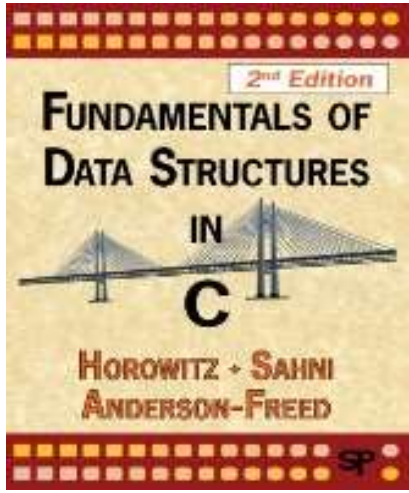
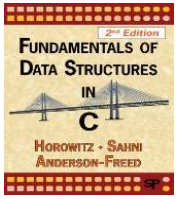
제 5 장

트리

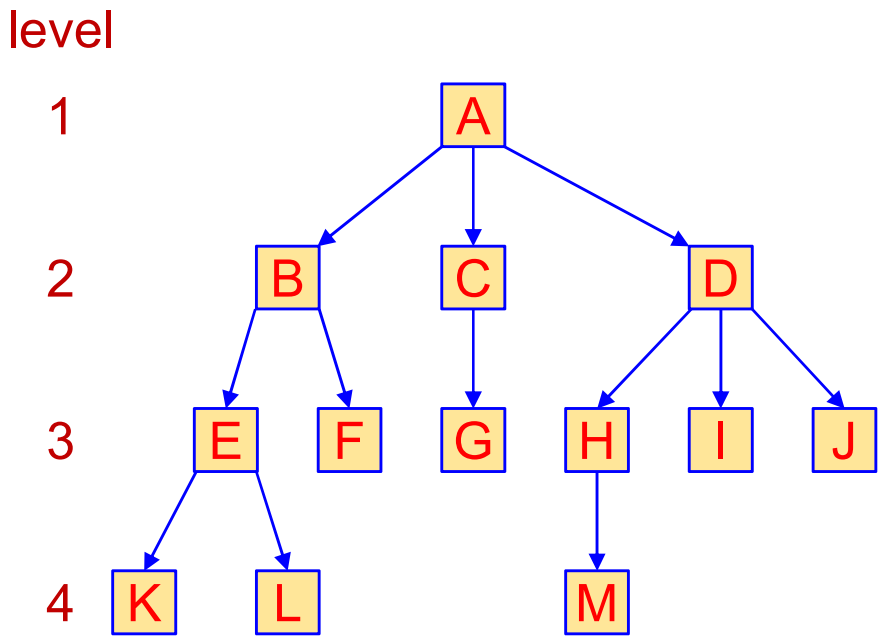
Trees


2nd Edition
FUNDAMENTALS OF
DATA STRUCTURES
IN
C
HOROWITZ • SAHNI
ANDERSON-FREED


2nd Edition
FUNDAMENTALS OF
DATA STRUCTURES
IN
C
HOROWITZ • SAHNI
ANDERSON-FREED

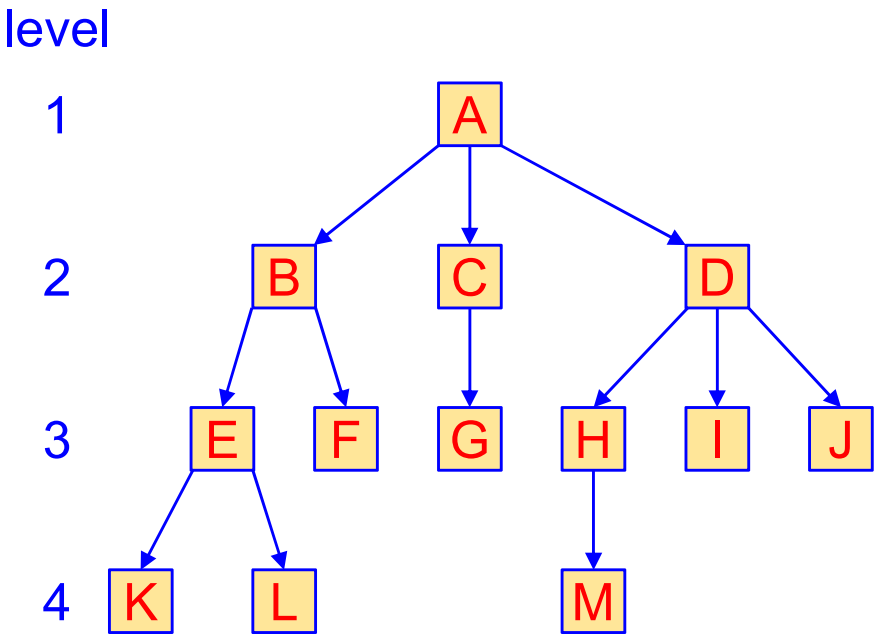
트리의 정의

- 트리는 데이터가 링크에 의하여 관련되도록 조직화
- 정의
  - 트리는 유한한 노드들의 집합으로 다음 조건을 만족한다.
    1. 루트(root)라는 특별한 노드가 있다.
    2. 나머지 노드들은 n개의 트리 T_1, T_2, …, T_n으로 분리되고, 각 T_1, T_2, …, T_n 들은 루트의 subtree라 부른다.
- 특징
  - 순환적으로 정의
  - 일반화된 리스트
  - 순환이 없는 그래프
  - 링크에 따라 조직화
- 용도
  - 족보
  - 혈통도
  - 계층구조



전문 용어

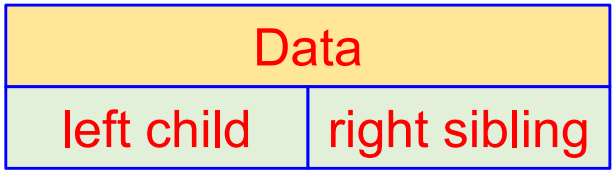
- 차수(degree)
  - 노드의 차수: 노드에서 서브트리의 갯수
  - 트리의 차수: 트리에서 노드의 최대 차수
- 터미널/내부 노드
  - 터미널:차수가 0인 노드(K, L, F, G, M, I, J)
  - 내부: 차수가 1 이상인 노드(A, B, C, D, E, H)
- 부모, 자식, 형제, 조상, 후손
  - 부모(parent): 바로 위 레벨의 노드
  - 자식(child): 서브트리의 루트
  - 형제(sibling): 부모가 같은 자식 노드
  - 조상(ancestor): 노드에서 루트까지의 노드
  - 후손(descendant): 서브트리의 모든 노드
- 레벨
  - 루트: 1(0)
  - 자식: 부모의 레벨 + 1



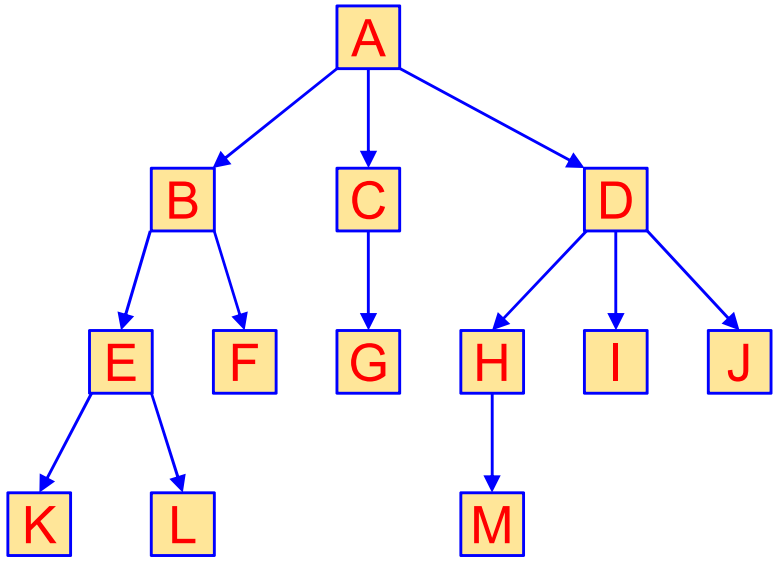
- 깊이(depth), 높이(height)
  - 최고 레벨
- 숲(forest)
  - 트리들의 집합

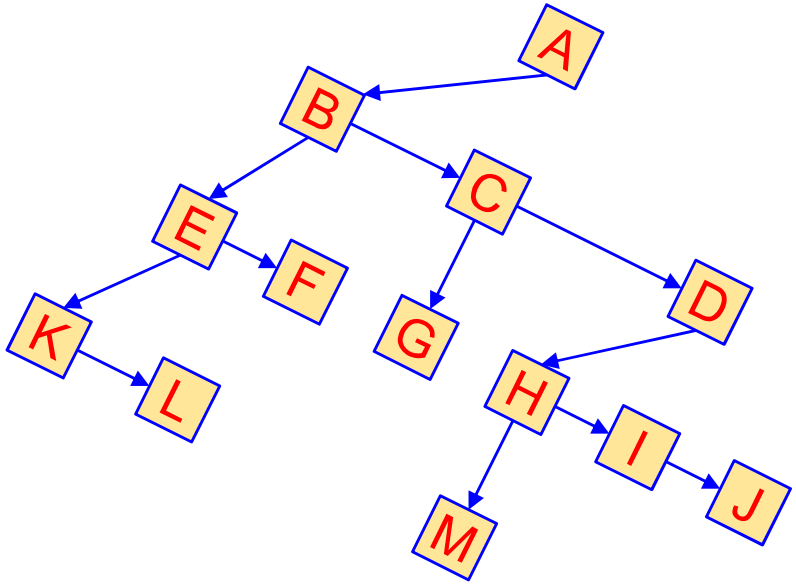
트리의 표현

- 리스트로 표현
  - 노드: (Data subtree$_1$ subtree$_2$ ... subtree$_n$)
  - (A (B (E (K) (L)) (F))
    (C (G))
    (D (H (M)) (I) (J)))
- 연결 리스트로 표현
  - 차수가 n인 트리
  - 

  - 노드의 크기가 가변
- 좌자식-우형제로 표현
  - 노드의 크기가 고정
    - 링크의 갯수가 2
  - 

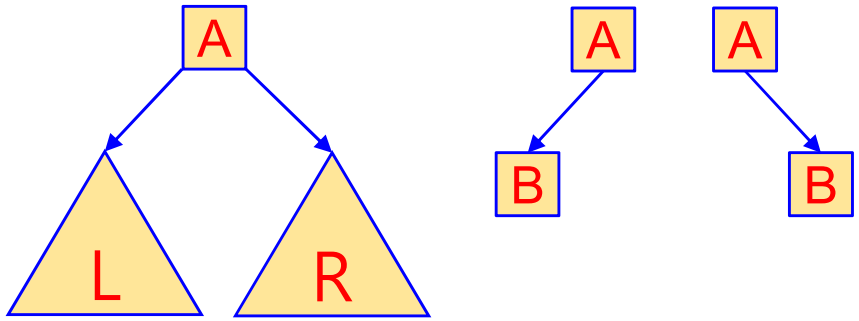
  - 이진 트리





이진 트리(Binary Tree)

- 정의
  - 유한한 노드들의 집합으로 다음의 하나를 만족한다.
    - 노드의 갯수가 0
    - 루트와 두 이진 트리로 분리된다.
      두 이진 트리는 왼쪽 서브트리, 오른쪽 서브트리라 불린다.



