6, && heap[child]=6 < heap[child+1]=7) **=> child=3**
   . temp(=4) < (heap[child]=7)
   . **heap[1] = 7**
   .parent = 3, child = 6



1	2	3	4	5	6
7	6	7	2	3	5

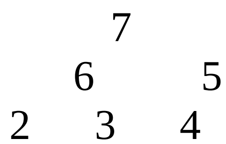
3) while (child <= 6)
   . child <6 && (heap[child] = 5 > heap[child+1]=4)=>child=6
   . temp(=4) < heap[child] = 5
   . **heap[3]=5** (heap[parent] = heap[child])
   . child = 12, parent = 6 => **exit while loop**

1	2	3	4	5	6
7	6	5	2	3	5



4) heap [6] = temp (4)

1	2	3	4	5	6
7	6	5	2	3	4



6. Binary Search Trees (BST)

⇨ HEAP is suitable for **priority queue** applications, but it's not well suited if we want to delete **arbitrary elements.**

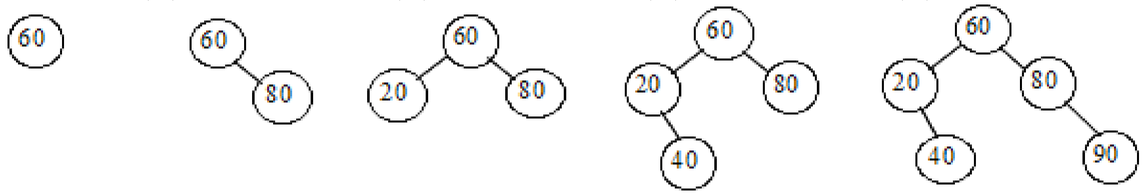
⇨ BST performs better than any data structure when we wish to perform INSERTION, DELETION, SEARCHING => BST can perform these operations by both KEY VALUE (예: delete element x), and BY RANK (예: delete 6th position)

Definition: BST is a binary tree. It may be empty, if not, it satisfies the followings;

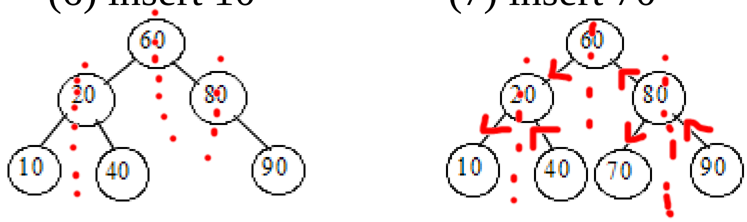
1) Every element has a key, and Keys are unique
2) The keys in leftsubtree must be smaller than the keys in ROOT of subtree root의 왼쪽에 있는 데이터값은 root보다 작아야됨
3) The keys in rightsubtree must be larger than the keys in ROOT of subtree root의 오른쪽에 있는 값은 root보다 커야됨

[예제#1] 60 80 20 40 90 10 70 순서로 삽입

(1) insert 60 (2) insert 80 (3) insert 20 (4) insert 40 (5) insert 90



(6) insert 10 (7) insert 70



- Representation : same as Binary Tree representation
- Tree operation: same as tree traversal (inorder, preorder, postorder)
  \+ Additional operations (insertion, deletion, search)

13일 LAB 에 넣어

1) **Searching a BST**

```
search (tree-ptr ptr,   int key)
{
   if (ptr = NULL)     return NULL;   //search unsuccessful
   else {
        if (key == x->data)   return ptr;
        else if (key < ptr->data)
                 ptr = search(ptr->left, key);   //search leftsubtree
         else if (key > ptr->data)
                  ptr= search(ptr->right, key); //search rightsubtree
    }
   return ptr;
 }
```

ptr

2) Inserting into a BST

* In order to insert, we must search the tree => if the search is unsuccessful, we insert the element at the point the search terminated

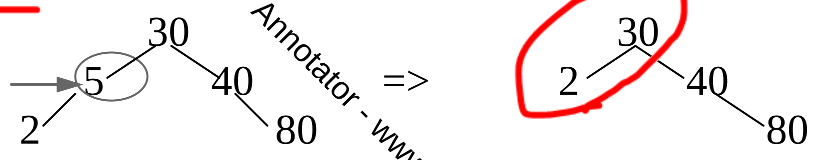
```
  INSERT (ptr, key) //recursive version
{
   if (ptr=NULL) {
            create new_node(ptr);
            ptr->data = key;
            ptr->left =   NULL;
             ptr->right = NULL;
    }
    else if   (key < ptr->data)
            ptr->left =   INSERT(ptr->left, key);
    elseif   (key >   ptr->data)
            ptr->right = INSERT(ptr->right, key);
   return ptr;
}
```

