- 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(Definition)
2) 이진트리 (Binary Tree: BT): definition and Representation
3) **BT algorithm:**
   - **Tree Build (exercise)**
   - **Traversing algorithm (Inorder, Preorder, Postorder)**
4) Threaded Tree: definition
5) **HEAP (Maxheap, Min Heap), HEAP sort**
6) **Binary Search Tree (BST) – Insert, Delete, Search 알고리즘**

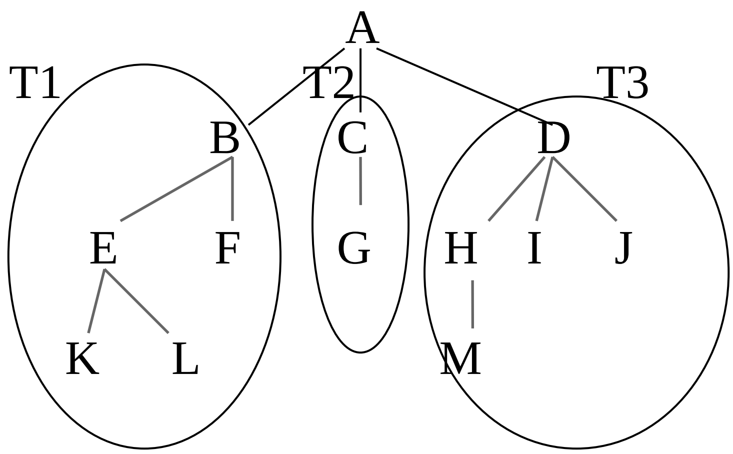
Applications: games, 조직도, 암호생성,…

1. Tree Introduction

definition: 하나 이상의 노드들로 구성된 유한집합

1) 한 개의 ROOT 노드
2) 나머지 노드들은 n(≥0)개의 subset $T_1, T_2, \ldots, T_n$ 으로 분할 (T_i : root 의 sub tree). Sub tree 들은 또한 tree 이다.

Ex) family tree



- **Definitions/terminologies**

1) **node** : item of information + branches to other nodes
2) **degree(차수)**: number of subtrees of a node
   ex) degree of A =3, C=1, F=0 (트리의 degree 는 최대차수 의미)

3) **leaves(leaf)**: nodes that have degree 0 (leaf node, terminal node)
   ex) 위 트리에서는 K,L,F,G,M,I,J 가 leaf 노드들이다.
   - leaf node 가 아닌 노드는 nonterminal node

4) **children**: nodes that are directly accessible from a given node in the lower level (children of B=> E,F)
5) **siblings(형제노드)**: children of same parent

6) **parent**: A node that has children (D is parent of H)
   - **grandparents, grandchildren**
     (D is grandparent of M , A is grandparent of EFGHIJ)

7) **path(경로)**: a sequence from a node N_i to N_k (두 노드 사이의 경로는 1 개이며, 1 개이상이면, 트리가 아니고, 그래프임)
   (ex, ABEL -> a path from A to L)
   - path length: edge 의 개수 (노드 A 와 L 의 path length 는 3)
   - edge: 경로와 경로 사이의 연결 선

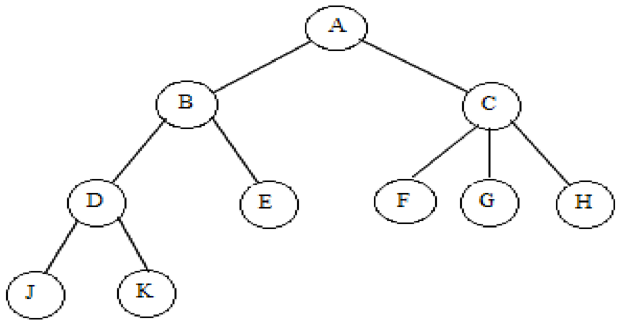
8) **ancestor(선조)**: all the nodes along the path from root to that node
   (ancestor of M-> A,D,H)
9) **descendants(자손)**: all the nodes that are in its subtrees
   (E,F,K,L are descendants of B)

10) **level**: let the root be at level 1
    (if a node is at level i, then its children is level i+1)
11) **height or depth**: maximum level of any node in the tree
    (ex. Depth of the figure is 4)

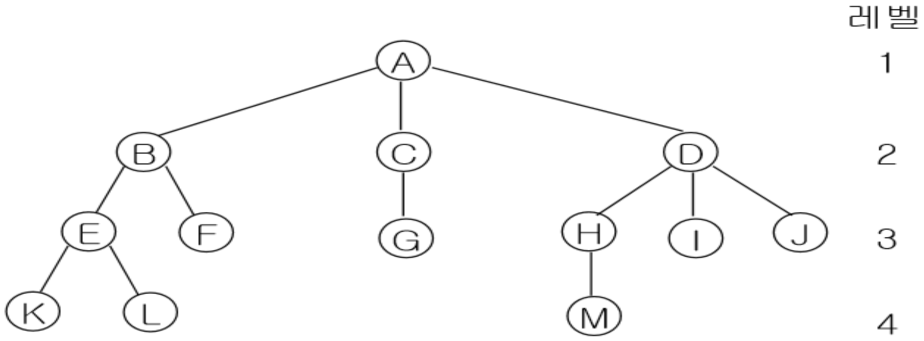
- **Representation of Trees**

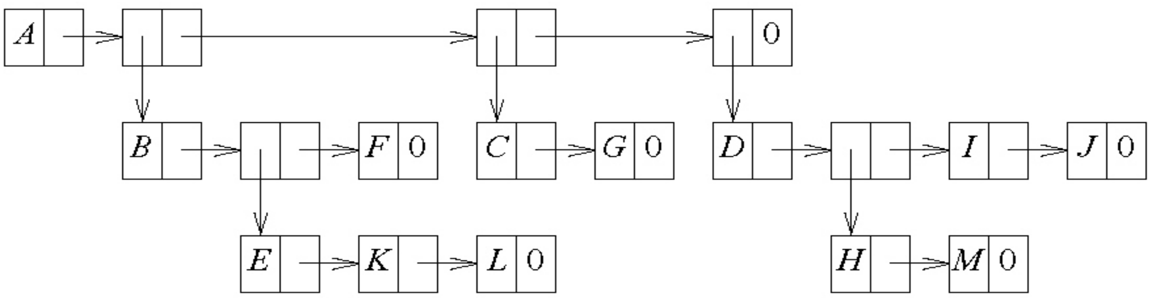
1) **List representation (**트리의 표현)

