

# 자료구조

## 링크드리스트 - 용목이형

배열의 장점 : 인덱스, 액세스가 가능, 간편하다 / 배열의 단점 : 값을 바꿀 수 없다,

\* 병렬 배열 => 김광일 / 김락현 / 송용목 / 김수정 <연결해서 씬 다음자리가 없으면 next

병렬배열 방법 : 김광일 : real data 김락현 : next node pointer / ...송용목, 장수정 포인터사용

\* 리스트 : 노드들의 집합 /

```
- 추상 자료형(ADT)
List
{
    node* head; // 노드에는 data(<=data type)를 저장하는 부분과 next (<=node*)는 다음 것을 가리킴
    methods:
    list();
    add(data);
}
```

(포인터) □->□->□->□->□-> 0(NULL)(없는 값) // 없는 값을 가리키면 tailPrev를 찾음

tail Prev / tail(tail은 끝값을 찾음)

\* 이중연결리스트 사용법 □ <-> □ <-> □ <-> □

```
//LinkedList.cpp파일//
#include <stdio.h>
#include "LinkedList.h"

CLinkedList::CLinkedList() {
    head = NULL; //헤드 값을 0으로
    tail = NULL; // 테일 값을 0으로
}

void CLinkedList::add(float data) { //여기에 float data를 추가함
    // 1. create a node(노드생성)
    CNode *n = new CNode; //CNode *n의 값에 new CNode의 값을 바라봄
    n->next = NULL; //n의 다음 값은 없는 값
    n->val = data;

    if(!head) { // if this is the first node(만약 첫 번째 노드라면)
        head = n; //헤드의 값은 n의 값이 됨
        tail = n; //테일의 값도 n의 값이 됨
        return;
    }
    // 3. add the new node to the tail (테일에 새로운 노드를 추가함)
    tail->next = n; //tail의 다음 값이 n
    tail = n; //tail의 값이 n이 됨
}

float CLinkedList::remove(void) {
    float retTemp;
    if(!head) return -99999999;
    if(!(head->next)) {
        retTemp = head->val;
        delete head;
        head = tail = NULL;
        return retTemp;
    }
    // 1. find the node just before the tail
    CNode *tailPrev = head;
    CNode *curTail = tailPrev->next;
    while(curTail->next) {
        tailPrev = curTail;
        curTail = curTail->next;
    }
    // 2. delete tail
    retTemp = tail->val;
    delete tail;
    tailPrev->next = NULL;
    tail = tailPrev;
    return retTemp;
}

void CLinkedList::show(void){
    CNode *n = head;
    while(n) {
        printf("[%f]", n->val);
        n = n->next;
    }
}
```

```
//LinkedList.h 파일// 1순위 헤더파일
#ifndef __LINKEDLIST_H
#define __LINKEDLIST_H
#include<iostream>
using namespace std;

struct CNode //class를 struct로 만들어도됨
{
    float val;
    CNode *next; //CNode를 가르키는 포인터
};

class CLinkedList
{
    CNode *head;
    CNode *tail;
public:
    CLinkedList();
    void add(float data); //add가 하는 것 node생성 ->data설정-> 마지막 노드찾기
    -> 마지막 노드 new node로 바꾸기
    float remove(void);
    void show(void);
};

#endif
```

```
// main.cpp 파일 //
#include "LinkedList.h"

CLinkedList myList;

int main(void) {

    char command;
    float num;
    while(1){
        printf("command> ");
        scanf("%c", &command);
        if(command=='a') {
            printf("value to be added: ");
            scanf("%f", &num); // c++에서는 cin >> num <<endl;
            myList.add(num);
        }
        else if(command=='r') {
            float removeValue = myList.remove();
            cout << "removed" << removeValue<<endl; //c++에서 출력
        }
        myList.show();
    }
    return 0;
}
```

**new\_bee** 2009.09.30 15:23

그렇지 않습니다. 99.99퍼센트 똑같고 0.01퍼센트가 다른 것은, 액세스 지정자를 지정하지 않았을 시 기본적으로 선택되는 것만 다릅니다. class의 경우는 private이고 struct의 경우 public입니다. 그 점만 빼고는 완전히 똑같습니다.

```
class Derived_class : Base
{
void PublicOrPrivate();
};
```

```
struct Derived_struct : Base
{
void PublicOrPrivate();
};
```

이렇게 두가지가 있을 경우, Derived\_class의 경우는 클래스이므로 Base로부터 private상속을 받게 되고 PublicOrPrivate는 private멤버가 됩니다.

