Priority Queues (Heaps)

Chapter 6 of textbook 1

Motivating Example

3 jobs have been submitted to a printer, the jobs have sizes 100, 10, 1 page.

Average waiting time for FIFO service, (100+110+111)/3 = 107 time units

Average waiting time for shortest job first service,
(1+11+111)/3 = 41 time units

Need to have a queue which does insert and delete Min based on priority:
Priority Queue

Another examples

- Some applications
  - ordering CPU jobs
  - simulating events
- Problems?
  - short jobs **should go first**
  - earliest (simulated time) events **should go first**

Outline

This topic will:

- Review queues
- Discuss the concept of priority queues
- Look at two simple implementations:
  - Arrays of queues
  - AVL trees
- Introduce heaps, an alternative tree structure which has better run-time characteristics
- Binary Heap
- Heap Sort

Definition

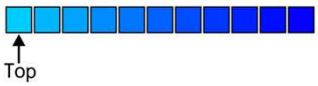
With queues
- The order may be summarized by *first in, first out*

If each object is associated with a priority, we may wish to pop that object which has highest priority

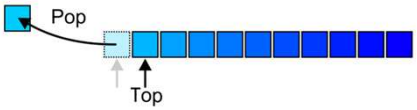
With each pushed object, we will associate a nonnegative integer (0, 1, 2, ...) where:
- The value 0 has the *highest* priority, and
- The higher the number, the lower the priority

Basic Operations

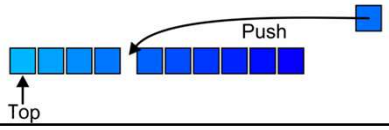
The top of a priority queue is the object with highest priority



Popping from a priority queue removes the current highest priority object:

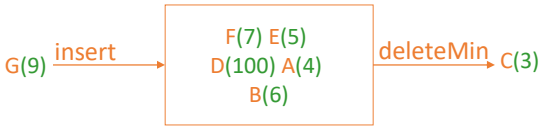


Push places a new object into the appropriate place



Priority Queue ADT

- Priority Queue operations
  - **DeleteMin (or Pop)**
  - Push (or Insert)
  - Top (or Find Min)



- Our goal is to make the run time of each operation as close to **Q**(1) as possible, especially Pop operation.

Implementations of this data structure

We will look at two implementations using data structures we already know:

- Multiple queues—one for each priority
- An AVL tree

The next topic will be a more appropriate data structure: the heap

Multiple Queues

Assume there is a fixed number of priorities, say M

- Create an array of M queues
- Push a new object onto the queue corresponding to the priority
- Top and pop find the first empty queue with highest priority

Multiple Queues

The run times are reasonable:

- Push is $\Theta(1)$
- Top and pop are both $\mathbf{O}(M)$

Unfortunately:

- It restricts the range of priorities
- The memory requirement is $\Theta(M + n)$

AVL Trees

We could simply insert the objects into an AVL tree where the order is given by the stated priority:

- Insertion is $\mathbf{O}(\ln(n))$
- Top is $\mathbf{O}(\ln(n))$ (or Find Min)
- Remove is $\mathbf{O}(\ln(n))$

There is significant overhead for maintaining both the tree and the corresponding balance. AVL Trees are useful for searching, but we don't search in this data structure

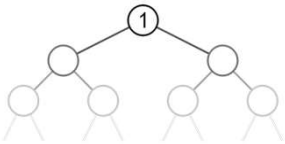
Heaps

Can we do better?

- That is, can we reduce some (or all) of the operations down to $\Theta(1)$?