- 특징
  - 자주 사용되는 트리 타입
    - 이진 탐색 트리
  - 좌우 이진 서브트리가 서로 구분
    - 서브트리의 순서가 중요
  - 링크 최대 2개

자료구조 이진트리

structure BTree
  declare Create() → *btree*
  IsEmptyBTree(*btree*) → Boolean
  MakeBTree(*btree*, *item*, *btree*) → *btree*
  LChild(*btree*) → *btree*
  RChild(*btree*) → *btree*
  Data(*btree*) → *item*
  for all *l*, *r* ∈ *btree*, *d* ∈ *item* **let**
  IsEmptyBTree(Create()) ::= *true*
  IsEmptyBTree(MakeBTree(*btree*, *item*, *btree*)) ::= *false*
  LChild(Create()) ::= *error*
  LChild(MakeBTree(*l*, *item*, *r*)) ::= *l*
  Data(Create()) ::= *error*
  Data(MakeBTree(*l*, *item*, *r*)) ::= *item*
  RChild(Create()) ::= *error*
  RChild(MakeBTree(*l*, *item*, *r*)) ::= *r*
  end
end BTree

이진 트리의 특성

- 최대 노드 수
  - 레벨 k: 2^{k-1}
    - 레벨 1에 노드 수는 1
    - 다음 레벨을 노드 수는 최대 2배
    - k 레벨의 노드 수는 2^{k-1}
  - 깊이 k인 트리의 노드 수: $2^k - 1 (2^{k+1} - 1)$
    - $2^0 + 2^1 + 2^2 + \ldots + 2^{k-1} = 2^k - 1$

레벨	노드수
1(0)	$1 = 2^0$
2(1)	$2 = 2^1$
3(2)	$4 = 2^2$
4(3)	$8 = 2^3$
:	:
k	$2^{k-1}(2^k)$

- 노드들의 관계

  n: 전체 노드 수

  n_0: 차수가 0인 노드 수

  n_1: 차수가 1인 노드 수

  n_2: 차수가 2인 노드 수

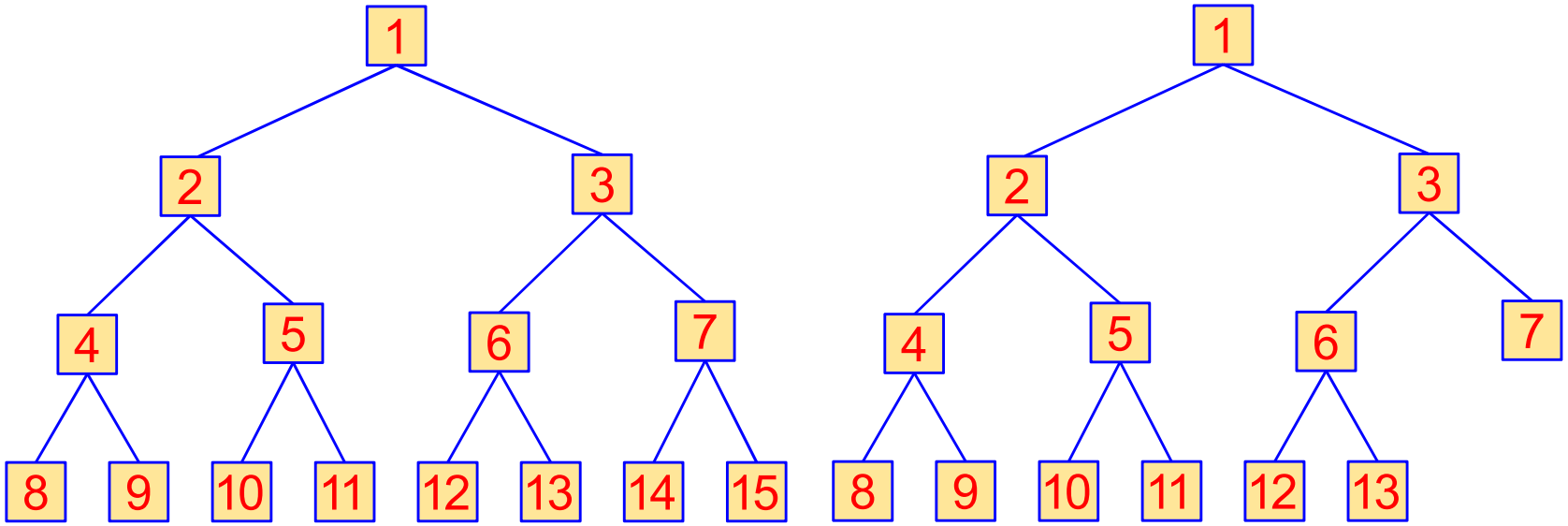
  $n = n_0 + n_1 + n_2 = link + 1 = n_1 + 2n_2 + 1$

  $n_0 + n_1 + n_2 = n_1 + 2n_2 + 1$

  $n_0 = n_2 + 1$

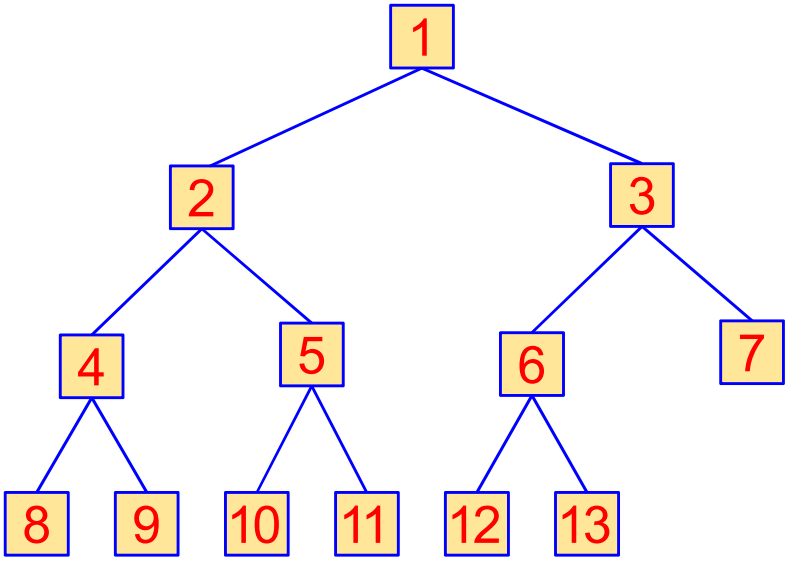
포화 이진 트리(Full Binary Tree)

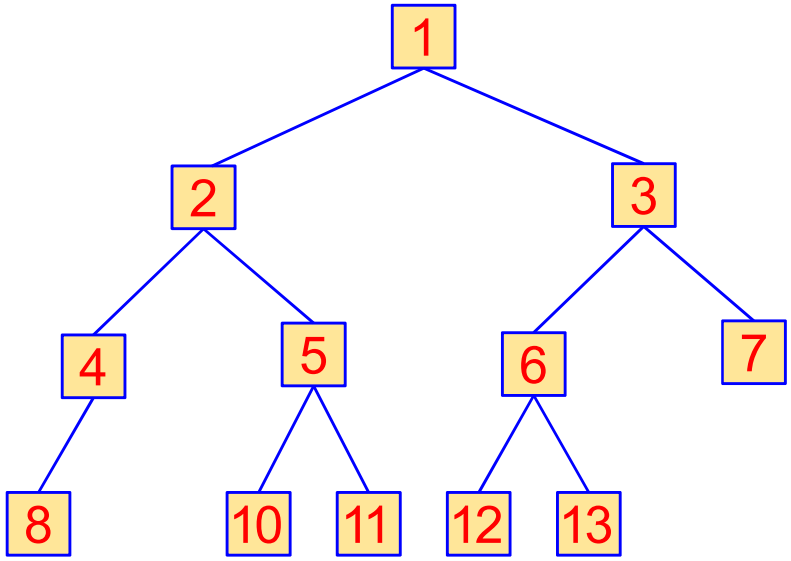
- 정의
  - 깊이가 k인 이진 트리가 최대의 노드(2^k-1)를 가지는 트리
  - 레벨 k까지 노드를 가득 채워야 한다.
- 이진 트리에서 노드 넘버링(Numbering)
  - 이진 트리에서 노드에 번호를 부여할 때 포화 이진 트리라고 가정
  - 루트 노드에서 1을 부여, 다음 레벨은 왼쪽에서 오른쪽으로 순서대로 부여
  - 번호가 n인 노드의 왼쪽 자식은 2n, 오른쪽 자식은 2n+1을 부여한다.



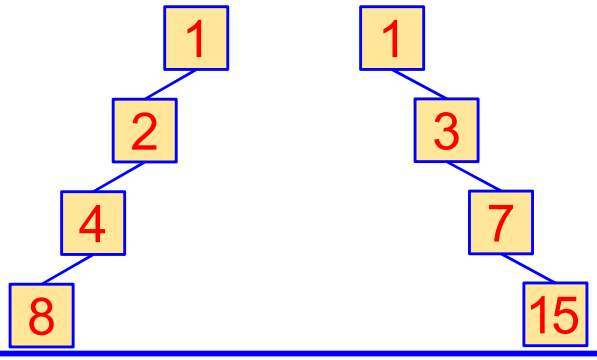
완전 이진 트리(Complete Binary Tree)

- 정의
  - 노드 수가 n인 이진 트리가 n까지 넘버링되는 트리
  - 마지막 레벨을 제외하면 포화 이진 트리이고, 마지막 레벨은 왼쪽부터



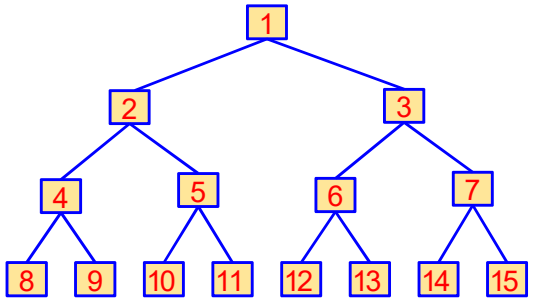


- 경사 트리(Skewed Tree)
  - 한 방향으로 기울어진 트리
  - 트리이지만 연결 리스트



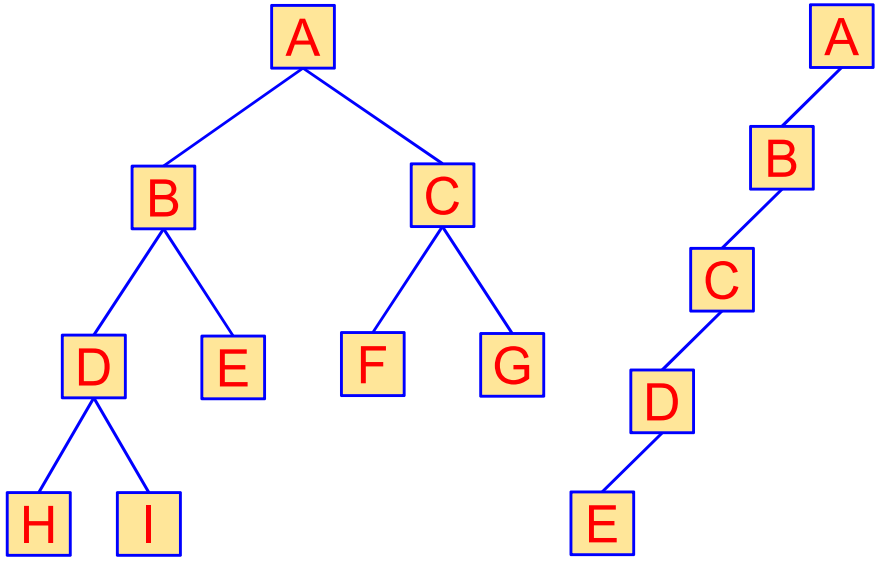
배열로 이진 트리 표현

- 배열로 표현
  - 이진 트리에 노드 넘버링에 대응하여 배열에 저장
    - 색인 n에 대응하는 노드에 대하여
      - Parent(n) n/2에 (루터가 아닐 경우)
      - LChild(n) 2n에 (왼 서브트리가 있을 경우)
      - RChild(n) 2n+1에 (오른 서브트리가 있을 경우)



  - 완전 이진 트리의 예

1	2	3	4	5	6	7	8	9
A	B	C	D	E	F	G	H	I



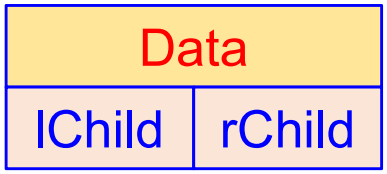
  - 경사 이진 트리의 예

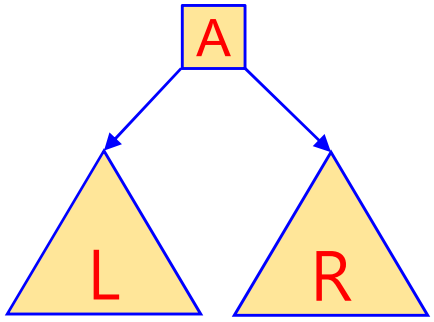
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	B		C				D								E

- 배열 표현의 문제점
  - 완전 이진 트리는 이상적인 표현
  - 경사 트리는 메모리 효율성 문제
  - 삽입/삭제가 어렵다(데이터 이동이 급격하게 증가)

링크로 이진 트리 표현

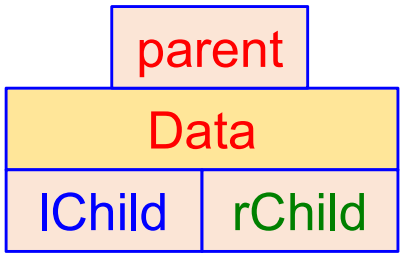
- 링크로 표현
  - 노드는 데이터, 왼쪽 이진 트리 링크, 오른쪽 이진 트리 링크로 구성
  -
```
typedef struct node {
        int             nData;
        struct node  *lChild;
        struct node  *rChild;
}       Node, *NodePtr, *TreePtr;
```