(ex) (A (B (D (J, K), E), C (F,G, H)))



2) **ex: (A(B(E(K,L),F),C(G),D(H(M),I,J)))**





⇨ 임의의 노드는 varying number of fields, depending on the number of branches (예: 어떤 노드는 1 개의 child, 어떤 노드는 5 개의 child, 즉 노드마다 서로 다르게 chile 구성)

⇨ 차수 n 인 트리에 대한 노드 구조

the possible representation may be: => 공간낭비

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

3) **왼쪽 자식 - 오른쪽 형제 표현**

(Left Child - Right sibling Representation)

. Since it is easier to work with fixed size => require exactly 2 links or pointer fields per node

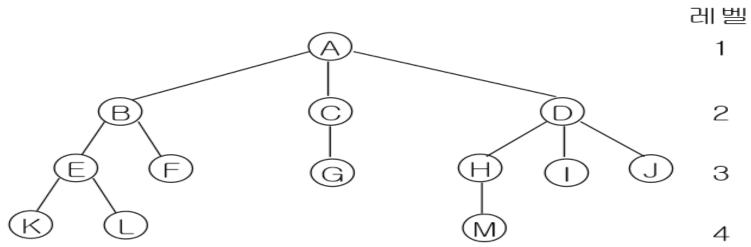
. Condition:

every node has one leftmost child and right siblings

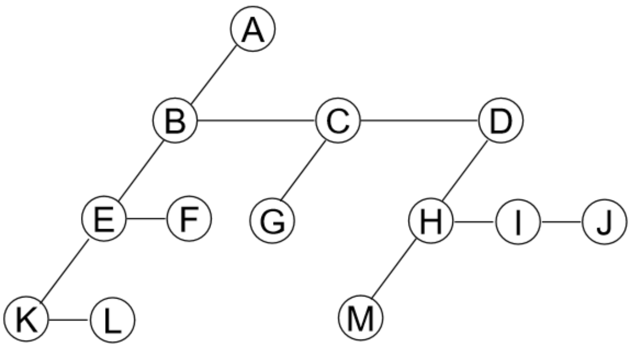
order of children in a tree is not important

. data representation:

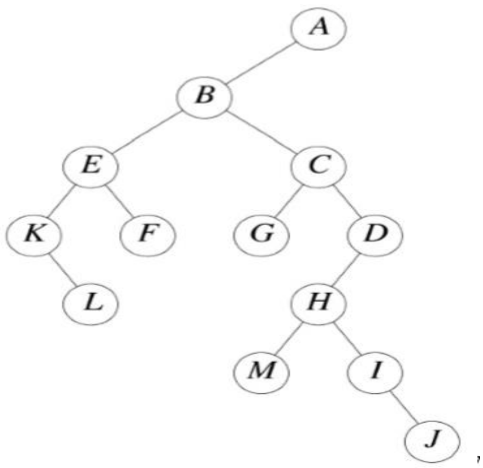
data	
left child	right siblings



. 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 45 degree

⇨ root 의 오른쪽 자식은 공백

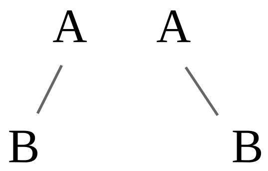
⇨ This tree is also known as BINARY TREE

2. 이진 트리(Binary Tree: BT)

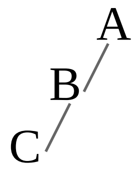
정의: consists of a **root** and and two disjoint binary trees called the **left subtree** and the **right subtree.** (have a maximum of two children)

- 이진트리와 일반트리의 차이점
  1) tree has no empty tree, BT has empty(공백이진트리)
  2) tree has no 순서(**order**), but BT has order

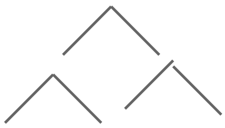
     two different BT ->



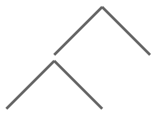
- Kinds of BT
  1) Skewed BT(편향 이진트리): node 들이 한쪽으로 치우친 형태



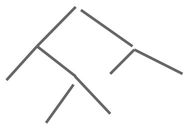
  2) Full BT: 모든 leaf 노드들이 같은 level 에 있으며, 모든 nonleaf 노드들이 two children 을 가지는 BT
  3) Complete BT:
     - BT 가 full BT 이거나,
     - 마지막 이전 level 까지는 full BT 이고, 마지막 level 에서는 왼쪽으로 치우친 경우에 complete BT 라 한다.



Full and complete



complete



not complete

- **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 => $2^k - 1$, $k \geq 1$

$$\sum_{i=1}^{k}(\text{레벨 } i\text{의 최대 노드 수}) = \sum_{i=1}^{k} 2^{i-1} = 2^k - 1$$

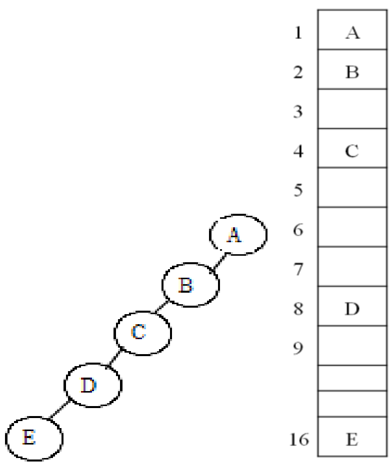
- **Relation between number of leaf nodes and nodes of degree 2**

"For any BT, T, if n_0 is the number of leaf nodes and
n_2 is the number of nodes of degree 2, then $\mathbf{n_0 = n_2+1}$"

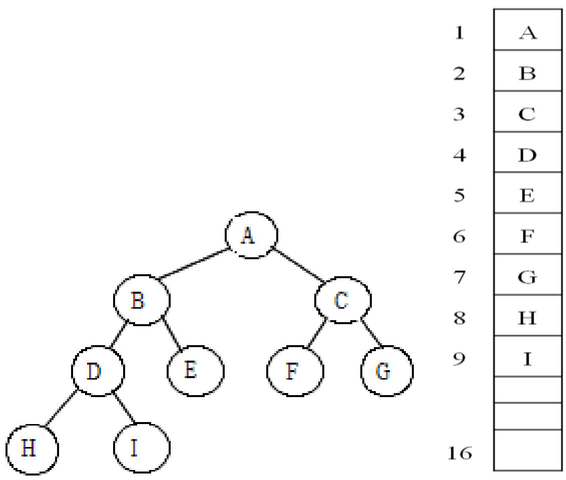
(Proof) n_1 = number of nodes of degree 1.
n = total number of nodes , then $\mathbf{n = n_0 + n_1 + n_2}$
If B is the branches, then $\mathbf{n = B+1}$
And all nodes stem from a node of degree 1 or 2,
then $B=n_1+2n_2$ So, we obtain $\mathbf{n = (n_1+2n_2)+1}$,
$n_0+n_1+n_2 = n_1+2n_2+1$. Therefore, $\mathbf{n_0 = n_2+1}$

