Computer Science

Dr. Y. Liow (August 1, 2020)

Contents

Chapter 109

Heaps and priority queues. Part 1: binary heaps.

`File: chap.tex`

TODO:

- Change max heap to maxheap
- Change min heap to minheap
- Change drawing code

File: priority-queue.tex

109.1 Priority queues

Recall a queue is a data structure that supports enqueue and dequeue (entering and leaving a queue). Think of people lining up at a ticket booth.

A **priority queue** is queue where people join a queue *but* they have priority numbers. So someone with a higher priority can jump the queue by going ahead of someone in front of him/her who has a lower priority. priority queue

Priority queues are very important. For instance modern computers run multiple processes at the "same time", i.e, the OS maintains a queue of processes and allow each process to execute a small time slice before that process pauses and goes back to the end of the queue. But this is not just a regular queue: the OS process queue is a priority queue. Why? Because some processes are more important than others. Try the following ...

Linux allows user to display running processes using the `ps` command: priority of processes. The following is a list of processes that I'm running:

```
[student@localhost n]$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  3148 26588  0  80   0 -  8250 poll_s pts/0    00:07:46 emacs
0 R  1000  7239 26588  0  80   0 -  1253 -      pts/0    00:00:00 ps
0 S  1000 26588 20828  0  80   0 -  1574 wait   pts/0    00:00:01 bash
0 S  1000 28473     1  1  80   0 - 75939 poll_s pts/0    00:00:34 atril
```

The priority is under the column labeled `NI`. Notice that `emacs` is running with priority `0`. I can change the priority of my `emacs` to this:

```
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  3148 26588  0  81   1 -  8250 poll_s pts/0    00:07:47 emacs
0 R  1000  8093 26588  0  80   0 -  1253 -      pts/0    00:00:00 ps
0 S  1000 26588 20828  0  80   0 -  1574 wait   pts/0    00:00:01 bash
0 S  1000 28473     1  1  80   0 - 76240 poll_s pts/0    00:00:35 atril
```

In Linux lower priority number means a higher priority.

Besides the prioritized use of resources (in OS or networks or web servers etc.) priority queues are also very important in algorithms. You have already seen examples where containers are used to hold "work to do". For instance in breadth-first traversals, queues are used to hold nodes to be processed. In depth-first traversals, stacks are used instead.

Many algorithms use priority queues to hold "work to do". For instance when you visit a graph or a tree, you might see a node that might provide a shorter processing time to reach a solution than the nodes (in a container) that are waiting to be processed. In that case, the new node you just discovered after being placed in the container (of nodes to processed) need to jump over some nodes. For instance suppose you are in an unknown maze and your goal is to locate a pot of curry. As you walk, you draw the maze as much as you can because you don't want to walk in a cycle forever and you want to know how to backtrack if you hit a deadend. You might want to be systematic and explore all possible passage ways. *But* if you can smell curry coming from a specific direction, you will probably explore that promising passageway right away, putting other passageways aside for the time being. In a nutshell, that's the main idea behind Dijstra's shortest path algorithm, probably the most famous algorithm that uses a priority queue. (Make sure you google for how to pronounce Dijkstra!) But there are many others.

Obviously, you can implement a priority queue, say implemented using a doubly linked list. After a node enters a queue at the tail, you compare that node with the node in front of it and swap them if necessary (based on their priorities). The runtimes are then as follows:

1. insert: $O(n)$
2. delete: $O(1)$

Can we do better? Yes we can. But we'll need something different. These are called heaps. And there are many different types of heaps.

There are some other operations that might be helpful for a priority queue. Although a priority queue is a self-organizing container, there are times when you want to access a particular item in the queue and make some modifications.

3. find a node in a priority queue and returning a pointer or index or location

Using the pointer or index or location of a node in the priority queue, you can make the following modifications:

4. delete an item in the priority queue
5. modifying the priority of an item in the priority queue

Another operations is

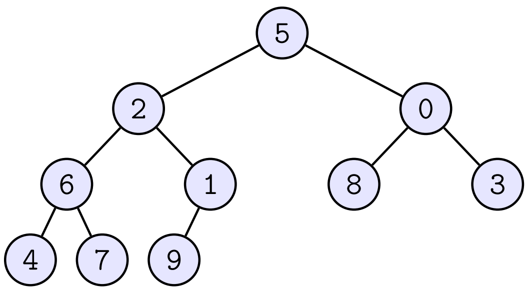
6. merging two priority queues into one

Of course there are other basic operations such as checking if the priority queue is empty, getting the size, clearing it, etc. For debugging it would be helpful if you can print it. Like any self-organizing container, sometimes it's helpful to look at the next item that will be removed (i.e., peek), without actually removing it.

File: array-tree.tex

109.2 Array trees

You have already seen trees using nodes which are dynamically allocated in the memory heap. Actually, it's possible to build trees using array! I'll just explain how to do this for the case of binary trees.



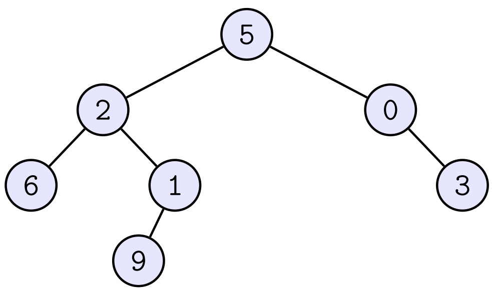
I can lay it out in an array like this:

5	2	0	6	1	8	3	4	7	9

In other words, traverse the tree using breadth-first left-to-right and put a value into the array as you see visit the value in the tree.

The above tree is complete. What if it's not?

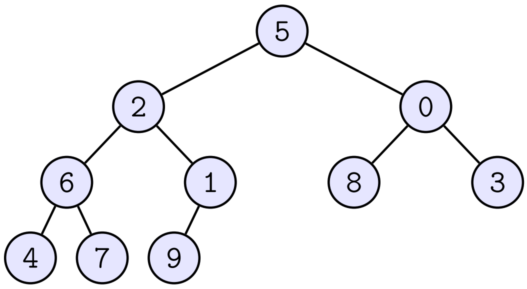
One option is to use a sentinel to denote the fact that an array element is not filled with a node value. For instance, say I have this tree:



5	2	0	6	1	-	3	-	-	9

Another option is to include a delete flag for each value in the array. If the element of the array represents a node, the delete flag is false. However if the element of the array does not hold a node, then the delete flag is set to true.

Of course a tree is not just a collection of data. It's a graph. You have to describe edges, i.e., you must connect the values. How can we do that? Well note that the left child of node with value `5` is the node with value `2`:



5	2	0	6	1	8	3	4	7	9

You notice that the index of `5` is 0 and the index of `2` is 1. Also, note that the left child of `2` is `6` which is at index 3. The left child of `0`, which is at index 2, is `8`, which is at index 5. For all the above cases, you notice the the left child of the value at index i is at index $2i + 1$. Correct?

Do you also notice that the value at index i, the right child is at index $2i + 2$?

Going to the parent is easy. If you look at the value at index j, then you have the followning fact:

- If j is odd, then the parent is at index $(j - 1)/2$. Why? Because in this case $j = 2i + 1$ and I want i. This is just $i = (j - 1)/2$.
- If j is even, then the parent is at index $(j - 2)/2$. Why? Because in this case $j = 2i + 2$ and I want i. Of course $i = (j - 2)/2$. Note that using integer division, this is the same as $(j - 1)/2$. Correct?

Let me summarize:

- The left child of the node at index i is at index $2i+1$.
- The right child of the node at index i is at index $2i+2$.
- The parent of the node at index j is at index $(j-1)/2$ where the division above is integer division.

```
ALGORITHM: left
INPUT:     i - an index in an array say x
OUTPUT:    index where x[index] is the left child of x[i]

return 2 * i + 1
```

```
ALGORITHM: right
INPUT:     i - an index in an array say x
OUTPUT:    index where x[index] is the right child of x[i]

return 2 * i + 2
```

```
ALGORITHM: parent
INPUT:     j - an index in an array say x
OUTPUT:    index where x[parent(i)] is the parent of x[i]

return (j - 1) / 2
```

So if I have a complete binary tree where the missing nodes are all on the right of the last level, I can use an array to represent the tree and the `left`, `right`, and `parent` functions can be used to form parent-child between the values of the array.

Sometimes when you read books on array trees, you will see that usually first element of the array, i.e., the element at index 0 is not used. In that case you have the following:

- The left child of the node at index i is at index $2i$.
- The right child of the node at index i is at index $2i+1$.
- The parent of the node at index j is at index $j/2$ where the division above is integer division, i.e., mathematically it should be $\lfloor j/2 \rfloor$

So be careful!

Whereas in the case of trees built with nodes in the heap and therefore we need pointers hold address values to the nodes, in the case of array trees, I will be using integer variables containing index values.

Exercise 109.2.1. Here's an array that represents a binary tree according to the scheme given above.

2	5	1	6	3	7	4	0	8	9

1. Locate the index position of the value `4`. What is the index and value of the parent of `4`?
2. Locate the index position of the value `1`. What is the index and value of the left child of `1`? What is the index and value of the right child of `1`?
3. What are the values of all the leaves?
4. What are the values of all the nodes with degree 1, i.e., with 1 child?
5. Draw the corresponding tree.

□

Exercise 109.2.2. How would you build a k–ary tree using arrays? Draw a 3–ary tree with about 20 nodes (with integer values).

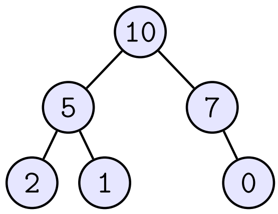
1. If `i` is an index, what is index of the d–th child?
2. If `j` is an index, what is index of the parent?

□

File: heap.tex

109.3 Binomial heaps

Look at this tree:

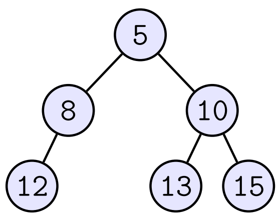


No is not a BST. But the numbers are not totally random: each node has value $\geq$ all the children. This is called a **maxheap**.

maxheap

Not too surprising, this is called **minheap**:

minheap



Each node has value $\leq$ all children.

Note that I have only defined max- and minheaps for *binary* trees. There are also called **binary heaps**. It's not too difficult to see that you can generalize these to k–**ary minheaps** or k–**ary maxheaps**.

binary heaps

k–ary minheaps

k–ary maxheaps

More generally you can define heaps with respect to an ordering relation. In the above the ordering is $\geq$ (for maxheap) and $\leq$ (for minheap).

Let me formalize the definitions for max- and minheap:

A tree satisfies the **maxheap property** if every node in the tree is greater than or equal to the children. A **maxheap** is a tree that satisfies the maxheap property. A k–ary tree satisfies the **minheap property** if every node in the tree is less than or equal to the children. A **minheap** is a tree that satisfies

maxheap property

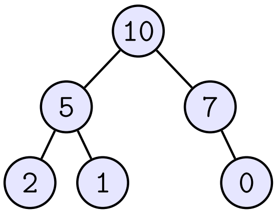
maxheap

minheap property

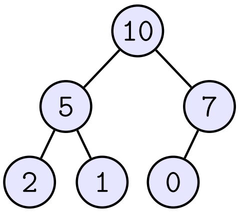
minheap

the maxheap property.

Although most of what we'll talk about regarding heaps works for all binary max/minheaps, we are usually only interested in complete heaps, i.e., all the levels are full except possibly for the last. This will ensure that the height is $O(\log n)$. Furthermore, the places where the level is not filled (if any) is "on the right". For instance instead of a maxheap like this:



we will usually consider this instead:



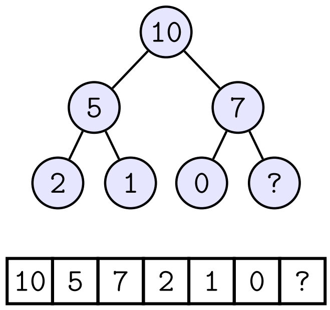
This will ensure that when this heap is implemented using an array, the values occupied are contiguous.

10	5	7	2	1	0

Cells which are not relevant are left blank. Of course there are integer values there – but we don't care about them. Furthermore, this array would usually come with a size or length variable (in the above example size would have value `6`) which is the case if I use `std::vector`. The size variable would tell me the index position of the first available cell in the array

10	5	7	2	1	0	?

which would correspond nicely with the next available node in the tree to keep the tree in the "heap shape", i.e., complete and unfilled nodes on a level (if any) on the right:



From now on, I will assume that a max- or minheap look like that, i.e., is complete where the unfilled slots (if any) are all on the same level and all "on the right".

Exercise 109.3.1. How many maxheaps can you draw if the heap contains the values 1, 2, 3, 4? □

Exercise 109.3.2. Suppose you have this array:

$$5, 4, 3, 2, 1$$

(index 0 is on the left). Assume that this array represents a binary tree.