The next topic defines a *heap*

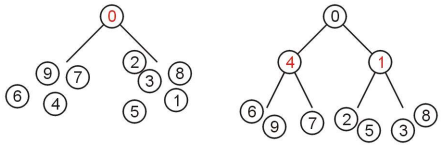
- A tree with the top object at the root
- We will look at binary heaps



Binary Min-Heap: A definition

A non-empty binary tree is **a min-heap** if

- The key associated with the root is less than or equal to the keys associated with either of the sub-trees (if any)
- Both of the sub-trees (if any) are also binary min-heaps



From this definition:

- A single node is a min-heap
- All keys in either sub-tree are greater than the root key => minimum is always at root (or at top)

Definition

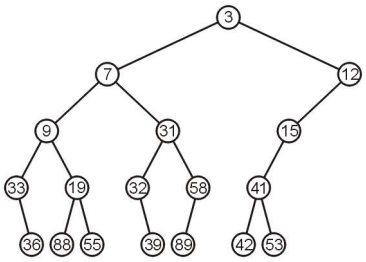
Important:

THERE IS NO OTHER RELATIONSHIP BETWEEN THE ELEMENTS IN THE TWO SUBTREES

Failing to understand this is the greatest mistake a student makes

Example

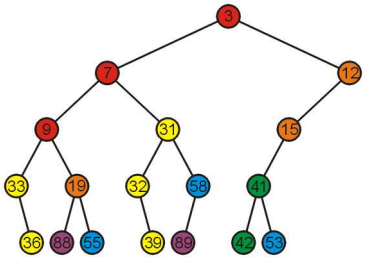
This is a binary min-heap:



Example

Adding colour, we observe

- The left subtree has the smallest (7) and the largest (89) objects
- No relationship between items with similar priority



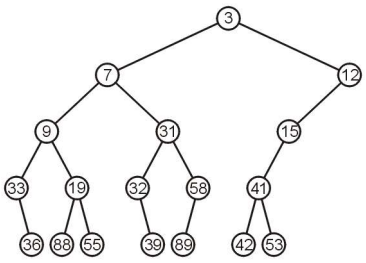
Operations

We will consider three operations:

- Top
- Pop
- Push

Example

We can find the top object in $\Theta(1)$ time: 3



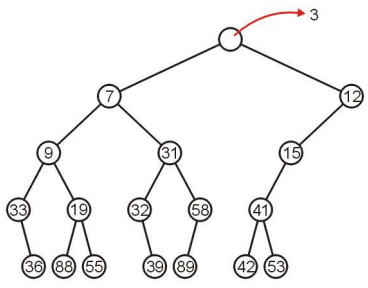
Pop

To remove the minimum object:

- Promote the node of the sub-tree which has the least value
- Recurs down the sub-tree from which we promoted the least value

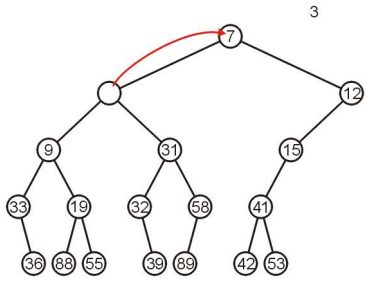
Pop

Using our example, we remove 3:



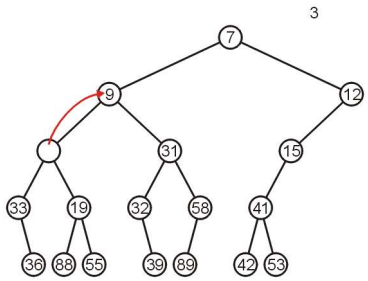
Pop

We promote 7 (the minimum of 7 and 12) to the root:



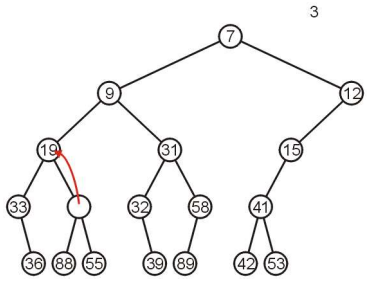
Pop

In the left sub-tree, we promote 9:



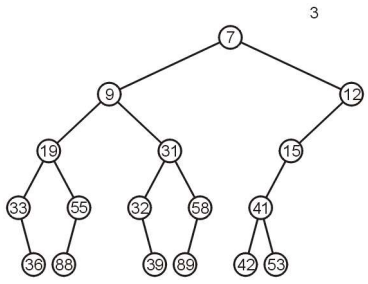
Pop

Recursively, we promote 19:



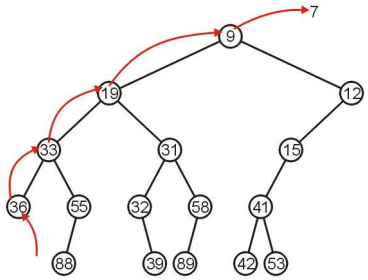
Pop

Finally, 55 is a leaf node, so we promote it and delete the leaf



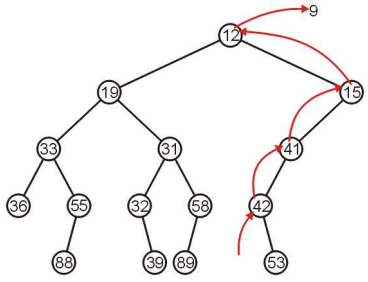
Pop

Repeating this operation again, we can remove 7:



Pop

If we remove 9, we must now promote from the right sub-tree:



Push

Inserting into a heap may be done either:

- At a leaf (move it up if it is smaller than the parent)
- At the root (insert the larger object into one of the subtrees)

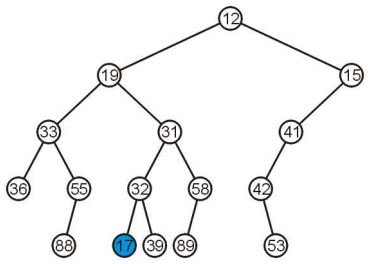
We will use the first approach with binary heaps

- Other heaps use the second

Push

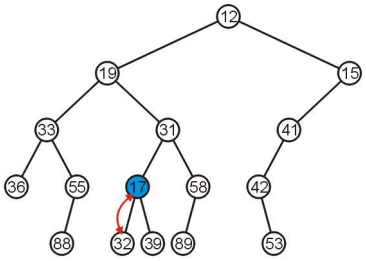
Inserting 17 into the last heap

- Select an arbitrary node to insert a new leaf node:



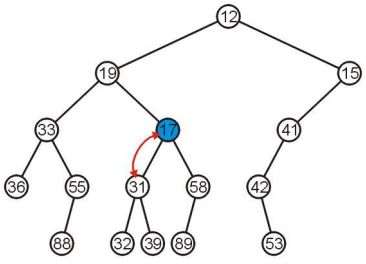
Push

The node 17 is less than the node 32, so we swap them



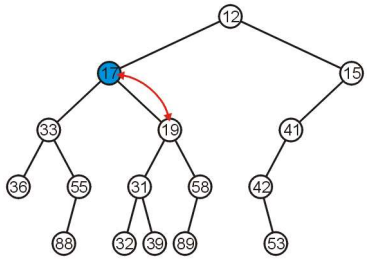
Push

The node 17 is less than the node 31; swap them



Push

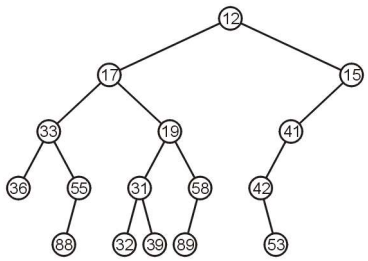
The node 17 is less than the node 19; swap them



Push

The node 17 is greater than 12 so we are finished

Observation: both the left and right subtrees of 19 were greater than 19, thus we are guaranteed that we don't have to send the new node down



Percolation

***percolation up**:* that is, the lighter (smaller) objects move up from the bottom of the min-heap

***percolation down**:* that is, the heavier (bigger) objects move down from the top of the min-heap

Complete Binary Tree as Heap

- Can we use simple binary Tree to implement Heap?
- Answer: No
  - Why ?
- We need a specific binary tree with balance: **Complete Binary Tree**

Complete Binary Tree as Heap

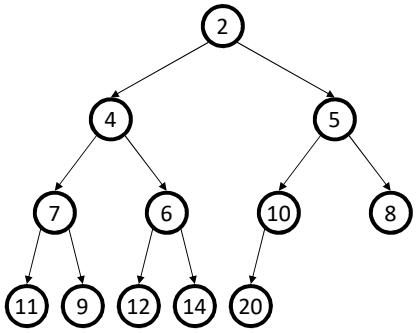
Complete binary tree is a binary tree in which all levels are complete **except the lowest one.**

- **Keeping the balance of binary tree**
- The height h of complete binary tree is O(log N)
- By using complete binary trees, we will be able to maintain, with minimal effort, the complete tree structure.