- <u>Binary Tree Implementation</u>

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



(skewed BT)



(Complete BT)

. Node i 의 parent 위치: parent(i) is at ($\lfloor i/2 \rfloor$, if $i \neq 1$),

. Node i 의 left-child 위치: left-child(i) is at (**2i**, if $2i \le n$),
If $2i > n$ then i has no left child

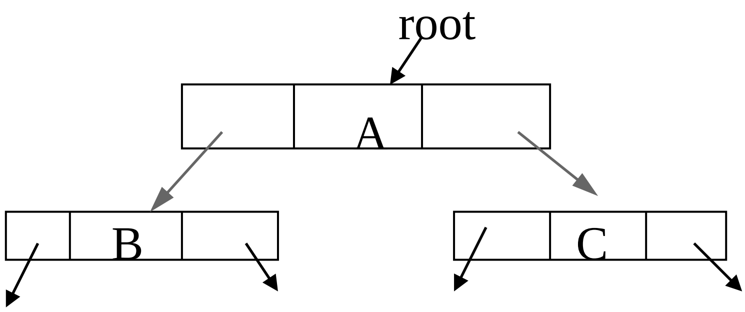
. Node i 의 right-child 위치: right-child(i) is at (**2i+1**, if $(2i+1) \le n$),
=> If $(2i+1) > n$, then node i has no right child

<u>**배열 표현의 문제점:**</u> full binary tree는 효과적이지만,
- Depth k 인 skewed BT에서 최악의 경우 $\mathbf{2^k-1}$ **space** 필요,
  실제로는 k space 만 사용. => **waste of a lot of space**)
- insert와 delete할 때 배열내에서 많은 data 이동 **(time waste)**

- **Linked Representation**

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 표현은 static (고정적)방법이고 linked list 표현은 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)

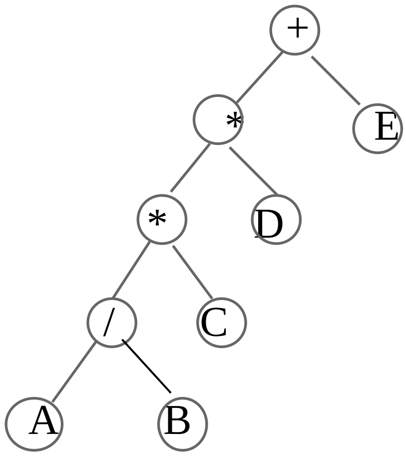
3. Binary Tree Traversal (이진 트리의 순회)

* traversing a tree: L- moving left, R- moving right, P- Print node

- Traversing method(순회방법):
  1) LpR(Inorder) : visit **L**eft, **p**rint current node, visit **R**ight
  2) LRp(Post order): visit **L**eft, visit **R**ight, **p**rint current node
  3) pLR(Preorder): **p**rint current node, visit **L**eft, visit **R**ight

* 산술식의 이진트리 표현 (A / B * C * D + E)

1) Inorder Traversal (LPR)




```
void Tree:: inorder(Node *p)
{
    if (p) {
        inorder(p->leftchild);
        print   p->data;
        inorder(p->rightchild);
    }
}
```

output : A / B * C * D + E

2) PostOrder Traversal (LRP)

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

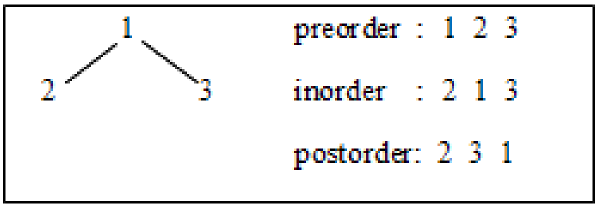
output: A B / C * D * E +

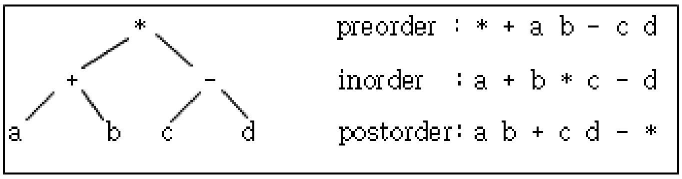
3) Preorder Traversal (PLR)

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

output: + * * / A B C D E

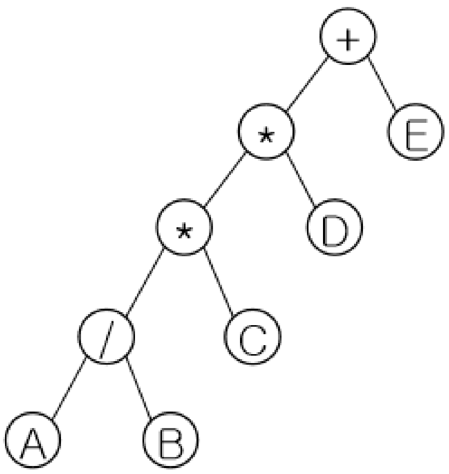
ex)





호출	currentNode의 값	결과	호출	currentNode의 값	결과
Driver	+		10	C	
1	*		11	0	
2	*		10	C	cout<<‘C’
3	/		12	0	
4	A		1	*	cout<<‘*’
5	0		13	D	
4	A	cout<<‘A’	14	0	
6	0		13	D	cout<<‘D’
3	/	cout<<‘/’	15	0	
7	B		Driver	+	cout<<‘+’
8	0		16	E	
7	B	cout<<‘B’	17	0	
9	0		16	E	cout<<‘E’
2	*	cout<<‘*’	18	0	

출력 : A/B*C*D+E



중위순회(Inorder)