1. Is this binary tree a maxheap?
2. How many arrays with the above values represent a maxheap?

□

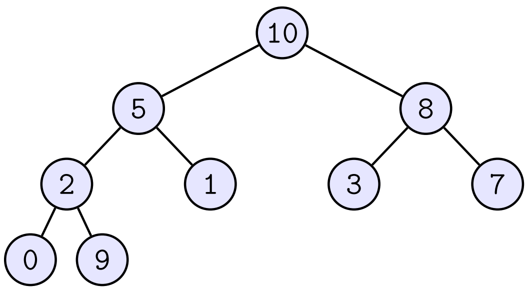
Exercise 109.3.3. In the array representation of a complete tree of size n where the leaves are all at the same depth and "on the left", what are the index values of all the non-leaves and all the index values of all the leaves? How many non-leaves are there altogether? $\square$

File: heapify-up.tex

109.4 Heapify-up

The following two operations are the basic tools for min- and maxheaps.

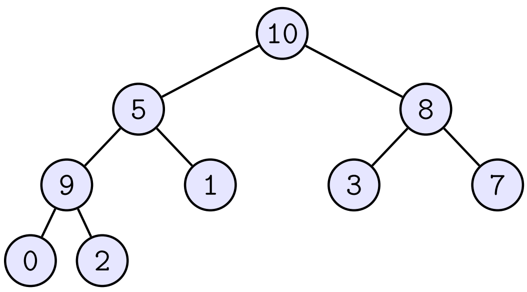
Look at this:



10	5	8	2	1	3	7	0	9

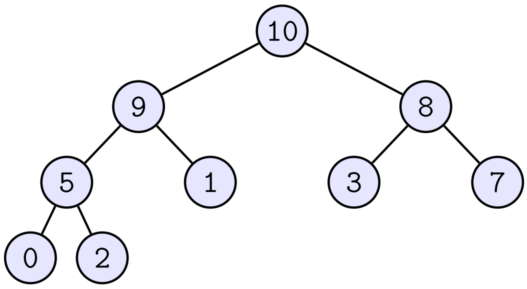
It's almost a maxheap except for the 9.

What is the simplest way to make it a maxheap? I compare 9 with it's parent, i.e., 2. Since this is a maxheap, something is wrong: parents should be larger than children. So I swap 9 and 2 to get:



10	5	8	9	1	3	7	0	2

I then compare 9 and 5 and decide to swap again to get:



10	9	8	5	1	3	7	0	2

Now it's a maxheap. Basically, you continually swap the value in question with its parent if the value is larger than the parent.

This is called **heapify-up**. It's also called **bubble up** or **percolate up**.

heapify-up bubble up percolate up

In general, if `x[0..n]` is an array and I can talk about heapify-up on array `x` treating `x` as a maxheap, starting at index `i` and going up along its path to the root (i.e., index 0).

You can also talk about heapify-up on an array treating it as a minheap, starting at some index.

For a maxheap, to heapify-up at an index `i`, you compare `x[i]` and `x[parent(i)]` and swap them if necessary, i.e. if `x[i]` $>$ `x[parent(i)]` and you keep following that value up to the root. If `x[i]` $\leq$ `x[parent(i)]`, you stop. That's it.

```
ALGORITHM: heapify-up (for maxheap)
INPUTS: x[0..n-1] -- array
        i         -- index of x

while 1:
    if parent(i) < 0: break
    if x[i] > x[parent(i)]:
        swap x[i], x[parent(i)]
        i = parent(i)
```

With a tiny of optimization:

```
ALGORITHM: heapify-up (for maxheap)
INPUTS: x[0..n-1] -- array
```

```
        i         -- index of x

v = x[i]
while 1:
    p = parent(i)
    if p < 0: break
    if v <= x[p]:
        x[i] = v
        break
    else:
        x[i] = x[p]
        i = p
```

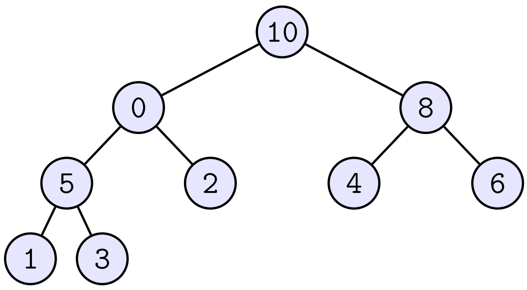
The same idea works for minheap.

File: heapify-down.tex

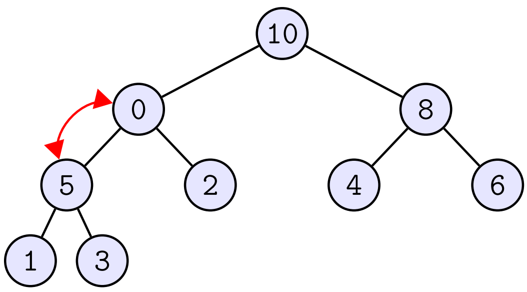
109.5 Heapify-down

Heapify-down is the opposite of heapify-up: you keep pushing a value `v` down, swapping with the largest child if that largest child is greater than `v`. Here's an example.

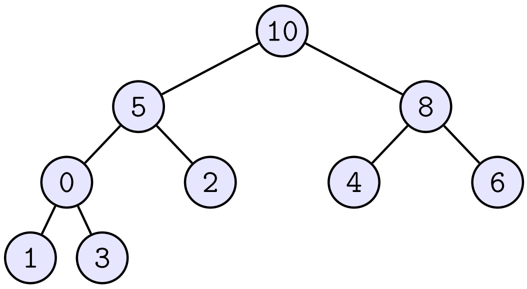
Suppose I have this tree:



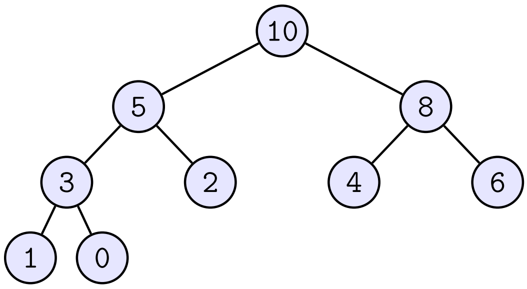
This is almost a maxheap except that `0` violates the maxheap property. I perform heapify-down at `0`, swapping with the larger of the children, i.e., `5`



to get



I do it again, swapping 0 with 3 to get



In this case, I stop at a leaf. In general, it's possible that the heapify-down stops before arriving at a leaf.

Note that I can heapify-down at any value even if the tree has multiple spots that violate the maxheap property.

```
ALGORITHM: heapify-down (for maxheap)
INPUTS: x[0..n-1] -- array
        i         -- index of x

while i < n and x[i] has a child larger than x[i]:
    swap x[i] with the largest child, say x[j]
    set i to j
x[i] = val
```

In more details:

```
ALGORITHM: heapify-down (for maxheap)
INPUTS: x[0..n-1] -- array
        i         -- index of x

v = x[i]
```

```
while 1:

    # Set j to index of the larger of the left and right
    # child of v. If there is no left and right child,
    # j is set to -1.
    l = left(i)
    r = right(i)
    if l == -1:
        # left child does no exist
        # (therefore right child does not exist)
        j = -1
    else:
        # left child exists
        if r == -1:
            # right child does not exists
            j = l
        else:
            # right child exists
            if x[l] > x[r]:
                j = l
            else:
                j = r

    if j != -1 and x[j] > v:
        # v heapify-down
        x[i] = x[j]
        i = j
    else:
        # v arrives at final index
        x[i] = v
        break
```

[REDUNDANT PARAGRAPH] You can heapify-up or heapify-down at any index position. There are times when heapifying-up or heapifying-down depends on the value at the given index. Let's call the function `heapify`. In such cases, if it's possible to heapify-up, then `heapify` will heapify-up and if it's possible to heapify-down, then `heapify` will heapify-down.

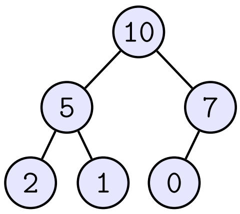
[What if there's a value that can heapify up and can heapify down?]

Exercise 109.5.1. In terms if actual real time, which cost more, one step of heapify-up or one step of heapify-down? □

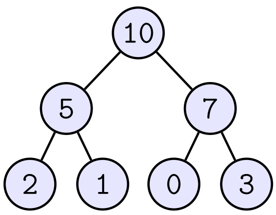
File: insert.tex

109.6 Insert

Look at this maxheap again:

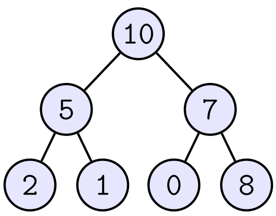


Suppose I want to insert a `3` into the above tree. To maintain the shape of a heap, I have to do it here:

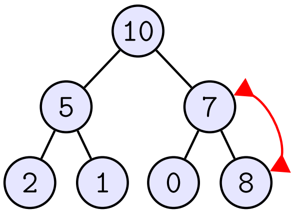


(Don't forget that heaps are implemented with arrays and therefore I can find the available slots in the array right away with the length variable of the array.) In this case the tree becomes perfect. It's also a maxheap.

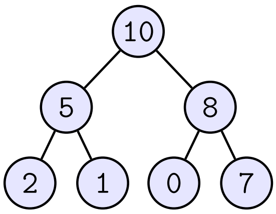
But what if I want to add `8` into the tree instead? I can again put it at the same spot:



Of course this is not a maxheap any more. What do I do? I heapify-up. I look at `8` and swap it with its parent if `8` is larger than its parent. In this case, the parent is 7, so I swap them:



to get



Now it's a maxheap again. In general recall that heapify-up might involve more than one swap.

In terms of the array implementation of the above heap, basically this:

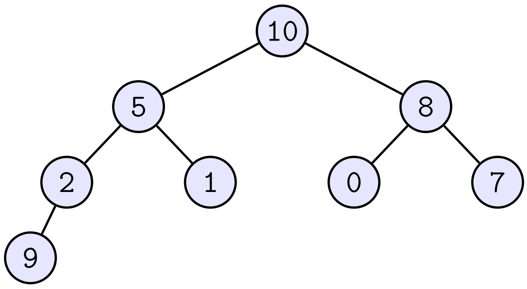
10	5	7	2	1	0	8

becomes this:

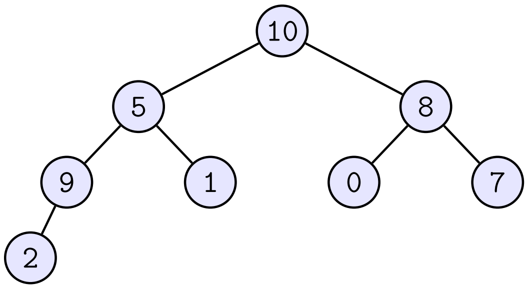
10	5	8	2	1	0	7

(Don't forget that technically speaking, there should also be a length variable.)

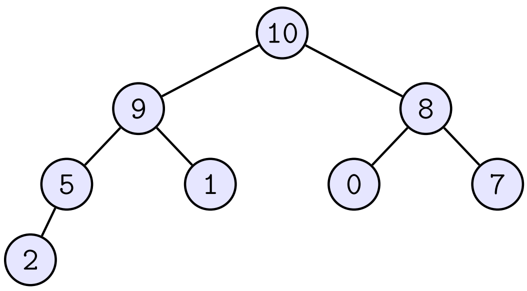
Suppose I do this again: I add a 9. It must go here:



(Draw the array implementation for the above.) I swap 9 and 2 to get this:

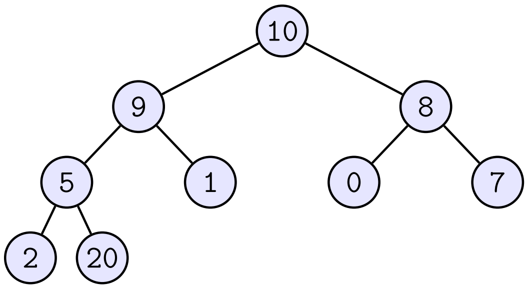


and then swap 9 and 5 to get

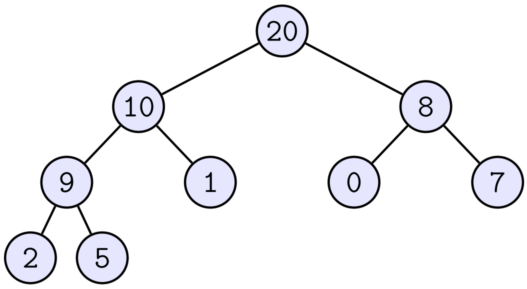


(Draw the arrays for the above so that you see how the array changes.)