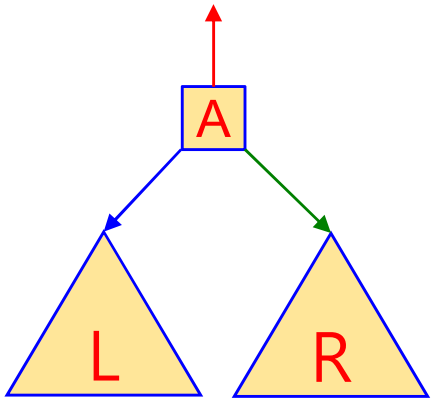




  - 부모 정보가 필요하면 링크를 하나 더 추가

```
typedef struct node {
        int             nData;
        struct node  *parent;
        struct node  *lChild;
        struct node  *rChild;
}       Node, *NodePtr, *TreePtr;
```

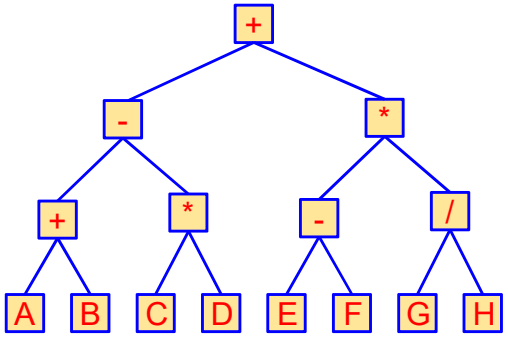




이진 트리 순회

- 원칙
  - 각 노드는 정확히 한 번만 방문
  - 항상 왼쪽 이진 트리를 오른쪽 이진 트리 보다 먼저 방문
  - 데이터, 왼쪽 이진 트리, 오른쪽 이진 트리에 대한 방문 순서
    - 전위 순회: D L R
    - 중위 순회: L D R
    - 후위 순회: L R D
- 이진 연산자들의 수식에 대한 이진 트리
  - 전위 순회로 노드의 데이터를 출력하면 전위 표기식
  - 중위 순회로 노드의 데이터를 출력하면 중위 표기식
  - 후위 순회로 노드의 데이터를 출력하면 후위 표기식

((A+B)-(C*D)) + (E-F)*(G/H)

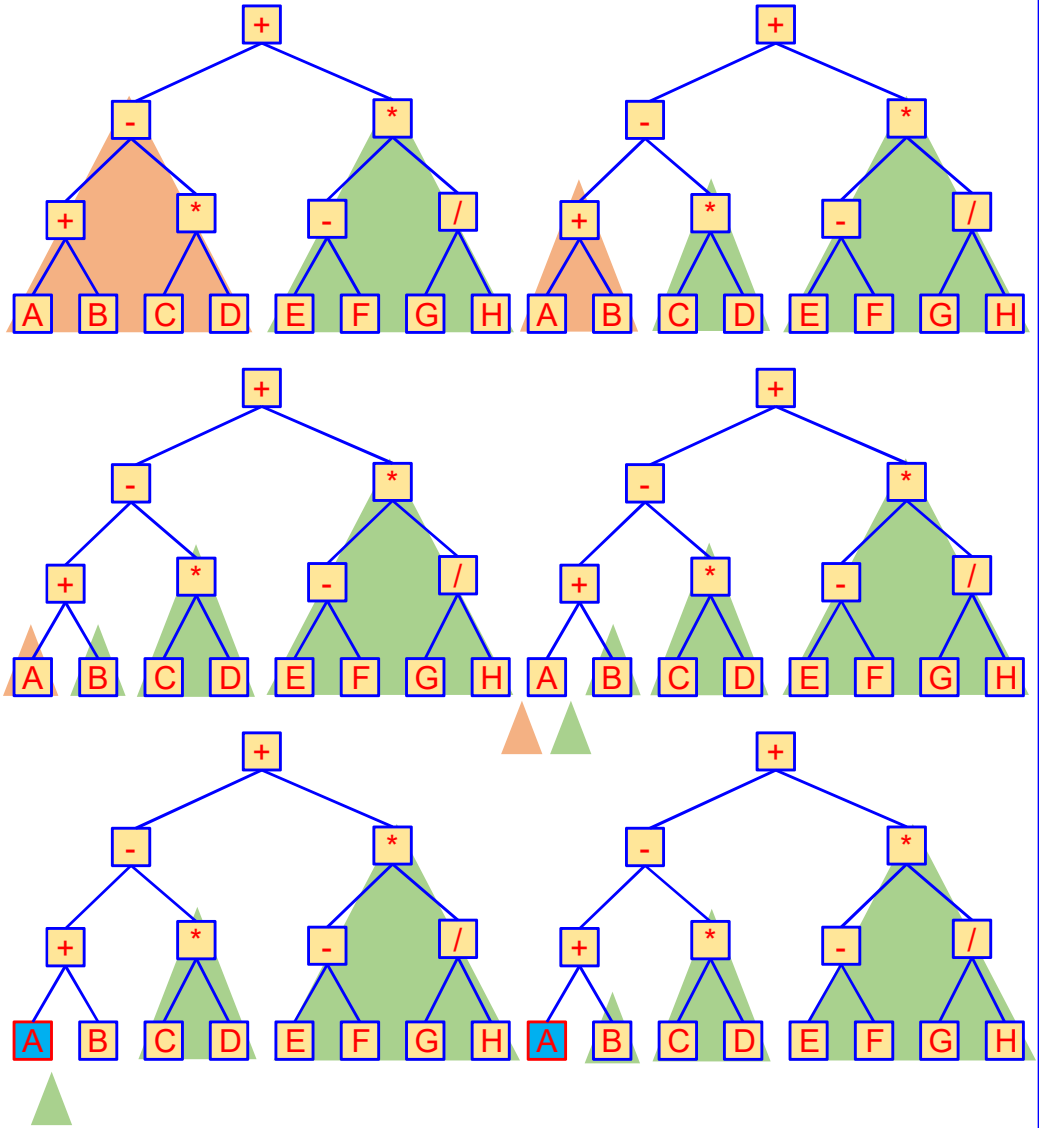


중위 순회(Inorder Traversal)

TreeOpr.cpp

- L D R
  1. 왼쪽 서브트리를 **중위 순회**를 하고,
  2. 루트를 방문하고,
  3. 오른쪽 서브트리를 **중위 순회**한다.
- 알고리즘

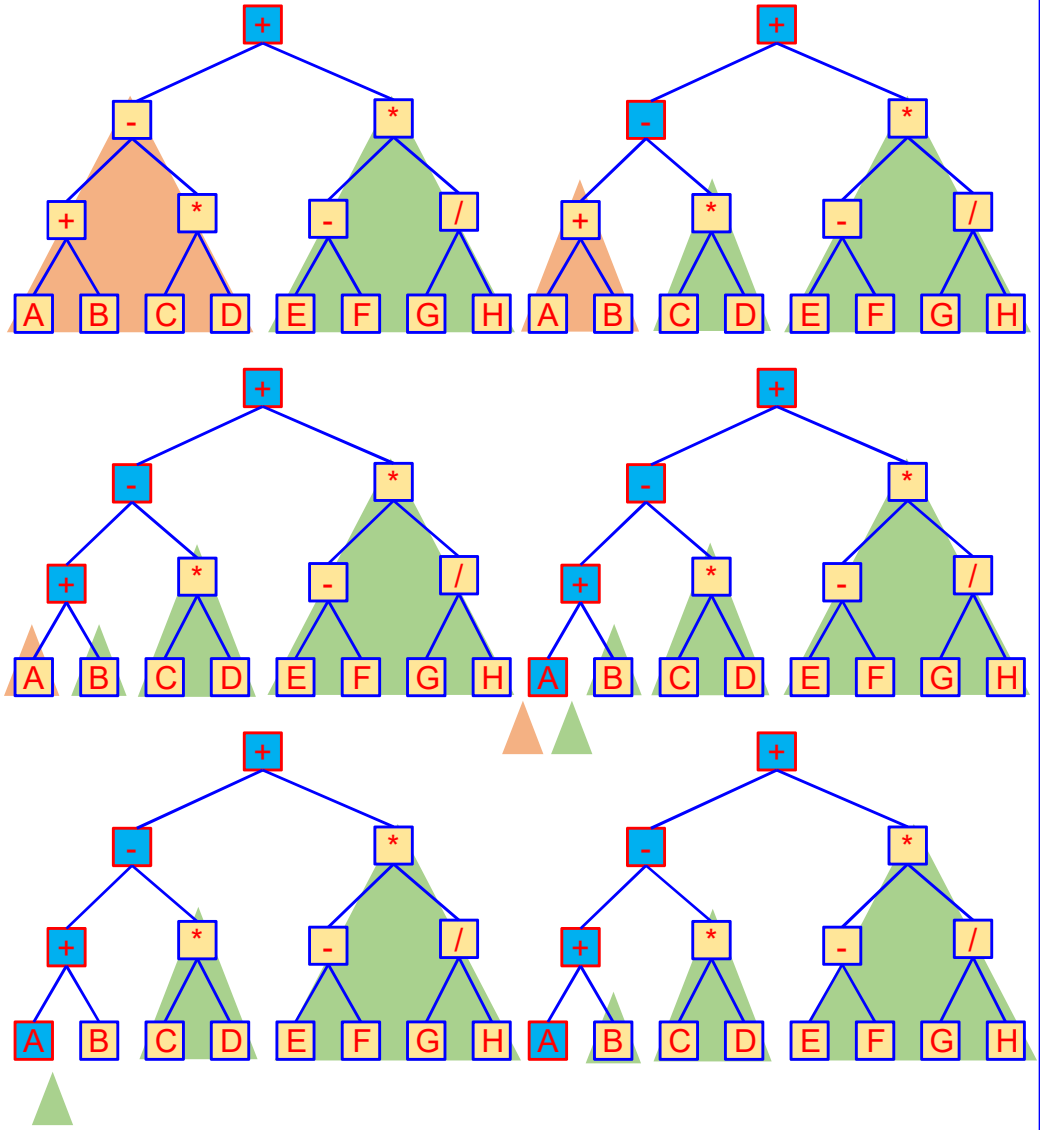
```
void inorder(TreePtr pTree)
{
    if (pTree) {
        inorder(pTree->lChild);
        printf("%c", pTree->nData);
        inorder(pTree->rChild);
    }
}
```



전위 순회(Preorder Traversal)

- D L R
  1. 루트를 방문하고,
  2. 왼쪽 서브트리를 **전위 순회**를 하고,
  3. 오른쪽 서브트리를 **전위 순회**한다.
- 알고리즘

```
void preorder(TreePtr pTree)
{
    if (pTree) {
        printf("%c", pTree->nData);
        preorder(pTree->lChild);
        preorder(pTree->rChild);
    }
}
```