Stack: +, *, *, /

Tree Build   * Building tree for mathematical expression

```
class Node {
  private:
        int data;
        Node *left;        // left    link
        Node *right;       // right    link
        int prio;          // priority number from precedence table
        Node(int value) {data = value; left = 0; right = 0; prio = }
  friend class Tree;        };

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

* Precedence Table

char prec[4][2] = {   '*', 2,   '/', 2,   '+', 1,   '-', 1};

Operator	'*'	'/'	'+'	'-'
priority	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 3    (if new-node -> data == prec[i][0])
                        then new-node ->prio = prec[i][1]
     . if (i==4)    then    call Operand(new-node)
                    else    call operator(new-node)     }    }
```

*** Operand(new-node)**

```
If    head==NULL    then     head=new-node     return
Node* 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**

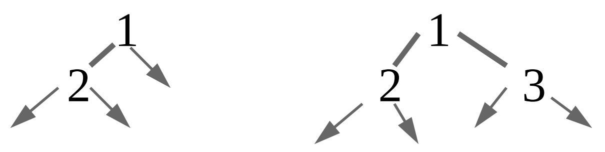
```
   int evalTree (p)          {

        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
          }

        }
        return value;
     }
```

4. 스레드 이진 트리 (Threaded Binary Tree)

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

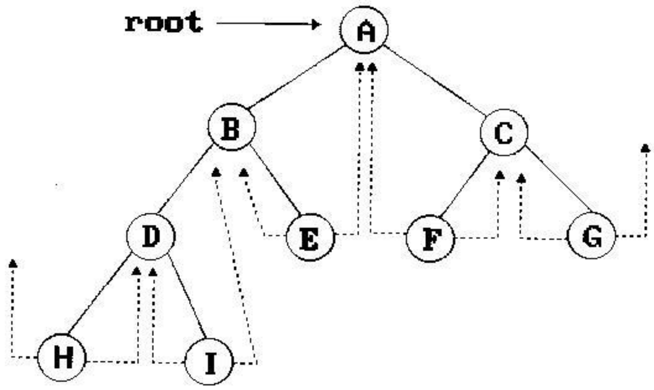


[n->2, Null->3] [n=3, null link=4]

- Threads 연결 규칙 (Ptr 이 현재 노드라 가정하면)

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

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



. 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

(1) normal pointer와 thread를 구분할 수 있어야 한다.

(2) leftThread와 rightThread field를 추가해야 한다

(3) structure 선언

```
struct Node {
        int leftThread;      Node *leftChild;
        char data;
        Node *rightChild;    int rightThread;
}
```

leftThread	leftChild	data	rightChild	rightThread
TRUE		'A'		FALSE

점선 실선

- thread 는 점선으로 표기, normal pointer 는 실선으로 표기

if thread field == true , then contains a thread

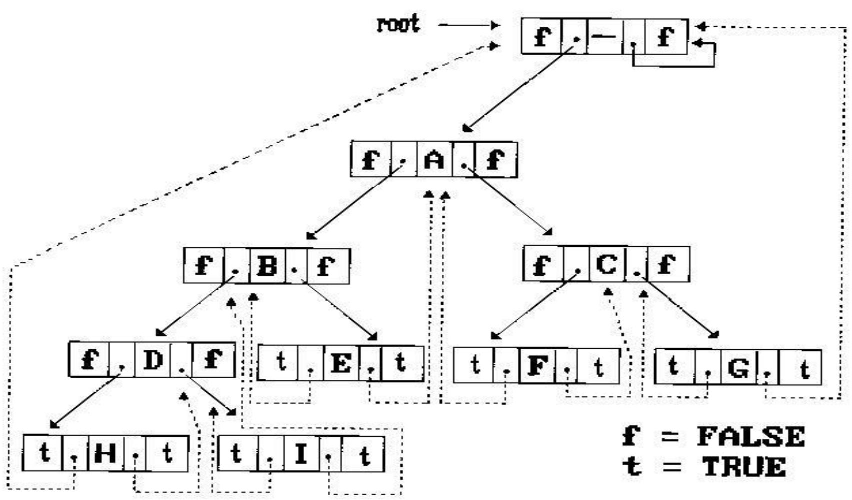
if thread field == false, then contains a pointer to a child

* 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 연결시켜 해결하며 그 결과는 아래 그림



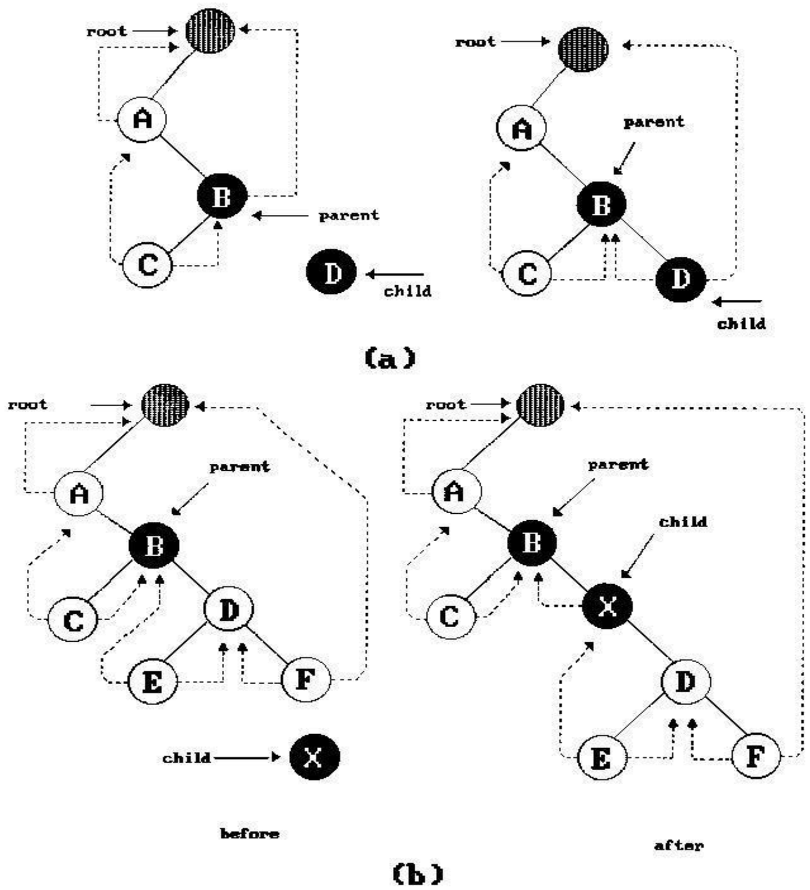
- Inorder: H D I B **E A F** C G

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

● 활용: inorder traversal 에 활용함 (스택 사용없이)

- recursive inorder traversal 을 간단한 non-recursive version 으로 구현할 수 있다.
- computing time은 마찬가지로 O(n)이지만 recursive call 에 따른 overhead는 없어짐

예)



[그림 a]: D 를 B 의 right 에 연결하기
(D 의 leftThread 는 B 에 연결, D 의 rightThread 는 Root 를 가르키게 한다)

[그림 ㅠ]: X 를 B 의 rightChild 에, D 를 X 의 rightChld 에 연결하기
(E 의 leftThread 는 X 에 연결. X 의 leftThread 는 B 를 가르키게 한다)

5. HEAP

5.1 HEAP creation

정의: HEAP is a special form of FULL/complete binary tree that is used in many applications (each node's data > it's children's)

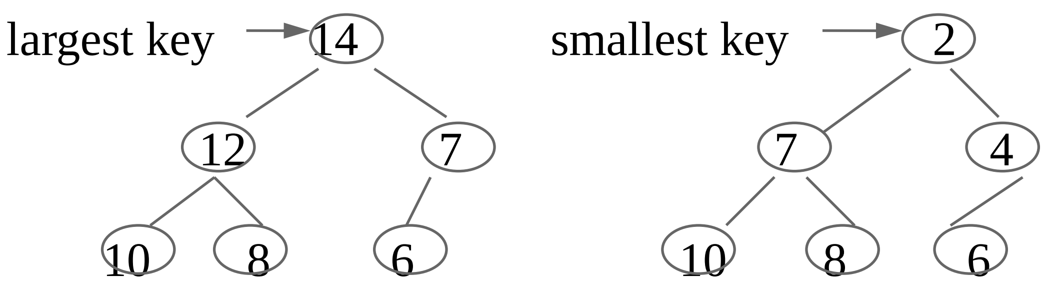
1) MAX TREE: is a tree in which the key value in each node is larger than the key values in its children (if any).

   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



- Representation - Use Arrays, (same as tree for array representation scheme)

- *MaxHeap ADT*

Template <class KeyType>
class MaxHeap: public MaxPQ<KeyType>
  //objects : 각 노드의 값이 그 자식들의 것보다 작지 않도록 조직된 n>0 원소의 완전 이진 트리
public:
  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
  Insert(heap, item, n); // if (!HeapFull(heap, n)),   item을 heap에 삽입
                                        else return error
  Boolean **HeapEmpty**(heap, n); //if (n==0) return TRUE
                                                    else return FALSE
  Delete(heap, n);   // if (!HeapEmpty(heap, n))
        Heap에서 가장 큰 원소를 제거 후 반환
        else return 에러

----

*class MaxHeap data member