This works even when you swap all the way to the root. Say I add a 20. It must go here:



After 3 swaps I get:

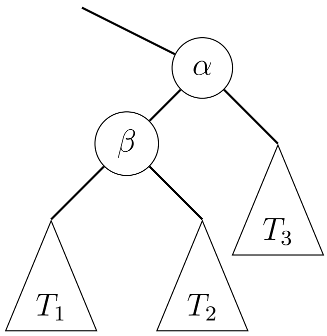


and I get a maxheap again.

This process is called **bubble-up** or **heapify-up**. or **percolate up**. bubble-up heapify-up percolate up

Exercise 109.6.1. Draw the maxheap after each of the above swaps and draw the corresponding array. ☐

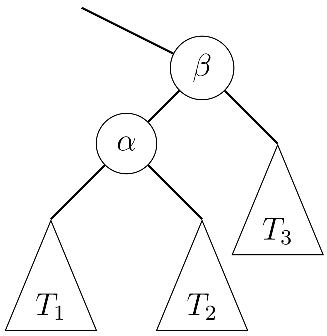
Now if you think about it, if you have the following



where

- the subtree at β is a maxheap,
- the subtree at α is also a maxheap if we ignore the its left subtree,
- and $\beta > \alpha$,

then on swapping α and β:



we have a maxheap at β.

```
ALGORITHM: heap_insert (for maxheap)
INPUT: x - an array representing a heap
       n - length of heap (pass by reference)
       key - value to be inserted

insert node with key as a leaf in the right place, i.e.,
x[n] = key
n = n + 1 (note that now the key is at index n - 1)
heapify_up(x, n - 1)
```

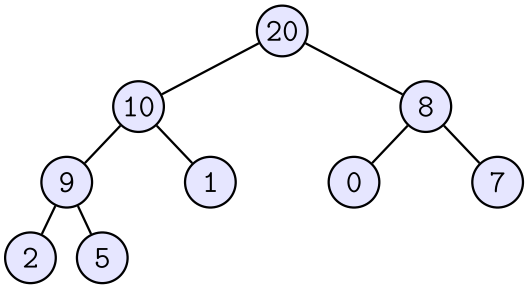
The corresponding algorithm for minheap is similar.

Note that the runtime is

$$O(\log n)$$

Why? Because the heapify-up basically "bubble up" the inserted key value from the point of insert (at leaf level) up to the root, possibly stopping before reaching the root. But in the worse case, this means that worse runtime depends on the height which is $O(\log n)$ since the tree is complete.

Exercise 109.6.2. Starting with this maxheap:



Do the following assuming that the array implementation of the above heap is an array of size 20.

1. Draw the array implementation of the above maxheap.
2. Insert `5`. Draw the maxheap after the insert and after each necessary swap (technically, until all the swaps are done the tree is not a max heap). Draw the array implementation after each swap.
3. Do the same with `15`.
4. Do the same with `11`.
5. Do the same with `22`.

□

Exercise 109.6.3. Starting with an empty *min* heap. Do the following, drawing the heap and the array implementation.

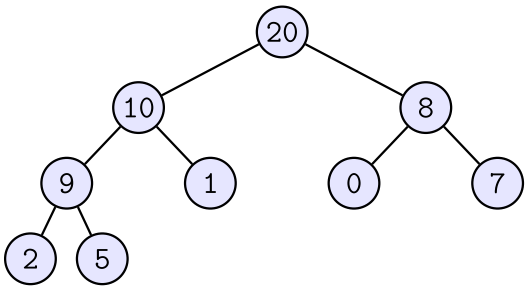
1. Insert `10`.
2. Insert `15`.
3. Insert `5`.
4. Insert `2`.
5. Insert `8`.
6. Insert `0`.
7. Insert `5`.

(First, you want to study the operations for maxheap very carefully. Then, you translate the operations to the case of minheap.) □

File: delete.tex

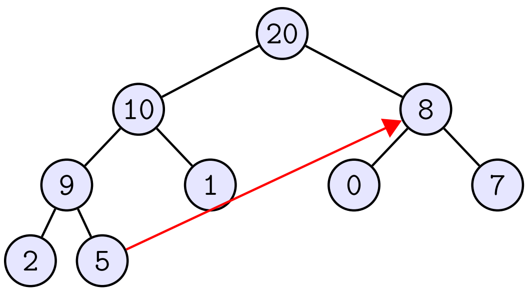
109.7 Delete

Suppose I want to delete `8` from this maxheap:

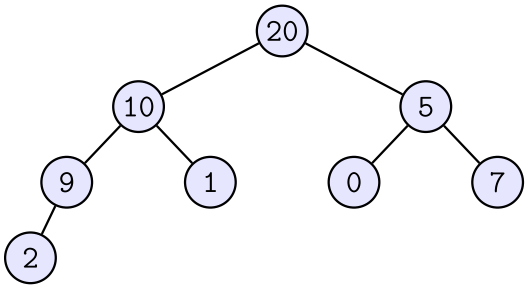


In other words I want to delete the value at index 2 in the array implementation of the above heap.

I'll do it this way: I look for the rightmost node of the last level – this corresponds to the last value in the array implementation. In the above case, that's the value `5`. I then overwrite `8` with `5`:

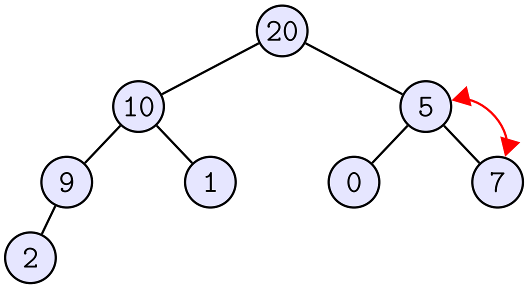


to get this:

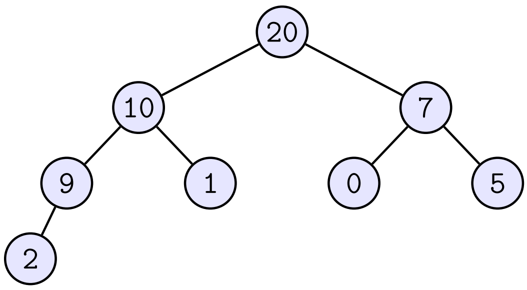


Now this is not a maxheap. Do you see why?

I look at the children of `5`: `0` and `7`. I swap `5` with the max of the children which is `7`



and get this:



Clearly, we keep pushing `5` down as much as possible until we get a maxheap again (i.e., heapify-down). It might take more than one swap.

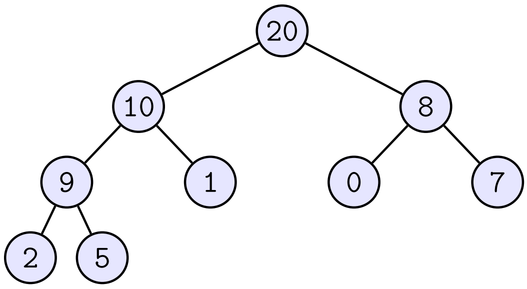
This above method works as long as `5` is a descendent of the value to be removed. Usually the value to be delete is in fact the root of the heap. When the root is to be deleted, then the operation is called **extract-max** for the case of maxheap and **extract-min** for the case of minheap. I'm going to call

extract-max

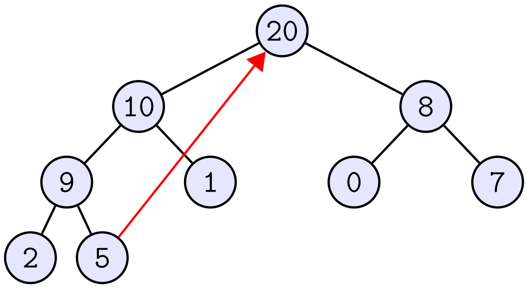
extract-min

both of them **extract-root**. extract-root

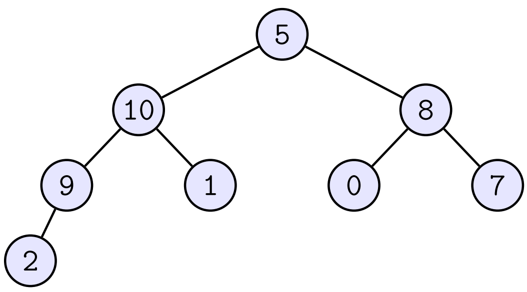
Let's do another example. Let's delete 20 (the root) from this maxheap:



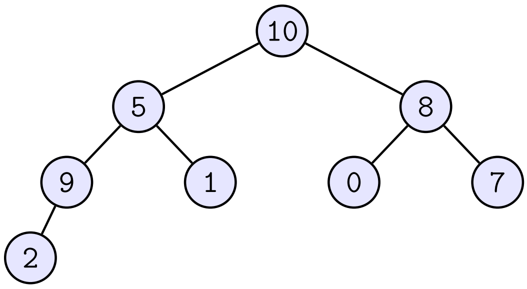
Again, I overwrite 20 with 5



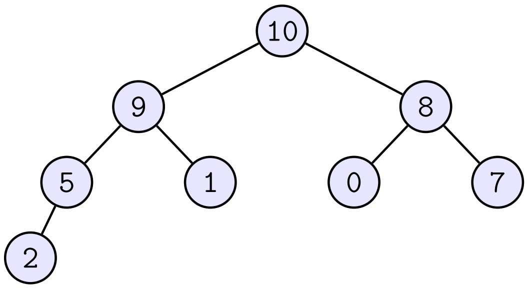
(5 is chosen, again because it's the rightmost in the last level, or equivalently, it's the last in the corresponding array implementation.) I get this:



I do the same again as above: I pick the larger of the children of 5, which in this case is 10 and swap with 5. I get this:



It's not a maxheap yet. I look at the children of 5 and choose the largest, which would be 9, and swap it with 5. Here's what I get:



Ahhh ... at this point I have a maxheap. I'm done!

```
ALGORITHM: heap_delete (for maxheap)
INPUT: x - an array representing a maxheap
       n - the length of the heap in x (pass by reference)
       i - index of value to be removed (note usually
           this is 0)

x[i] = x[n - 1]
n = n - 1
heapify_down(x, i)
```

(Note that because of the shape of the tree – a complete tree – a node cannot have a right child but no left child.)

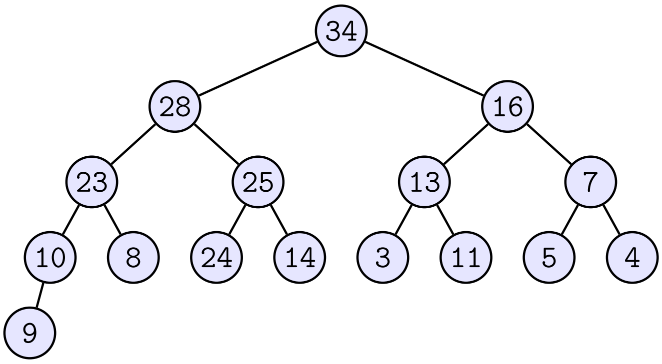
Recall what I just said: usually the delete operation for maxheap occurs at index 0, i.e., you're removing the maximum value in the maxheap. In that case the operation is also called extract-max or delete-max. You'll see why when we use this delete operation to perform heapsort and when we use this for priority queues.

Note that the runtime is

$$O(\log n)$$

since the heapify-down operation basically moves a value in the tree down to the leaf level, possibly stopping early. This means that the worse runtime must be the height of the tree which is $O(\log n)$.

Exercise 109.7.1. You are given this maxheap:



Do the following operations one after another, drawing the tree including the swaps and the final maxheap. Also, draw the corresponding array.

1. Delete `28`.
2. Delete `34`
3. Delete `16`
4. Delete the maximum value in the maxheap, i.e., perform extract max.
5. Delete the maximum value in the maxheap, i.e., perform extract max.

Exercise 109.7.2. Draw a minheap with 15 distinct values. Extract the minimum and draw the minheap. Do it again. ☐

Exercise 109.7.3. The extract-root removed the root of the heap. If I give you an index of a heap element, how would you delete that value at that index?

