### Heaps

（2a）在每个函数中添加一个含H的断言。

void pq\_insert(heap H, elem e)

//@requires is\_heap(H) && !pq\_full(H);

//@ensures is\_heap(H);

{

H->data[H->next] = e;

(H->next)++;

//@assert **!pq\_full(H)**;

int i = H->next - 1;

while (i > 1 && priority(H,i) < priority(H,i/2))

//@loop\_invariant 1 <= i && i < H->next;

//@loop\_invariant is\_heap\_except\_up(H, i);

{

swap(H->data, i, i/2);

i = i/2;

}

//@assert is\_heap(H);

return;

}

elem pq\_delmin(heap H)

//@requires is\_heap(H) && !pq\_empty(H);

//@ensures is\_heap(H);

{

int n = H->next;

elem min = H->data[1];

H->data[1] = H->data[n-1];

H->next = n-1;

if (H->next > 1) {

//@assert **is­­\_heap(H) && !pq\_empty(H)**;

sift\_down(H, 1);

}

return min;

}

（2b） 完成函数pq\_max，该函数从基于数组的最小堆中返回但不删除具有最大priority value的元素,同时只检查那些可能包含最大值的元素。

elem pq\_max(heap H)

//@requires is\_heap(H) && !pq\_empty(H);

//@ensures is\_heap(H);

{

int max = **0 ;**

for (int i = **0** ; i < **H->limit** ; i++)

if (priority(H, i) > priority(H, max)) max = i;

return **H->data[max]** ;

}

（1c）pq\_build接受一个数据元素数组（忽略数组的索引0），并构建基于数组的最小堆。也就是说，它在我们的堆结构中使用给定的数组，并且不分配一个新的数组。

heap pq\_build(elem[] elements, int arraylength)

//@requires \length(elements) > 0 && \length(elements)==arraylength;

//@ensures is\_heap(\result);

{

heap H = alloc(struct heap\_header);

H->limit = arraylength;

H->next = 1;

H->data = elements;

for (int i = 1; i < arraylength; i++)

pq\_insert(H, elements[i]);

return H;

}

完成以下代码，以便pq\_empty函数会因为不满足前提条件而提前退出。

**:**

**//@assert \length(E) = 16;**

**heap H = pq\_build(E, 16);**

**:**

**if( pq\_empty(H) ) return;**

### Priority Queues as an Abstract Data Type

（2a） 优先级队列什么时候会像堆栈一样运行？

对于基于队列实现的堆：因为队列的特点时先进先出，可以用最小优先队列来实现，设置插入元素的优先级属性就用基于最小堆的优先级队列来实现，因为一个元素插入的越晚，他的priority value的值越大，在这一点上，最小堆的优先级队列和队列是保持一致的，这样子就可以实现在队列中插入元素的同时，通过增量设置的方法使插入元素的priority value大于前一个元素，当出队的时候，extract\_min()就可以把最先进入队列的元素出队。

对于基于栈实现的堆：因为栈的特点是后进先出，所以我们将入栈的第一个元素的priority value的值设置的足够大，这样子在插入元素的过程中，我们采用递减的方法设置插入元素的priority value，这样就可以实现基于栈的优先级队列。

（4b） 回顾优先级队列ADT（抽象数据类型）的客户端和库接口：

假设我们的客户需要处理 代表股票市场报告的 非常长的 stream S：

struct stock\_report {

string company;

int value; // stock value in whole dollars

};

typedef struct stock\_report\* elem;

stream S由由以下两个函数的数据类型流表示：

elem get\_report(stream S);

// 返回数据流中的下一个股票报告

bool stream\_empty(stream S);

// 判空，若空return true

由于stream非常，非常长，我们不知道它最终会有多大，我们不能把所有的股票报告存储在一个非常大的数组中。编写一个客户端函数total\_value，它返回1000个具有最高股票价值的股票报告的总价值。你可能会假设该stream有超过1000份股票报告。您的解决方案必须使用一个优先级队列。你不知道优先级队列是如何实现的，因为你只有它的接口。请确保包含一个库可用于解决此问题的合适elem\_priority函数的C0定义。

int elem\_priority(elem e) {

return  **e->value** ;

