<자료구조 기말 강의자료 요약>

CH5-1. TREES

* 기본적인 tree terms *

- height : 세대

 = depth = number of levels

- node degree : 자녀 수

 = # of children

- tree degree : 부모 노드가 가진 자녀의 수 중 최댓값

 = max node degree

<conditions>

- tree : a finite set of one or more nodes (empty X)

- leaf (terminal) : node degree = 0 (child X)

<Binary Tree>

- def) a finite set of nodes that is either empty or consists of a root and two disjoint binary trees (left / right subtree)

- empty 가능 (nonempty의 경우, root 존재)

- full binary tree의 k번째 깊이의 node의 개수 => 2^k - 1 nodes

=> array representations

- complete binary tree with n nodes => depth(height) <= $\log_2 n + 1$

- parent의 index = i 일 경우, leftChild는 2i번째, rightChild는 2i+1번째에 존재

(단, 2i < n , 2i+1 < n일 때)

- node structure (linked list ver.)

```
typedef struct node *tree_pointer;

typedef struct node {

        int data; tree_pointer left_child, right_child;

}
```

<Binary Tree Traversal>

1) preorder => V L R

```
void preorder (treePointer ptr){
	if (ptr){
		visit(ptr);				=>  V
		preorder(ptr->leftchild);		=>  L
		preorder (ptr->rightchild);		=>  R
	}
}   **** visit = print ****
```

2) inorder => L V R

3) postorder = > L R V

4) level order => FIFO queue

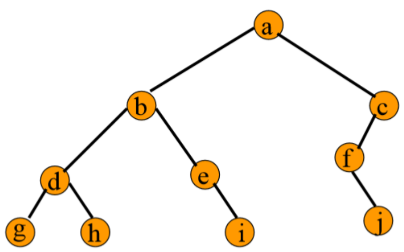
```
void level_order(treePointer ptr){
	int front=rear=0;
	treePointer queue[MAX_SIZE];
	if(!ptr) return;
	addq(front,&rear, ptr);
	for (;;){
		ptr = deleteq(&front,rear);
		if(ptr) {
			printf("%d",ptr->data);
			if(ptr->leftChild) addq(front,&rear,ptr->leftchild);
			if(ptr->rightChild) addq(front,&rear,ptr->rightchild);
		}
		else break;
	}// for문 fin. }
```

5) Iterative Inorder => LIFO stack

* left node가 null을 만날때까지 stack에 push -> stack pop -> right node가 push

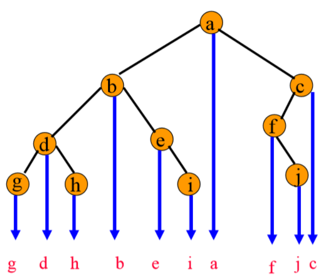
```
void iter_inorder (treePointer node){
        treePointer stack[MAX_SIZE];
        for (;;){
                for (;node;node= node->leftChild)   push(&top,node);
                node = pop(&top);
                if(!node) break;
                printf("%d",node->data);
                node = node->rightChild; }// outer for문 fin.
}
```

** 출력 순서



1) preorder : a b d g h e i c f j

2) inorder : g d h b e i a f j c => projection (squishing)



3) postorder : g h d i e b j f c a

4) level order : a b c d e f g h i j

<Binary Search Tree>

- (key, value)로 이루어짐

- x를 가진 노드의 left subtree => x보다 작은 값들

- x를 가진 노드의 right subtree => x보다 큰 값들

<Time complexity of BST>

1) ascending operation => inorder traversal : O(n)

2) searching operation : O(height)

```
① treePointer search (treePointer root, int key){
        if (!root) return NULL:
        if (key == root->data) return root;
        if (key < root -> data) return search (root->leftChild, key);
        return search (root->rightChild , key);
} // recursive
```

```
② treePointer search2 (treePointer tree, int key) {
        while (tree) {
                if (key == tree->data) return tree;
                if (key < tree-> data) tree = tree->leftChild;
                else tree = tree->rightChild;
                return NULL;
        }
} // iterative
```

3) Inserting operation : O(height)

```
 void insert_node (treePointer *node, int num){
        treePointer ptr, temp = modified_search(*node, num);
        // modified_search : empty/해당 숫자가 존재=> NULL
                                      나머지 경우 => tree의 last node
```

```
        if (temp || !(*node)) {
            ptr = (treePointer)malloc(sizeof(node));
            ptr->data = num;
            ptr->leftChild = ptr->rightChild = NULL;
            if (*node) {
                        if (num<temp->data) temp->leftChild = ptr;
                        else temp->rightChild = ptr; }
            else *node = ptr;
        }
}
```

4) Deleting operation

4-1) element가 없을 때

4-2) 자식이 없는 노드 (leaf)

4-3) 1개의 자식이 있는 노드 (degree 1)

=> 그냥 지워버리고 남은 parent와 child를 연결시켜준다.