[PUT SOMEWHERE]

Exercise 109.7.4. Using an array, build a maxheap by inserting the following into an empty tree: 1, 3, 5, 7, 6, 4, 2, 0, 8, 9. Make sure the tree is complete after each insert and the leaves at the lowest level are all on the left. Call the array `x` (assume it has size at least 10) and use integer variable `len` for the length of `x`. Of course initially `len` is 0. ☐

Exercise 109.7.5. Using the maxheap (using an array) from the previous question, remove the following values one after another left to right:

$$1, 3, 5, 7, 6, 4, 2, 0, 8, 9$$

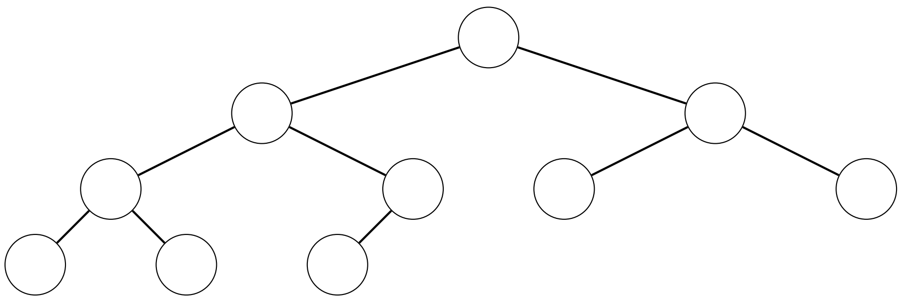
Call the array `x` and use integer variable `len` for the length of `x`. Of course initially `len` is 10. Make sure that after each delete, the tree is complete after each delete and the leaves at the lowest level are all on the left. □

File: heap-and-complete-tree.tex

109.8 Heap and complete trees

Recall that we have complete freedom in choosing where to insert a new node. We also have complete freedom in choosing any leaf to use to overwrite a node to be deleted as long as the leaf is a descendent of the node whose value is to delete. In particular, if we are remove the value of the root of the heap, we can choose any leaf.

It's because of the above, after every insert and root value removal, we can alway ensure that the heap is complete. Recall that a complete binary tree that is almost full except that the last level might not have all the leaves. are at the same level. Furthermore, we can force to have all the leaves to be all on the left side of the whole tree. This is what I mean by "left side" of the tree:



This can be achieved by

- During insert, always insert a leaf just to the right of the rightmost leaf at the last level.
- During delete, always using the right most leaf of the last level whenever we remove the root.

Exercise 109.8.1. Build a maxheap by inserting the following into an empty tree: 1, 3, 5, 7, 6, 4, 2, 8, 9, 0. Make sure the tree is complete after each insert and the leaves at the lowest level are all on the left. □

Exercise 109.8.2. Using the maxheap from the previous question, remove the following values one after another: 1, 3, 5, 7, 6, 4, 2, 8, 9, 0. Make sure that after each delete, the tree is complete after each insert and the leaves at the lowest level are all on the left. □

Exercise 109.8.3. Suppose that in maxheap, there are two 5's. One 5 was inserted at 9AM and the second 5 was inserted at 10AM. If I delete the maximum value of the heap until the heap is empty, will the 5 inserted at 9AM be deleted from the heap before or after the 5 inserted at 10AM? Or are both scenarios possible?

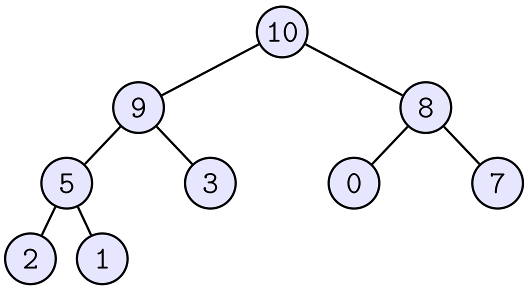
Being complete means that the heaps can have the minimal possible height. At this point, you ought to know that this is a good thing.

This also implies right away that the worse runtimes for insert and delete is $O(\log n)$ for both insert and delete as long as we keep the heap complete.

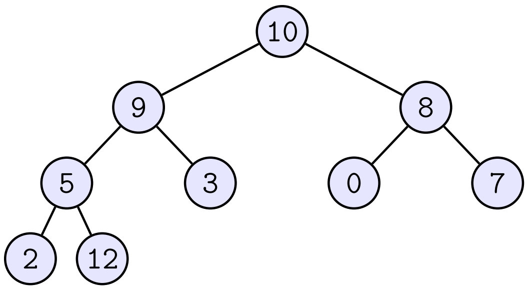
File: increase-key-decrease-key.tex

109.9 Increase-key and decrease-key

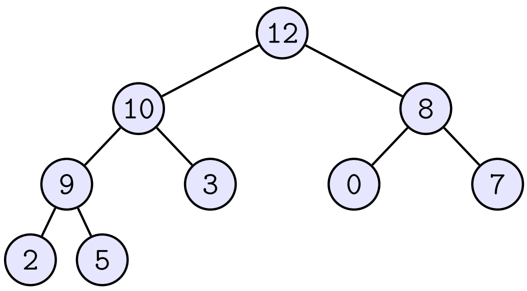
So suppose I already have a priority queue (as a heap):



Suppose 1 is increased to 12:



I'm sure you can see very quickly that in order to make this back to a maxheap, I need to heapify-up 12. In this case I need to swap 3 times to get this:

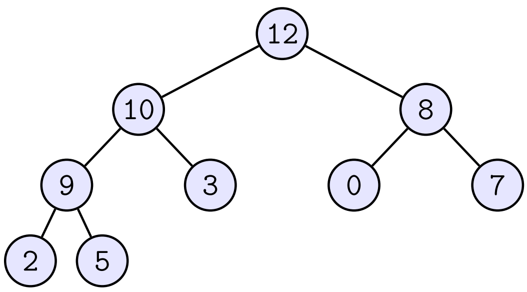


```
ALGORITHM: increase_key (for max heap)
INPUT: x - heap (using an array)
       n - length of heap in x
       i - index where key will increase
       k - new key value (k > x[i])

x[i] = k # originally x[i] < k
Perform heapify-up on x starting at index i.
```

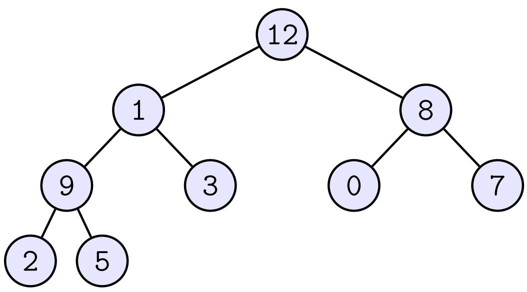
So if the above represent processes with priorities, the low priority process moves up, in fact to the top, so it will be the next process to be executed. The worse runtime of increase-key for max heap is

$$O(\log n)$$

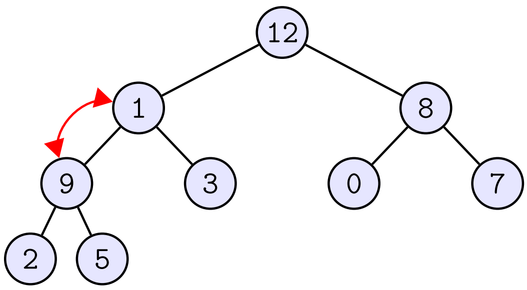
Using the above tree



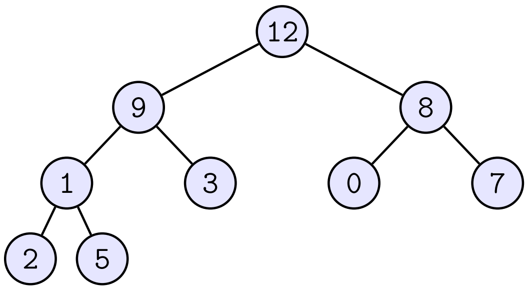
suppose `10` is decreased to `1`:



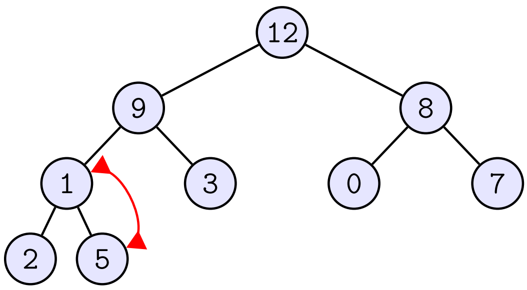
Of course this is not a maxheap anymore. To make this back to a maxheap, clearly the simplest thing to do is to heapify-down. In this case I need two swaps. First I do this swap



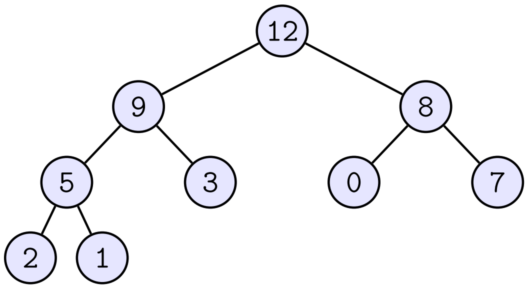
to get this:



Next I do this swap



to get this:



Here's decrease key:

```
ALGORITHM: decrease_key (for maxheap)
INPUT: x - heap (using an array)
       n - length of heap in x
       i - index where key will increase
       k - new key value (k < x[i])

x[i] = k # originally x[i] > k
Perform heapify_down on x at starting at index i.
```

The worse runtime of decrease-key for maxheap is clearly

$$O(\log n)$$

It's clear that given a heap (max or min), you can heapify-up and heapify-down. I prefer to use "heapify-up" and "heapify-down" because the important thing is whether a node moves up or down. In the case of maxheap, heapify-up and heapify-down are respectively increase-key and decrease-key whereas in the case of minheap heapify-up and heapify-down are respectively decrease-key and increase-key.

Exercise 109.9.1. Let x be the array

$$4, 9, 2, 6, 3, 8, 0, 1, 5, 7$$

First heapify it. Next do the following:

1. Perform increase-key from on the key with value `2` and change it to `11`.
2. Perform decrease-key from on the key with value `7` and change it to `-1`.

□

Exercise 109.9.2. Continuing the implementation of the heap ADT using functions, implement the increase-key and decrease-key functions:

```
int x;
std::vector< int > heap;

heap.resize(5);
heap[0] = 5;
heap[1] = 7;
heap[2] = 8;
heap[3] = 10;
heap[4] = 2;
maxheap_build(heap);      // [10, 7, 8, 5, 2]

maxheap_increasekey(heap, 2, 12); // heap[2] is changed
                          // to 12. heap has to be
                          // reorganized to become
                          // maxheap again.

maxheap_decreasekey(heap, 2, 0); // heap[2] is changed
                          // to 0. heap has to be
                          // reorganized to become
                          // maxheap again.
```

File: build-heap.tex

109.10 Build heap

Frequently, you want to make an array into a heap. This is called **build-maxheap** (if you want to make the array into a maxheap) or otherwise it's called **build-minheap**. I'll just call it **build-heap** if the type of heap is clear from the context. It's also called **max-heapify** or **min-heapify** or **heapify** (if the context is clear).

build-maxheap
build-minheap build-heap
max-heapify min-heapify
heapify

109.10.1 Slow method

We can use the heaps to sort arrays. For instance suppose you have an array `x` of 10 values. Looking just at the first value, `x[0]`, you have a heap of one value. Now insert `x[1]` into the heap with only `x[0]`. At this point `x[0..1]` is a heap – say we want to sort it in ascending order, which means that we're using maxheap (you see why later). Now we repeat to get `x[0..2]` to be a maxheap. Etc. When we're done, we have a maxheap of `x[0..9]`. Inserting into a heap requires $\log_2 n$ steps where n is the size of the heap. Therefore the runtime to create the heap from an array of n values is, informally speaking, $\log_2 1 + \log_2 2 + \cdots + \log_2 n$ which is $\log_2 n! \leq \log_2 n^n = n \log_2 n$. There's a faster algorithm ... the real build-heap.

109.10.2 Fast and right method

Now for the real build-heap or build-maxheap or build-minheap.

As mentioned at the beginning of this section, you can create the maxheap by continually inserting values into the the maxheap. The runtime is $O(n \log n)$. Instead of doing that you can also execute heapify-down on all the non-leaves positions of the given array is a systematic way: from the non-leaf at the lowest level to the root, more or less the opposite of the breadth-first traversal (ignoring the leaves).

Note that if the size of the array is n, then the indices of the leaves are $n/2$ (integer division), $n/2 + 1$, ..., $n - 1$. Therefore you can convert the array to a maxheap if you perform heapify-down at indices $n/2, n/2 - 1, n/2 - 2, ..., 0$, you will get a maxheap too.