Derived\_struct의 경우 Base로부터 public상속을 받게 되고 PublicOrPrivate는 public멤버가 됩니다.

C의 습관 때문에 구조체를 쓴다는 그런 이유는 아닐 것 같고..  
저는 나름대로(?) 용도를 나눠서 쓰고 있습니다. 다른 분들도 그러실 것입니다.  
물론 class만 사용할 수도 있겠구요.. 모두 취향에 따라 다를 것입니다.

```

#include <iostream>
using namespace std;
template <typename Object>
class LinkedList{
protected:
    struct Node{
        Object element; //원소
        Node* prev; //앞 노드
        Node* next; //뒷 노드
        Node(const Object& e = Object(), Node* p = NULL, Node* n = NULL)
            : element(e), prev(p), next(n){}
        //생성자 초기 값도 매개변수 라인에 지정 후 아래 : 에 대입
    };
    typedef Node* NodePtr; //노드에 대한 포인터 정의

private: //멤버 데이터
    NodePtr head; //head세팅을 위한 포인터
    NodePtr tail; //tail세팅을 위한 포인터
    int sz; //원소들의 수

public:
    LinkedList(){
        head = new Node;
        tail = new Node;
        head->next = tail; //head의 다음은 tail
        tail->prev = head; //tail의 앞은 head로 초기화
        sz = 0;
    }

    ~LinkedList(){
        delete head;
        delete tail;
        head = NULL;
        tail = NULL;
    }

    void insertFirst(const Object& e){ //Node의 생성자의 매개변수 중 Object의 참조형 변수 타입이기 때문
        NodePtr oldFirst = head->next;
        NodePtr newFirst = new Node(e, head, oldFirst);
        oldFirst->prev = newFirst;
        head->next = newFirst;
        sz++;
    } //맨처음 head와 tail이 붙어있는 경우에도 이는 수행되어야함. 따라서 헤드 안에서 왔다갔다
    //하는 것보다 포인터 선언으로 안전하게 하는 것이 가시화가잘됨

    void insertLast(const Object& e){
        NodePtr oldLast = tail->prev;
        NodePtr newLast = new Node(e, oldLast, tail);
        oldLast->next = newLast;
        tail->prev = newLast;
        sz++;
    }

    bool isEmpty(){
        if(sz <= 0)
            return true;
        else
            return false;
    }

    void removeFirst(){
        if(isEmpty()){
            cout<<"노드가 존재하지 않습니다."<<endl;
            return;
        }
        NodePtr oldFirst = head->next;
        NodePtr newFirst = oldFirst->next;
        newFirst->prev = head;
        head->next = newFirst;
        delete oldFirst;
        sz--;
    }

    void removeLast(){
        if(isEmpty()){
            cout<<"노드가 존재하지 않습니다."<<endl;
            return;
        }
        NodePtr oldLast = tail->prev;
        NodePtr newLast = oldLast->prev;
        newLast->next = tail;
        tail->prev = newLast;
        delete oldLast;
        sz--;
    }

    Object& first(){
        return head->next->element;
    }
    Object& last(){
        return tail->prev->element;
    }

    int size(){
        return sz; //현재 크기 반환
    }

    void showAllLinkedList(){ //원소 모두 출력
        int num = 0;
        NodePtr np = head->next;
        while(np != tail){
            cout<<num<<"번째 요소는"<<np->element<<endl;
            num++;
            np = np->next;
        }
    };

    int main(){
        LinkedList<int> lk;
        lk.insertFirst(2);
        lk.insertFirst(2);
        lk.insertFirst(3);
        lk.insertFirst(4);
        lk.insertFirst(2);
        lk.insertFirst(6);
        lk.insertFirst(7);

        cout<<"현재 링크드 리스트의 크기는"<<lk.size()<<"이다."<<endl<<"그 원소로는"<<endl;
        lk.showAllLinkedList();
        lk.removeLast();
        lk.removeFirst();
        cout<<"현재 링크드 리스트의 크기는"<<lk.size()<<"이다."<<endl<<"그 원소로는"<<endl;
        lk.showAllLinkedList();
        return 0;
    }
}

```

## 스택과 큐(Queue)

**스택** : 삽입되는 곳과 삭제되는 곳이 한끝에서 일어나는 자료구조의 형태 (-LIFO후입선출)

- top : 데이터가 입력 및 삭제를 가리킬 변수
- isEmpty(); : 스택에 입력된 값이 있는지 없는지 확인하는 함수
- isFull(); : 스택의 배열 사이즈를 초과했는지 확인하는 함수
- push(); : 배열에 데이터를 넣어주는 함수
- pop(); : 배열에서 데이터를 빼어주는 함수

**큐** : 한쪽 끝에 데이터가 삽입, 그 반대쪽 끝에서 삭제가 일어나는 형태 (-FIFO선입선출)

- 두 개의 인덱스를 설정해야함 / rear이 증가/ front는 처음 들어온 데이터를 삭제하는형태

\* **선형 큐** : 선형 큐는 삭제하고 나면 빈 공간이 생성됨.