```
    private:
          Element<Type> *heap;
          int n;                 // 최대 히프의 현재 크기
          int MaxSize;     // 히프의 최대 크기

   MaxHeap::MaxHeap(int sz=DefaultSize) {
        MaxSize = sz;
        n = 0;
        heap = new Element<Type>[MaxSize+1]; //heap[0]은 사용되지 않음
    }
```

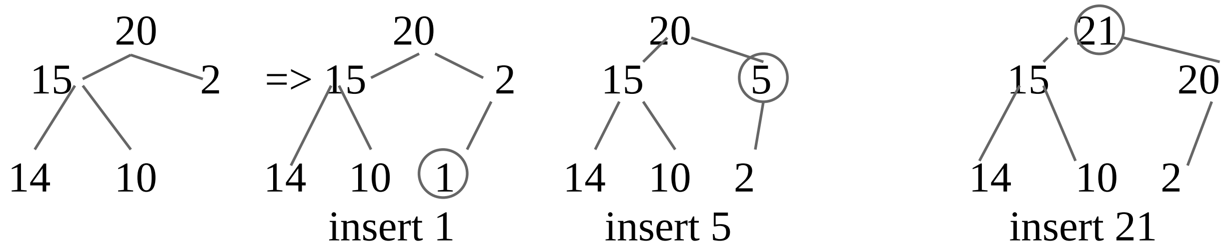
- **PRIORITY QUEUE**

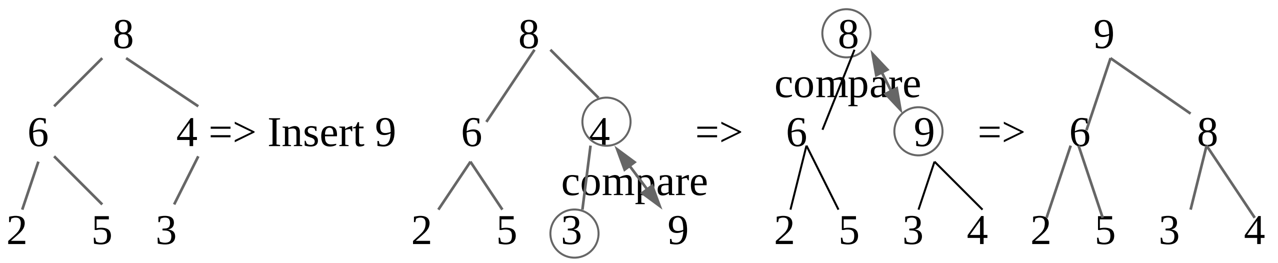
. HEAPs are used to implement PRIORITY QUEUE

- ⇨ 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

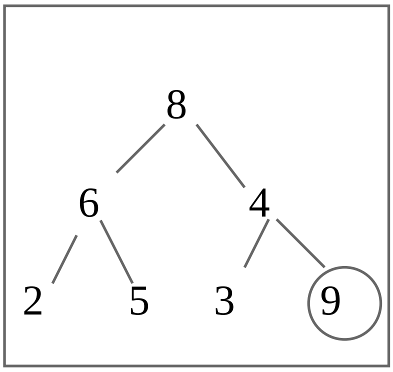
- ⇨ **ARRAY** is a simple representation of a priority queue (easily add to P.Q. by placing the new item at the current end of array) => insertion complexity O(1)

- **Insertion into a MAX HEAP**





