**———————————————————————————————————————————————————**

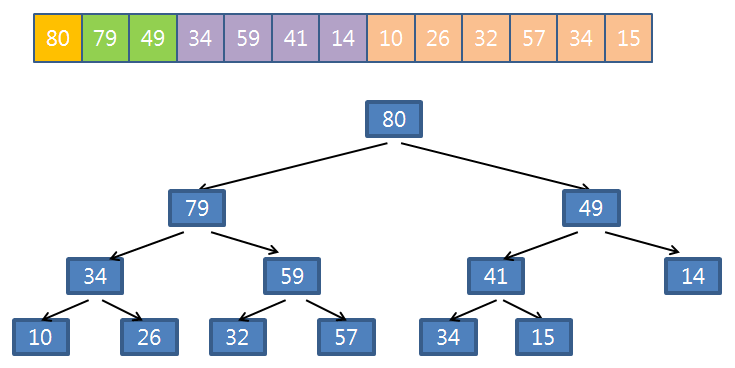
[1] 기초 문제

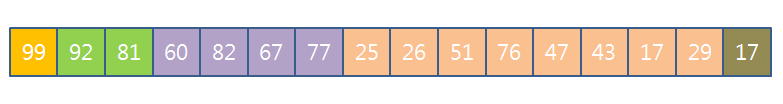
———————————————————————————————————————————————————

1. **우선 순위 큐 용어와 관련된 문제다. 빈 칸에 들어가는 단어를 맞춰보도록 한다.**
   1. ① 히프 ② 최대 또는 최소 ③ 완전 이진 트리 ④ 중복 ⑤ 정렬
   2. ① 완전 이진 트리 ② 루트 ③ 단말 ④ 꽉 찬 ⑤ 배열
2. **우선 순위 큐를 사용하고 있는 곳을 세 가지씩만 나열해 보자.**
   1. PB, 승진, TODO, 긴급 차량, 응급실
   2. 메시지 큐, 히프 정렬, 스레드 스케줄링, 네트워크 관리 패킷, 랭킹

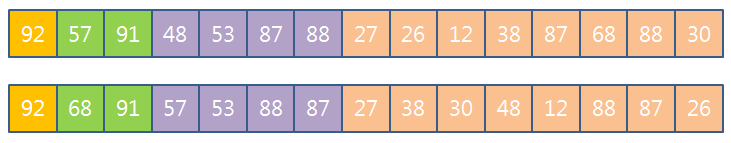
|  |
| --- |
| 해답  정답으로 간주할 수 있는 것들이 너무 많아서 이번 장에서 설명한 것만 실었습니다. |

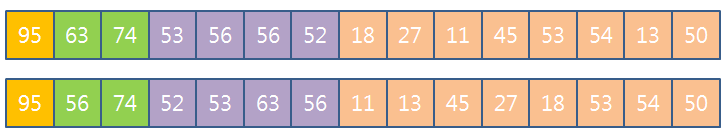
1. **배열을 이용한 완전 이진 트리를 구성해서 히프 자료 구조를 구현했다고 가정하자. 그림을 통해 물리적인 구조와 논리적인 구조의 연관 관계를 이해했는지 확인해 본다.**





1. **15개의 요소를 갖는 난수 배열이 있다. 이들 배열을 히프의 정의에 맞게 재배치 한다. 히프는 느슨한 정렬을 지원하기 때문에, 각각 2가지 이상의 방법으로 배치한다. 저자가 친절하게도 빈 칸까지 만들어 놓았다. 잘 하자!**





1. **본문에 나왔던 코드에 대해 설명을 하는 곳이다. 우선 순위 큐를 배우려는 친구에게 설명한다는 생각으로 차분하게 기술해 보자.**

|  |
| --- |
| 해답  자세한 설명이 2장의 심화 학습에 있습니다. 설명이 너무 길어 답안은 생략했습니다. |

**———————————————————————————————————————————————————**

[2] 기본 실습 문제

———————————————————————————————————————————————————

1. **우선 순위 큐에서 나올 수 있는 가벼운 코드들이다. 직접 구현해 보도록 하자.**
   1. 완전 이진 트리이므로 시프트 연산으로 쉽게 구할 수 있습니다.

int getLevelSize(PRIOR\_QUEUE\* pq, int level)

{

return 1 << level;

}

* 1. 앞에서 만든 getLevelSize 함수를 사용했습니다. 구하려는 계층이 우선 순위 큐에 존재하지 않으면 -1을 반환합니다. count-1은 현재 계층의 시작 위치가 되고, count\*2-1은 트리가 가질 수 있는 노드의 최대 개수를 의미합니다. 4계층이라면, 4계층에 포함될 수 있는 최대 개수는 8이 되고 노드의 전체 개수는 최대 15(8\*2-1)가 됩니다. 검증을 위해 검사하게 되는 요소들을 출력했습니다.

int getMaxInLevel(PRIOR\_QUEUE\* pq, int level)