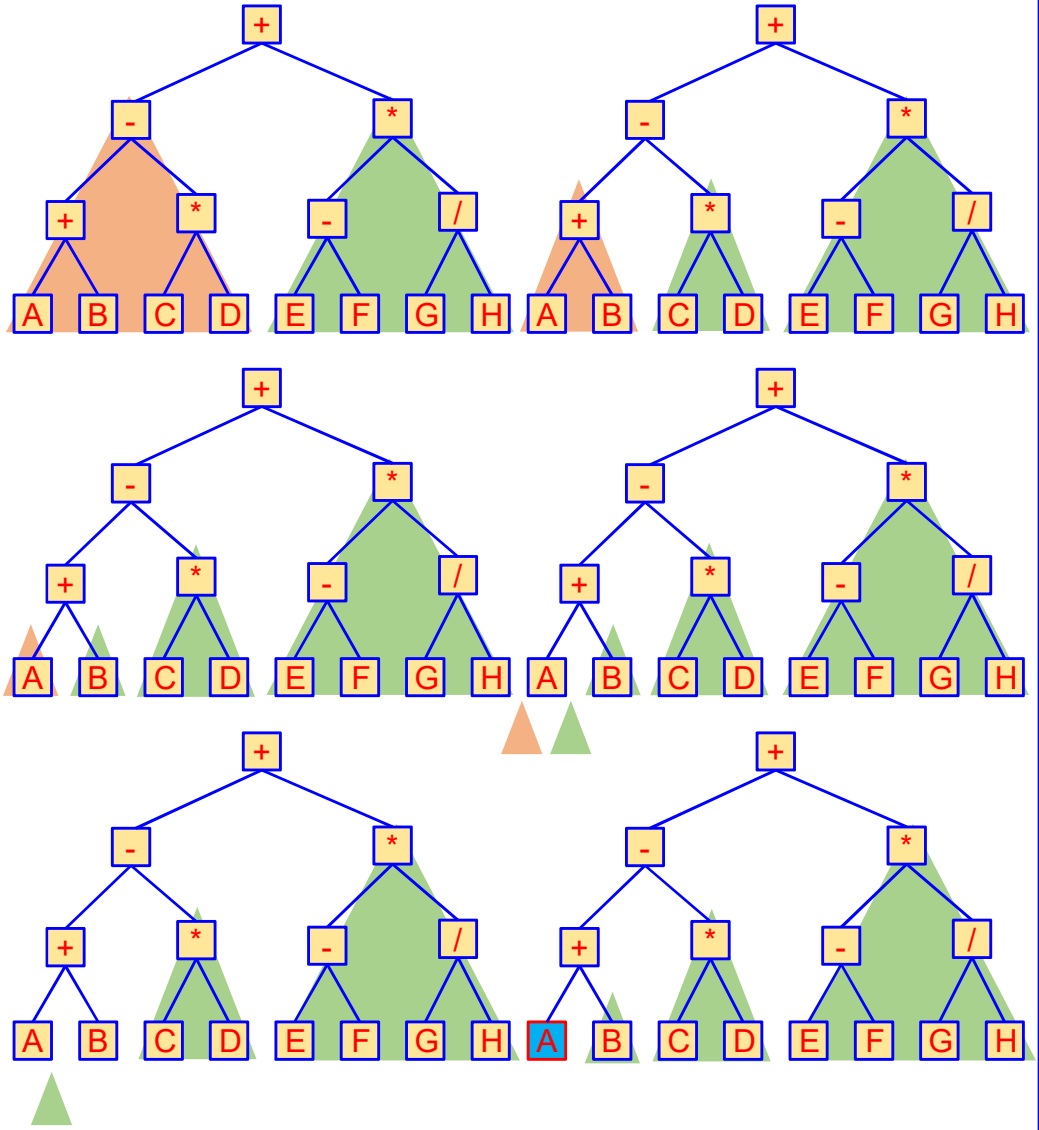


후위 순회(Preorder Traversal)

Tree4Trvsl.cpp

- L R D
  1. 왼쪽 서브트리를 **후위 순회**를 하고,
  2. 오른쪽 서브트리를 **후위 순회**하고,
  3. 루트를 방문한다.
- 알고리즘

```
void postorder(TreePtr pTree)
{
    if (pTree) {
        postorder(pTree->lChild);
        postorder(pTree->rChild);
        printf("%c", pTree->nData);
    }
}
```

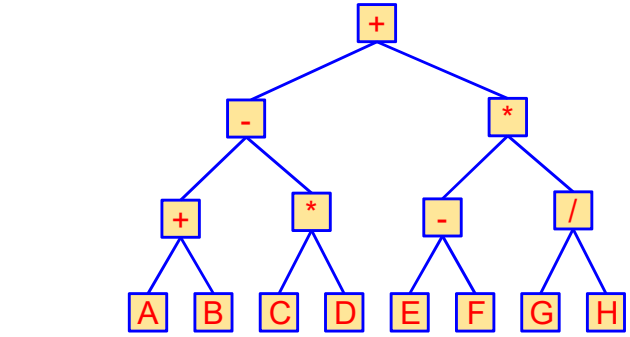


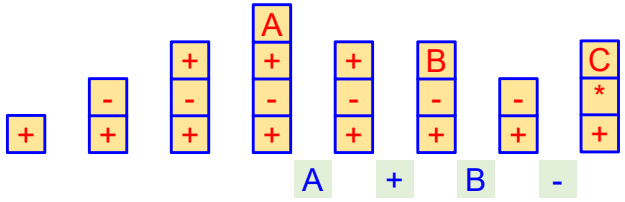
중위 순회(반복)

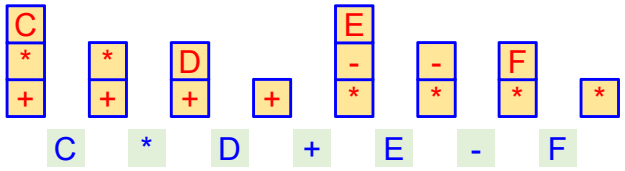
- 스택을 사용
  - 두 서브트리를 처리
    - 하나는 처리할 수 있겠지만
    - 나머지는 스택에 저장하여 나중에 처리

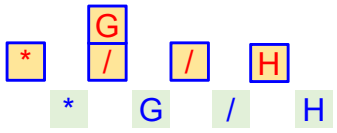
```
typedef struct node {
        TreePtr   items[StackSize];
        int       nTop;
}       Node, *NodePtr, Stack, *StackPtr;

void iterInorder(TreePtr pTree)
{
        Stack aStack;
        aStack.nTop = -1;
        while (1) {
                for ( ; pTree; pTree = pTree->lChild)
                        Push(&aStack, pTree);
                if ((pTree = Pop(&aStack)) == NULL)
                        break;
                printf("%c", pTree->nData);
                pTree = pTree->rChild;
        }
}
```







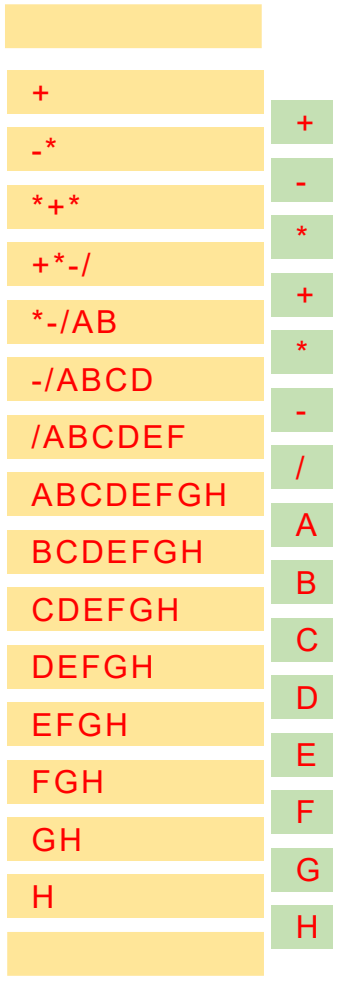


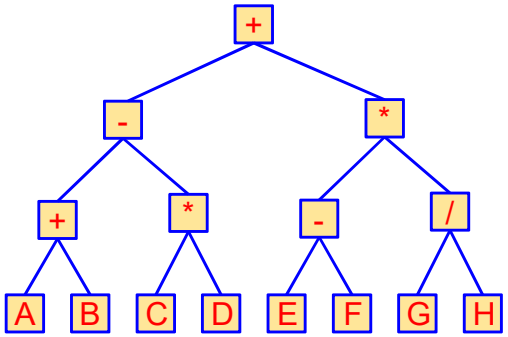
레벨 순회

- 레벨 순회
  - 상위 레벨 우선으로 순회
  - 같은 레벨은 왼쪽에서 오른쪽으로
  - 큐를 사용

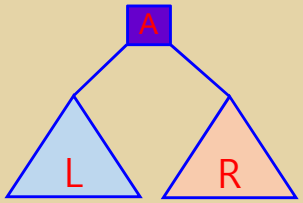
```
void leveOrder(TreePtr pTree)
{
    if (pTree == NULL)
        return;
    Queue aQueue;
    aQueue.nFront = aQueue.nRear = 0;
    AddQ(&aQueue, pTree);
    while (1) {
        if ((pTree = DeleteQ(&aQueue)) == NULL)
            break;
        printf("%c", pTree->nData);
        if (pTree->lChild)
            AddQ(&aQueue, pTree->lChild);
        if (pTree->rChild)
            AddQ(&aQueue, pTree->rChild);
    }
}
```

```
typedef struct node {
    TreePtr   items[QueueSize];
    int       nFront, nRear;
}   Node, *NodePtr, Queue, *QueuePtr;
```



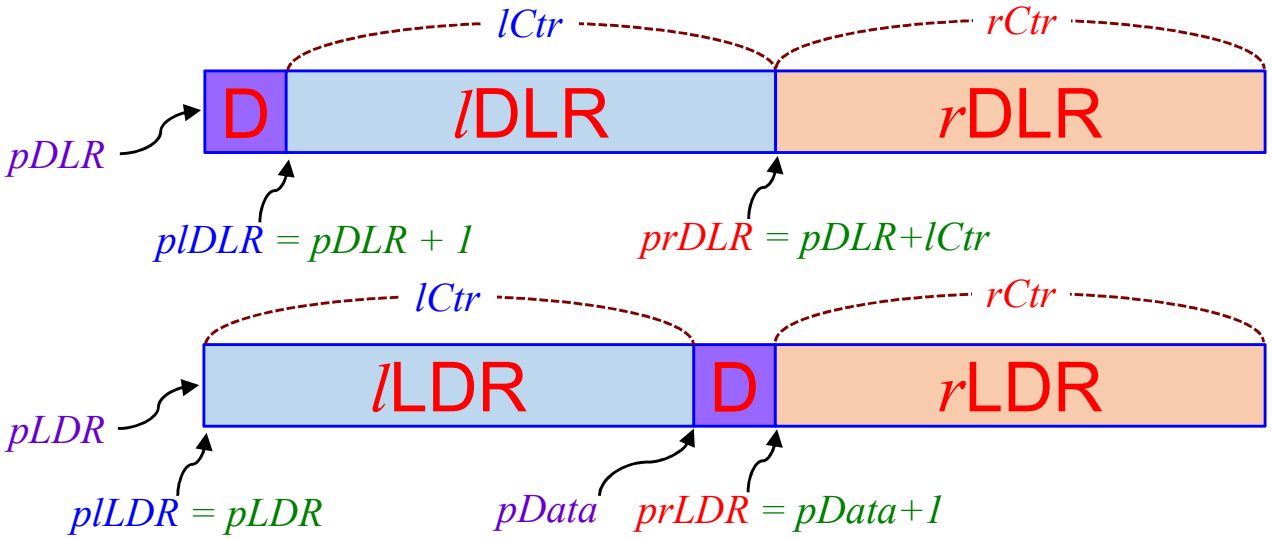


이진트리 복원



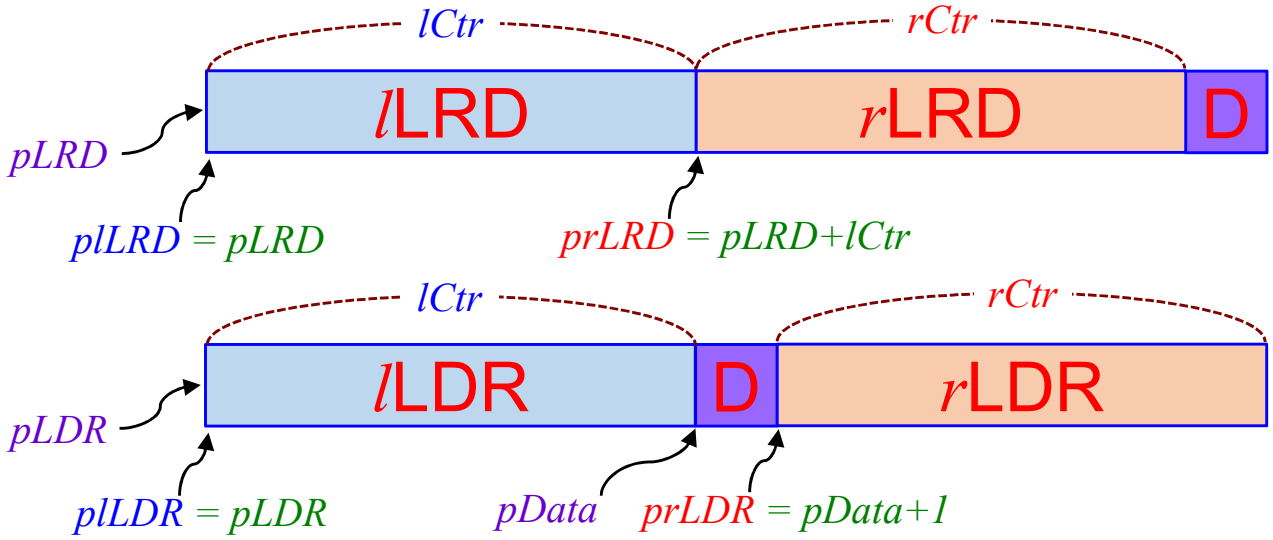
➢전위 순회 – 중위 순회

```
pData = strchr(pLDR, pDLR[0]);
lCtr = pData - pLDR;
rCtr = nCtr – lCtr - 1;
plDLR = pDLR + 1;
prDLR = plDLR + lCtr;
plLDR = pLDR;
prLDR = pData + 1;
```