The runtime of build-heap is

$$\lg(n/2) + \lg(n/2+1) + \cdots + \lg n$$

Each of these terms are $\leq \lg n$ and there are $n/2$ such heapify operations. So the runtime is at most $(n/2)\lg n = O(n \lg n)$. But that's an over-approximation. It can be shown that the runtime is actually

$$O(n)$$

which is faster than the earlier build max heap algorithm at the beginning of this section. However this does not improve the overall heapsort since the second part of the heapsort process will still run in $O(n \log n)$.

```
ALGORITHM: build_maxheap (or heapify)
INPUT: x - array containing n values x[0..n-1] that will
           represent a maxheap at the end of this
           algorithm.
       n - length of x

Perform heapify bottom-up from the first nonleaf to the
root, i.e.,

for i = n/2, n/2 - 1, n/2 - 2, ..., 0:
    perform heapify-down on x at i
```

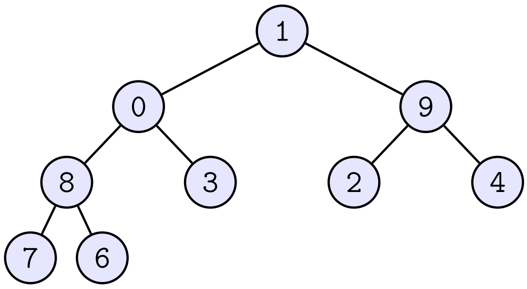
As mentioned at the beginning of this section, the runtime of this build-maxheap has runtime

$$O(n)$$

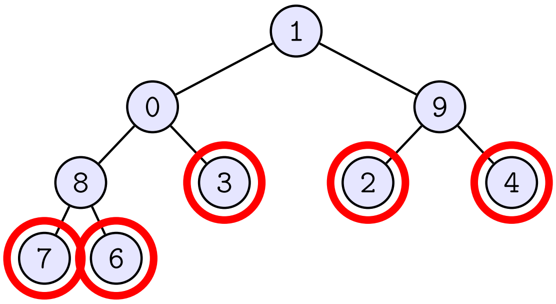
Let me show you how to build a maxheap given an array. Let's say we're given this array:

$$1, 0, 9, 8, 3, 2, 4, 7, 6$$

Here's the array drawn as a complete tree:

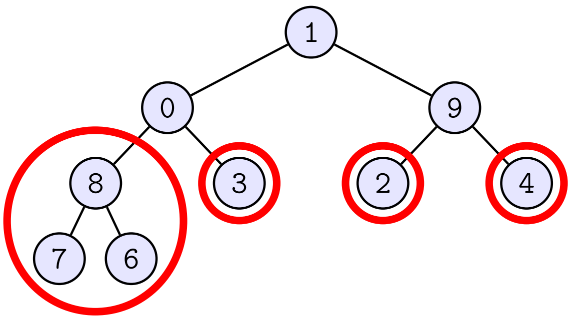


The main idea of build-maxheap is to maintain a collection of subheaps. Each leaf is already a heap. So I actually start with 5 subheaps:

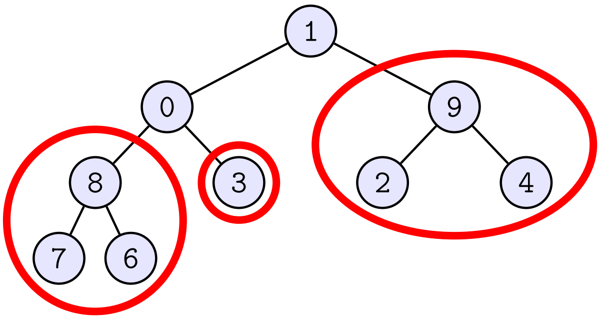


I now heapify in this order: 8,9,0,1. By this I mean `8` (at index 3) is going to start off at the root postion of the heap that combines two subheaps at 7 and 6.

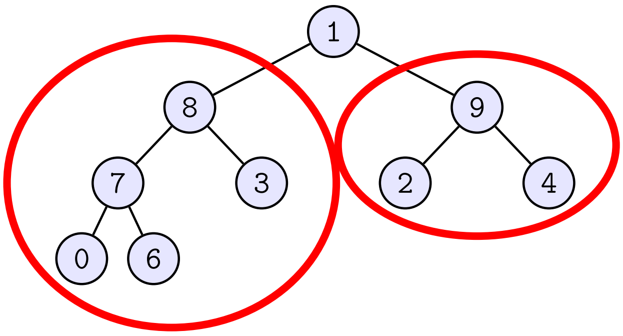
Step 1. I heapify-down at `8` (at index 3). There's no change since the subtree at `8` is a maxheap.



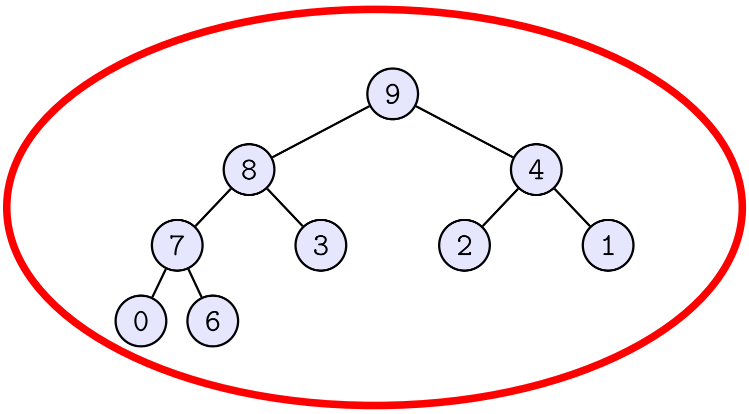
Step 2. Heapify-down at index 2 with value `9`: No change since the subtree at `9` is a maxheap.



STEP 3. Heapify-down at index 1 with value `0`: I need 2 swaps. After that the subtree at position where `0` originally was a maxheap.



STEP 4. Heapify-down at index 0 with value `1`: I need 2 swaps. After that the subtree at the place where `1` was is a maxheap.



I'm done!

Hence I get this array (which represents the above maxheap):

$$9, 8, 4, 7, 3, 2, 1, 0, 6$$

Exercise 109.10.1. Draw the corresponding array at the end of each stage for the above computation □

Exercise 109.10.2. Perform build-maxheap on the following array:

$$5, 3, 0, 7, 1, 2, 6, 9, 4, 8$$

showing every step (like in the above example). $\square$

Exercise 109.10.3. Perform build-maxheap on the following array:

$$2, 1, 5, 0, 3, 7, 9, 6, 4, 8$$

showing every step (like in the above example). □

Exercise 109.10.4. Perform build-maxheap on the following array:

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9$$

showing every step (like in the above example). $\square$

File: heapsort.tex

109.11 Heapsort

HEAPSORT GIVEN A MAXHEAP. Now we delete the root from the heap. Because this is a maxheap, the root, is the maximum value of `x[0..9]`. We swap `x[0]` and `x[9]` and re-heapify to get a heap `x[0..8]`. In other words we're essentially doing a delete of the value at `x[0]` (the extract-max operation), putting this value at `x[9]`.

We repeat the above to get a heap `x[0..7]`. At this point the largest value of the array is at `x[9]` and the second largest is at `x[8]`.

We repeat until the heap has only one value, i.e., the heap is `x[0]`. The whole array must be sorted in ascending order.

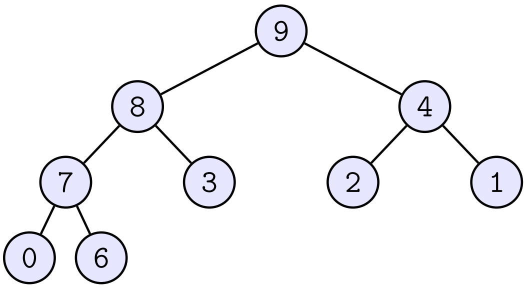
The runtime is, informally, $\log_2 n + \log_2(n-1) + \cdots$ which is $\log_2 n! \leq \log_2 n^n = n \log_2 n$.

So the total time for the above process, creating the heap and then removing roots from the heap is roughly $2n \log_2 n$ and therefore the runtime for heapsort is $O(n \log_2 n)$.

Note that the heapsort has a worse runtime of $n \log_2 n$ whereas quicksort can have a worse runtime of n^2 although on the average quicksort is $O(n \log n)$ and typically quicksort is faster than heapsort by a constant factor. Although mergesort does achieve $n \log_2 n$ for worse runtime, remember that mergesort needs $O(n)$ space. However heapsort is not stable whereas mergesort is stable.

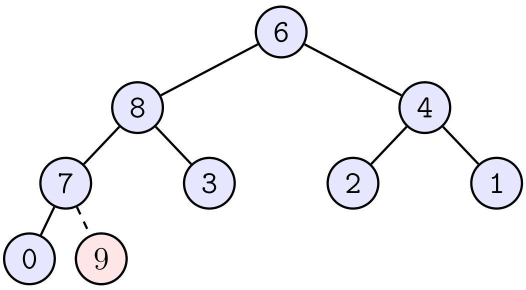
Exercise 109.11.1. Give an example showing that heapsort is not stable.

Here is a maxheap say constructed using build-maxheap:

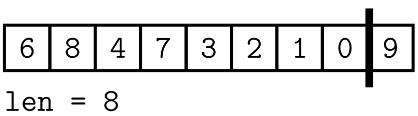


Let's perform heapsort on this maxheap. Recall that extract-root will throw away the root by replacing the value at the root with the last value in the tree (i.e., the rightmost value at the last level of the tree). Instead of replacing the root with the last value, I will *swap* the root value and last value. Otherwise it's the same extract-root operation. Let's do it.

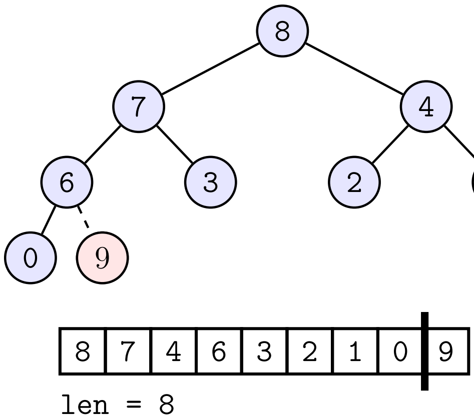
Step 1. First I swap the root value and the last value:



I color `9` in red to remind myself that the `9` should not be considered part of the maxheap, but of course it's in the array. In terms of an array the above diagram would correspond to this array:

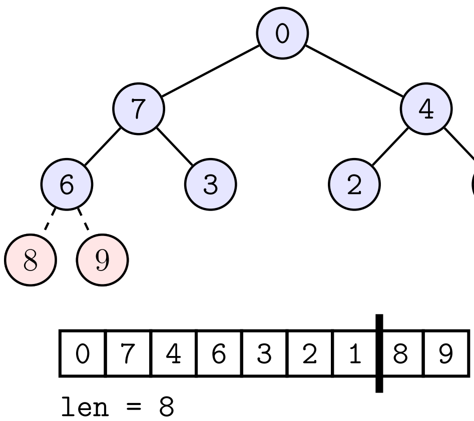


I still need to heapify-down the `6` to get this:

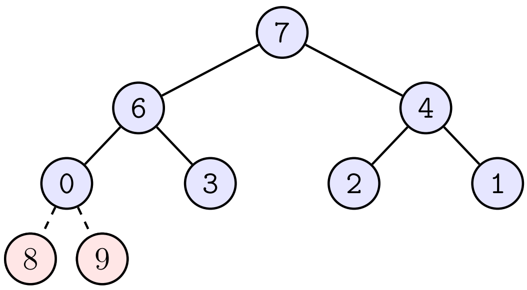


Because I started with a maxheap, `9`, the value I get from extract-root, is the largest value in the array. Since it moved to the last index position in the array, this means that `9` has found its right place.

Step 2. Now I repeat the above process: first I swap the root (i.e., `8`) and the last value of the tree (i.e., `0`) to get this:



(at this point `9` and `8` are not part of the heap) then I heapify-down `0` to get this:



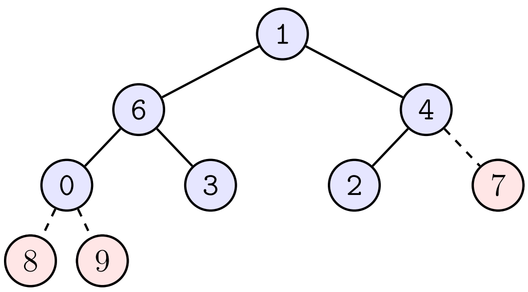
The array is

7	6	4	0	3	2	1	8	9

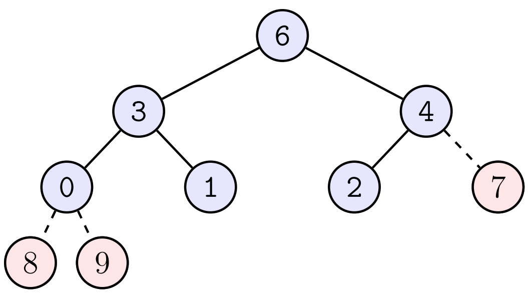
`len = 7`

This is the second extract-max. This means that `8` is the second largest value in the array. Note also that it's in its right place.

