자료구조 자료[redacted]

① 큐(선입선출목록) ~▷ 목록에 들어온지 가장 오래된 것 꺼내기.

– queue의 구현(array)

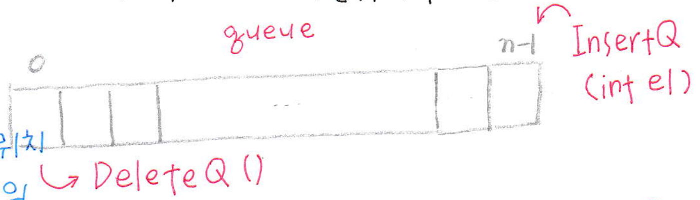
front : 큐의 첫번째요소 = 삭제될 원소의 위치
rear : 큐의 마지막 원소의 다음위치 = 삽입될 위치

```
class Queue{  <queue.h> ①
public:
   Queue();
   bool IsEmpty(){return(rear==front)};
   bool IsFull(){return(rear==MAXQ)};
   bool InsertQ(int el);
   int DeleteQ();
private:
   int arr[MAXQ];
   int front;
   int rear;
}
```

queue덜적음.

```
queue.cpp>②
#include queue.h
⊕V #include <iostream> using ~;
Queue::Queue(){  초기화
  front=0;
  rear=0; }
void Queue::InsertQ(int el){
if(rear==MAXQ) return false;
arr[rear++] = el; }

int Queue::DeleteQ(){
if(rear==front) return -1;
return arr[front++];
```



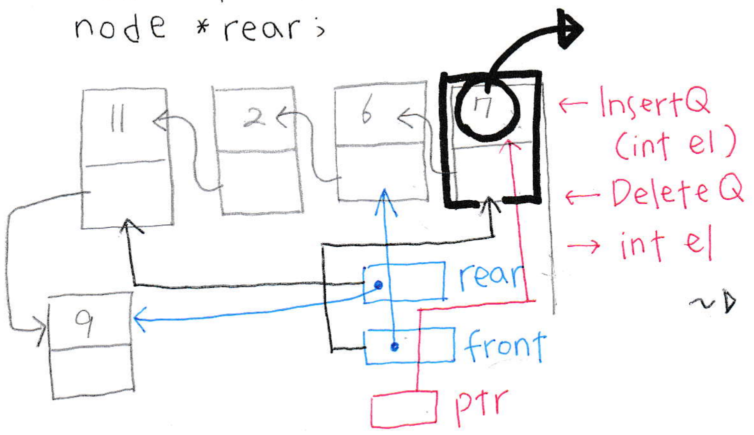
front : 큐의첫번째원소위치
rear : 큐의 마지막원소의 위치

– queue의 구현(pointer)

```
#define ERROR -1 ; //양의정수만 다룬다.
class node {  <node.h> ①
public:
  node(int d, node *n=0){
    next=n;
    data=d; }

#include node.h <queue.h>②
class Queue{
public: Queue();
  bool Isempty()
  bool IsFull()
  bool InsertQ(int el);
  int DeleteQ();
private:
  node *front;
  node *rear;
```

```
#include queue.h ⊕V <queue.cpp>③
Queue::Queue(){ front=0; rear=0; }
bool Queue::IsEmpty(){
  if(front==0) return true;
  return false; }
bool Queue::IsFull(){
  return false; }
void Queue::InsertQ(int el){
if(IsEmpty()){} if(IsFull()) return;
  front = rear = new node(el);
  return;
}
rear→next = new Node(el);
rear = rear→next; }
int Queue::DeleteQ(){
if(IsEmpty()) return -1;
int el = front→data;
Node *ptr = front;
front = front→next;
if(front==0) rear=0;
delete ptr;
return el; }
```



~▷ 9 삽입
7 삭제

② 스택 (후입선출) 재귀적인 구조를 표현하는데 적절

- Stack의 구현 (array)