Complete Binary Tree with **Heap-order property:**

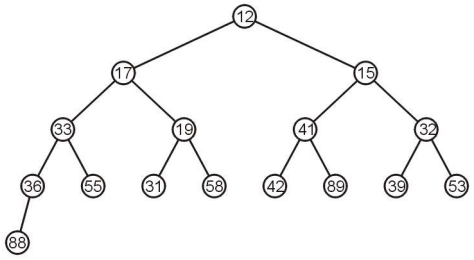
parent's key is less than children's keys
result: minimum is always at the top

THERE IS NO OTHER RELATIONSHIP BETWEEN THE ELEMENTS IN THE TWO SUBTREES



Complete Trees

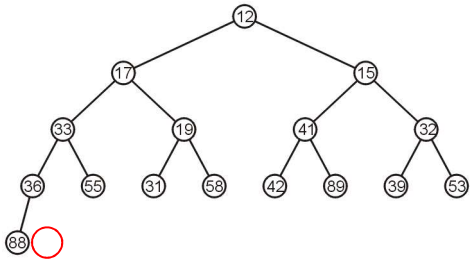
For example, the previous heap may be represented as the following (non-unique!) complete tree:



If we use complete tree, pop and push strategy will be difference to previous slides to maintain the complete tree with small effort.

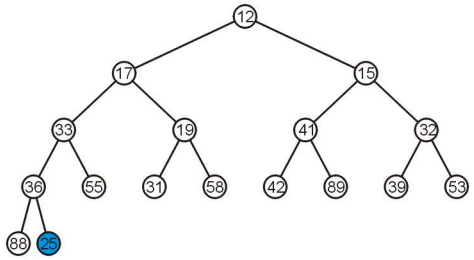
Complete Trees: Push

If we insert into a complete tree, we need only place the new node as a **leaf node** in the appropriate location and percolate up



Complete Trees: Push

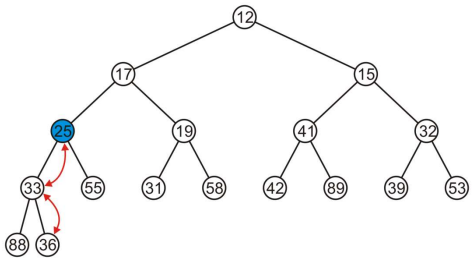
For example, push 25:



Complete Trees: Push

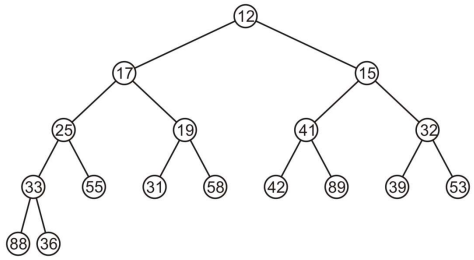
We have to percolate 25 up into its appropriate location

- The resulting heap is still a complete tree



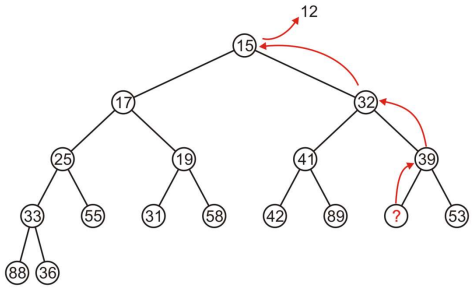
Complete Trees: Pop

Suppose we want to pop the top entry: 12



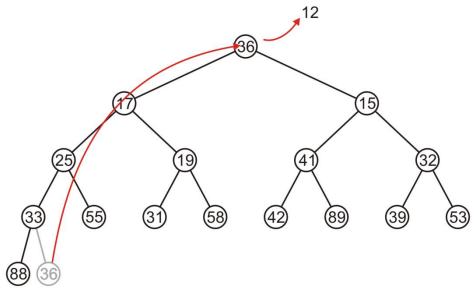
Complete Trees: Pop

If we use the previous strategy, Percolating up creates a hole leading to a non-complete tree