➢후위 순회 – 중위 순회

```
pData = strchr(pLDR, pLRD[nCtr - 1]);
lCtr = pData - pLDR;
rCtr = nCtr – lCtr - 1;
plLRD = pLRD;
prLRD = plLRD + lCtr;
plLDR = pLDR;
prLDR = pData + 1;
```



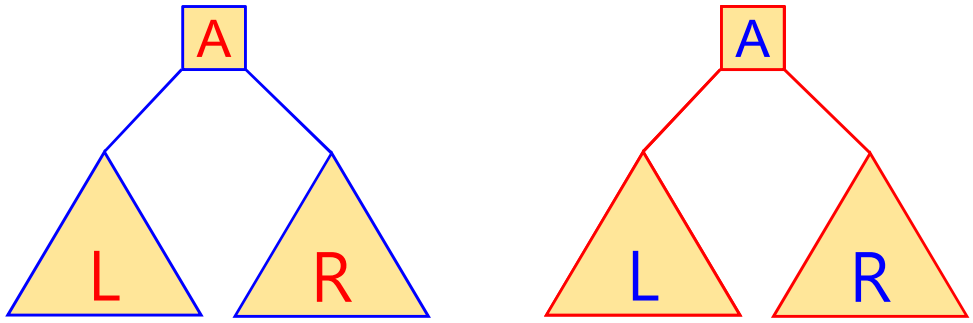
이진 트리 연산자

Numbering.cpp

CountNode.cpp

➢이진 트리 복사

```
TreePtr copy(TreePtr pTree)
{
      TreePtr pNew = NULL;
      if (pTree) {
            pNew = new Tree;
            if (pNew) {
                  pNew->nData = pTree->nData;
                  pNew->lChild = copy(pTree->lChild);
                  pNew->rChild = copy(pTree->rChild);
            }
      }
      return pNew;
}
```



➢이진 트리가 같은가?

```
int equal(TreePtr t1, TreePtr t2)
{
      if (t1 == NULL && t2 == NULL)
            return true;   // 둘 다 널
      if (t1 == NULL || t2 == NULL)
            return false;  // 하나는 널, 하나는 널 아님
      // 둘 다 널 아님
      return t1->nData == t2->nData &&
            equal(t1->lChild, t2->lChild) &&
            equal(t1->rChild, t2->rChild);
}
```

➢재귀함수

- ➢일반 규칙:
  - ➢루트의 데이터가 서로 같고,
  - ➢왼쪽 서브트리끼리 같고(순환),
  - ➢오른쪽 서브트리마저 같다(순환).
- ➢진행 방향: 높이가 줄어드는 방향
- ➢중단 조건: 높이가 0
  - ➢둘 다 0: 같음
  - ➢하나만 0: 다름

스레드 이진 트리

- 널 링크
  - 실제 포인터 보다 널 링크가 더 많다.
    - 노드 수가 n이면 총 링크는 2n, 실제 포인터는 n-1, 널 링크는 n+1
  - 널 링크를 효과적으로 사용하기 위하여 스레드로 대치
    - 스레드: 원래는 널인데, 널 대신 다른 노드를 가리킨다.
      - lChild: 중위 순회에서 앞 노드
      - rChild: 중위 순회에서 뒤 노드
    - 스레드인지 실제 링크인지 구별하도록 한 비트를 추가
      - pNode->blThread == false
        - pNode->lChild는 왼쪽 자식을 가리킴
      - pNode->blThread == true
        - pNode->lChild는 스레드이고
        - pNode->lChild는 중위 순회에서 앞 노드를 가리킴
      - pNode->brThread == false
        - pNode->rChild는 오른쪽 자식을 가리킴
      - pNode->brThread == true
        - pNode->rChild는 스레드이고
        - pNode->rChild는 중위 순회에서 뒤 노드를 가리킴

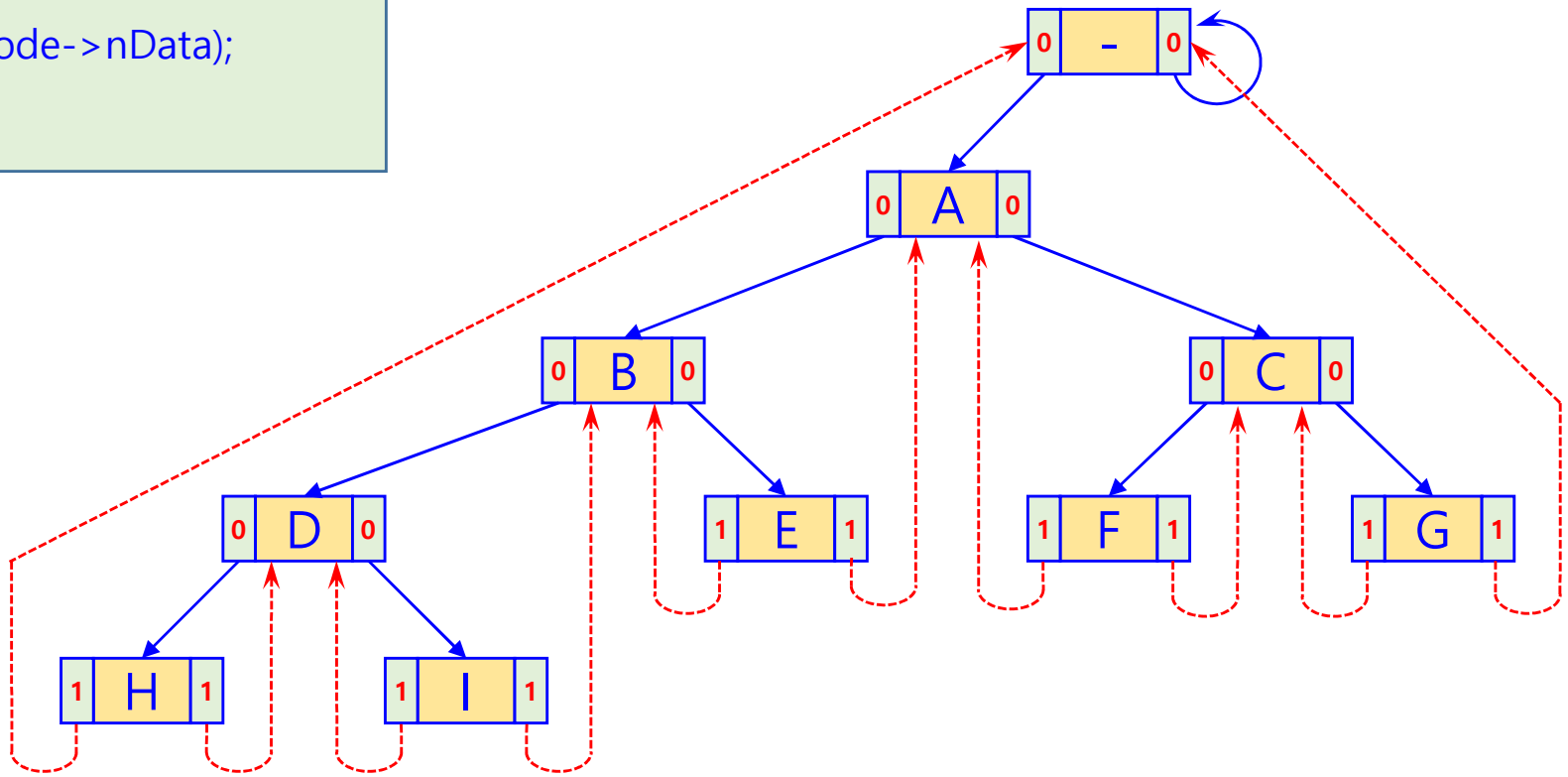
```
typedef struct node {
        int             nData;
        short int       blThread;
        short int       brThread;
        struct node   *lChild;
        struct node   *rChild;
}       Node, *NodePtr, Tree, *TreePtr;
```

스레드 이진 트리

➢스레드 이진 트리의 중위 순회

```
void threadInorder(TreePtr pHead)
{
    TreePtr pNode = pHead;
    while (1) {
        pNode = successor(pNode);
        if (pNode == pHead)
            break;
        printf("%c", pNode->nData);
    }
}
```

```
TreePtr successor(TreePtr pTree)
{
    TreePtr pNode = pTree->rChild;
    if (pTree->brThread == false)
        while (pNode->blThread == false)
            pNode = pNode->lChild;
    return pNode;
}
```



Heap

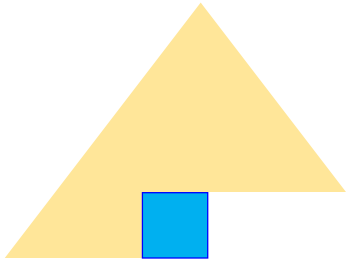
- 최대 트리(Max Tree)
  - 루트의 데이터가 자식들의 데이터보다 크거나 같다.
  - 루트가 가장 큰 데이터를 가지고 있다.
- 최소 트리(Min Tree)
  - 루트의 데이터가 자식들의 데이터보다 작거나 같다.
  - 루트가 가장 작은 데이터를 가지고 있다.
- 최대 heap
  - 최대 트리
  - 완전 이진 트리
- 최소 heap
  - 최소 트리
  - 완전 이진 트리
- heap의 표현
  - 완전 이진 트리이기 때문에 배열로 표현

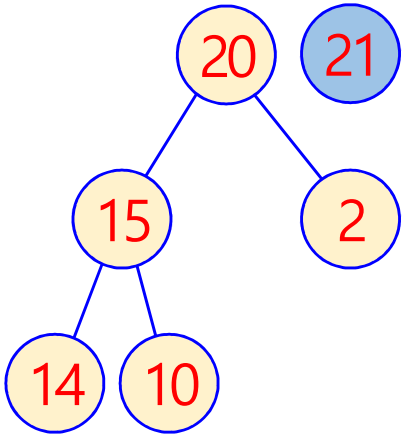
최대 Heap 삽입

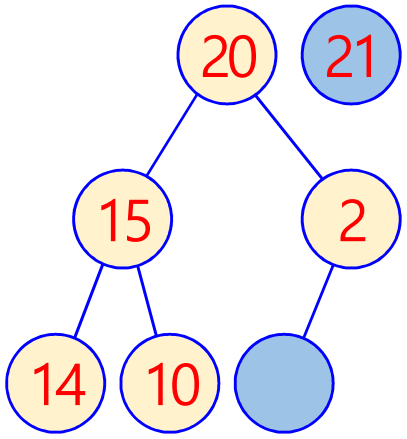
- 데이터 삽입
  - 가장 바닥 레벨에서 빈 공간의 가장 왼쪽에 삽입
    - 완전 이진 트리는 만족
    - 최대 트리는 아님
  - 데이터 조정

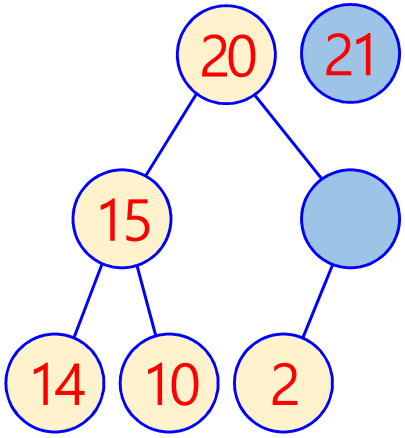
    while (부모의 데이터 < 삽입된 데이터)

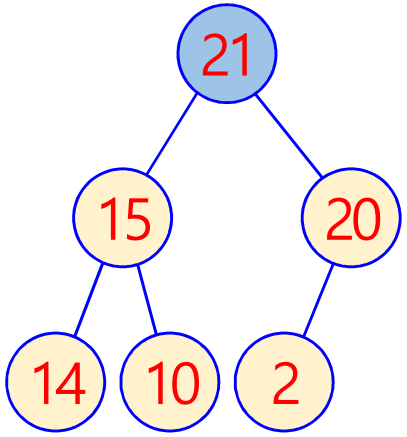
    서로 바꾼다.