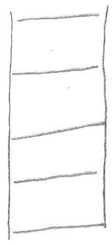
top : 다음 삽입될위치 = 다음번 원소가 들어갈위치 ~▷ 스택의 자료삽입삭제는 모두 top에서 이루어짐.

```
class Stack{  <stack.h>
 public:
  Stack(int s=100){
   size = s;
  storage = new int[size];
   top = 0; }
bool IsEmpty(){return top==0;};
bool IsFull(){return top==size;};
bool Push(int el);
 int Pop();  int Top;
private:
 int * storage; int top; int size; }
```

```
#include stack.h → 삽입
bool Stack::Push(int el){
  if(IsFull()) return false;
 storage [top++] = el;
 return true;  }
int Stack::Pop(){ → 삭제.
  if (IsEmpty()) return 0;
  return storage [--top]; }
 int Stack :: Top(){
  if(IsEmpty()) return 0;
  return storage[top-1]; }
```

top==0이면 삭제할 원소가 없는 경우

top →



- Stack의 구현 (pointer)

<node.h> 큐와동일.

```
Class Stack{  <stack.h>
public: Stack();
 bool IsEmpty();
 bool IsFull();
 bool Push (int el);
 int PoP ();
 int Top();
private: node *top; }
```

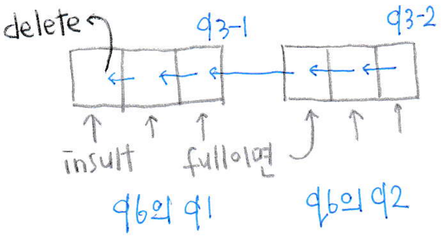
```
#include stack.h ⊕ V   <stack.cpp>
void Stack::Stack(){ top=0; }
void Stack::IsEmpty(){ return (top==0); }
void stack::IsFull(){ return false; }
bool Stack::Push(int el){
 if (IsFull ()) return false;
 top = new node (el, top); return true; }
int Stack::PoP(){
 if(IsEmpty ()) return 0;
 node * ptr = top;  top=top→next;
 int el = ptr→data;  delete ptr;
 return el; }
int Stack ::Top(){
if(IsEmpty()) return 0; return top→data; }
```

③ 큐의 응용 - 과제.

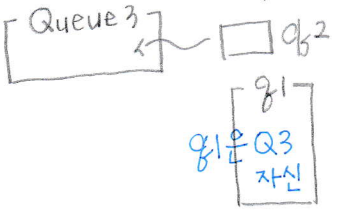
```
class queue3{
public:  queue3(){ rear=0; }
bool IsEmpty() { return (rear==0); };
bool IsFull() { return (rear==3); };
bool InsertQ(int el){
 if(rear == 3) return false;
 arr[rear++] = el; return true; }
int DeleteQ(){
 if (rear==0) return -1;
 int el=arr[0]; rear--;
for(int i=0; i<rear; i++)
 arr[i]=arr[i+1];  return el; }
 Private:  int arr[3]; int rear; };
```

```
queue6::queue6(){ q2 = new queue3();}
bool queue6::IsEmpty(){ return q1.IsEmpty();}
bool queue6::IsFull(){ return q1.IsFull(); }
bool queue6::InsertQ (int el) {
 if(q1.IsFull()) return q2→InsertQ(el);
 q1.InsertQ(el); return true; }
int queue6:: DeleteQ(){
if(q2→IsEmpty()){ return q1.DeleteQ();}
else{
 int i=q1.DeleteQ();
 q1. InsertQ(q2→DeleteQ());
return i; }}
```

큐3을만들고 주소를준다.



queue6



Queue6

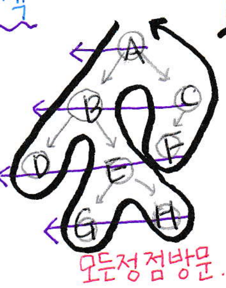
```
q2=new queue3();
q1. IsEmpty();
q2 → IsEmpty();
```

④ 트리. 이진트리 : 최대 2개의 child node를 가진다.

Binary Search Tree. ↗ 임의의노드기준 · 왼(자기보다작은값) / 오(큰값)

BFS : 큐를써서모든노드방문. → 너비우선탐색

```
void binaryTree :: BFS(){
 queue q;
 q.InsertQ(root_);
while(!q.IsEmpty()){
node* ptr = q.DeleteQ();
visit(ptr);
if(ptr→lc_!=0) q.InsertQ(ptr→lc_);
if(ptr→rc_!=0) q.InsertQ(ptr→rc_);
}  };
```



모든정점방문.

