# 5.6 Heaps

5.6.1 simple heap

heap은 priority queues를 구현하는 데 가장 빈번히 사용된다. priority queue에서 우선 순위가 가장 높은 (또는 가장 낮은) element가 삭제된다. 임의 priority를 갖는 element가 queue에 삽입되는 데 queue의 rear에 들어가지 않고 priority에 따라 처음 또는 중간에 삽입이 된다. 이와 같이 first come, high priority serve를 지원하는 자료구조를 max( min) priority queue라고 부른다. Priority queue를 abstract class로 정의한 다음에 class MaxHeap 또는 class MinHeap를 subclass로 구현한다.

소스코드 5.4: Heap

// 5.6 heap

#include <iostream>

//simple queue에 대한 성능 비교 실험, 난수를 사용하여 시간 측정

//MinPQ를 구현

using namespace std;

const int HeapSize = 1000;

enum Boolean { FALSE, TRUE };

template <class Type>

class Element {

public:

Type key;

Element() {}

Element(Type key) :key(key) {}

};

template <class Type>

class MaxPQ {

public:

virtual void Insert(const Element<Type>&) = 0;

virtual Element<Type>\* DeleteMax(Element<Type>&) = 0;

};

template <class Type>

class MaxHeap : public MaxPQ<Type> {

public:

MaxHeap(int sz = HeapSize)

{

MaxSize = sz; n = 0;

heap = new Element<Type>[MaxSize + 1]; // Don't want to use heap[0]

};

void display();

void Insert(const Element<Type>& x);

Element<Type>\* DeleteMax(Element<Type>&);

private:

Element<Type>\* heap;

int n; // current size of MaxHeap

int MaxSize; // Maximum allowable size of MaxHeap

void HeapEmpty() { cout << "Heap Empty" << "\n"; };

void HeapFull() { cout << "Heap Full"; };

};

template <class Type>

void MaxHeap<Type>::display()

{

int i;

cout << "MaxHeap:: (i, heap[i].key): ";

for (i = 1; i <= n; i++) cout << "(" << i << ", " << heap[i].key << ") ";

cout << "\n";

}

template <class Type>

void MaxHeap<Type>::Insert(const Element<Type>& x)

{

int i;

if (n == MaxSize) { HeapFull(); return; }

n++;

for (i = n; 1; ) {

if (i == 1) break; // at root

if (x.key <= heap[i / 2].key) break;

// move from parent to i

heap[i] = heap[i / 2];

i /= 2;

}

heap[i] = x;

}

template <class Type>

Element<Type>\* MaxHeap<Type>::DeleteMax(Element<Type>& x)

{

int i, j;

if (!n) { HeapEmpty(); return 0; }

x = heap[1]; Element<Type> k = heap[n]; n--;

for (i = 1, j = 2; j <= n; )

{

if (j < n) if (heap[j].key < heap[j + 1].key) j++;

// j points to the larger child

if (k.key >= heap[j].key) break;

heap[i] = heap[j];

i = j; j \*= 2;

}

heap[i] = k;

return &x;

}

int main() {

int select = 0;

int data = 0;

MaxHeap<int> heap(20);

Element<int> ele;

Element<int>\* deletedEle = nullptr;

do {

cout << endl << "Max Tree. Select: 1 insert, 2 display, 3 delete, >=5 exit" << endl;

cin >> select;

switch (select) {

case 1:

cout << "input value: ";

cin >> data;

heap.Insert(Element<int>(data));

heap.display();

break;

case 2:

heap.display();

break;

case 3:

deletedEle = heap.DeleteMax(ele);

if (deletedEle != nullptr) {

cout << "deleted element: " << deletedEle->key << endl;

}

cout << "current max heap: " << endl;

heap.display();

break;

case 5:

exit(1);

break;

