

# 자료구조 응용 과제-기말고사 과제 정리

2021113772 이수민

앞부분-실습 문제 중 그려야하는 것

뒷부분-단원별로 분류해서 문제 풀이에 이용한 수업시간에 배운 함수나 제가 만든

함수에 주석 달면서 정리해놓은 것

순서로 정리해두었습니다.

# Tree

## 1. max heap

```
void push(element item, int* n) //n은 heap 내 원소의 수
{
    int i;
    if (HEAP_FULL(*n)) {
        fprintf(stderr, "The heap is full.\n");
        exit(EXIT_FAILURE);
    }
    i = ++(*n); //인덱스는 1부터 시작
    //n에 1을 더함->i는 원소의 수+1
    // \\\새로 들어오는 원소를 넣을 빈 공간까지 확보한 상태의 노드 수가 i

    //i가 1이면 n은 0이었다->힙 안에 들어있는 게 없었다
    //->그냥 바로 heap[1]=item(whileX)

    //인덱스를 2로 나누면(i/2) 부모노드를 가리킴
    //heap[i/2]와 넣으려는 값(item) 비교해서
    //\\\maxheap은 클수록 위어가니까
    //넣으려는 값이 부모노드보다 더 크면
    //원래 부모노드에 있던 값을 밑에 새로 생긴 노드로 밀고
    //item값을 비교하던 부모노드의 부모노드와 비교(i/=2)
    //그 후 제자리를 찾을 때까지 그 위의 부모노드와 또 비교
    while ((i != 1) && (item.key > heap[i / 2].key)) {
        heap[i] = heap[i / 2]; //부모노드를 아래로 내리고
        i /= 2; //복합대입연산자->원본값(i) 수정됨 -> 계속 위로 올라가기
    }
    //item값이 부모노드보다 더 작으면 바로 item 넣기
    heap[i] = item; //조정된 i 위치에 값을 넣는다
}
```

**element pop(int\* n)**

```
{
    int parent, child;
    element item, temp;
    if (HEAP_EMPTY(*n)) {
        fprintf(stderr, "The heap is empty.\n");
        exit(EXIT_FAILURE);
    }
    //item에 루트 노드, temp에 제일 끝의 노드 넣기

    item = heap[1]; //제일 위의(root) 값 item에 넣기(빼내기)
    temp = heap[*n--]; //제일 마지막 노드 위치 이동
    //후위연산->처음에는 n값을 그대로 하고 그 다음에 빼기
    //temp에 마지막 노드의 값을 넣은 후 원소의 수(*n)을 1 빼기
}
```

```

parent = 1; /\빈 공간이 항상 parent
child = 2; //child+1=그 옆의 자식노드
//root 를 빼주면 그 빈자리를 메우고, 다시 또 그 빈자리를 메우면서
// 재배치하기 위해 while문 돌림

//자식 둘 중 더 큰 것 위로 올리기
while (child <= *n) {
    //child와 그 옆의 자식(형제노드)와 temp를 비교
    //if문에서 child<*n 이유는 child==*n이면 오른쪽 (형제)자식노드 없이
    //그 child가 트리의 마지막 노드->형제끼리 비교할 필요 없음
    //->첫번째 if문 그냥 넘어가기
    if ((child < *n) && (heap[child].key < heap[child + 1].key))
        child++; //자식이 두명 있을 때&& 둘 중 더 큰 자식이 child
    if (temp.key >= heap[child].key) break;
    //temp의 값보다 child의 값이 더 크면 child를 위로 올리기
    //제일 첫 단계에서는 child의 값이 temp보다 크겠지만
    //계속 진행하다 보면 temp보다 작은 child가 생김
    heap[parent] = heap[child];
    parent = child; //원래있던 child를 그 위의 부모 자리로 넣어서
    //child자리가 빈 자리니까 거기를 parent로 재설정해서
    //처음 과정 다시 진행
    child *= 2; //왼쪽 자식노드를 child로 재설정
}
heap[parent] = temp; //temp를 child에 넣기
return item;
}