{

int i, max, end,

count = getLevelSize(pq, level);

if(count >= pq->size)

return -1;

end = (pq->size <= count\*2-1) ? pq->size : count\*2-1;

max = pq->array[count-1];

printf("%d ", max);

for(i = count; i < end; i++)

{

printf("%d ", pq->array[i]);

if( max < pq->array[i])

max = pq->array[i];

}

printf("\n");

return max;

}

1. **우선 순위 큐에서 나올 수 있는 조금 무거운 코드들이다. 직접 구현해 보도록 하자.**
   1. 단말 노드는 전체 노드의 절반에 해당하므로 배열의 뒷부분 절반만 반복합니다. 정확하게 하는 방법으로는 마지막 노드(size-1)의 부모 노드 또는 부모 노드+1부터 반복하지만, 쉽게 처리하기 위해 크기/2를 시작 위치로 삼았습니다. 당연히 잘 동작합니다. 여기서 필요한 건 부모 노드가 아니라 단말 노드의 처음이니까요. 이렇게 하면 크기가 1일 때 별도로 처리할 필요가 없습니다.

#include <limits.h>

int getMin(PRIOR\_QUEUE\* pq)

{

if(pq->size > 0)

{

int i, min = pq->array[pq->size/2];

for(i = pq->size/2+1; i < pq->size; i++)

{

if( min > pq->array[i])

min = pq->array[i];

}

return min;

}

return INT\_MIN;

}

* 1. 정렬되지 않았기 때문에 배열 전체를 순차 탐색했습니다.

int\* findData(PRIOR\_QUEUE\* pq, int data)

{

int i;

for(i = 0; i < pq->size; i++)

{

if(pq->array[i] == data)

return pq->array+i;

}

return NULL;

}

느슨하게 정렬되었다는 사실을 이용하기 위해 재귀 함수를 사용합니다. 하위 계층으로 내려가는 도중, 찾으려는 숫자보다 작은 숫자를 만나면 그만 찾도록 꾸몄습니다. 그런데, 코드의 복잡함은 예외로 두고, 앞에 있는 순차 탐색과 성능에서 많은 차이가 나지 않습니다. 느슨하게 정렬되었기 때문에 큰 숫자는 앞에 있으니까 결국 순차 탐색에서도 일부만 반복합니다. 함수 호출과 분기에 대한 비용을 지불하고 나면 오히려 성능이 떨어질 수도 있을 것 같습니다. 그러나, 존재하지 않는 요소를 찾는다면 그땐 재귀 함수가 나을 수도 있어 보입니다.

int\* findDataRec(PRIOR\_QUEUE\* pq, int data, int cur)

{

int\* p;

if(cur >= pq->size || pq->array[cur] < data)

return NULL;

printf("cur[%d] %d\n", cur, pq->array[cur]);

if(pq->array[cur] == data)

return pq->array+cur;

if(p = findDataRec(pq, data, cur\*2+1))

return p;

return findDataRec(pq, data, cur\*2+2);

}

int\* findDataWrapper(PRIOR\_QUEUE\* pq, int data)

{

return findDataRec(pq, data, 0);

}

1. **저자는 여러분이 복잡한 코드까지 직접 구현하기를 바라지는 않는다. 대신 비어있는 곳의 코드를 채울 정도는 되어야 한다고 생각한다. 빈 칸에 들어가는 코드는 무엇인가?**
   1. ① if(cur < pq->size)

② printShape(pq, cur\*2+2, level+1);

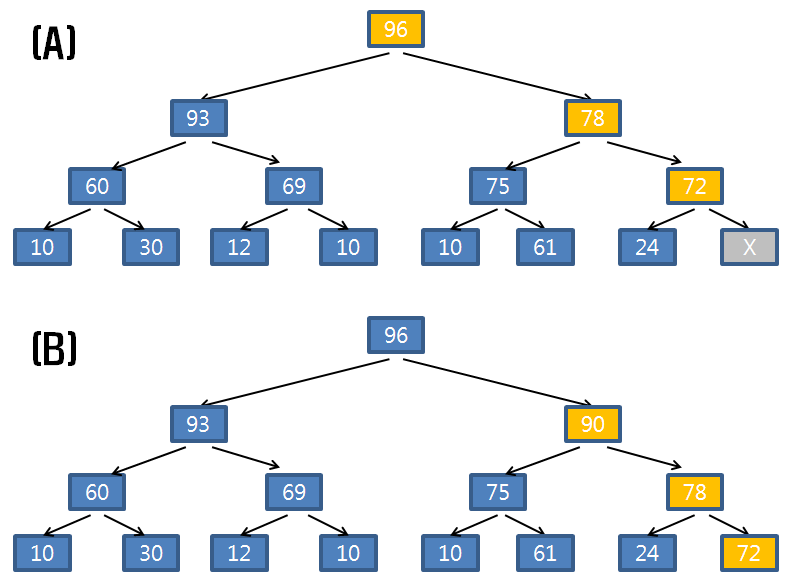
③ printShape(pq, cur\*2+1, level+1);