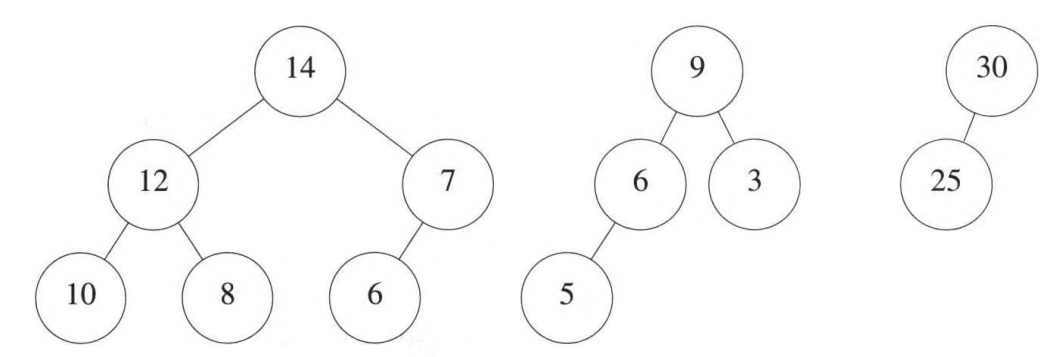
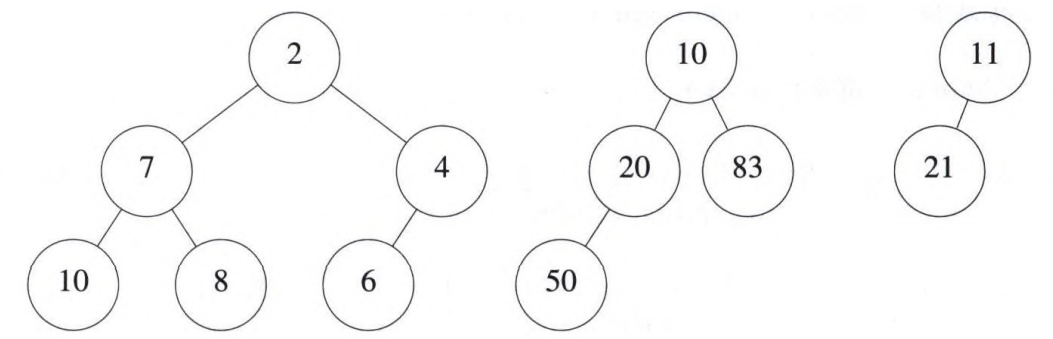
}

} while (select < 5);

return 0;

}

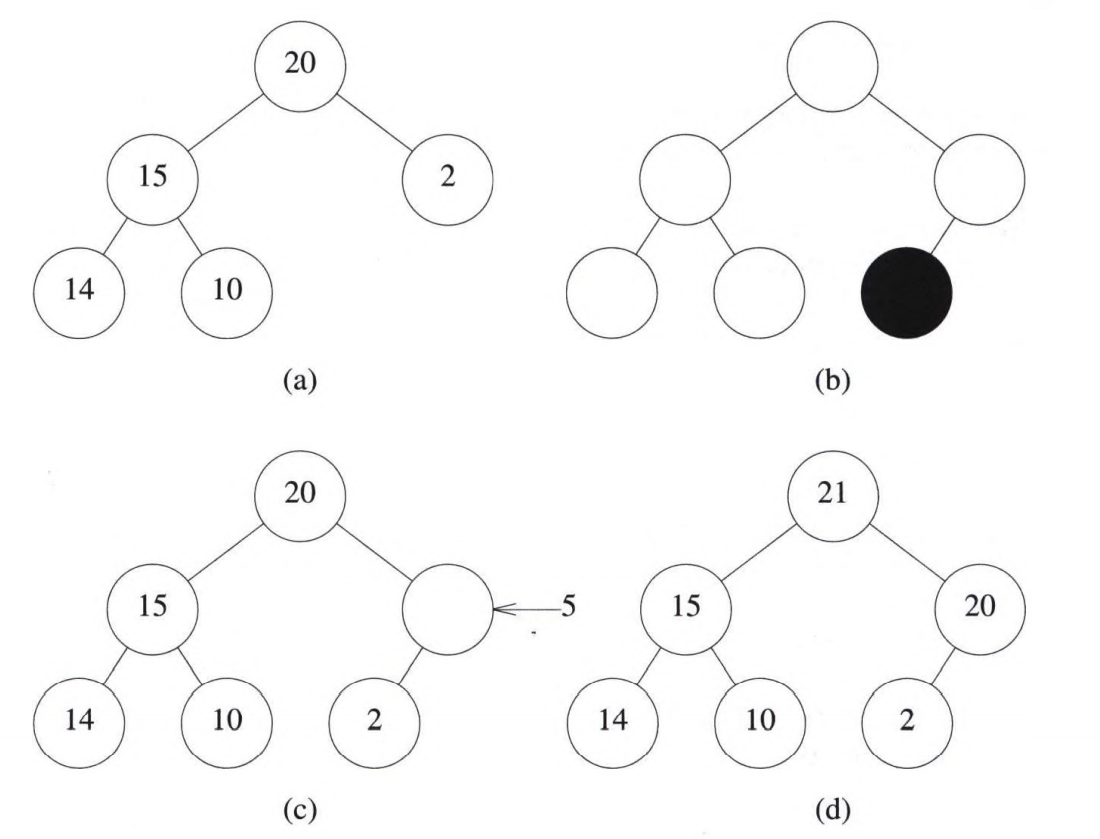
MaxPQ는 다양한 data structures로 구현할 수 있다. unordered list로서 구현할 수도 있고 ordered list로 구현할 수 있다. unordered linear list로 MaxPQ를 구현할 경우에 list의 rear에 insert는 O(1)이며 largest key를 갖는 element를 찾는 삭제는 O(n)이다. abstract class MaxPQ의 sublcass로 구현할 수 있다. ordered linear list를 사용하여 MaxPQ를 구현하면 delete time은 O(1)이며 insert time은 O(n)이 된다. abstract class MaxPQ의 sublcass로 ordered linear list로 구현할 수 있다. ordered list 대신에 max heap 사용하면 insert, delete 모두 O(log n)으로 처리할 수 있다. log n인 이유는 heap이 complete binary tree의 depth이기 때문이다.