Step 3. Next I swap the root value (i.e., `7`) and the last value of the tree (i.e., `1`)



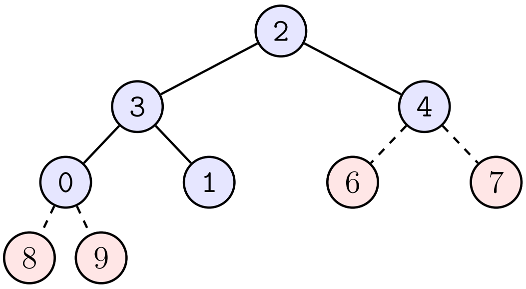
and heapify at `1` to get this:



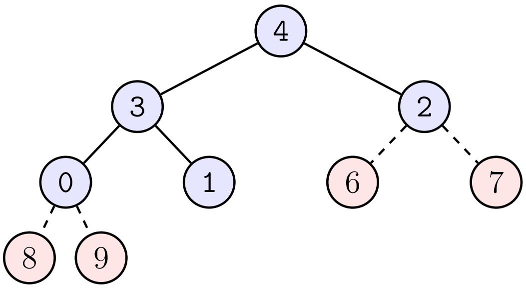
6	3	4	0	1	2 ‖	7	8	9

`len = 6`

STEP 4. I swap `6` and `2`:



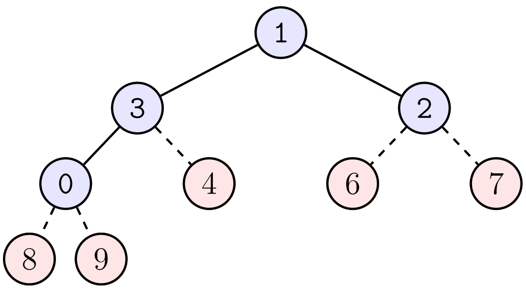
and heapify-down at `2` to get:



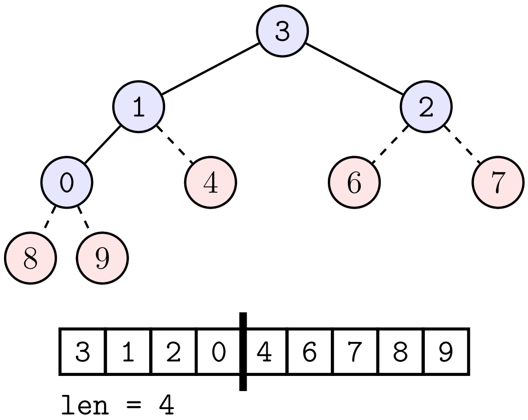
4	3	2	0	1 ‖	6	7	8	9

`len = 5`

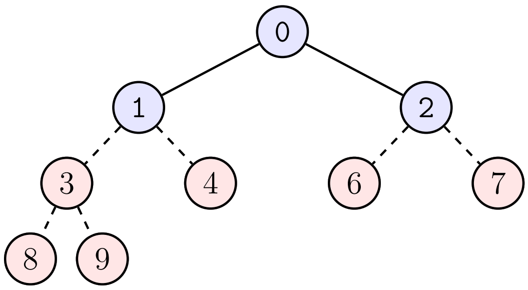
STEP 5. I swap `4` and `1`:



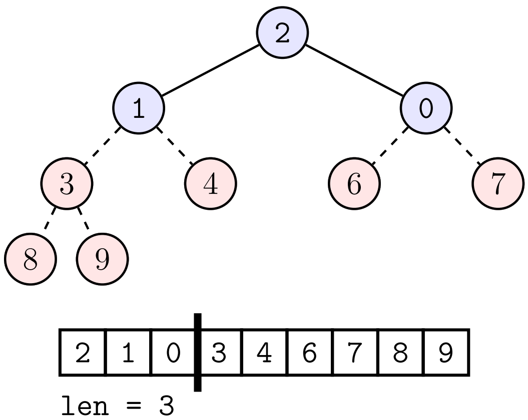
and heapify-down at `1` to get this:



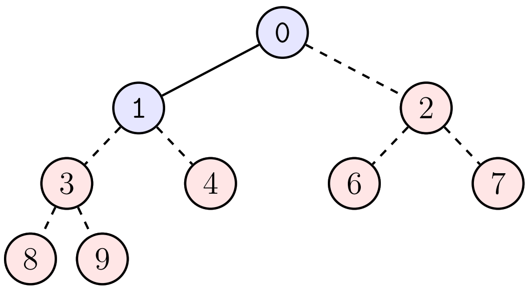
STEP 6. I swap `3` and `0`:



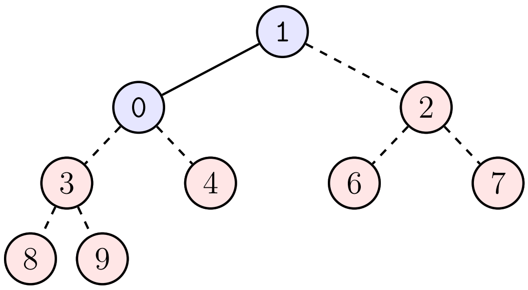
heapify-down at `0`:



STEP 7. Swap `2` and `0`:



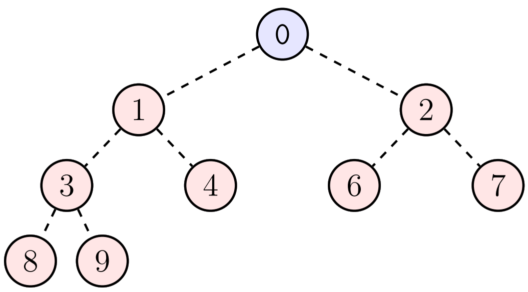
heapify-down at `0`:



1	0	2	3	4	6	7	8	9

`len = 2`

Step 8. I swap `1` and `0`



Clearly there's no need to heapify-down since the tree now has size 1. At this point, the array is

0	1	2	3	4	6	7	8	9

`len = 1`

It's sorted!

Here's the algorithm:

```
ALGORITHM: heapsort
INPUT:     x - array
           n - length of x

Perform build_maxheap on x[0..n - 1]
for i = n - 1, n - 2, ..., 1:
    swap x[0] and x[i]
    perform heapify_down on x[0..i - 1] at index 0
```

Exercise 109.11.2. Perform heapsort on given arrays below (in ascending order). Execute build-maxheap and then continually perform extract-max and heapify. Draw the tree after every extract-max and heapify and draw the corresponding array.

- `[5, 7, 4, 0, 8]`
- `[3, 9, 0, 7, 2, 5, 4, 1, 0]`
- `[1, 3, 5, 7, 6, 4, 2, 0, 8, 9]`
- `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
- `[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]`

Exercise 109.11.3. Now do the same as above except that you perform heap-sort in *descending* order.

- `[5, 7, 4, 0, 8]`
- `[3, 9, 0, 7, 2, 5, 4, 1, 0]`
- `[1, 3, 5, 7, 6, 4, 2, 0, 8, 9]`
- `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
- `[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]`

Exercise 109.11.4. An array x of size n represents a maxheap. What is the fastest way to find the minimum value in x? □

Exercise 109.11.5. Figure out a good algorithmn to merge two maxheaps. □

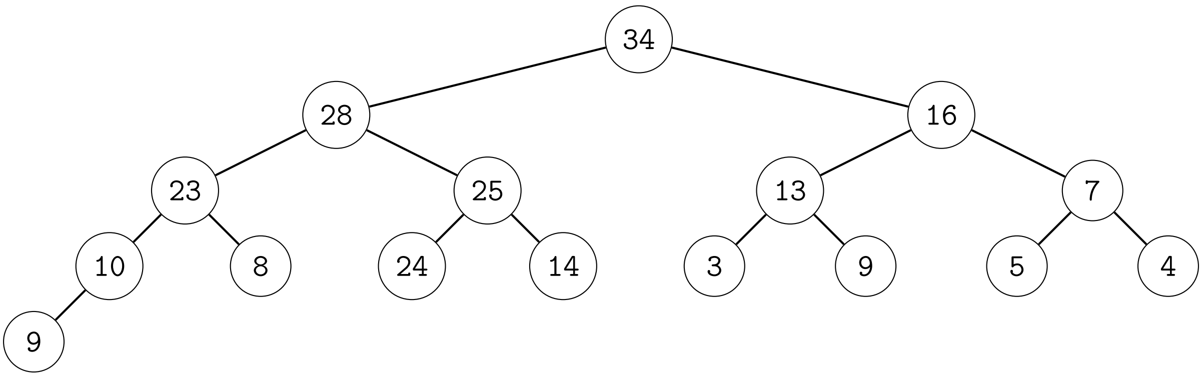
File: merge.tex

109.12 Merge

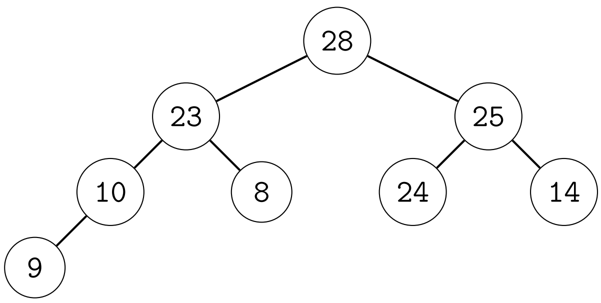
Suppose you're two max heaps. Is there an efficient way to put them together to get a max heap? This is called a heap **merge** or a heap **union**. merge

For instance suppose, suppose a machine has two exact same server process running the same software, answering to HTTP requests. Each server software maintains it's own queue of server requests. However something bad happened: one of the server process is stuck (hackers?) A main master process that watches the two server processes sees that one of server processes is stuck. So it transfers the queue of requests to the one that is still alive and kills the server that is stuck. Somehow the two queues of requests has to be merged.

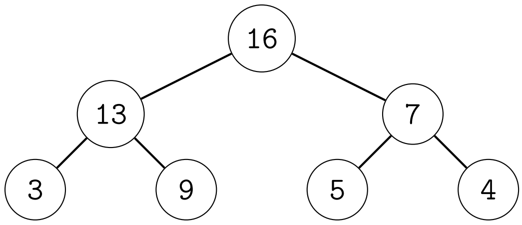
By the way. I hope you realize that if you have a max heap and you remove the root, then the two subtrees are themselves max heaps. For instance if you remove the root from this max heap:



you get the left subtree which is a max heap:



and the right subtree which is also a max heap:



OK. Back to the algorithm of merging two (max) heaps H_0 and H_1. Suppose the total number of nodes in H_1 is m and the total number of nodes in H_2 is n. Without using the existing heap structure in H_1 and H_2 and just build a heap from scratch requires $O(n)$ using the build-heap operation. Also, note that if we can merge two heaps quickly, then another way to build a heap is to do it recursively: recursive divide a collection of values into two piles, make them into heaps, then merge the heaps. So what's there not to like?

However, for our max heap, the only way seems to be to copy the values from one heap to another and perform build-heap which will run in $O(m+n)$ where m is the size of the first heap and n is the size of the second.

Exercise 109.12.1. Can we merge in $O(\log(m+n))$ time? [ANSWER: You'll need a different kind of heap. See if you can figure this out on your own.]

File: api.tex

109.13 API

Here I'm assuming the implementation is an array-based implementation (for us, this means C array or C++ `std::vector` objects).

Here are some very common operations:

`is_heap`	true if it's a heap
`insert`	insert a value into a heap
`delete`	delete max (min) in a maxheap (minheap)
`heapify`	create a heap from a collection of data (usually an array)
`find_max/find_min`	find max (min) in maxheap (minheap)
`heapsort`	heapsort on array
`increase_key/decrease_key`	increase (decrease) key of a node in maxheap (minheap)
`merge`	merge two heaps

It's kind of annoying to have to use max/min in the names above. The simplest thing to do is to define a comparison function that the heap functions/class uses. The boolean function basically compares two nodes and tells you which node should be "higher" in the heap tree.

109.13.1 C++

C++ provides heap functions, which operations on sequential container such as `std::vector`. To use them include this at the top:

```
#include <algorithm>
```

The functions are

`is_heap`	`std::is_heap`
`is_heap_until`	`std::is_heap_until`
`insert`	`std::heap_push`
`delete`	`std::heap_pop`
`heapify`	`std::make_heap`
`find_max/find_min`	`std::front`
`heapsort`	`std::sort_heap`
`increase_key/decrease_key`	
`merge`	

C++ provides an STL container `priority_queue` for ... priority queues (duh).

File: compare.tex

109.14 Compare heap against queue and BST/AVL

As mentioned earlier, the point of the heap is to organized data so that you want remove an item with the highest priority (use maxheap or minheap depending on whether "higher prioriy" mean "larger priority number" or "smaller priority number".

Here is the runtimes of the heap (min or max):

- Insert: $O(\lg n)$
- Delete: $O(\lg n)$ (i.e., delete the root)

Of course if you just want to peek at the highest priority value without deleting it, it's $O(1)$.

If you use a queue (using double linked list) to implement a priority queue,

- Insert: $O(n)$
- Delete: $O(1)$ (i.e., delete the root)

You might think this is OK since the delete is fast. But hang on cowboy ... if you perform n inserts and deletes say alternativing them, the average is going to be $O(n)$ for each. The average for the heap is going to be $O(\lg n)$. So using a queue is a bad idea.

What about a BST? Say we use the AVL tree.