```

## 2. bst

**treePointer modifiedSearch(treePointer tree, int k)**

```

{
    //k값을 key값으로 가지는 노드가 있는지 search
    treePointer tmp=tree;
    while (tree) { //k가 tree 안에 있는지 계속 찾기
        if (k == tree->data.key)
            return NULL;
        if (k < tree->data.key) {
            tmp = tree;
            tree = tree->leftChild;
        }
        else {
            tmp = tree;
            tree = tree->rightChild;
        }
    }
    return tmp;
    //tmp 값으로 k를 key값으로 가지는 노드가 들어가야하는
    //위치의 "부모 노드" 주소가 반환됨
}

```

```

}

void insert(treePointer * node, int k, itemType theItem)
{
    treePointer ptr;
    treePointer temp = modifiedSearch(*node, k);
    //modifiedSearch로 트리에 이미 key값으로 k가 존재하는지 검사
    //존재하거나 트리가 공백이면 temp=NULL
    //마지막까지 검사했는데 같은 값이 없으면
    //마지막으로 검사한 노드의 포인터 반환

    if (temp || !(*node)) {
        //같은 키 값이 트리 안에 없어서 추가 가능한 경우
        //OR 트리가 공백인 경우
        MALLOC(ptr, sizeof(*ptr));
        ptr->data.key = k;
        ptr->data.item = theItem;
        ptr->leftChild = ptr->rightChild = NULL;

        if (*node) { //트리에 노드 이미 존재
            //temp에는 삽입할 노드의 부모노드의 주소값
            if (k < temp->data.key)
                temp->leftChild = ptr;
            else
                temp->rightChild = ptr;
        }
        else//빈 트리면 쥔 위에 붙이기
            *node = ptr;
    }
}

```

```

element* search(treePointer root, int k)
{
    if (!root) {
        printf("there is no such an element\n\n");
        return NULL;
    }

    if (k == root->data.key) {
        printf("the element's item is %d\n\n", k);
        return &(amp;root->data);
    }

    if (k < root->data.key)
        return search(root->leftChild, k);
    return search(root->rightChild, k);
}

```

### 3. winner tree

```
int initWinner(int cur, int k)
{
    if (cur >= k)
        return winTree[cur] = (cur + 1) - k; //record 번호 그대로 넣기
    int leftChild = initWinner(cur * 2, k); //왼쪽 자식
    int rightChild = initWinner(cur * 2 + 1, k); //오른쪽 자식

    return winTree[cur]
        = nums[leftChild][sortedIdx[leftChild]] < nums[rightChild][sortedIdx[rightChild]] ?
        leftChild : rightChild; //둘 중 작은 값 넣기
}
```

```
void adjustWinner(int min, int k)
{
    //변할 필요없는 값은 그대로 두고 올라간 값 때문에 변해야하는 노드부터 수정
    //->min=k - 1 + winTree[1]
    if (min == 1)
        return;
    int parent = min / 2;
    int leftChild = winTree[parent * 2];
    int rightChild = winTree[parent * 2 + 1];

    winTree[parent]=
        nums[leftChild][sortedIdx[leftChild]]
        < nums[rightChild][sortedIdx[rightChild]] ?
        leftChild : rightChild;
    adjustWinner(parent, k);
}
```

## Linked List

### 1. Palindrome

```
tListNodePointer initList(char* str)
{
    //뒤에 삽입하기
    tListNodePointer newNode, lead, tmp;
    int i = 0;
    lead = createNode(str[i]);
    for (str[++i]; str[i]; i++) {
        newNode = createNode(str[i]);
        //tmp는 newNode 바로 전 노드
        tmp = lead;
        while (tmp->link)
            tmp = tmp->link;
        //tmp 뒤에 newNode 연결
        tmp->link = newNode;
    }
}
```

```

    }
    return lead;
}