Complete Trees: Pop

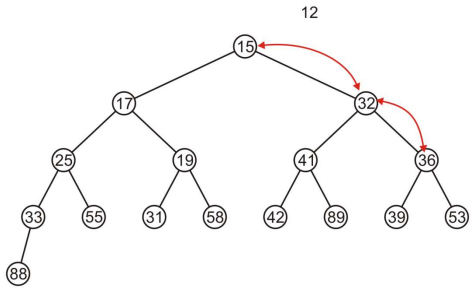
Alternatively, copy the last entry in the heap to the root



Complete Trees: Pop

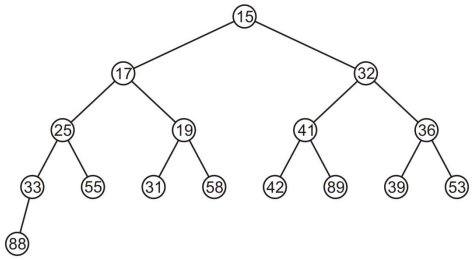
Now, percolate 36 down swapping it with the smallest of its children

- We halt when both children are larger



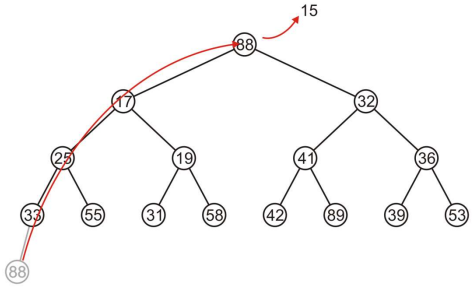
Complete Trees: Pop

The resulting tree is now still a complete tree:



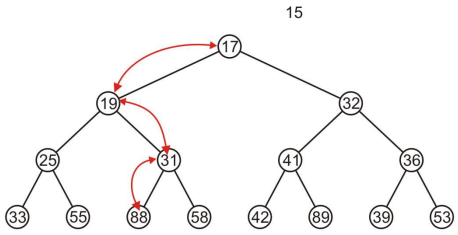
Complete Trees: Pop

Again, popping 15, copy up the last entry: 88



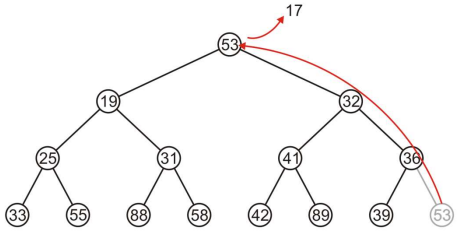
Complete Trees: Pop

This time, it gets percolated down to the point where it has no children



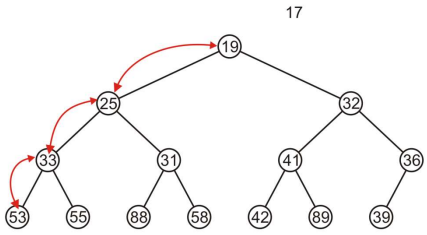
Complete Trees: Pop

In popping 17, 53 is moved to the top



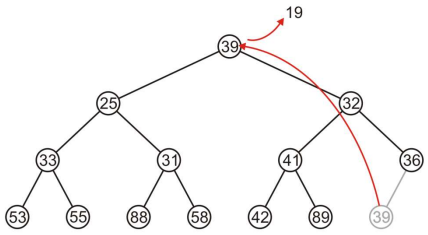
Complete Trees: Pop

And percolated down, again to the deepest level



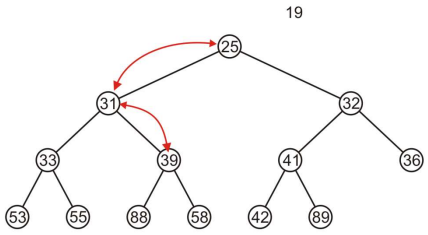
Complete Trees: Pop

Popping 19 copies up 39



Complete Trees: Pop

Which is then percolated down to the second deepest level



Complete Tree

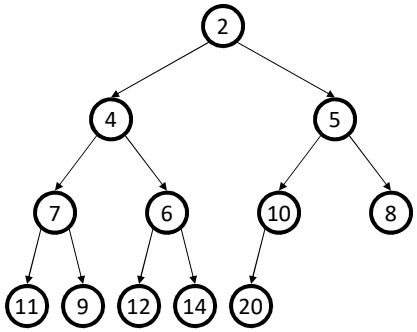
Therefore, we can maintain the complete-tree shape of a heap