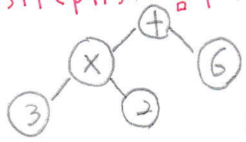
DFS : Stack(재귀함수)를써서모든노드방문. → 깊이우선탐색.

```
void binaryTree :: DFS(node*ptr){
 if(ptr==0) return;
 DFS(ptr→lc_);
 DFS(ptr→rc_); };
```

(pre / In / Post) Order Traversal(node*ptr) 전위 · 중위 · 후위탐색. ⇝ Stack을 이용한 재귀.

'visit(ptr);' 함수의위치에 따라 기능이 다르다.

```
 if(ptr==0) return; <
(/ /) Order Traversal(ptr→lc_); <
          "           (ptr→rc_); <
```

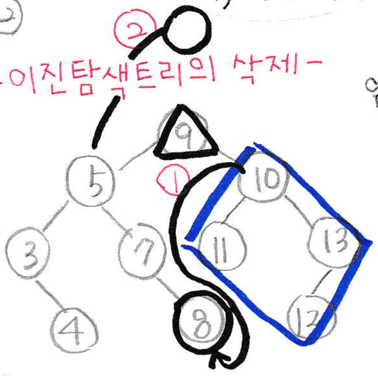


+ * 3 2 6
3 * 2 + 6

- 이진탐색트리의 삽입 -

```
void bst :: Insert(int el){
 if(root_==0) root_=new node(el);
 else Insert(root_, el); }
void bst :: Insert(node* ptr, int el){
 if(ptr→d_ == el) return;
 else if(ptr→d_→ el){
 if(ptr→l_==0)ptr→l_=new node(el, ptr);
 else Insert(ptr→l_, el);
 } else {  ──  l을 r로바꾸고 동일 } }
```

- 이진탐색트리의 삭제 -



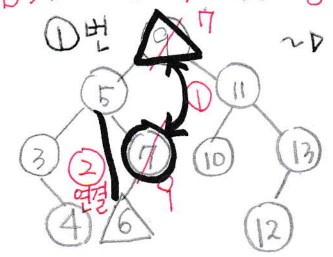
임의의노드 9의 right subtree 전체를 왼쪽으로 1번 가서 오른쪽으로 끝까지 간것(preorder)에 붙임.

⇝ 9삭제

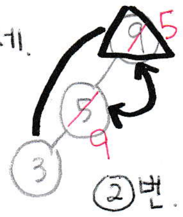
a) Delete By Merging
: 어떤임의의노드를삭제하고 만들어진 subtree의 루트를 돌려주는일. → 9를삭제하고 5를돌려주는일.

b) DeleteByCopying : preorder 7과 9 교체.

①번



⇝ 9삭제



②번.

```
a) node * BST :: DeleteByMerging(node*ptr){
 if(ptr→l_==0 && ptr→r_==0) return 0;
 if(ptr→l_==0) return ptr→r_;
 if(ptr→r_==0) return ptr→l_;
 node * tptr = ptr→l_;
 while(tptr→r_!=0) tptr = tptr→r_;
 tptr→r_ = ptr→r_;
 return ptr→l_; }
b) node * BST :: DeleteByCopying(node*ptr){
 ⊕ ptr→d_ = tptr→d_;
 if(tptr→p_==ptr) ptr→l_=tptr→l;
 else tptr→p_→r_ = tptr→l_;
 return ptr; }
```

```
bool BST :: Delete(int el){ → 실제삭제
 node*ptr = Search(el);
 if(ptr==0) return false;
 if(ptr→p_→d_ > ptr→d_)
 ptr→P_→l_ = DeleteByMerging(ptr);
 else ptr→P_→r_ =   "
 delete ptr;
 return true; }
```

⊕ ↗ ↘

→ 이진트리!!

```
int binaryTree :: Height(node*ptr){
 if(ptr==0) return 0;        else return hr+1;
 int hl = Height(ptr→lc_);   };
 int hr =   "   (ptr→rc_);
 if(hl>hr) return hl+1;
```

```
bool binaryTree :: Search(int el){
 if(Search(root_, el)==0) return false;
 return true;
};
```

~▷ 순서를 갖는 원소들의 집합.