```

**tListNodePointer copyNode(tListNodePointer lead)**

```

{//lead를 copyLead로 복사
    tListNodePointer copyLead = createNode(lead->data);
    //첫번째 데이터 복사
    lead = lead->link;
    tListNodePointer newNode, tmp;
    while (lead) {
        newNode = createNode(lead->data);
        //tmp는 새로 삽입할 노드 바로 전 노드
        tmp = copyLead;
        while (tmp->link)
            tmp = tmp->link;
        //tmp 뒤에 newNode 연결
        tmp->link = newNode;
        lead = lead->link;
    }
    return copyLead;
}

```

**int isPalindrome(tListNodePointer head)**

```

{
    tListNodePointer tmp, copyHead, inv;

    printf("initial linked list : ");
    printLinkedList(head);

    copyHead = copyNode(head);

    inv = invert(copyHead);
    printf("inverted linked list: ");
    printLinkedList(inv);

    while (head && inv) {
        //역순 연결리스트(inv)랑 데이터 하나라도 다르면 회문 아님
        if (head->data != inv->data)
            return 0;
        //다음 노드 비교
        head = head->link;
        inv = inv->link;
    }
    //while문 모두 통과했으면 회문
    return 1;
}

```

# Graph

## 1. DFS

```
void dfs(int v)
{
    nodePointer w;
    visited[v] = TRUE; //시작하는 v정점 TRUE로 바꾸고
    printf("%5d", v); //v부터 출력
    //인접연결리스트(adjList[])로 정리된 graph[]
    //시작할 때는 graph[v]에 연결된 노드부터 w에 집어넣고
    for (w = graph[v]; w; w = w->link) {
        if (!visited[w->ver]) //visited[node]==0 이면
            dfs(w->ver); //graph[node]에 대해 dfs 진행
        //visited[node]==1 이면 0인 노드 찾을 때까지 옆 노드로 이동
        //옆으로 끝까지 갔는데 visited[node]==0인 게 없으면 backtracking
    }
    //v에 연결된 노드 모두 프린트
}
```

## 2. BFS

```
void addq(int item)
{
    queuePointer tmp;
    MALLOC(tmp, sizeof(*tmp));

    tmp->vertex = item;
    tmp->link = NULL;

    if (front) //큐에 뭐가 있으면
        rear->link = tmp; //마지막 노드 다음에 새 노드 추가
    else
        front = tmp;

    rear = tmp; //추가한 노드를 마지막노드로 설정
}

int deleteq()
{
    queuePointer tmp = front; //삭제할 노드(제일 앞의 front)
    int item;

    item = tmp->vertex;
    front = tmp->link; //front의 정보 빼고 없애기 (다음 노드를 front로)

    return item;
}
```

```
}
```

```
void bfs(int v)
```

```
//한 줄씩 해서(graph[0]에 연결된 것 모두 출력 후 graph[1]) 모든 원소 출력
```

```
{
```

```
    front = rear = NULL;
```

```
    printf("%5d", v); //v번 정점 출력하고
```

```
    visited[v] = TRUE; //v번 정점 1로 바꾸고
```

```
    addq(v); //큐에 추가
```

```
    while (front) { //큐에 뭔가 있으면
```

```
        v = deleteq(); //큐 제일 앞에 있는 거 빼서 그거에 대해 for문
```

```
        //처음에는 graph[v]에 연결된 첫번째 노드를 w로 설정
```

```
        for (queuePointer w = graph[v]; w; w = w->link) {
```

```
            if (!visited[w->vertex]) {
```

```
                printf("%5d", w->vertex); //연결된 노드 원소 출력하고
```

```
                addq(w->vertex); //큐에 넣고
```

```
                visited[w->vertex] = TRUE; //visited=>1로 바꾸기
```

```
            } //다음에 연결된 노드가 없으면 while문으로 돌아가기
```

```
        }
```

```
    }
```

```
    //큐에 아무것도 남지않으면 종료
```

```
}
```