```
void insert_maxheap (element item, int *n)
 {
   int i
   if (HEAP_FULL(*n)) {
          print (“Heap is full....\n”); return;
   }
   i = ++(*n);
   while ((i!=1) && (item    > heap[i/2] ))
     {        heap[i] = heap[i/2];
              i = ⌊i/2⌋;
     }
   heap[i] = item;
 }
```



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**

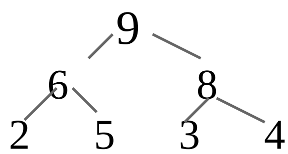
for i = i/2 = 3,

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

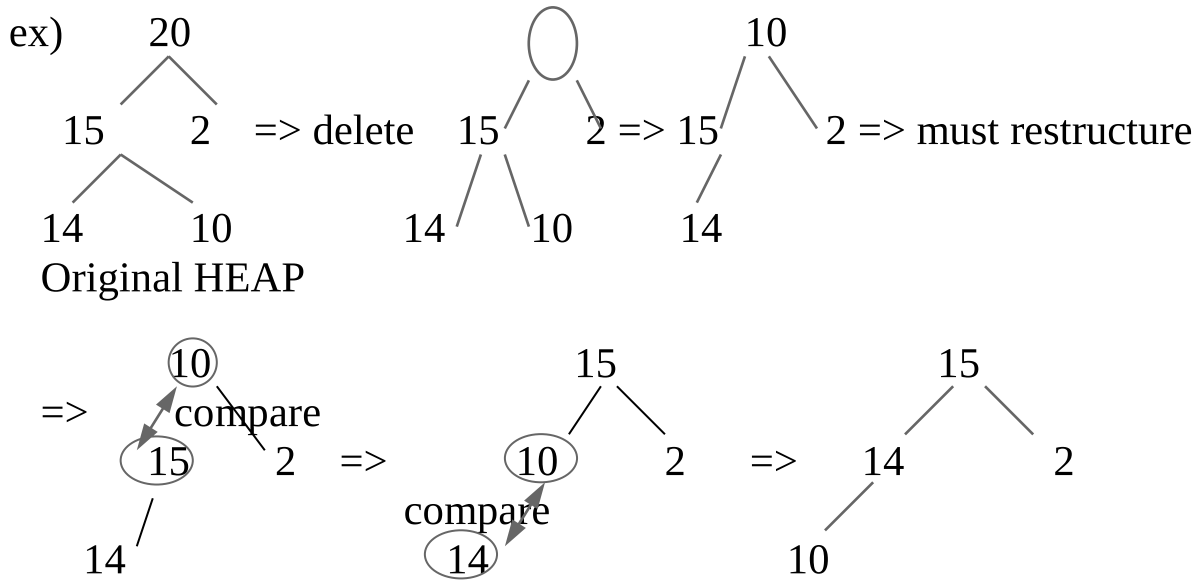
heap[1] = item => (heap[1] = 9)



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

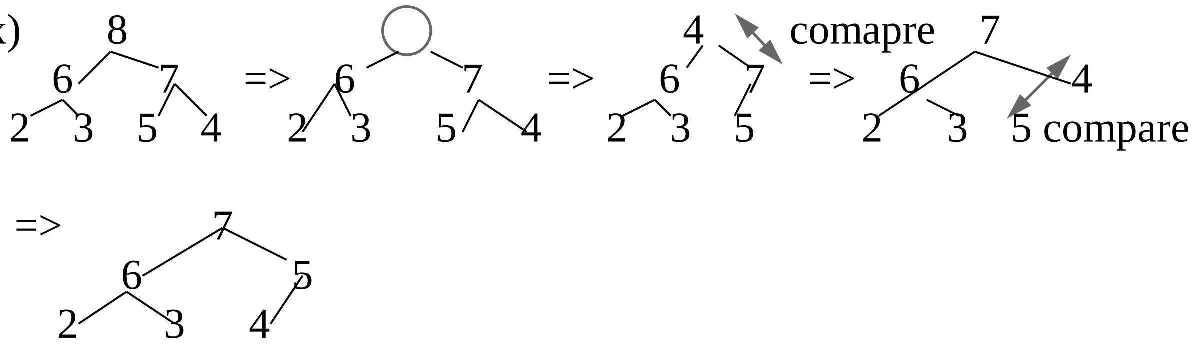
- **Deletion from a MAX HEAP**

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




```
element delete-maxheap (int *n)
{    int parent, child;
     element item, temp;
       …………………
   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;      }
```

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

7
6 7

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

3) while (child <= 6)
   . 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

7
6 5
2 3 4

4) heap [6] = (temp =4) //parent =6

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

7
6 5
2 3 4

5.2 HEAP Sort : （ O(nlogn) - worst, average case）

* Heap sort 의 두 단계

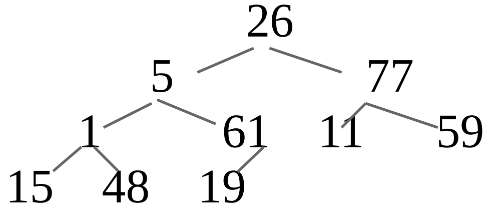
1) 화일 표현하는 tree 를 max HEAP 변환
2) ROOT 출력하고 나머지 tree 를 다시 HEAP 으로 만드는 process 계속 (**<u>Heapify</u>**)

```
void HEAPSORT (list[], int n)
{
   for (i = n/2; i>0; i--) {
        adjust(list, i, n);        //heap 변환

    for (i=n-1; i>0; I--) {
        swap(list[1], list[i+1], temp);    //root 출력
        adjust(list, 1, i);                //heap 변환
     }
   }
```

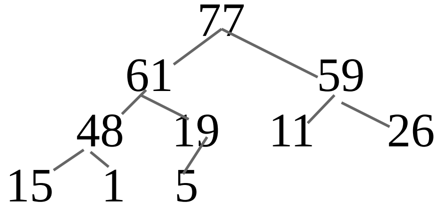
*** ADJUST (first FOR LOOP adjust => make MAX HEAP)**

ex) 26, 5, 77, 1, 61, 11, 59, 15, 48, 19

1) interpret as BT



2) Convert into HEAP



⇨ to convert into MAX HEAP follow the adjust algorithm

** Adjust main code

```
for (i=n/2;    i>0;    i--)    adjust(Heap,i,n);