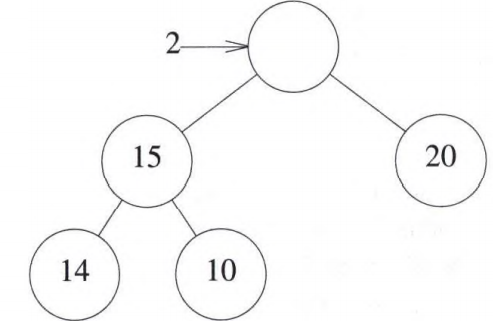
max heap은 앞의 그림[1]처럼 각 노드의 key 값이 child보다 항상 큰 tree로 정의된다. min heap은 각 노드의 key 값이 child보다 항상 작은 tree로 정의된다. max(min) heap은 배열로 표현되는 complete binary tree이며 insert는 배열의 마지막에, delete는 root에서 처리된다. heap에 insert, delete를 한 후에 heap의 조건(child의 key 값이 parent보다 작아야 한다는 조건)을 만족하기 위해서 heap의 insert(), delete()알고리즘의 구현이 필요하다.

max heap에 삽입은 다음 그림[1]처럼 heap을 표현한 배열의 마지막 위치에 저장한다. heap에 삽입은 complete binary tree에 insert한 후에도 complete binary tree가 유지되어야 한다. heap에 n개가 있을 때 새로 삽입될 위치는 heap[n++]이다. 현재 heap이 (20, 15, 2, 14, 10)일 때 5를 insert한 결과가 (c)이고, 21을 insert한 결과가 (d)이다. heap insertion의 구현은 삽입되는 element 부터 parent로 이동하면서 크기를 비교하여 insert되는 키가 크면 parent와 맞바꾼다.

heap의 삭제는 항상 root에서 발생하며 root가 비어 있게 되므로 heap[n]이 배열의 마지막 element이므로 heap[n]을 root에 이동한다. 다음에 root는 child보다 항상 커야 하므로 child가 크면 현재 root와 맞바꾼다. 이 과정을 leaf node까지 반복한다.



현재 heap[] = {21, 15, 20, 14, 10, 2}일 때 root인 21를 삭제하면 heap[6] = 2가 root로 이동하게 되고 20이 크므로 20이 root가 되고 2는 다음 그림[1]처럼 root의 우측 child로 가게 된다. 삭제한 후에 heap[] = {20, 15, 2, 14, 10}이 된다.



난수로 k개의 정수를 입력받아 배열 data [] = {12, 3, 5, 19, 22, 14, 21}이 있을 때 새로운 배열을 만들지 않고 제자리에서(in-place라는 의미로 data[]만을 사용하는 것을 말한다) max heap을 만드는 알고리즘을 구현한다. heap에서 최대값을 delete하여 sort된 결과를 display하는 알고리즘을 구현한다.

난수를 사용하여 106 ~ 107개의 heap을 만들고 insert, delete 처리 시간과 ordered list로서 sort된 결과를 유지하고 insert후에도 sort 결과를 유지하고, delete후에도 sorted 결과를 유지하기 위한 자료구조를 제안하고 구현후 처리 시간을 측정한다.