### 3. adjacency list

```
void addEdge(nodePointer* list, int ver)
```

```
{
```

```
    nodePointer tmp;
```

```
    MALLOC(tmp, sizeof(*tmp)); //노드 생성
```

```
    tmp->data = ver;
```

```
    if (*list) { //노드에 노드 연결
```

```
        // *list는 "마지막으로" adj[i]에 연결한 노드의 주소값
```

```
        tmp->link = *list;
```

```
        *list = tmp;
```

```
        //tmp의 주소가 adj[i]의 주소에 저장
```

```
        //tmp->adj[i]는 마지막으로 저장된 노드를 가리킴
```

```
    }
```

```
    else { //노드 연결된 거 없으면 첫 노드 연결
```

```
        tmp->link = NULL;
```

```
        *list = tmp;
```

```
        //tmp의 주소가 adj[i]에 저장됨
```

```
    }
```

```
}
```



## 4. connected components

```
void connected(int num)
{
    int cnt = 1;

    for (int i = 0; i < num; i++) {
        if (!visited[i]) {
            //1번이랑 다르게 visited 초기화 안 해주니까
            //n번 vertex에서 visited[i]가 모두 1로 바뀌면
            //<=>모든 정점이 출력되면
            //더이상 출력되지 않음
            printf("connected component %d : ", cnt);
            dfs(i);
            printf("\n");
            cnt++;
        }
    }
}
```

## Sorting

### 1. insertion sort

```
void insert(element e, element a[], int i)
{
    //삽입하려는 값(e)보다 작거나 같은 인덱스(i)를 찾아서
    //그 다음 인덱스에 e 삽입
    //맨 뒤 원소부터 비교
    a[0] = e; //e가 가장 작은 경우에도 별도 조건 없이 while탈출
    while (e.key < a[i].key) { //a[i].key가 e보다 작으면 index=i
        a[i + 1] = a[i]; //뒷칸으로 밀기 (e가 들어갈 곳 마련)
        i--; //그 앞의 원소와 비교
    }
    a[i + 1] = e;
}

void insertionSort(element a[], int n)
{
    //각 원소에 대해 insert 수행
    //첫번째 원소는 비교 대상 없으니까 그냥 넘어가기
    for (int j = 2; j <= n; j++) {
        element temp = a[j];
        //temp를 a배열의 이미 정렬된 j-1개의 원소와 비교해서
        //들어가야할 위치에 삽입
        insert(temp, a, j - 1);
    }
}
```

## 2. quick sort

```
void quickSort(element a[], int left, int right)
{
    int pivot, i, j;
    element temp;
    cnt++;
    for (int k = 0; k < num; k++)
        printf("%2d ", a[k].key);
    printf("\n");

    if (left < right) {
        i = left; j = right + 1;
        pivot = a[left].key;
        do {
            do i++; while (a[i].key < pivot);
            //pivot보다 큰 a[i]를 찾을 때까지 오른쪽으로 올라가기
            //a[i].key>pivot 이면 정지
            do j--; while (a[j].key > pivot);
            //pivot보다 작은 a[j]를 찾을 때까지 왼쪽으로 내려가기
            //a[j].key<pivot 이면 정지
            if (i < j) SWAP(a[i], a[j], temp);
        } while (i < j);
        SWAP(a[left], a[j], temp); //pivot와 j 바꾸기
        //j를 기준으로 왼쪽에 있는 거, 오른쪽에 있는 거 quickSort
        quickSort(a, left, j - 1);
        quickSort(a, j + 1, right);
    }
}
```