\* **원형 큐** : 선형 큐의 빈 공간이 생기는 것을 방지하기 위해 만든 것

(빈 공간 한군데는 포화와 오류상태를 구분하기 위해 남겨둠)

- front : 데이터를 삭제할 곳을 가리키는 변수
- rear : 데이터를 입력할 곳을 가리키는 변수
- isEmpty(); : 스택에 입력된 값이 있는지 없는지 확인하는 함수
- isFull(); : 스택의 배열 사이즈를 초과했는지 확인하는 함수

(스택과 차이점 : 프론트,리얼의 위치로 파악함)

- add(); : 배열에 데이터를 넣어주는 함수
- remove(); : 배열에 데이터를 삭제하는 함수

```
//Stack.h파일//
#ifndef STACK_H
#define STACK_H

#include <iostream>
using namespace std;

#define MAX_SIZE 100

class Stack
{
private:
    int n[MAX_SIZE]
    int top;
public:
    bool isEmpty();
    bool isFull();
    void push(int data);
    int pop(); // data를 넣을 필요 없음. top에 있는 애가 나오면 되니깐
    void display();
};

#endif STACK_H

//Stack.cpp파일//
#include "Stack.h"

Stack::Stack() {top = -1;}

bool Stack::isEmpty()
{
    return top == -1;
    /*if(top == -1)
    {
        return true;
    }
    else
        return false;*/
}

bool Stack::isFull()
{
    if(top == (MAX_SIZE-1))
    {
        return true;
    }
    else
        return false;
}

void Stack::push(int data)
{
    if(!isFull()) n[++top] = data; //if(isEmpty()) {printf("Stack Full!\n"); return;}
    else
        /*top++;
        n[top] = data;*/
}

int Stack::pop() // data를 넣을 필요 없음. top에 있는 애가 나오면 되니깐
{
    return isEmpty()? -999999 : n[top--];
    /*if(!isEmpty())
    {
        return n[top--];
    }
    else
        return -999999;
    */
}

/* 현재 지우려는 top의 값을 지우지 않아도 다음번에 덮어쓰기 때문에 코드가 줄어듬
top--; return n[top+1] = NULL;*/

void Stack::display()
{
    for(int i=0; i<top; i++){
        printf("[%d]", n[i]);
    }
    printf("\n");
};

#endif STACK_H
```