⑤ 우선순위큐와 힙정렬.

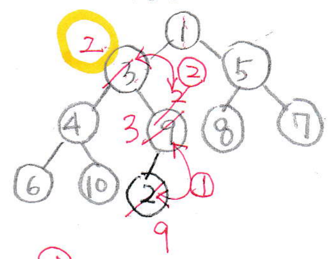
스택 : 삽입된 시간이 늦을수록 우선순위가 높다.

최대 우선순위를 갖는 원소를삭제 → 큐 : 〃 빠를수록 〃 높다.

– 우선순위큐의 구현 (힙) – (cbt)

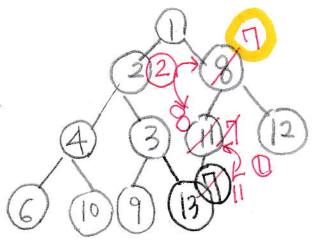
최대힙 (max heap) : 완전이진트리 . 자식노드값 < 부모노드값. / 최소힙 (min heap) : 자식 > 부모.

마지막줄을제외하고 모든노드가 꽉차있어야한다. 마지막은 순서대로 ~!

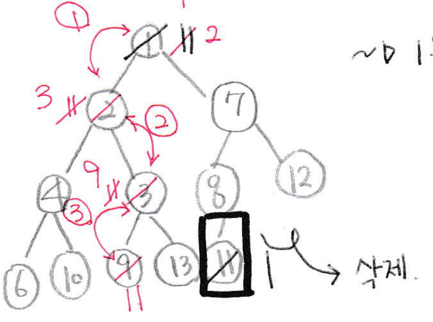


~▷ 2를 삽입!

⇒ parent와 비교하여 머무른다.



~▷ 13과 7을 삽입!



~▷ 1을 삭제!

== 완전이진트리를 배열로 저장했을때 ==

0 ~ n번까지 배열에 저장 – 표현

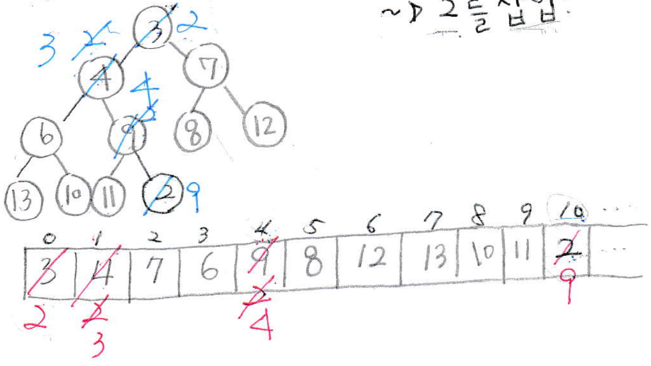
(i−1)/2 : 자신의 parent (i의 parent)

(i×2)+1 : left child

(i×2)+2 : right child.

< minheap에 삽입 >

~▷ 2를 삽입!

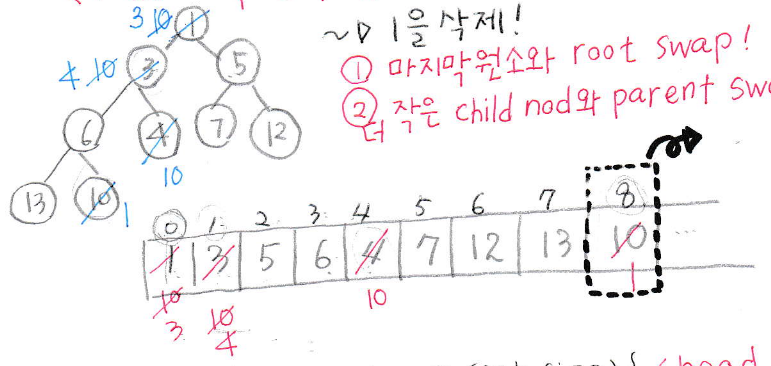


< minheap에 삭제 >

~▷ 1을 삭제!

① 마지막원소와 root swap!