- Insert: $O(\lg n)$
- Delete: $O(\lg n)$ (i.e., delete the minimum)

If you want to peek at the minimum without removing it, the runtime is $O(\lg n)$. Excluding the peek, the insert and delete are the same as the heap. But wait ...

When you insert a value into an AVL, it can be anywhere in the AVL. But when about a heap? Say a minheap? If the minheap has values $1, 2, 3, ..., n$

and you insert value $n/2$, notice that because the tree is a minheap, it would be nearer to the the bottom of the tree. Assuming the tree is full. How many leaves are there? If the height is h, then there are 2^{h+1} leaves. And how many nonleaves are there? Well it's $1 + 2 + \cdots + 2^h = 2^{h+1} - 1$. So there's a 50% chance that the inserted value is a leaf! Which in the case of a heap, it means that there's not a whole lot of heapify up at all!!! In fact you can show (but you would need probability theory) that the *average* runtime of heap insert is $O(1)$. Wonderful!

File: cpp.tex

109.15 C++ priority queues and heaps

109.15.1 Function pointers and functors

Suppose you have a function with the following prototype:

```
int f(double, char, bool);
```

then you can think of `f` as a variable. Except that you have to think of `f` as a pointer – it's a function pointer. Try this:

```
#include <iostream>

int f(double x, char y, bool z)
{
    std::cout << "f\n";
}

int main()
{
    int (*v) (double, char, bool) = f;
    v(3.14, 'A', true);
    return 0;
}
```

With this, a function can now work with a function:

```
#include <iostream>
#include <cmath>

double derivative(double (*f) (double), double x, double h=0.0001)
{
    return (f(x + h) - f(x)) / h;
}

double square(double x)
{
    return x * x;
}

```

```
int main()
{
    std::cout << derivative(sin, 0) << '\n';
    std::cout << derivative(cos, 0) << '\n';
    std::cout << derivative(tan, 0) << '\n';
    std::cout << derivative(square, 0) << '\n';

    return 0;
}
```

Exercise 109.15.1. Write a bubblesort function that accepts the equivalence of a "$\leq$".

```
bool less_than_or_equal(int x, int y)
{
    return (x <= y);
}

bool greater_than_or_equal(int x, int y)
{
    return (x >= y);
}

// cmp(x, y) is true if x appears before y when sorted
void bubblesort(int x[], int n,
                bool (*cmp) (int, int))
{
    // TODO
}
```

In that case

```
bubblesort(x, n, less_than_or_equal);
```

will sort `x[0..n-1]` in ascending order while

```
bubblesort(x, n, greater_than_or_equal);
```

sorts in descending order.

Exercise 109.15.2. Now make $\leq$ the default case so that

```
bubblesort(x, n);
```

will sort `x[0..n-1]` in ascending order while

```
bubblesort(x, n, greater_than_or_equal);
```

sorts in descending order.

It's not too surprising that since $\leq$ and $\geq$ are so common that in fact C++ has already defined `std::less`, `std::greater`, `std::less_equal`, and `std::greater_equal` that does the obvious except that these are not functions – they are `structs` with function call operator `operator()`. These are called **functors**. In other words `std::less` is something like

functors

```
namespace std
{
    template < typename T >
    struct less
    {
        operator()(const T & x, const T & y)
        {
            return (x < y);
        }
    };
}
```

Of course you can also do it this way:

```
namespace std
{
    template < typename T >
    class less
    {
    public:
        operator()(const T & x, const T & y)
        {
            return (x < y);
        }
    };
}
```

```
#include <iostream>
```

```
#include <functional>

int main()
{
    std::cout << std::less< int >()(1, 2) << '\n';
    std::cout << std::less< int >()(1, 1) << '\n';
    std::cout << std::less_equal< int >()(1, 1) << '\n';
    return 0;
}
```

Now why does C++ create a struct that contains an operator instead of creating a function? Because functors are structs variables (or they can be objects if you use classes instead of struct), they can hold values in their instance variables. That makes them more flexible. For instance a functor can keep the last 10 computations as a history and use them if necessary to speed up computation by avoiding computations and simply use a lookup table.

Exercise 109.15.3. Now make everything into templates:

```
template < typename T >
bool less_than_or_equal(const T & x, const T & y)
{
    return (x <= y);
}

template < typename T >
bool greater_than_or_equal(const T & x, const T & y)
{
    return (x >= y);
}

template < typename T >
void bubblesort(T x[], int n,
                bool (*cmp) (const T &, const T &))
{
}
```

```
#include <iostream>

bool le(int x, int y)
{
    return (x <= y);
}
```

```
void f(bool (*g)(int, int))
{
    std::cout << g(2, 3) << '\n';
}

int main()
{
    f(le);
    return 0;
}
```

With this, you can write a heapify-up (and heapify-down) function that works for both maxheap and minheap simply by passing in a comparison function or functor.

```
#include <iostream>

bool le(int x, int y) # minheap
{
    return (x <= y);
}

bool ge(int x, int y) # maxheap
{
    return (x >= y);
}

void heapify_up(int x[], int n, int i, bool (*cmp)(int, int))
{
    int v = x[i];
    while (1)
    {
        int p = (i - 1) / 2;
        if (cmp(x[i], x[p]))
        {
            x[p] = x[i];
            i = p;
        }
        else
        {
            x[i] = v;
            break;
        }
```

```
    }
}

int main()
{
    f(le);
    return 0;
}
```

The next thing is to make heapify up into a template:

```
template < typename T >
bool le(const T & x, const T & y) # minheap
{
    return (x <= y);
}

template < typename T >
bool ge(const T & x, const T & y) # maxheap
{
    return (x >= y);
}

template < typename T >
void heapify_up(T x[], int n, int i, bool (*cmp)(T, T))
{
    T v = x[i];
    ...
}
```

Another way to achieve this is through functors.

```
template < typename T >
struct LE
{
    bool operator()(const T & x, const T & y) # minheap
    {
        return (x <= y);
    }
};

template < typename T >
struct GE
{
```

```
    bool operator()(const T & x, const T & y) # minheap
    {
        return (x <= y);
    }
};

LE * le = new LE;
template < typename T, typename F >
void heapify_up(T x[], int n, int i)
{
    T v = x[i];
    ...
}
```

109.15.2 ???

C++ STL heap works on `std::vector` objects.

To use it do this at the top of your cpp file:

```
#include <algorithm>
```

Let v be a `std::vector< T >` object (for some type `T`).

- `std::make_heap(v.begin(), v.end())`: make `v` in to a maxheap
- `std::is_heap(v.begin(), v.end())`: returns true if `v` is maxheap
- `std::is_heap_until(v.begin(), v.end())`:
- `v.front()`: peek at root
- `std::push_heap(v.begin(), v.end())`: Performs insert with value at `v.end()`. The new value is first placed in `v` using `push_back()`.
- `std::pop_heap(v.begin(), v.end())`: extract-root
- `std::sort_heap(v.begin(), v.end()`: heapsort

C++ provides heap functions

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>

std::ostream & operator<<(std::ostream & cout,
                          const std::vector< int > & v)
{
    std::string sep = "";
    cout << '{';
    for (auto & e: v)
    {
        cout << sep << e;
        sep = ", ";
    }
    cout << '}';
    return cout;
}

int main()
{
    int x[] = {5,3,0,1,2,6,7,4};
    std::vector< int > v(x, x + 8);
    std::cout << v << '\n';
    std::make_heap(v.begin(), v.end());
    std::cout << v << '\n';
    return 0;
}
```

```
[student@localhost heap] g++ heapsort.cpp -std=c++11
```

```
[student@localhost heap] ./a.out
{5, 3, 0, 1, 2, 6, 7, 4}
{7, 4, 6, 3, 2, 5, 0, 1}
```

Of course you can also use an array:

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>

void println(int v[], int size)
{
    std::string sep = "";
    std::cout << '{';
    for (int i = 0; i < size; ++i)
    {
        std::cout << sep << v[i];
        sep = ", ";
    }
    std::cout << "}\n";
}

int main()
{
    int x[] = {5,3,0,1,2,6,7,4};
    println(x, 8);
    std::make_heap(x, x + 8);
    println(x, 8);
    return 0;
}
```

```
[student@localhost heap] g++ heapsort2.cpp -std=c++11
[student@localhost heap] ./a.out
{5, 3, 0, 1, 2, 6, 7, 4}
{7, 4, 6, 3, 2, 5, 0, 1}
```

File: nonarray-implementation.tex

109.16 Implementation

Frequently heaps are implemented using arrays, i.e., they are array trees.

Exercise 109.16.1. Johnny is writing a scheduler for his game and wanted to use a balanced binary search tree (AVL) to handle jobs with priority (collision computation, collisiion resolution, shadow computation, drawing, sound, etc.) with instead of heaps. What are the advantages and disadvantages? Compare and contrast. □

Exercise 109.16.2. Analyze and implement a FIFOMinHeap. This is a minheap that is FIFO on values with the same priority. In other words, if two values of the same priority are inserted into a minheap, then value inserted earlier will be removed before the other value. (There are at least two different ways of doing the above.)

There are 2 ways of doing the above.

Method 1: For object x, instead of using $x.priority$, use $(priority, time)$ where the minheap maintains a time that is incremented every time an insert occurs. The ordering of $(priority, time)$ is dictionary. For the case of maxheap, decrement the time counter.

Disadvantage: If the heap runs for a very long time, the time might overflow.

Method 2: The heap is made up of linked list of values. Each linked list contains values of the same priority. When a value x is inserted, if $x.priority$ occurs in the minheap, insert x into the correct linked list. If x's priority is new, create a new linked list and insert into heap. Note that we need to know if a linked list of the correct priority exists in the heap. We can have a hash table of (priority, heap index). During heap operations (heapify-up and -down), the hash table must be updated during the swaps. If the root linked list is empty, it is removed from the heap. If the heap allows priority modification, then a non-root linked list can become empty – this should not be delete. Therefore during the extract-root operation, need to loop over the extract-root operation until the new root is a non-empty linked list.

Exercise 109.16.3. Write a class `MaxHeap` that works like this:

```
int x;
MaxHeap< int > heap;
x = heap.size();       // x = 0
heap.insert(5);        // [5]
heap.insert(7);        // [7, 5]
heap.insert(9);        // [9, 5, 7]
x = heap.size();       // x = 3
std::cout << heap;     // Prints "[9, 5, 7]".

int a = heap.delete(); // [7, 5], deleting at index 0
                       // a = 9
x = heap.max();        // x = 9. The root is not deleted.

heap[0] = 1;           // [1, 5], heap is also like a
                       // std::vector.
heap.heapify_down(0);  // [5, 1]

heap[1] := 10;         // [5, 10]
heap.heapify_up(1);    // [10, 5]

heap.resize(5);
heap[0] = 5;
heap[1] = 7;
heap[2] = 8;
heap[3] = 10;
heap[4] = 2;
std::cout << heap;     // Prints "[5, 7, 8, 10, 2]".
heap.build();          // build-max-heap with 5 values
                       // already in the heap.
std::cout << heap;     // Prints "[10, 7, 8, 5, 2]".

heap.clear();          // []
```

Clearly the following must work: copy constructor, destructor, `operator=`, `operator==`. The class should look like this:

```
template < typename T >
class MaxHeap
{
public:
private:
    std::vector < T > x;
};
```

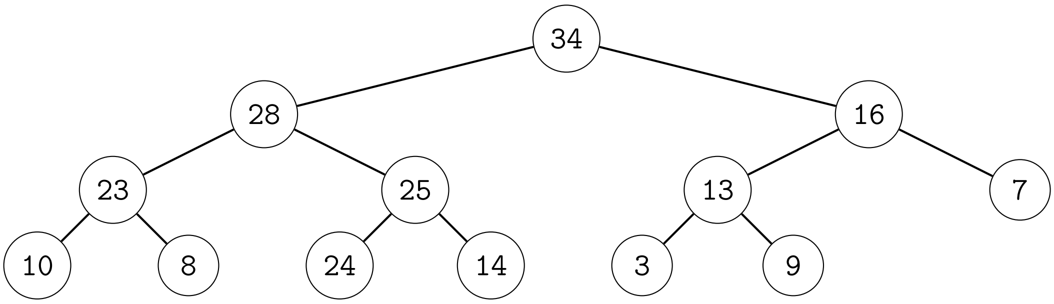
Exercise 109.16.4. It's common to implement a heap by using arrays and implement the ADT using functions instead of using a class with methods. For instance this would be allow one to implement heapsort easily to sort an array directly. Implement the heap ADT described below where an a vector is used to represent the heap.

```
int x;
std::vector< int > heap;

maxheap_insert(heap, 5);   // [5]
maxheap_insert(heap, 7);   // [7, 5]
maxheap_insert(heap, 9);   // [9, 5, 7]

int a = maxheap_delete(heap); // [7, 5]
                           // a = 9
x = maxheap_max(heap);     // x = 9. Root is not deleted.

heap[0] = 1;               // [1, 5]
maxheap_heapify_down(heap, 0); // [5, 1]

heap[1] = 10;              // [5, 10]
maxheap_heapify_up(heap, 1); // [10, 5]

heap.resize(5);
heap[0] = 5;
heap[1] = 7;
heap[2] = 8;
heap[3] = 10;
heap[4] = 2;
maxheap_build(heap);       // [10, 7, 8, 5, 2]

heap.resize(5);
heap[0] = 2;
heap[1] = 6;
heap[2] = 8;
heap[3] = 10;
heap[4] = 5;
maxheap_heapsort(heap)     // [2, 5, 6, 8, 10]
```