## 우선순위 큐(Priority Queue) - 광일이형

우선순위 큐 : 큐지만 데이터의 순서와 상관없이 우선순위가 높은 데이터가 먼저 나가는 큐

- \* 우선순위 큐의 구현 : 힙(Heap)
- \* 힙 : 중간에 빈 곳이 없이 채워진 트리, 배열을 이용하여 구현(핵심적인 아이디어) / 정렬할 때도 사용
  - 포화 이진트리는 2승씩 올라간다.
- \* 포화 이진트리가 되려면 배열의 0번 과 마지막 번을 비워줌(비율에 맞게)
  - MAX heap : 부모 노드의 키값이
  - MIN heap
    - 왼쪽 자식 :  $4 = i(2)*2$  / 부모노드찾기 :  $(i)/2$
    - 오른쪽 자식 :  $5 = i(2)*2+1$  / 부모노드찾기 :  $(i)/2-1$
- \* 삽입 연산 : 들어온 노드의 키 값을 보고 부모 노드와 비교 후 위치를 바꾼다.
  - 제일 밑바닥부터 시작해 이겨서 올라감
- \* 삭제 연산 : 맨 위의 노드를 제거 후 제일 마지막 노드를 가져다 끼운다. 그 후 아래위를 바꿔나간다.
  - 자식 둘 중 작은 놈과 바꾼다.

```
//PriorityQueue.h //
#ifndef PRIORITYQUEUE_H
#define PRIORITYQUEUE_H
#include <string>
#include <iostream>

using namespace std;

struct Node{
    string data;
    int key;
};

class PriorityQueue {
    Node *heap;
public:
    PriorityQueue();
    void Push(string data, int key);
    void Pop();
    void Show();
};
#endif

// main.cpp //
#include "PriorityQueue.h"

int main(){
    PriorityQueue aa;

    aa.Push("a", 0); aa.Push("b", 2); aa.Push("c", 3); aa.Push("d", 7); aa.Push("e", 8);
    aa.Push("f", 5); aa.Push("g", 12); aa.Push("h", 10); aa.Push("i", 1);
    aa.Show();

    aa.Pop(); aa.Show(); aa.Pop(); aa.Show(); aa.Pop(); aa.Show();
    aa.Pop(); aa.Show(); aa.Pop(); aa.Show();

    return 0;
}
```

```

// PriorityQueue.cpp //
#include "PriorityQueue.h"

void Swap(Node &a, Node &b){ // 키 값 비교 후 교환을 위한 Swap
    Node tmpNode;
    tmpNode = a;
    a = b;
    b = tmpNode;
}

PriorityQueue::PriorityQueue(){
    heap = new Node[100];
    heap[0].key = 0;
}

void PriorityQueue::Push(string data, int key){
    heap[0].key++;
    heap[heap[0].key].key = key;
    heap[heap[0].key].data = data;

    while(1){
        if(heap[0].key < 1) // 1보다 작다는 것은 아무것도 없다는 뜻
            break;
        if(heap[heap[0].key].key < heap[heap[0].key/2].key){
            Swap(heap[heap[0].key], heap[heap[0].key/2]);
        } // 삽입 후 부모노드와 키 값 비교
        else break;
    }
}

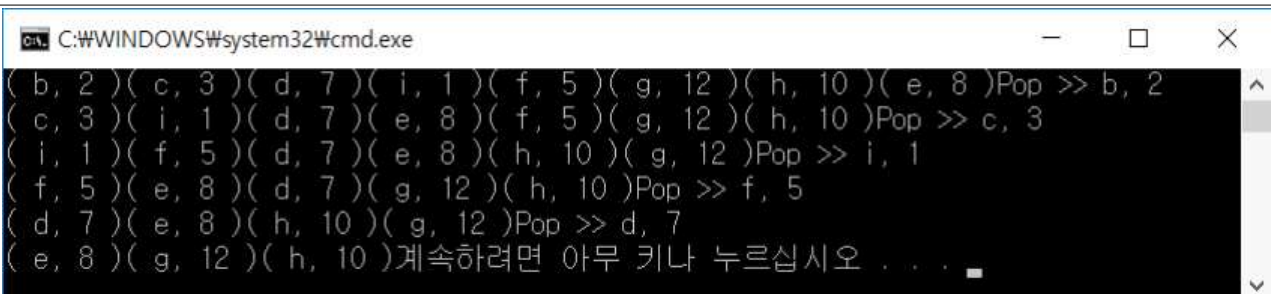
void PriorityQueue::Pop(){
    if(heap[0].key <= 1){
        cout << "Not pop data" << endl;
    }
    else{
        cout << "Pop >> " << heap[1].data << ", " << heap[1].key << endl;
        heap[1].key = 99999;
        heap[1].data = "\0";
        int num = 1;

        Swap(heap[1], heap[heap[0].key]); //마지막 노드를 최상위 노드로
        heap[0].key--;

        while(1){
            if(num > heap[0].key/2)
                break;
            if(heap[num].key > heap[num*2].key || heap[num].key > heap[num*2+1].key){
                if(heap[num*2].key > heap[num*2+1].key){ //최상위 노드와 자식 노드 비교
                    Swap(heap[num], heap[num*2]);
                    num = num*2;
                }
                else {
                    Swap(heap[num], heap[num*2+1]);
                    num = num*2+1;
                }
            }
            else break;
        }
    }
}

void PriorityQueue::Show(){
    for(int i=1; i<=heap[0].key; i++){
        cout << "(" << heap[i].data << ", " << heap[i].key << " )";
    }
}

```



```

C:\WINDOWS\system32\cmd.exe
(b, 2)(c, 3)(d, 7)(i, 1)(f, 5)(g, 12)(h, 10)(e, 8)Pop >> b, 2
(c, 3)(i, 1)(d, 7)(e, 8)(f, 5)(g, 12)(h, 10)Pop >> c, 3
(i, 1)(f, 5)(d, 7)(e, 8)(h, 10)(g, 12)Pop >> i, 1
(f, 5)(e, 8)(d, 7)(g, 12)(h, 10)Pop >> f, 5
(d, 7)(e, 8)(h, 10)(g, 12)Pop >> d, 7
(e, 8)(g, 12)(h, 10)계속하려면 아무 키나 누르십시오 . . .