★4-4) 2개의 자식이 있는 노드 (degree 2)

=> 해당 노드를 left subtree의 largest key나, right subtree의 smallest key로 대체

(위의 노드들은 항상 degree 0 이거나 1)

대체하고자 하는 대상의 원래 노드를 삭제

Time Complexity : O(height)

- Height of a binary search tree => $\log_2 n$

<Winner Tree>

1) Min winner Tree => smaller element wins

height = $\log_2 n$

O(1) : playing match at each match node (n-1 개의 match nodes)

O(n) : initialize n player winner tree

- sorting할 때의 min winner tree 사용하기 -> sorted array 이용

1) initialize n player winner tree : O(n)

2) remove winner & replay : O(log n)

3) repeat 2번 for n times : O(nlogn)

<Loser Tree>

- left child winner를 저장한다 => O(n)

- replay matches => O(log n)

<Forest>

- find(i) : i를 원소로 가지는 tree의 root에 있는 element를 리턴

-> table[i]에서 시작해서 root node에 있는 element를 리턴

- simple union => parent[i] = j; // make one tree a subtree of the other

- simple find => while (parent[i] >= 0)   i = parent[i]; // move up the tree

* union

 -> height rule : height가 더 작은 tree가 subtree가 된다.

 -> weight rule : node의 개수가 더 작은 tree가 subtree가 된다.

- root노드의 parent은 -1이다.

```
* void weightedUnion (int i, int j){
        int temp = parent[i] + parent[j];
        if (parent[i] > parent[j]) {
                parent[i] = j; parent[j] = temp;
        } // j가 새로운 root
        else {
                parent[j] = i; parent[i] = temp;
        } // i가 새로운 root
} // 위로 갈수록 값이 작아지는 tree
```

O(n + m $\log_2 n$) -> n-1 union , m find operations

<CH 5.4-5.6>

- Binaty Tree의 equality 확인하기

1) !first && !second

2) first && second && first->data == second->data && equal (first->leftChild,second->leftChild) && equal (first->rightChild && second->rightChild)

<Threaded Binary Trees>

Thread => left : Inorder Predecessor / right : Inorder Succesor

thread 0   == Child 존재 / thread 1 == Ancestor

* Inorder Successor

```
threaded_pointer insucc(threaded_pointer tree){

        threaded_pointer temp;

        temp = tree->right_child;

        if(! tree->rightthread) while(!temp->left_thread) temp= temp->left_child;

        return temp;

}
```

* InsertRight in Threaded Binary Tree

```
        child->rightChild = parent->rightChild;
        child->rightThread = parent->rightThread;
        child->leftChild = parent;
        child->leftThread = 1;
        parent->rightChild = child;
        parent->rightThread = 0;
        if (!child->rightThread){
            temp = insucc(child);
            temp->leftChild = child;
        }
```

<Insertion into a max heap> => O(log2n)

```
void insert_max_heap(element item, int *n) {
/* insert item into a max heap of current size *n */
int i = ++(*n);
while ((i != 1) && (item.key > heap[i/2].key)) { heap[i] = heap[i/2]; i /= 2; }
heap[i] = item; }
```

<Deletion in a max heap> => O(log2n)

```
element delete_max_heap(int *n){
/* save value of the element with the largest key */
item = heap[1];
 /* use last element in heap to adjust heap  */
 temp = heap[(*n)--]; parent = 1;    child = 2;
while (child <= *n)  {  /* find the larger child of the current parent  */
  if ((child < *n) &&(heap[child].key< heap[child+1].key)) child++;
  if (temp.key >= heap[child].key)    break;
  /* move to the next lower level  */
  heap[parent] = heap[child]; parent = child; child *= 2; }
heap[parent] = temp; return  item;
}
```