The functions above should be function templates. □

109.17 Non-array implementation

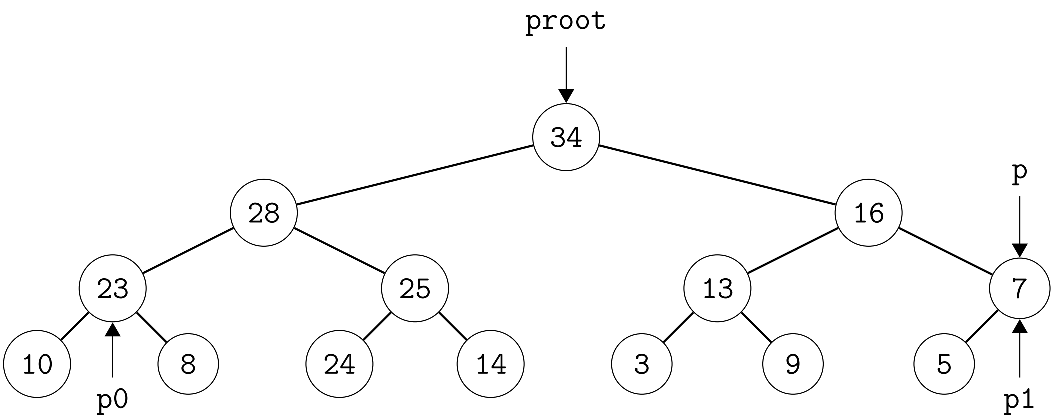
For a non-array implementation of heaps, consider this:



Of course I need a pointer to the root. That's pretty obvious. Another thing to note is that because of the way I add nodes, i.e., always at the lowest level and left-to-right, I should have a pointer to the parent where I have a new node.

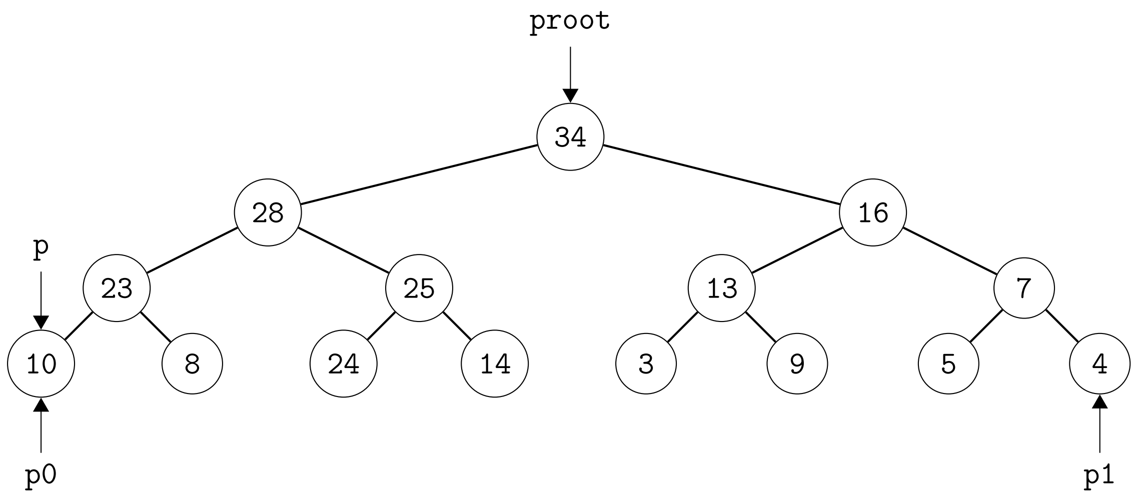
Exercise 109.17.1. Write a function that takes the pointer-to-root of a max heap (or more generally a complete binary tree) and returns the pointer to the node that will be the parent of the next insert. □

However that takes $(\log n)$ time. So we might as well have a pointer that records where the insert should go. But that's not enough. Once a level is full, we need to go to the next level. Therefore we need two (and not one) extra pointer. Like this:

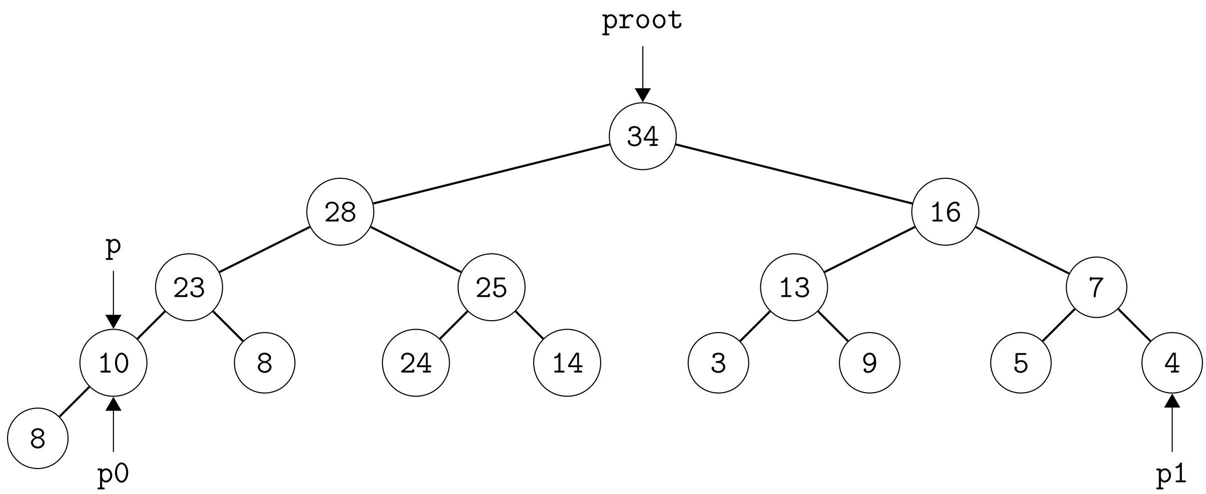


Note that each node has not just left child and right child pointer. When we do delete, we might need to go to the previous level. That means that each node must have a parent pointer too. Of course the parent pointer is also needed when I do heapify-up, right?

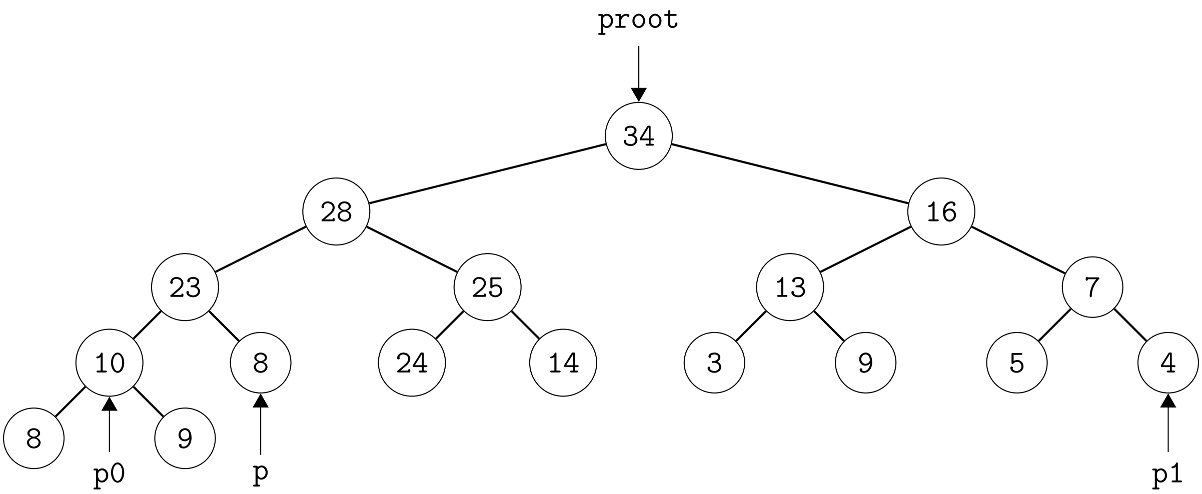
If I insert `4` to the above, then I get this:



If I add `8`, I get this:



Once I add a `9`, I get this:



Get it?

Clearly the above idea works for complete trees where nodes are added left-to-right at the last level.

Note that `p` points to the parent where new nodes should be attached but does not have information on whether it should be as a left or as a right child. You can also include an integer variable `num_children` to tell you how many children there are. If the number of children is 0, then the new node should be attached as a left child. If the number of children is 1, then the new node should be attached as a right child.

Exercise 109.17.2. Implement a max heap using the above idea. □

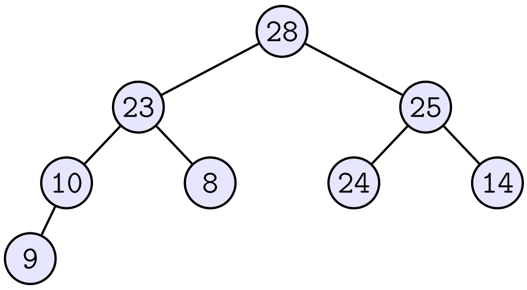
Exercise 109.17.3. Investigate the following implementation: Each level is linked up as a doubly linked list. What are the pros and cons compared to the above? □

File: fakerootheap.tex

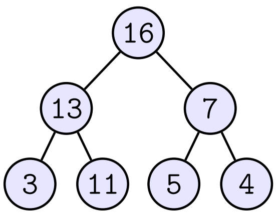
109.18 Fake root heap

Suppose you have two max heaps. One quick way to merge them is to simply create a new root and join the new root to the two roots as parent. The value of the new root can be any arbitrary value that is larger than the values of the two roots. To remind myself that the new root is not really a value in the heap, I can use a flag in each node. Another thing that I can do it to use a value that is sentinel. In the case of max heap, say I call this value ∞.

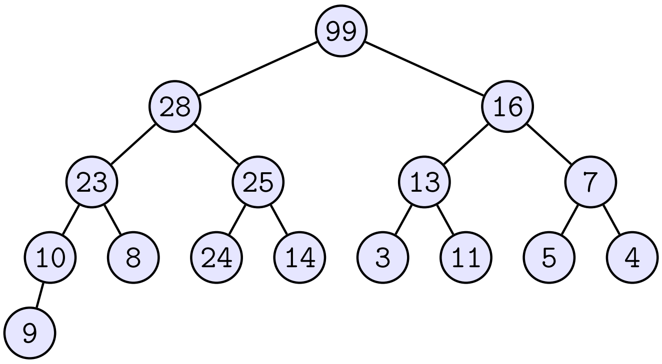
For instance say you have these two max heaps:



and

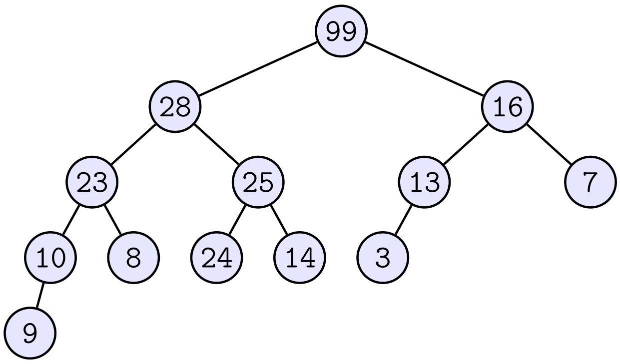


You can merge them together to get this:



where 99 is a fake value, i.e., it's should not be considered part of the values in the heap container.

Of course if two heaps have very different heights, then you won't be able to achieve $O(\lg n)$ runtimes for insert and delete. For instance if the second heap from the above has only 4 values (i.e., 16, 13, 7, 3), then the resulting merged heap (using the above method) would look like this:



File: questions.tex

109.19 Questions

The minheap basically have pointers going to values smaller than the root. But there's no organization about the children. What if the children are organized? What if left subtree have values $\leq k$ and right subtree have values $> k$? Say root is 10. Insert 5 goes into left child first. k is set fo 5. What happens with insert 3? We can change the left, right pointers. So that left points to 3, right points to 5 and $k = 4$? Are there such trees? After insert 1 and 2 which will go into 3, what if I insert 0?

What is we do lazy extract-max? Suppose we extract-root and do not heapify and only heapify during the second extract-root? What is the heapify like if we delete two roots?

Index