```

## 버블소트, 퀵 정렬 - 정환이

- \* 버블소트 : 앞에서 탐색하여 스왑해주는 것
- \* 퀵 정렬 : 맨 앞자리 숫자(a)를 선택하고 정렬을 반으로 나누고 low, high를 사용해 탐색 하여 a의 자리를 차려줌.
  - 장점 : a의 개수 :  $n = 1$ 개,  $n-1 = 2$ 개,  $n-3 = 4$ 개 ..... 로 빨리 탐색 가능

```
//Bubblesort.h//
#ifndef __MYSORT_H_
#define __MYSORT_H_
class Bubblesort{
private:
    void Swap(int & ref1, int &ref2);
    int Partition(int arr[], int left, int right);
public:
    void BubbleSort(int arr[], int len);
    void QuickSort(int arr[], int left, int right);
};
#endif

#include"Bubblesort.h"
#include<iostream>
using namespace std;

void main()
{
    Bubblesort s;

    int arr[6] = { 2,4,1,6,5,3 };

    cout << "[퀵소트 전]" << endl;
    for (int i = 0; i <= 5; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;

    s.QuickSort(arr, 0, 5);

    cout << "[퀵소트 후]" << endl;
    for (int i = 0; i <= 5; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;

    int arr2[] = { 2,4,5,6,7,1,2,4,5,2,6,1 };
    cout << "[버블소트 전]" << endl;

    for (int i = 0; i < sizeof(arr2) / sizeof(int); ++i)
    {
        cout << arr2[i] << " ";
    }

    s.BubbleSort(arr2, sizeof(arr2)/sizeof(int));

    cout << "[버블소트 후]" << endl;
    for (int i = 0; i < sizeof(arr2) / sizeof(int); ++i)
    {
        cout << arr2[i] << " ";
    }
    cout << endl;
}
```

```
#include "Bubblesort.h"

void Bubblesort::Swap(int & ref1, int & ref2)
{
    int temp = ref1;
    ref1 = ref2;
    ref2 = temp;
}

int Bubblesort::Partition(int arr[], int left, int right)
{
    int pivot = arr[left]; //선택한 숫자(제일 왼쪽 숫자)
    int low = left+1, high = right;
    // 인덱스 값을 나타내기 때문에 위치정보를 줌 / left +1은 비교할 숫자이기 때문에 +1해줌

    while (low <= high)
    {
        while (pivot >= arr[low] && low<=right) { //교차했을 때 끝낼 것이기 때문
            low++;
        }
        while (pivot <= arr[high] && high>left) {
            high--;
        }

        if(low<=high)
            Swap(arr[low], arr[high]);
    }
    Swap(arr[left], arr[high]);
    return high;
}

void Bubblesort::BubbleSort(int arr[], int len)
{
    for (int i = 0; i < len - 1; i++)
    {
        for (int j = 0; j < len - 1 - i; j++)
        {
            if (arr[j] > arr[j + 1])
                Swap(arr[j], arr[j + 1]);
        }
    }
}

void Bubblesort::QuickSort(int arr[], int left, int right) //퀵소트
{
    if (left >= right)
        return;
    int mid = Partition(arr, left, right);
    QuickSort(arr, left, mid - 1);
    QuickSort(arr, mid + 1, right);
}
```

//버블소팅 다른 방법//

```
void Bubblesort::BubbleSort(int *a, int n)
{
    int nSorted = 0;
    int i = 0;
    while(nSorted<n) {
        if(i==n-nSorted-1) { nSorted++; i=0; continue;}
        if(a[i]>a[i+1]) swap(a[i],a[i+1]); i++;
    }
}
```

```
C:\Windows\system32\cmd.exe
[퀵소트 전]
2 4 1 6 5 3
[퀵소트 후]
1 2 3 4 5 6
[버블소트 전]
2 4 5 6 7 1 2 4 5 2 6 1
[버블소트 후]
1 1 2 2 2 4 4 5 5 6 6 7
계속하려면 아무 키나 누르십시오 . . .
```



## 탐색, 그래프 - 수정이

DFS : 깊이우선탐색 - 스택 (왼쪽루트부터 내려가고 더 이상 갈림길이 없을 때 전루트로 올라가서 오른쪽으로 탐색 반복)

BFS : 넓이우선 탐색 - 큐