}

int total\_value(stream S) {

pq P = pq\_new( **1000** );

while (!stream\_empty(S)) {

// Put the next stock report into the priority queue:

**pq\_insert( P, get\_report(S));**

// If the priority queue has more than 1000 reports,

// delete the report with the smallest value.

if (**pq\_full(P)**)

**pq\_delmin(P);**

}

// Add up the values of all 1000 reports in the priority queue

int total = 0;

while ( **!pq\_empty(P)** ) {

total += **elem\_priority ( pq\_min(P) )** ;

**pq\_delmin(P);**

}

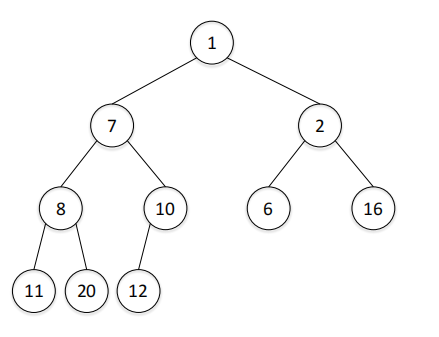
return total;

}

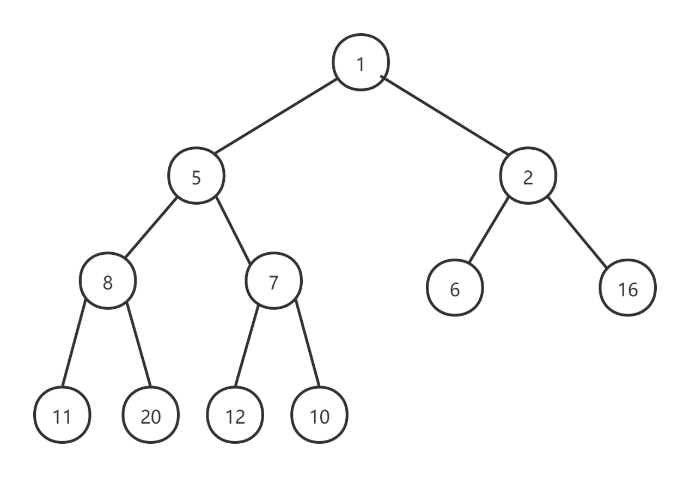
### Heaps

最小堆是一个满足两个数据结构不变量的层次数据结构：

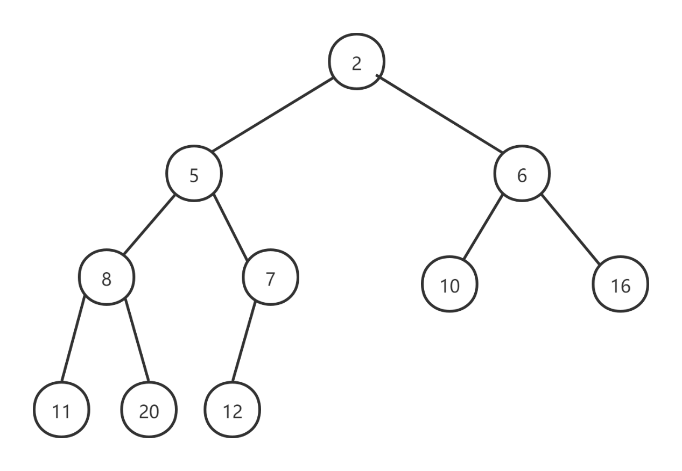
对于每个非根元素，其值都大于或等于其父元素的值。最小堆的每一级(level)都是完全满的，除了最后一个level，它的所有元素尽可能存储在最后一个level。