② 더 작은 child nod와 parent swap!



```
class minheap{  <heap.h>
public: minheap(){minheap(100); };
 minheap (int size);
 ~minheap ();
 bool IsEmpty ();
 bool IsFull ();
 void InsertMH (int el);
  int DeleteMH ();
protected:                cul
   Void UpHeap(int ~~el~~);
    void DownHeap();
private:
   int hsize_; // 힙의최대크기
   int last_;  // 현재힙의크기.
   int * storage_; };
```

```
  minheap:: minheap (int size){ <head.cpp>
    hsize_ = size;
    last_ = 0;   Storage_ = new int[100]; };
  minheap:: ~minheap:: (){ delete storage_;};
 bool minheap:: IsEmpty(){ return last_==0; };
 bool minheap:: IsFull(){ return last_==hsize_;};
void minheap::InsertMH (int el) {
   storage_[last_++] = el;
   UpHeap (last-1); };
void minheap:: UpHeap (int cur){
   int par =(cur-1)/2;
          부모
   while((cur!=0) && (storage_[cur]< storage
                                        _[par]))
  {swap (cur, par);
   cur = par;   par=(cur-1)/2; } }
 int  minheap:: DeleteMH (){
                                   배열에
   int el = storage_[0];         0번째인자
   storage_[0] = storage_[--last_];
   Down Heap(0);        바꾸고.
   return el; };
```

```
void minheap:: DownHeap(int cur){
while ((cur*2+1 < last_){
 int mc = cur*2+1; //mc = lc;
       lc
if (cur*2+2 < last_) && (Storage_[mc] > Storage_[cur*2+2]))
   mc = cur*2+2   // mc = rc;
    rc
if(Storage_[cur] < Storage_[mc]) break;
 Swap(cur, mc);    cur=mc; }  };
```

⑥ 예년문제.

* 자료구조란 무엇인가? 추상자료형 : 객체의 명세와 그 연산의 명세가 객체의 표현과 연산의 구현으로부터 분리된 자료형.

리스트 · 큐 · 스택 · 그래프 · 트리 · 사전기계.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
초기상태	~~3~~	4	~~5~~	6	7	8	~~9~~	10	11	12	13	14	15	16	~~0~~	0
	1 el		~~1~~ 3				~~1~~ 5								~~1~~ 9	

① insert(1)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Insert(1)후	1	~~4~~	3	~~6~~	7	8	5	~~10~~	11	12	13	14	15	16	9	~~0~~ ~~2~~ 10
		2		~~2~~ 4				~~2~~ 6								

② insert(2)

Insert(2)후	1	2	3	4	7	8	5	6	11	12	13	14	15	16	9	10
Delete()후①	4	6	5	10	7	8	9	16	11	12	13	14	15			
Delete()후②	5	6	8	10	7	14	9	16	11	12	13	15				

-BST의 Search-

```
bool bst:: Search(int el){
 if(Search(root_, el)!=0)
  return true;
 else return false; }

node*bst :: Search(node *ptr, int el){
 if(ptr == 0) return 0;
 if(ptr→d_ == el) return ptr;
 else if(ptr→d_ >el) return Search(ptr→l_, el);
else return Search(ptr→r_,el);
 }
```

-최대·최소-

```
Maxn (node* ptr){
  if(ptr==0) return 0;
 Static int max = ptr→data_;
if(max <ptr→data_)max =ptr→data;
 Maxn(ptr→lc_);
 Maxn(ptr→rc_);
```

Minn은 Max와반대~!

```
     int BTS :: Sum(node*ptr){
     Static  int S;
   if(ptr==o) return 0;
int x = sum(ptr→lc)
int y = sum( ptr→rc)
    S = x+y     return S;
```

- Height와 numberof nodes -

```
Height( node * ptr){ <트리의높이>
* if (ptr == 0) return 0;
int hl = Height(ptr→lc_);자신과
int hr = Height(ptr→ rc_); 똑같은것을호출
if(hl>hr) return hl+1; 둘중큰값에(+1) };
else return hr+1; };
```

```
Numberof nodes(node *ptr){ <나를기준으로하는노드의수>
*⊕ int nl = NumberofNodes (ptr→lc_);
   nr          "                →rc_);
 return nl+nr(+1);
```