Procedure adjust(Heap, i, n)    {
  int child,    j;
  child = 2*i;
  temp = Heap[i];

  while (n >= child) {
        if (n>=child && Heap[child] < Heap[child+1])
           child =child+1;    //right child  선택
        if(temp >= Heap[child]) break;
        j=child/2;
        Heap[j] = Heap[child];
        child=2*child;
  }
  j=child/2;        Heap[j]=temp;
  return;
}
```

1	2	3	4	5	6	7	8	9	10
26	5	77	1	61	11	59	15	48	19

1) 1^{st} loop, I = 5,

Temp = list[5] = 61   child = 10 (root*2)

if   61 > 19, break;

2) 2^{nd} loop, I=4,

Temp= list[4] = 1, child = 4*2 = 8,   (list[child] <list[child+1])

=> child = 9

compare (1 , 48) => list[4] = list[9]   //j=9/2

list[9] = temp=1

1	2	3	4	5	6	7	8	9	10
26	5	77	**48**	61	11	59	15	**1**	19

3) 3rd lop, I= 3,

temp = list[3] = 77, child = 3*2 = 6

(list[6] < list[7] => child = 7)

compare(77, > list[7]=59) =>break

1	2	3	4	5	6	7	8	9	10
26	5	**77**	48	61	11	59	15	1	19

4) 4th loop, I = 2,

temp = list[2] = 5, child = 4 & (since l[4] <L[5], child => 5)

compare(5, list[5]=61) => list[2] =list[5],

1	2	3	4	5	6	7	8	9	10
26	**61**	77	48	61	11	59	15	1	19

child = child*2 = 10

compare (5, < list[10]=19) => list[5]=list[10], list[10]=temp

1	2	3	4	5	6	7	8	9	10
26	61	77	48	**19**	11	59	15	1	**5**

5) 5th loop, I = 1,

temp = 26, child = 2,3=>child=3

compare(26, <list[3]=77) => list[1] = list[3] child= 6

1	2	3	4	5	6	7	8	9	10
77	61	77	48	19	11	59	15	1	5

child=6,7 => child=7

compare(26, <list[7]=59) list[3]=list[7], child=12 => list[7]=26

1	2	3	4	5	6	7	8	9	10
77	61	**59**	48	19	11	**26**	15	1	5

결과는 MAXHEAP

77

61 59

48 19 11 26

15 1 5

- 다음은 max heap 을 HEAPIFY 한다.

- **Heapify**

1	2	3	4	5	6	7	8	9	10
77	61	59	48	19	11	26	15	1	5

1)1st LOOP: I=9, SWAP (list[1], list[I+1], temp)

1	2	3	4	5	6	7	8	9	10
5	61	59	48	19	11	26	15	1	77

** adjust(HEAP, i, n)

* adjust(list, 1, i)

temp=5. Child=2 n=9

- 2<9, child=(2,3) = 2 선택,
- compare(5, list[2]=61) => list[1]=list[2], child=4

1	2	3	4	5	6	7	8	9	10
61	61	59	48	19	11	26	15	1	77

- 4<9, child=(4,5)=4 선택,
  compare(5, list[4]=48)=>list[2]=list[4], child=8

1	2	3	4	5	6	7	8	9	10
61	**48**	59	48	19	11	26	15	1	77

- 8<9, child=(8,9)=8 선택,
- compare(5, list[8]=15)=>list[4]=list[8], child=16

1	2	3	4	5	6	7	8	9	10
61	48	59	**15**	19	11	26	15	1	77

- list[8]=5

1	2	3	4	5	6	7	8	9	10
61	48	59	15	19	11	26	**5**	1	77

결과=>

```
              61
         48          59
     15       19   11     26
  5    1
```

2) 2nd LOOP: I=8, SWAP (list[1], list[I+1], temp)

1	2	3	4	5	6	7	8	9	10
1	48	59	15	19	11	26	5	61	77

* adjust(list, 1, I) , I = 8

temp=1, Child=2 n=8

- 2<8, child=(2,3) = 3 선택,
- compare(1, list[3]=59) => list[1]=list[3], child=6

1	2	3	4	5	6	7	8	9	10
59	48	59	15	19	11	26	5	61	77

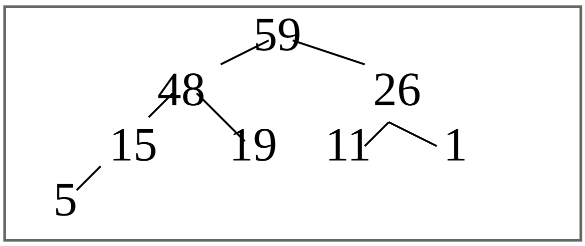
- 6<8, child=(6,7)=7 선택,
- compare(1, list[7]=26)=>list[3]=list[7], child=14

1	2	3	4	5	6	7	8	9	10
59	48	**26**	15	19	11	26	5	61	77

- list[7]=1

1	2	3	4	5	6	7	8	9	10
59	48	26	15	19	11	**1**	5	61	77

결과=>



3) 3rd LOOP: I=7,   SWAP (list[1], list[I+1], temp)

1	2	3	4	5	6	7	8	9	10
5	48	26	15	19	11	1	59	61	77

* adjust(list, 1, I) , I = 7

temp=5, Child=2 n=7

- 2<7, child=(2,3) = 2 선택,
- compare(5, list[2]=48) => list[1]=list[2], child=4

1	2	3	4	5	6	7	8	9	10
48	48	26	15	19	11	1	59	61	77

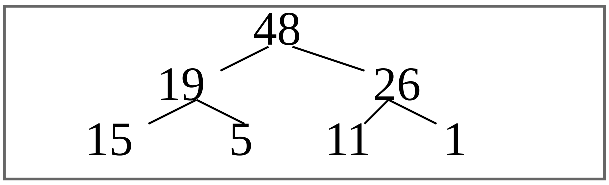
- 4<8, child=(4,5)=5 선택,
- compare(5, list[5]=19)=>list[2]=list[5], child=10

1	2	3	4	5	6	7	8	9	10
48	**19**	26	15	19	11	1	59	61	77

- list[5]=5

1	2	3	4	5	6	7	8	9	10
48	19	26	15	**5**	11	1	59	61	77

결과=>



■ 생략..

- Sorted List: 1, 5, 11, 19, 26, 48, 59, 61, 77

- Heap sort 요약　　28, 7, 79, 3, 64, 15, 58, 17, 46,23

*** create MAX HEAP**

28
7　79
3　64　15　58
17　46　23

n div 2 => 5,　5 번노드부터 시작

1) 5 번노드 -> ok

2) 4 번노드 -> 46
17　3

3) 3 번노드 -> ok

4) 2 번노드 -> 64
46　23
17　3　7

5) 1 번노드　79
64　58
46　23　15　28
17　3　7

* Heapify

(79)
64
46　58
17　23　15　28
7　3

(79, 64)
58
46　28
17　23　15　3
7

(79, 64, 58)
46
23　28
17　7　15　3

(79, 64, 58, 46)
28
23　15
17　7　3

(79, 64, 58, 46, 28)
23
17　15
3　7

(79, 64, 58, 46, 28, 23)
17
7　15
3

(79, 64, 58, 46, 28,23, 17)
15
7　3

(79, 64, 58, 46, 28,23, 17, 15)
7
3

결과 =>　(79, 64, 58, 46, 28,23, 17, 15, 7, 3)

6. 이진 탐색트리 (Binary Search Trees :BST)

⇨ HEAP 은 우선순위 큐(**priority queue)** 응용에는 적합하지만, 임의의 노드를 삭제하기에는 적합치 않다.

⇨ BST 는 Insert, Delete, Search 등을 수행하기에 편리한 자료구조 이다.

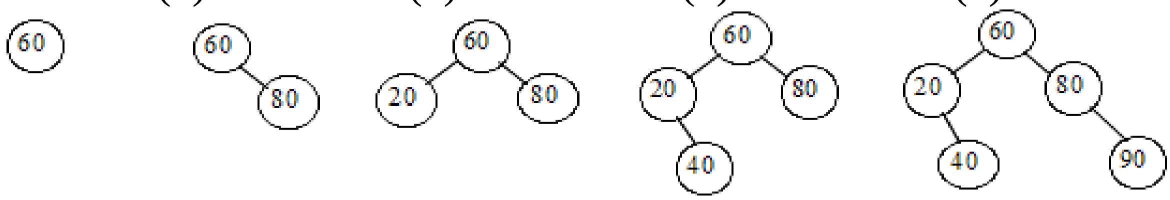
⇨ BST 는 이러한 연산을 Key Value (예: delete element x), 와 RANK (예: delete 6th position)로 수행한다.

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

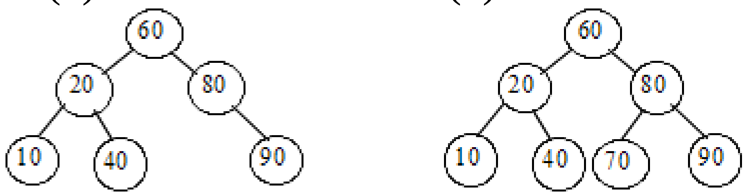
1) 모든 노드에는 Key 값이 있다. 동일한 Key 값은 존재하지 않음.
2) Left subtree 에 있는 Key 는 ROOT node 의 Key 값 보다 작다.
3) Right subtree 에 있는 Key 는 ROOT node 의 Key 값 보다 크다.

[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



[예제#3] 65 30 50 80 60 10 70 순서로 삽입 **[예제#4] 10 20 30 40 50 60**

[예제#5] box cow owl monkey zebra

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

1) **Searching a BST**

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

2) Inserting into a BST

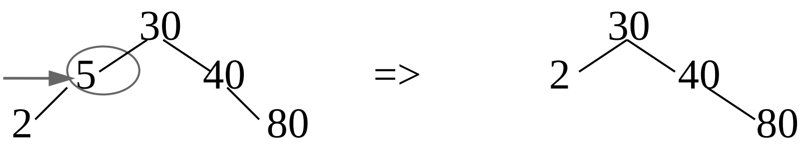
* In order to insert, we must search the tree => if the search is unsuccessful, we insert the element at the point the search terminated