## 3. merge sort

```
void merge(element initList[], element mergedList[], int i, int m, int n)
{
    //i부터 시작하는 정렬된 부분집합과 m+1부터 시작하는 정렬된 부분집합을
    // 정렬해서 작은것부터 mergedList에 넣기
    //부분집합 한 뭉텅이를 merge해서 정렬
    //initList[i:m]+[m+1:n]
    int j, k, t;
    j = m + 1;
    k = i; //mergedList index i부터 시작

    while (i <= m && j <= n) { //하나라도 끝까지 가면 종료
        //앞의 부분집합과 뒤의 부분집합. 각자의 부분집합 내에서 이동
        //i와 j 중 작은 거부터 mergedList에 넣기
        if (initList[i].key <= initList[j].key)
```

```

        mergedList[k++] = initList[i++];
    else
        mergedList[k++] = initList[j++];
}
//남는 데이터 삽입
//앞부분과 뒷부분의 리스트 중 남는 거 append
if (i > m) //앞부분이 먼저 종료->뒷부분 그대로 붙이기
    for (t = j; t <= n; t++)
        mergedList[k++] = initList[t];

else //뒷부분이 먼저 종료->앞부분 그대로 붙이기
    for (t = i; t <= m; t++)
        mergedList[k++] = initList[t];

//mergedList[k + t - i] = initList[t]; ->피피티 내용
//앞부분에 있던 거니까 mergedList랑 인덱스 달라짐
//현재 mergedList의 인덱스 k
//=뒷부분 원소 개수(n-m-1)+정렬된 앞부분 원소 개수(i)
//앞부분 원소 인덱스=t부터 시작
}

```

```

void mergePass(element initList[], element mergedList[], int n, int s)

```

```

{
    int i, j;
    //initList를 2s개씩 묶어서 merge 수행
    for (i = 1; i <= n - 2 * s + 1; i += 2 * s)
        //sublist 1개의 크기가 s->2개씩 합치니까 증감식에서 2s 더하기
        // 2s개 원소->merge 끝나는 부분이 i+2s-1 -> 조건식에서 이 부분이 n보다 작다
        // i+2s-1<=n <=> i<=n-2s+1
        //merge(initList, mergedList, 시작, 첫 segment 끝, 두번째 segment 끝)
        merge(initList, mergedList, i, i + s - 1, i + 2 * s - 1);

    //2s개씩 묶어서 merge 하고 나서 남은 부분 처리
    //
    // 남은 부분이 s개 보다 더 많을 때
    //남은 부분 중 앞부분은 s크기로 자르고 뒷부분은 다른 크기일 때
    //남은 부분 첫 인덱스를 i로 두고 initList 끝(n)까지 merge
    if (i + s - 1 < n)
        merge(initList, mergedList, i, i + s - 1, n);
    //남은 부분이 s보다 적을 때
    //이미 정렬된 상태니까 따로 합병할 필요없이 그대로 mergedList에 삽입(복사)
    else
        for (j = i; j <= n; j++)
            mergedList[j] = initList[j];

    printf("segment size :%3d\n", s);
}

```

```

int num = logB(s,2); //s가 2의 몇 제곱인지->짝수 제곱승이면 a,extra 순서
if (num % 2 ==0) {
    //merge 전
    printf("\t  %s :", "a");
    for (int h = 1; h <= n; h++)
        printf("%3d", initList[h].key);
    printf("\n");
    //merge 후
    printf("\t%s:", "extra");
    for (int h = 1; h <= n; h++)
        printf("%3d", mergedList[h].key);
    printf("\n\n");
}
else {
    //merge 전
    printf("\t%s:", "extra");
    for (int h = 1; h <= n; h++)
        printf("%3d", initList[h].key);
    printf("\n");
    //merge 후
    printf("\t  %s :", "a");
    for (int h = 1; h <= n; h++)
        printf("%3d", mergedList[h].key);
    printf("\n\n");
}
}

void mergeSort(element a[], int n)
{
    int s = 1; //current segment size
    element extra[MAX_SIZE];

    while (s < n) { //부분집합이 전체 원소 수보다 큰 동안 진행. 같아지면 종료
//mergePass( initList, mergedList, int n, int s)
        //a(전체 데이터)를 extra라는 sublist에 정렬해서 넣음
        //->extra 배열 속의 데이터는 2s개씩 끊어서 모두 정렬된 상태
        //printf("mergePass(a,extra);
        mergePass(a, extra, n, s);
        s *= 2;
        //정렬된 extra가 이번에는 a 역할
        //s개 단위로 정렬되어있는 extra를 2s개씩 묶어 정렬 후 a로 merge
        //printf("mergePass(extra,a);
        mergePass(extra, a, n, s);
        s *= 2;
    }
}

```