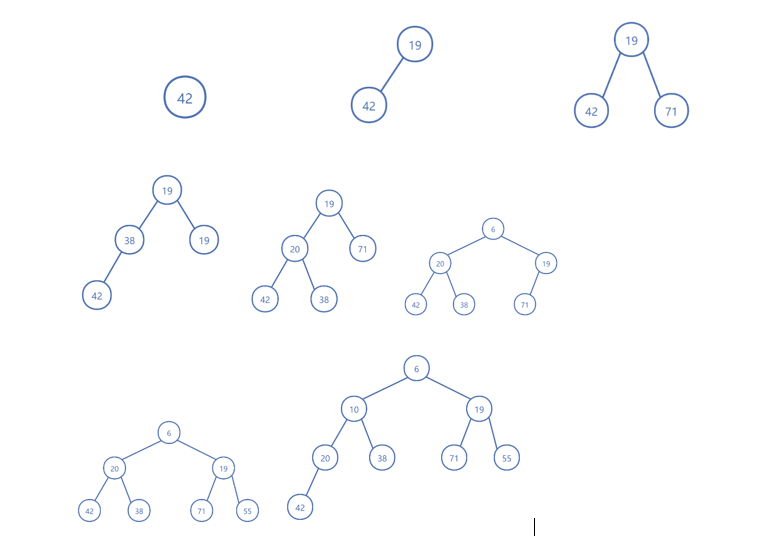
（1a） 在插入一个值为5的元素后，绘制最小堆的最终状态。



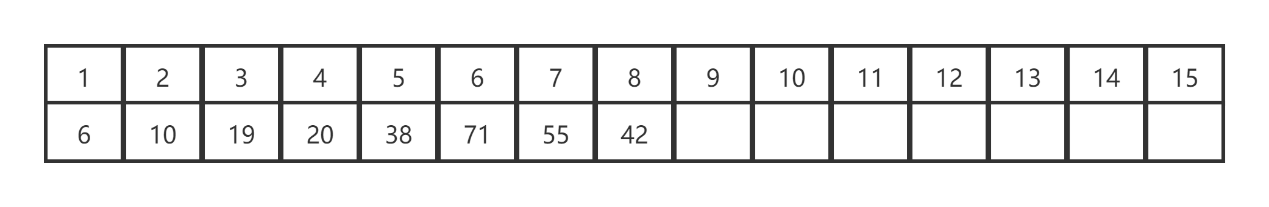
（1b）在删除具有最小值的元素后，绘制最小堆的最终状态。



按照42, 19, 71, 38, 20, 6, 55, 10插入到初始空的最小堆中，绘制最小堆的最终状态。



（1d）使用final mini-heap，显示每个元素将存储在数组中的位置。



（1e）在非空最小堆（作为树，不一定存储为数组）中，最大值必须在哪里？

**假设树高为h，最大值在处于深度为h或者h-1的叶节点**

（1f）如果最小堆有n个元素，那么在最小堆中找到最大值的最坏情况下的运行时复杂度是什么？为什么？

**O(n);对深度为k的堆，调整算法进行的关键字比较次数至多为2(k-1)次**

（1g）我们得到了一个包含n个整数的数组A，我们希望对其进行排序，然后我们决定使用一个最小堆来帮助进行排序。我们的最小堆将使用课堂中描述的数组来表示。对于数组中的每个整数，一次插入一个，我们将该整数插入到最小堆中。然后，我们从最小堆中重复删除（最小值），将每个删除的值存储回数组中，一次一个，从数组的开头开始。这个使用大O符号的排序算法的最坏情况下的运行时复杂度是什么？

**O(nlogn);**

**理由：对于深度为k的堆，调整算法进行的关键字比较次数至多为2(k-1)次；建立n的元素的初始堆，调用调整算法n/2次，进行的关键字比较不超过4n次；需要调整的层为h-1层至1层，以第i层某结点为根的二叉树深度对应为h-i+1,第i层结点最多为2i-1个，故调整时比较关键字最多为：4n；最坏情况下，算法的实践复杂度为O(4n+nlogn)，即O(nlogn)**;