```
INSERT (ptr, key)
{
  if (ptr=NULL) {    // if empty
      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;
}
```

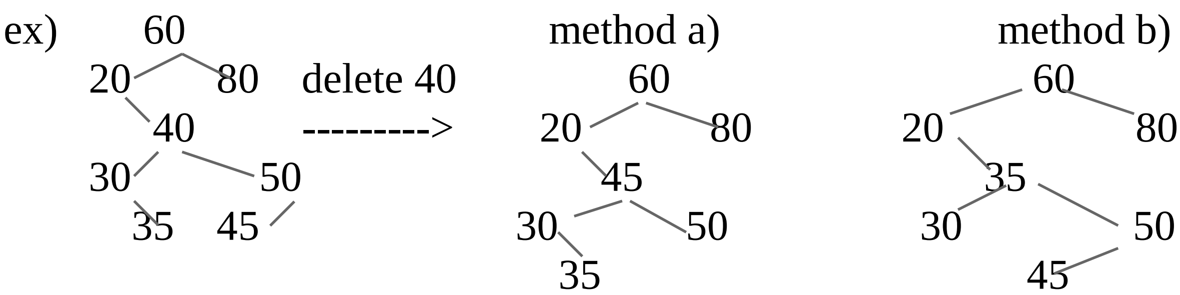
```
freeBSTree(Node *p){
   if (p != 0) {
      freeBSTree(p->left);
      freeBSTree(p->right);
      delete p;
   }
}

traverseTree() {inorderTraverse(root);}


inorderTraverse(Node *p){
if (p != 0) {
     inorderTraverse(p->left);
     cout << p->data << endl;
     inorderTraverse(p->right); }
```

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 pointer from parent to single child



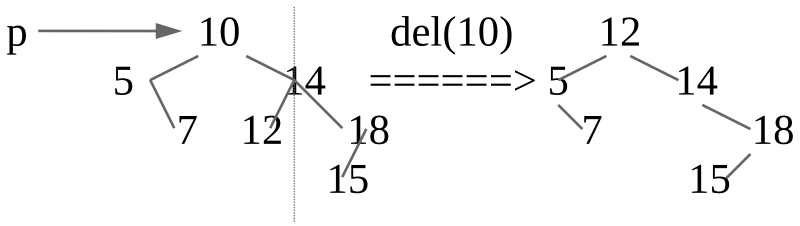
* Nonleaf node with two children
   a. replace with smallest element in rightsubtree(findMin)
   b. replace with largest element in leftsubtree (findMax)



delete (ptr, key) {

```
  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*/
      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    temp = find_min(ptr->right)    /*both child exists */
                 ptr->data = temp->data;
                  ptr->right = delete(ptr->right, ptr->data);
   }
   else    print("Not found");5

 return ptr;
```

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

int findMax( node *p)    /*right subtree 에서  가장  큰것  선택  */
     {
       if (p==NULL) return NULL;
       else if (p->right ==NULL) return p;
       else return findMax(p->right);
  }
```



- <u>선언</u>

```
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 *); //same as postOrder, preOrder
        void drawTree();
        void drawBSTree(Node *, int);
        Node *findmin(Node *p);      //findmax
        int tree_empty();
        void freeBSTree(Node *);
        //etc
    };

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

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

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 << 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);
    }
}
```