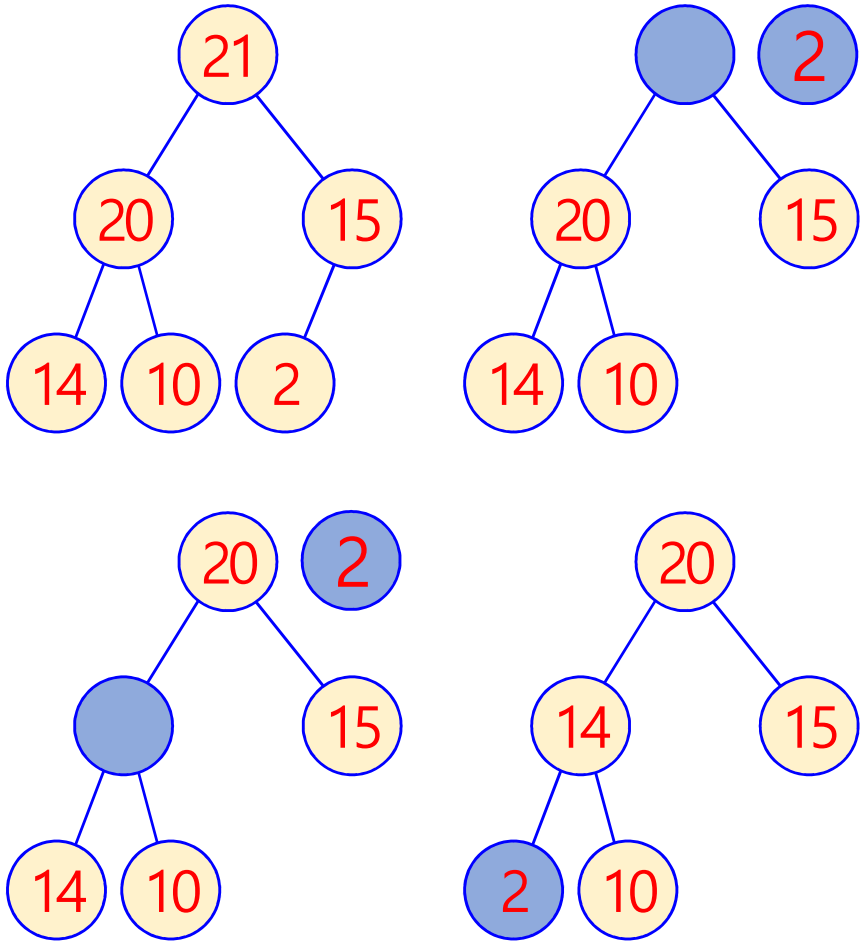



```
void InsertMaxHeap(int nData, int& n)
{
	if (n >= HeapSize)
		return;
	int nNdx = ++n;
	while (nNdx != 1 && nData > heap[nNdx / 2]) {
		heap[nNdx] = heap[nNdx / 2];
		nNdx /= 2;
	}
	heap[nNdx] = nData;
}
```

최대 Heap 삭제

- 데이터 삭제
  - 루트의 데이터를 삭제
  - 마지막 노드의 데이터를 루트로 이동
  - 자식보다 작으면 서로 교환
  - 만족할 때까지 반복

```
Item DeleteMaxHeap(int& n)
{
      Item item = -1;   // 비정상 값
      if (n) {
            item = heap[1];
            Item temp = heap[n--];
            int parent = 1, child = 2;
            while (child <= n) {
                  if (child < n && heap[child] < heap[child + 1])
                        child++;
                  if (temp >= heap[child])
                        break;
                  heap[parent] = heap[child];
                  parent = child;
                  child *= 2;
            }
            heap[parent] = temp;
      }
      return item;
}
```



이진 탐색 트리

SearchTree.cpp

- 이진 탐색 트리
  - 모든 데이터는 다르다.
  - 왼쪽 서브트리의 데이터는 루트 보다 작다.
  - 오른쪽 서브트리의 데이터는 루터 보다 크다.
  - 두 서브트리는 이진 탐색 트리이다.
- 탐색
  - 순환 함수

```
Tree *search(Tree *pRoot, int nData)
{     /* nData를 가지는 노드를 반환(없으면 NULL) */
      if (pRoot == NULL)
            return NULL;
      if (nData == pRoot->nData)
            return pRoot;
      if (nData < pRoot->nData)
            return search(pRoot->lChild, nData);
      return search(pRoot->rChild, nData);
}
```

  - 반복 함수

```
Tree *iterSearch(Tree *pTree, int nData)
{
      while (pTree) {
            if (nData == pTree->nData)
                  return pTree;
            pTree = (nData < pTree->nData)
                  ? pTree->lChild : pTree->rChild;
      }
      return NULL;
}
```

이진 탐색 트리 삽입

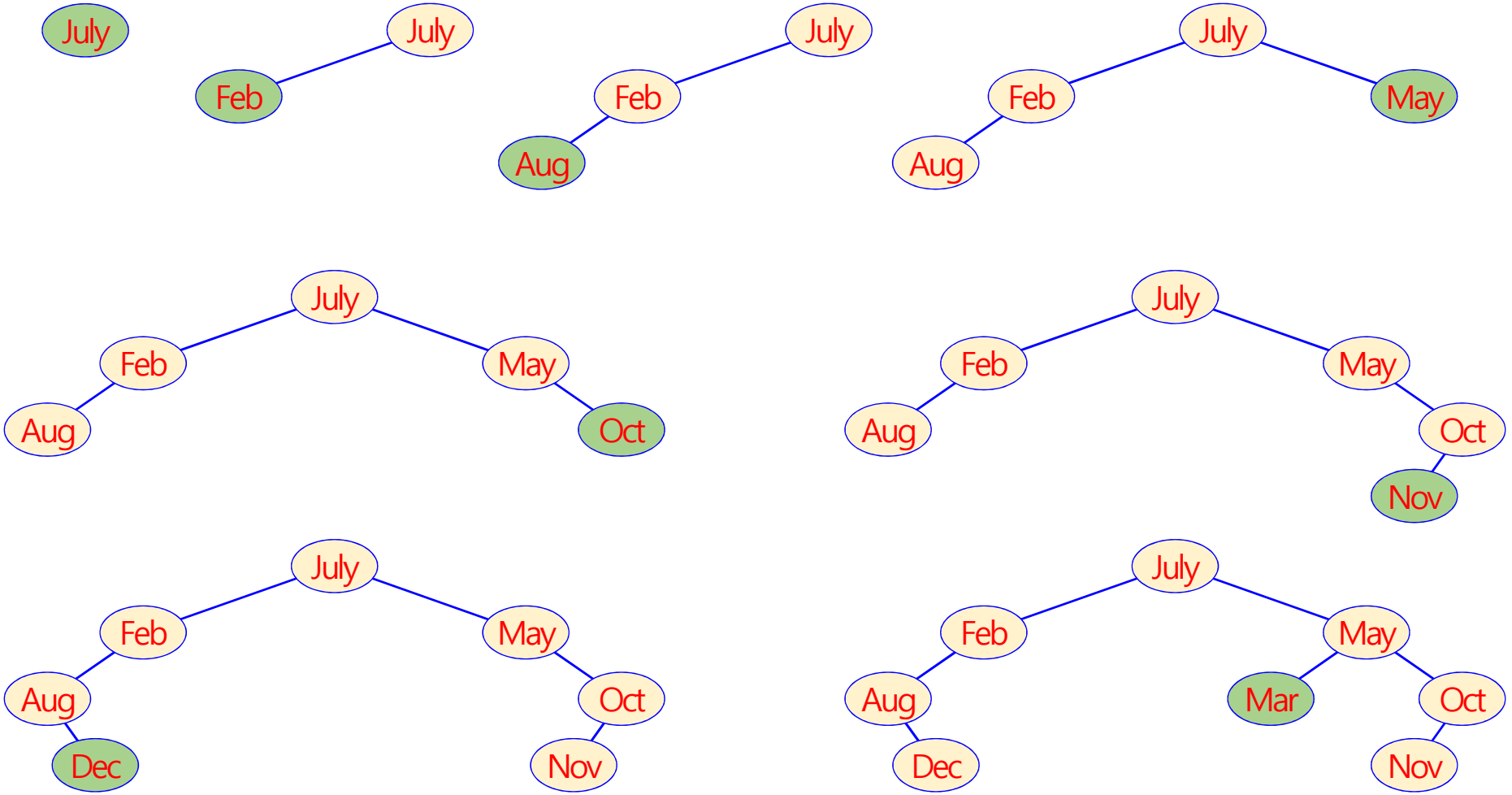
- 삽입
  - 루트에서 시작하여 삽입되는 데이터가
    - 작으면 왼쪽 자식에 재귀적으로 호출
    - 같으면 이미 삽입되어 있어 그냥 return;
    - 크면 오른쪽 자식에 재귀적으로 호출
  - 트리 높이가 작아지는 방향으로 진행
    - 중단 조건: 높이가 0
    - 단순 해답: 데이터를 가지는 노드 생성

```
TreePtr insert(TreePtr pNode, int nData)
{
	if (pNode == NULL) {
		pNode  = new Tree;
		if (pNode) {
			pNode->nData = nData;
			pNode->lChild = pNode->rChild = NULL;
		}
	}
	else if (nData < pNode->nData)
		pNode->lChild = insert(pNode->lChild, nData);
	else if (nData > pNode->nData)
		pNode->rChild = insert(pNode->rChild, nData);
	return pNode;
}
```

```
void insert(TreePtr& pNode, int nData)
{
	if (pNode == NULL) {
		pNode  = new Tree;
		if (pNode) {
			pNode->nData = nData;
			pNode->lChild = NULL;
			pNode->rChild = NULL;
		}
	}
	else if (nData < pNode->nData)
		insert(pNode->lChild, nData);
	else if (nData > pNode->nData)
		insert(pNode->rChild, nData);
}
```