13일 LAB에 넣어

3) DELETE (3 cases)

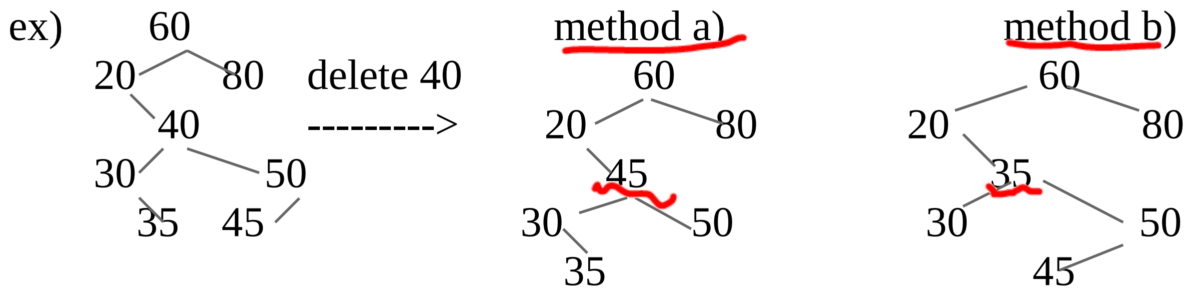
* leaf node => set the child field of the node's parent pointer to NULL and free the node

* nonleaf node with one child => change the pointer from parent to single child



* Nonleaf node with two children 얘가 어렵

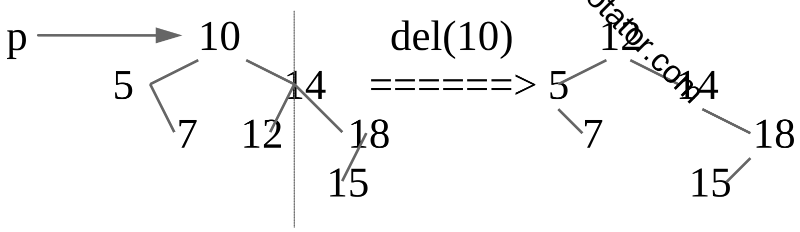
a. replace with smallest element in rightsubtree
b. replace with largest element in leftsubtree



method(a)

```
delete (key, ptr)    {
 if (ptr != NULL)    {
    if (key < ptr->data)
          ptr->left = delete(key, ptr->left)        /* move to the node    */
    else if (key > ptr->data)
          ptr->right = delete (key, ptr->right)    /* arrived at the node*/
    else if ((ptr->left == NULL) && (ptr->right==NULL))
          ptr=NULL                                  /*leaf*/  child가 있나 없나
    else if (ptr->left == NULL) {
          p = ptr;    ptr=ptr->right;    delete(p);    /rightchild only*/    }
    elseif    (ptr->right == NULL) {
          p = ptr;      ptr=ptr->left;    delete(p);    /*left child only */    }
    else    ptr->data = find_min(ptr->right)    /*both child exists */
 }
 Else    printf("Not found");
 return ptr;    }
```

```
int find_min(ptr)      /*right subtree 에서  가장  작은것  선택  */
{
    if (ptr->left ==NULL)
    {
        temp = ptr->data;      ptr = ptr->right;
    }
    else    temp = find_min (ptr->left);
    return temp;
}
```



root인 10을 지우면 right subtree에서 가장 작은 12가 root가 됨

- 선언

```
class Node {
    ……………………
};
class Tree {
    private:
        Node *root;
    public:
        Tree() {root = 0;}
        ~Tree();
        void insertTree(int);
        void deleteTree(int);
        Node *deleteBSTree(Node *, int);
        void searchTree(int);
        Node *searchBSTree(Node *, int key);
        void traverseTree();
        void inorderTraverse(Node *);
        void drawTree();
        void drawBSTree(Node *, int);
```

```
        Node *findmin(Node *p);
        int tree_empty();
        void freeBSTree(Node *);
    };

Tree::~Tree()      {    freeBSTree(root);    }

void Tree::drawTree()    {    drawBSTree(root, 1);    }
```

그대로 쓰면 됨 LAB 할때

```
void Tree::drawBSTree(Node *p, int level) {
    if (p != 0 && level <= 7) {
        drawBSTree(p->right, level+1);
        for (int i = 1; i <= (level-1); i++)
                cout << "  ";
        cout << setw(2) << p->data;
        if (p->left != 0 && p->right != 0)    cout << " <" << endl;
        else if (p->right != 0)               cout << " /" << endl;
        else if (p->left != 0)                cout << " \\" << endl;
        else                                  cout << endl;
        drawBSTree(p->left, level+1);
    }
}
```