* 1. ① if(left >= size)

② if(array[parent] >= array[max])

③ parent = max;

1. **15개의 요소를 갖는 히프 자료 구조가 있다. 히프에 들어있는 데이터가 아래와 같을 때, 문제를 해결하고 난 후의 트리 모습을 그려보자. 저자가 바로 입력할 있도록 빈 칸으로만 구성된 트리를 그려놨다. ^^**



**———————————————————————————————————————————————————**

[3] 응용 실습 문제

———————————————————————————————————————————————————

1. 배열 범위에 존재하는지, 단말 노드인지, 자식은 어떻게 되는지 등을 회피하기 위해 무조건 자식이 존재하는 걸로 가정하고, 없을 경우 INT\_MIN을 넣었습니다.

#include <limits.h>

int able2Swap(PRIOR\_QUEUE\* pq, int pos1, int pos2)

{

if(pos1 >= 0 && pos1 < pq->size && pos2 >= 0 && pos2 < pq->size)

{

int left1, right1, left2, right2;

left1 = (pos1\*2+1 < pq->size) ? pq->array[pos1\*2+1] : INT\_MIN;

right1 = (pos1\*2+2 < pq->size) ? pq->array[pos1\*2+2] : INT\_MIN;

left2 = (pos2\*2+1 < pq->size) ? pq->array[pos2\*2+1] : INT\_MIN;

right2 = (pos2\*2+2 < pq->size) ? pq->array[pos2\*2+2] : INT\_MIN;

printf("pos1 : %d %d %d\n", pq->array[pos1], left1, right1);

printf("pos2 : %d %d %d\n", pq->array[pos2], left2, right2);

if( pq->array[pos2] >= left1 && pq->array[pos2] >= right1 &&

pq->array[pos1] >= left2 && pq->array[pos1] >= right2)

return 1;

}

return 0;

}

1. 동적으로 메모리를 할당하는 방법과 자체 메모리만으로 해결하는 두 가지 방식으로 구현해 봤습니다. 두 가지 방법 모두 히프를 구성하기 위해 만든 함수들을 사용합니다. 다만 두 번째 버전은 히프의 성질을 만족시키기는 하지만, 원본과 동일한 구조는 아닙니다. 문제에서는 원본 배열을 그대로 유지하자고 했기 때문에, 엄밀하게 말하면 첫 번째만 답이 됩니다. 참, 히프 구조에 대한 문제이므로 우선 순위 큐 구조체를 사용하지 않았습니다.

#include <stdlib.h>

#include <string.h>

void reheapDown(int\* array, int size, int parent);

void reheapUp(int\* array, int child);

void swap(int\* p1, int\* p2);

// 동적 할당 버전

void printBiggie1(int\* array, int size, int count)

{

int i;

int\* backup = malloc(sizeof(int)\*size);

memcpy(backup, array, sizeof(int)\*size);

if( count > size)

count = size;

for(i = size-1; i >= size-count; i--)

{

printf("%d ", array[0]);

// backup에 저장했기 때문에 교환할 필요가 없습니다.

array[0] = array[i];

reheapDown(array, i, 0);

}

printf("\n");

memcpy(array, backup, sizeof(int)\*size);

free(backup);

}

// 자체 메모리 버전

void printBiggie2(int\* array, int size, int count)

{

int i;

if( count > size)

count = size;

for(i = size-1; i >= size-count; i--)

{

printf("%d ", array[0]);

// 변수 교환

swap(array, array+i);

reheapDown(array, i, 0);

}

printf("\n");

// 반대로 히프에 속하지 않은 요소부터 히프를 구성합니다.

for(i = size-count; i < size; i++)

reheapUp(array, i);

}

1. 히프에서 다루었던 reheapDown과 swap 함수를 호출하지 않고 붙여 넣었습니다. 문제에서 복습의 의미로 직접 구현하자고 했었습니다. 역시 연습 문제에서 나온 findData 함수도 직접 구현해서 넣었습니다.

int deleteData(PRIOR\_QUEUE\* pq, int data)

{

int parent, left, right, max, temp;

// 찾기

for(parent = 0; parent < pq->size; parent++)

{

if(pq->array[parent] == data)

break;

}

if(parent >= pq->size)

return 0;

// 히프 크기 감소

pq->size--;

pq->array[parent] = pq->array[pq->size];

// reheapDown()

while(1)

{

left = parent\*2 + 1;

right = parent\*2 + 2;

if(left >= pq->size)

break;

if(left == pq->size-1) max = left;

else if(pq->array[left] > pq->array[right]) max = left;

else max = right;

if(pq->array[parent] >= pq->array[max])

break;

// swap()

temp = pq->array[parent];

pq->array[parent] = pq->array[max];

pq->array[max] = temp;

parent = max;

}

return 1;

}