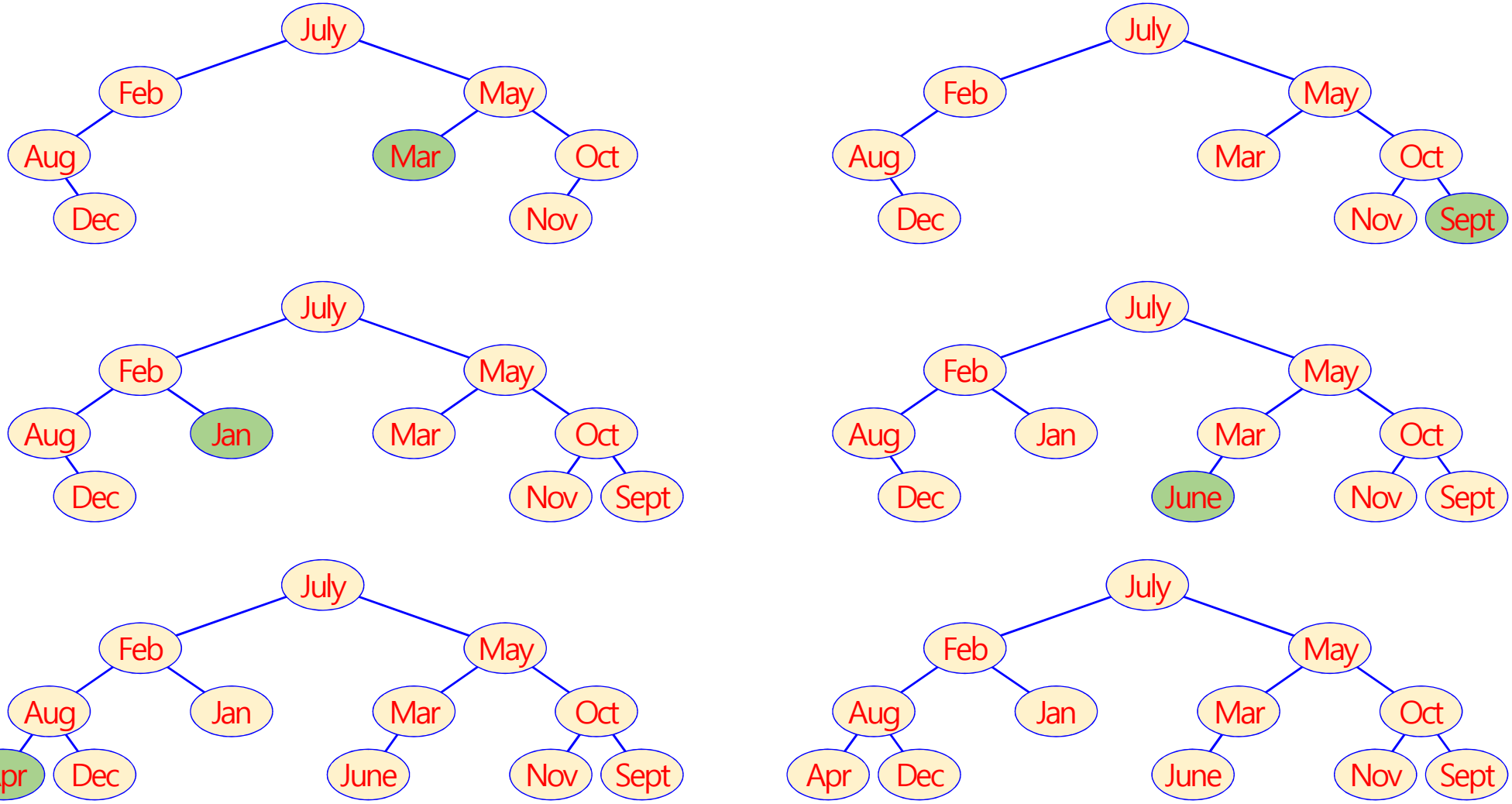
이진 탐색 트리 삽입

- 탐색 트리에 노드 삽입
  - July Feb Aug May Oct Nov Dec Mar Sept Jan June Apr 순서로 삽입



이진 탐색 트리 삽입

- 탐색 트리에 노드 삽입
  - July Feb Aug May Oct Nov Dec Mar Sept Jan June Apr 순서로 삽입



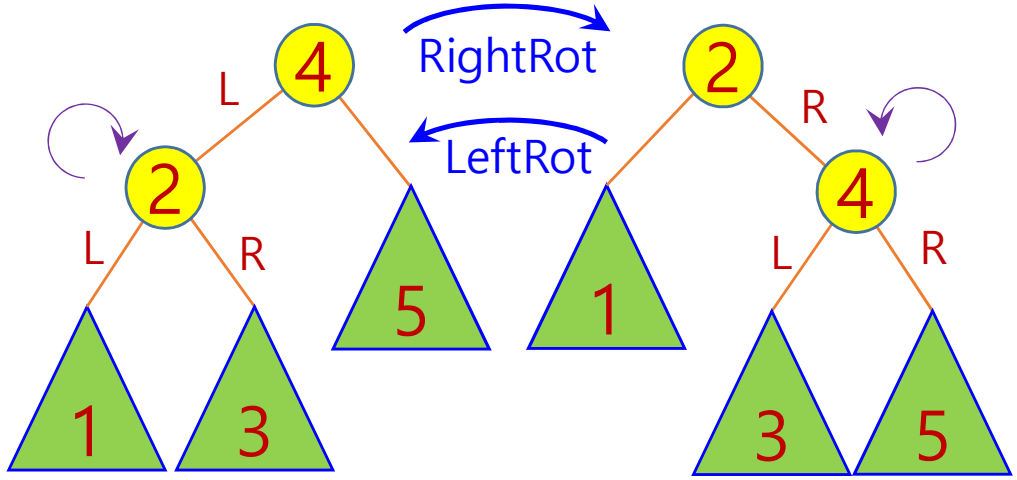
이진 탐색 트리

- 이진 탐색 트리의 효율성
  - 트리의 높이는 입력되는 데이터의 특성에 따라 결정됨
    - 입력되는 데이터가 오름(내림)차순이면 경사 트리를 형성
    - 루트의 데이터가 상대적으로 작으면 왼쪽 서브트리가 작아지고 오른쪽은 커짐
    - 루트의 데이터가 상대적으로 크면 왼쪽 서브트리는 커지고 오른쪽은 작아짐
  - 탐색의 효율성은 트리의 높이에 좌우(노드 수가 n인 경우)
    - 이상적인 경우: 한 레벨 내려가면 노드 수가 n/2으로 감소
      - 완전 이진 트리이면 가장 이상적
      - 포화 이진 트리 $n = 2^h - 1$, $h \approx \log_2 n$
    - 최악의 경우: 연결 리스트처럼 한쪽으로 경사지면 노드 수가 n-1으로 감소
      - 경사 트리이면 최악의 상황
    - 평균적으로 $O(\log_2 n)$, 최악의 경우 $O(n)$
- AVL 트리(Georgy Adelson-Velskii, Evgenii M. Landis)
  - 데이터의 입력에 상관없이 양쪽 서브트리의 균형을 맞춤
    - 삽입 후 양쪽 높이 차가 2가 되면 루트를 조정: LL, LR, RL, RR 회전
  - 노드 개수가 n인 AVL 트리의 높이는 항상 $O(\log_2 n)$을 보장

AVL 트리

- 정의
  - 이진 탐색 트리
  - 양쪽 트리의 높이 차이가 -1 0 1
- 회전하여 스스로 높이 차이를 조정
  - 단일 회전(single rotation): LL, RR
  - 이중 회전(double rotation): LR, RL
- 회전
  - LL
    - rightRotation(parent)
  - RR
    - leftRotation(parent)
  - LR
    - leftRotation(lChild), rightRotation(parent)
  - RL
    - rightRotation(rChild), leftRotation(parent)
- Visual Insertion:

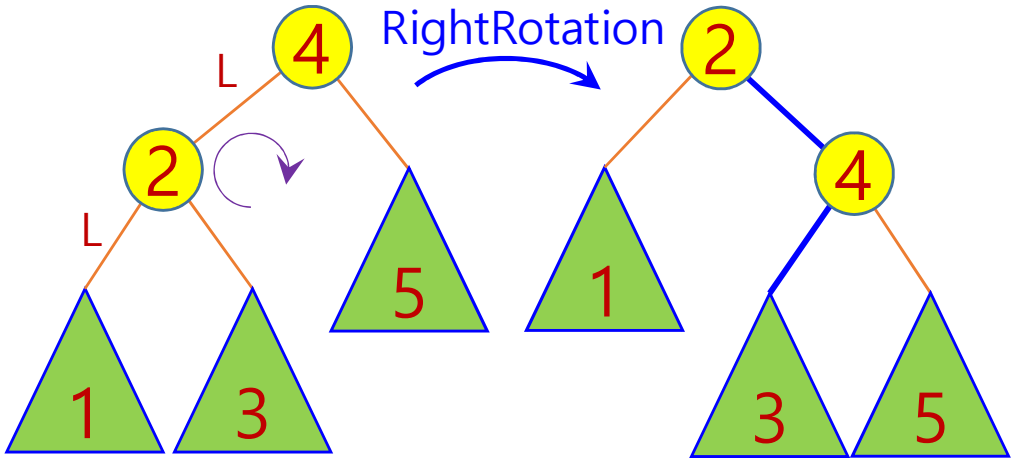
  https://www.cs.usfca.edu/~galles/visualization/AVLtree.html




```
TreePtr AVLInsert(pTree, nData)
{
        pTree = BSTInsert(pTree, nData);    // 탐색 트리 삽입
        if (좌우 트리의 높이 차이가 1 이상) {
                if (LL type)
                        rightRotation(pTree);
                else if (RR type)
                        leftRotation(pTree);
                else if (LR type) {
                        leftRotation(pTree->lChild);
                        rightRotation(pTree);
                }
                else {          // RL type
                        rightRotation(pTree->rChild);
                        leftRotation(pTree);
                }
        }
        return pTree;
}
```

rightRotation, leftRotation

- 두 함수는 서로 대칭
  - 오른쪽 ←→왼쪽
- 이진 탐색트리
  - 순서: 1, 2, 3, 4, 5
  - LL Rotation(함수 RightRotation)
    - 4가 너무 커서 왼쪽으로 쏠림
    - 2를 root로 세워 균형 트리로 변형
  - RR Rotation(함수 LeftRotation)
    - 2가 너무 작아서 왼쪽으로 쏠림
    - 4를 root로 세워 균형 트리로 변형

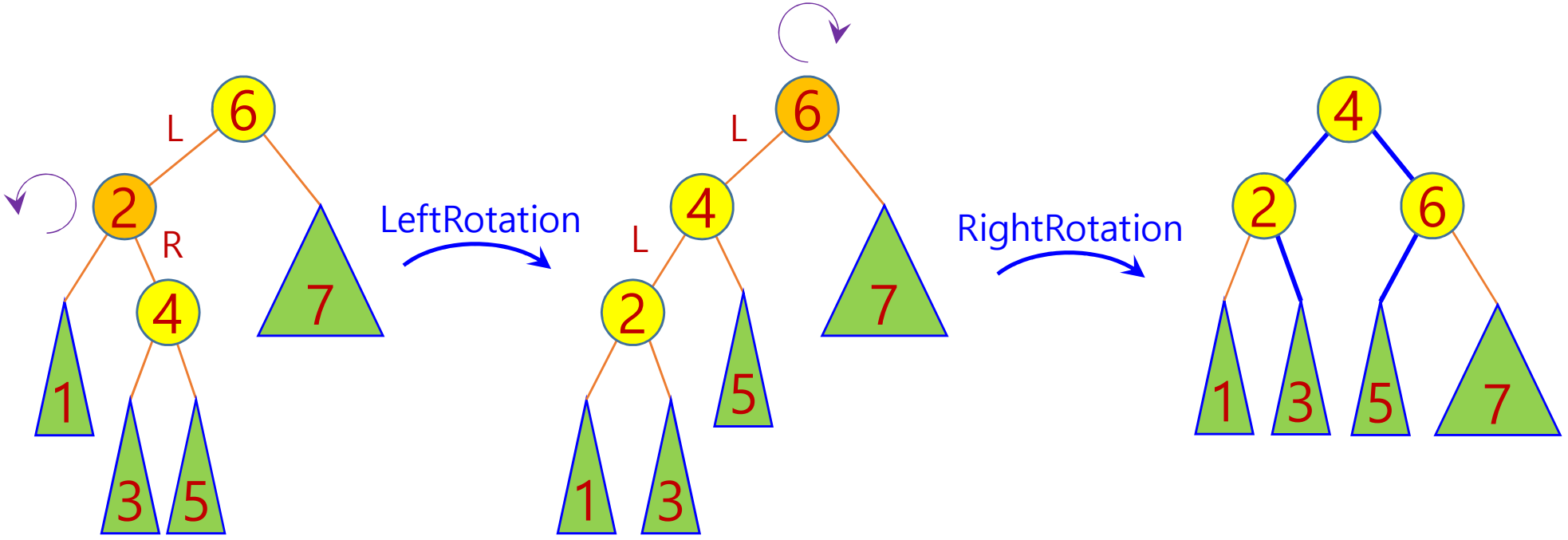



```
TreePtr RightRotation(TreePtr pTree)
{
    TreePtr pNewRt = pTree->lChild;
    pTree->lChild = pNewRt->rChild;
    pNewRt->rChild = pTree;
    return pNwRt;
}
```

```
TreePtr LeftRotation(TreePtr pTree)
{
    TreePtr pNwRt = pTree->rChild;
    pTree->rChild = pNewRt->lChild;
    pNewRt->lChild = pTree;
    return pNwRt;
}
```

Double Rotation

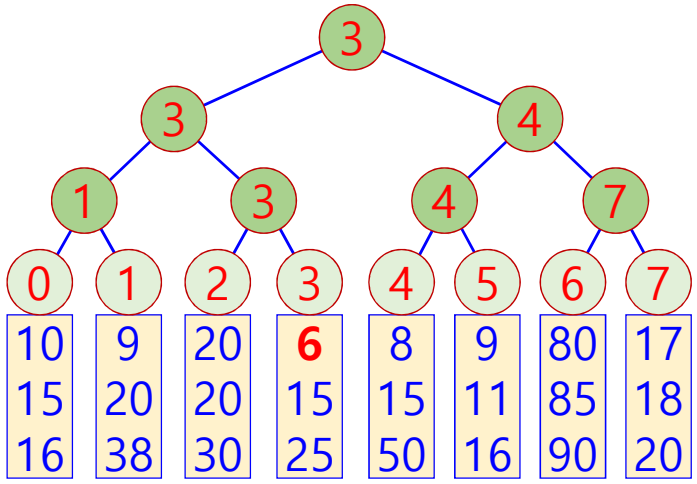
- LR Rotation
  - LeftRotation for left child
  - RightRotation for parent
- RL Rotation
  - RightRotation for right child
  - LeftRotation for parent

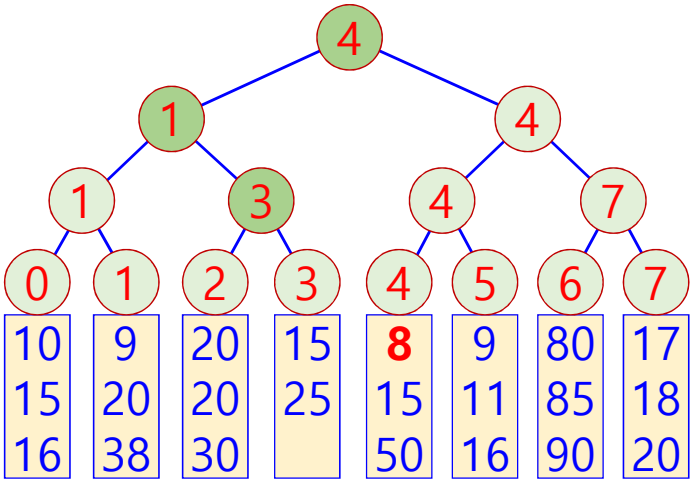


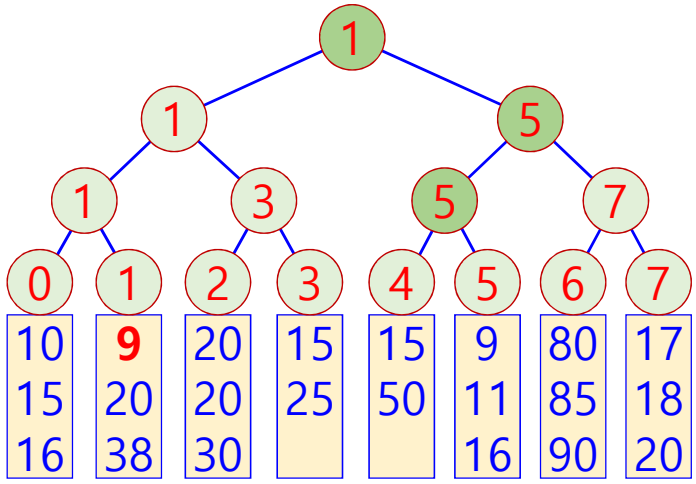
선택 트리(Selection Tree)

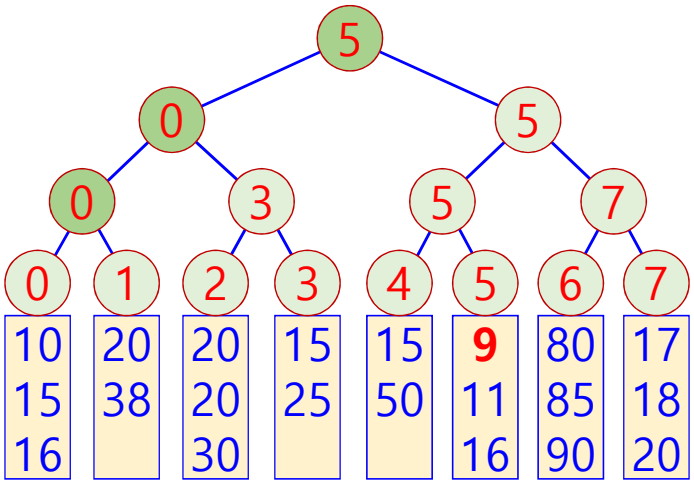
- 선택 트리
  - 부모 노드는 두 자식 노드 중 승자(오름차순/내림차순)를 올림
  - 토너먼트 경기
  - 최소값을 찾는 경우
    - 부모 노드는 두 자식 노드 중 작은 자식을 올림
    - 최종적으로 루트는 최소 값을 가지는 노드가 올라 옴
- 병합
  - 런(run): 일부분이지만 정렬된 데이터의 집합
  - 2 병합: 16M = 2^{24}, 24번
  - 4 병합: 16M = 4^{12}, 12번
  - 8 병합: 16M = 8^{8}, 8번
  - 16 병합: 16M = 16^{6}, 6번
  - k 개의 런에서 최소값을 찾아 병합된 런에 출력
    - 최소값을 찾기 위하여 k-1번 비교
    - 효과적으로 최소값을 찾기 위하여 선택 트리를 사용
  - 8 런 → 1 런으로 병합: 8 개의 값의 토너먼트 경기에서 최종 승자를 찾음

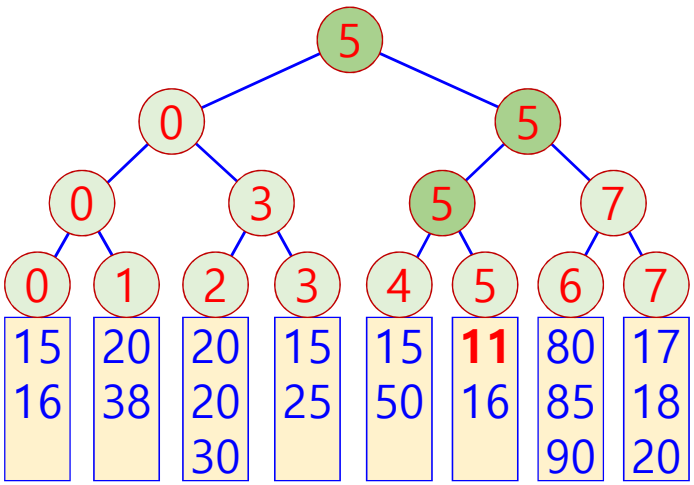
선택 트리에 의한 병합

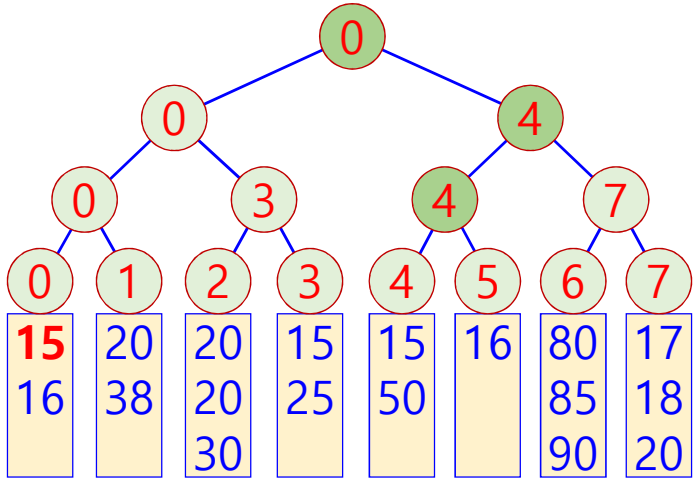

3
3 4
1 3 4 7
0 1 2 3 4 5 6 7
10 9 20 6 8 9 80 17
15 20 20 15 15 11 85 18
16 38 30 25 50 16 90 20


4
1 4
1 3 4 7
0 1 2 3 4 5 6 7
10 9 20 15 8 9 80 17
15 20 20 25 15 11 85 18
16 38 30 50 16 90 20


1
1 5
1 3 5 7
0 1 2 3 4 5 6 7
10 9 20 15 15 9 80 17
15 20 20 25 50 11 85 18
16 38 30 16 90 20


5
0 5
0 3 5 7
0 1 2 3 4 5 6 7
10 20 20 15 15 9 80 17
15 38 20 25 50 11 85 18
16 30 16 90 20


5
0 5
0 3 5 7
0 1 2 3 4 5 6 7
15 20 20 15 15 11 80 17
16 38 20 25 50 16 85 18
30 90 20


0
0 4
0 3 4 7
0 1 2 3 4 5 6 7
15 20 20 15 15 16 80 17
16 38 20 25 50 85 18
30 90 20

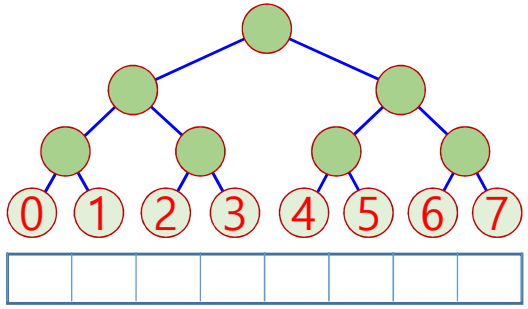
병합

- 초기 선택 트리 생성
  - k-1번 비교
- 선택된 데이터 출력 후 선택 트리 재조정
  - 생성된 선택 트리를 재조정하여 효과적으로 최소값 결정
  - $\log_2 k$번 비교 후 최소값 결정
  - 8개를 병합
    - 선택 트리 사용하지 않을 경우: 7번
    - 선택 트리를 사용할 경우: 3번
  - 런에 데이터가 n이면
    - 선택 트리 사용하지 않을 경우
      - $n(k-1)$ $7n(k=8)$
    - 선택 트리 사용
      - $n \cdot \log_2 k$ $3n(k=8)$
- 선택 트리 재조정: $O(\log_2 k)$
- n개 데이터 병합: $O(n \cdot \log_2 k)$

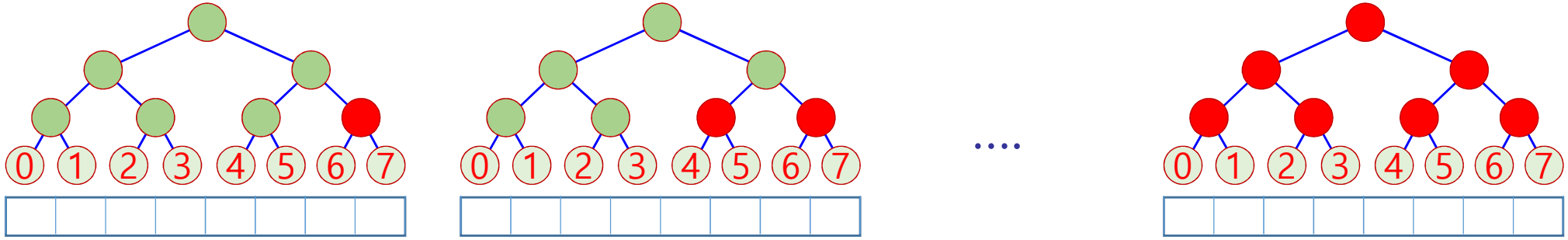
Section Tree 실습

SelectionTree.cpp

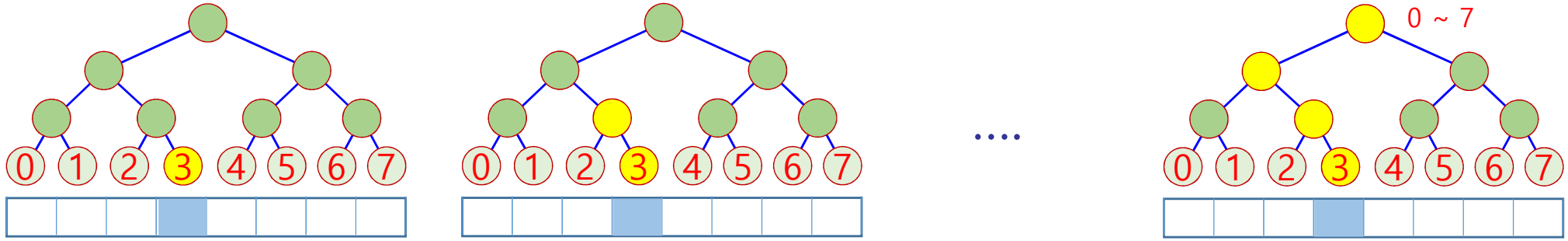
- SetupLeafNode: 노드의 데이터는 런의 색인



- FirstSelectonTree



- RebuildSelectionTree



Section Tree Example

```
                                                10
                                07                                              10
                        02                              07                              10                              13
                00              02              05              07              09              10              13              14
            00      01      02      03      04      05      06      07      08      09      10      11      12      13      14      15

          [00]    [01]    [02]    [03]    [04]    [05]    [06]    [07]    [08]    [09]    [10]    [11]    [12]    [13]    [14]    [15]
          2436    3926    2129    3926    2870    2457    2843    1574    4738    2793    1360    2198    2089    1841    3395    3579

Winner[10]: 1360, New data[10]: 3237

                                                07
                                07                                              13
                        02                              07                              11                              13
                00              02              05              07              09              11              13              14
            00      01      02      03      04      05      06      07      08      09      10      11      12      13      14      15

          [00]    [01]    [02]    [03]    [04]    [05]    [06]    [07]    [08]    [09]    [10]    [11]    [12]    [13]    [14]    [15]
          2436    3926    2129    3926    2870    2457    2843    1574    4738    2793    3237    2198    2089    1841    3395    3579

  Winner[07]: 1574
```

Q & A