We may store a complete tree using an array:

- A complete tree is filled in breadth-first traversal order
- The array is filled using breadth-first traversal

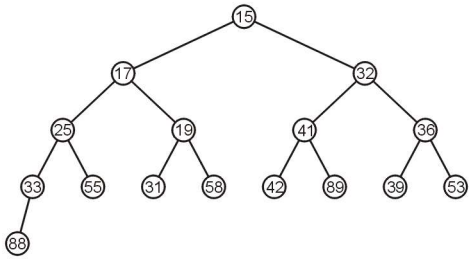
Quiz

- Show the step-by-step when we call DeleteMin operation?
- Show the step-by-step when we insert 3 into this Heap



Array Implementation

- Advantages of storing a heap as an array rather than a pointer-based binary tree
  - It is easy to store a complete tree as an array and still be able to find children and parents easily.
    - For element at array i: left child (2i), right child (2i+1), parent (i/2)
  - Lower memory usage
  - Easier memory management
- Disadvantage is that an estimate of the maximum heap size is required in advance

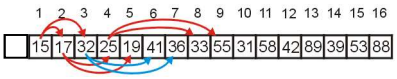


	15	17	32	25	19	41	36	33	55	31	58	42	89	39	53	88

a breadth-first traversal yields

Array Implementation

Recall that If we associate an index–starting at 1–with each entry in the breadth-first traversal, we get:



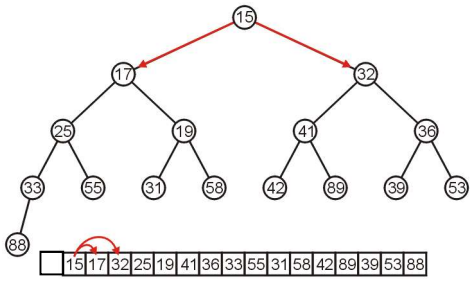
Given the entry at index k, it follows that:

- The parent of node is a $k/2$
- The children are at $2k$ and $2k + 1$

Cost (trivial): start array at position 1 instead of position 0

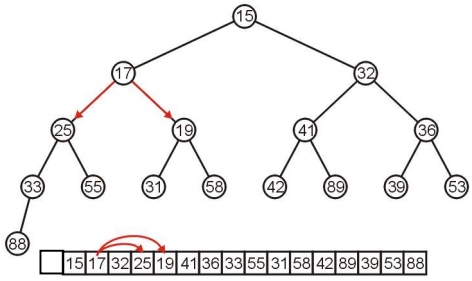
Array Implementation

The children of 15 are 17 and 32:



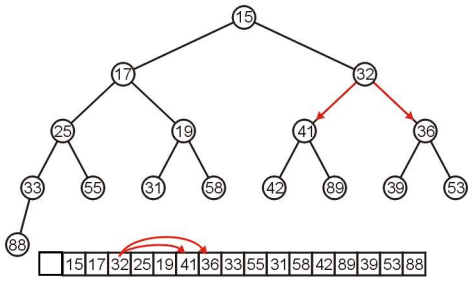
Array Implementation

The children of 17 are 25 and 19:



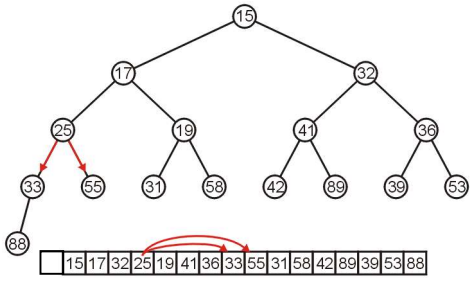
Array Implementation

The children of 32 are 41 and 36:



Array Implementation

The children of 25 are 33 and 55:



Array Implementation: pushing strategy

If the heap-as-array has **`count`** entries, then the next empty node in the corresponding complete tree is at location **`posn = count + 1`**

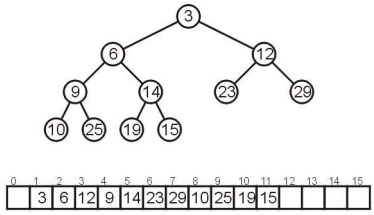
We compare the item at location **`posn`** with the item at **`posn/2`**

If they are out of order

- Swap them, set **`posn /= 2`** and repeat

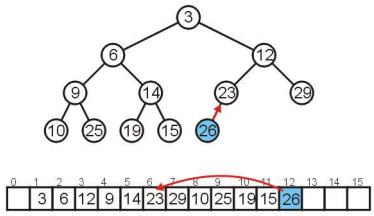
Array Implementation

Consider the following heap, both as a tree and in its array representation



Array Implementation: Push

Inserting 26 requires no changes



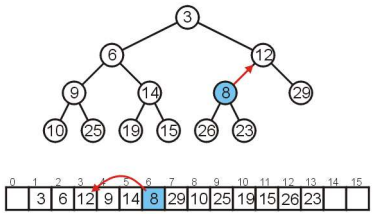
Array Implementation: Push

Inserting 8 requires a few percolations:

- Compare 8 and 23 => Swap 8 and 23

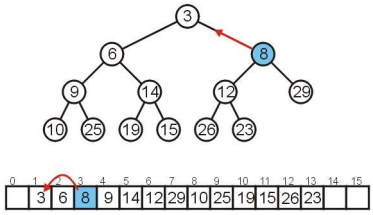
Array Implementation: Push

Swap 8 and 12



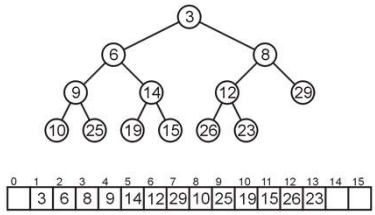
Array Implementation: Push

At this point, it is greater than its parent, so we are finished



Array Implementation: Pop

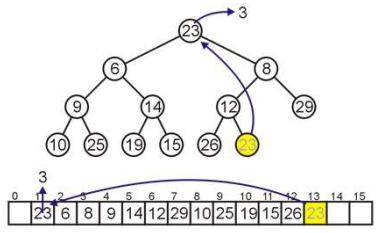
As before, popping the top has us copy the last entry to the top



Array Implementation: Popping strategy

Instead, consider this strategy:

- Copy the last object, 23, to the root

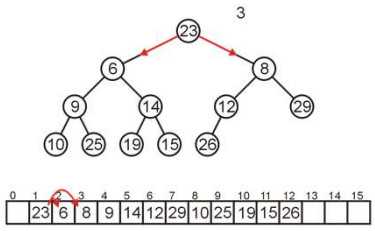


Array Implementation: Popping strategy

Now percolate down

Compare Node 1 with its children: Nodes 2 and 3

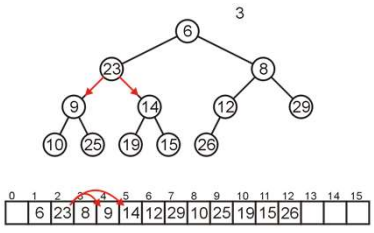
- Swap 23 and 6



Array Implementation: Popping strategy

Compare Node 2 with its children: Nodes 4 and 5

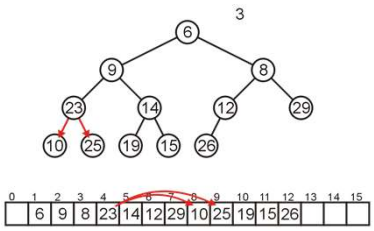
- Swap 23 and 9



Array Implementation: Pop

Compare Node 4 with its children: Nodes 8 and 9

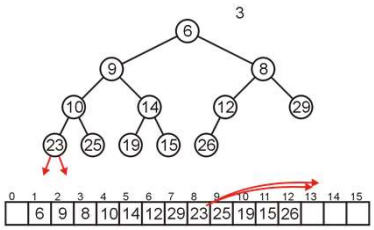
- Swap 23 and 10



Array Implementation: Pop

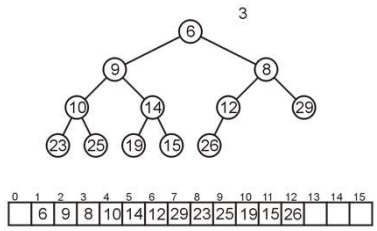
The children of Node 8 are beyond the end of the array:

- Stop



Array Implementation: Pop

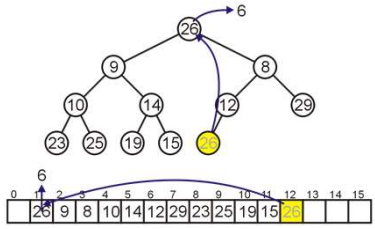
The result is a **binary min-heap**



Array Implementation: Pop

Dequeuing the minimum again:

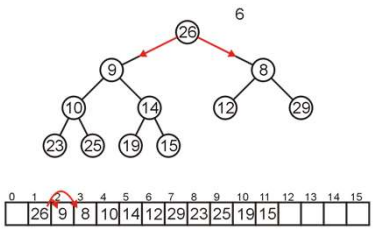
- Copy 26 to the root



Array Implementation: Pop

Compare Node 1 with its children: Nodes 2 and 3

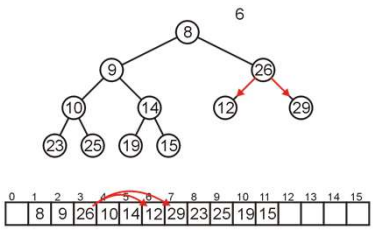
- Swap 26 and 8



Array Implementation: Pop

Compare Node 3 with its children: Nodes 6 and 7

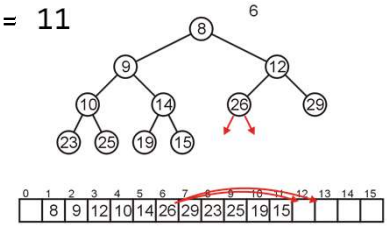
- Swap 26 and 12



Array Implementation: Pop

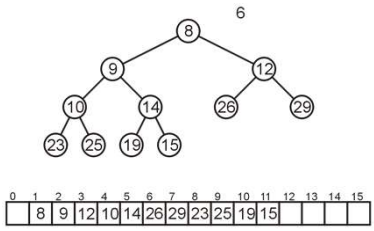
The children of Node 6, Nodes 12 and 13 are unoccupied

- Currently, `count == 11`



Array Implementation: Pop

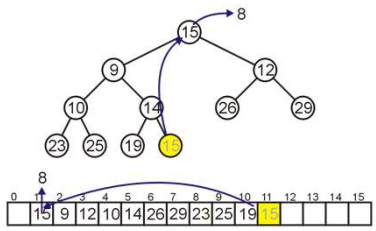
The result is a min-heap



Array Implementation: Pop

Dequeuing the minimum a third time:

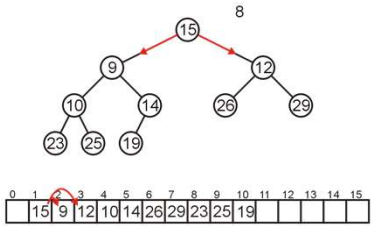
- Copy 15 to the root



Array Implementation: Pop

Compare Node 1 with its children: Nodes 2 and 3

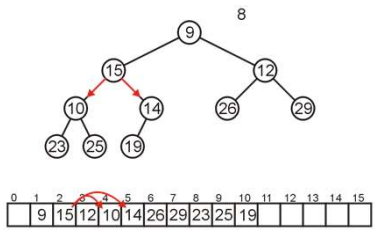
- Swap 15 and 9



Array Implementation: Pop

Compare Node 2 with its children: Nodes 4 and 5

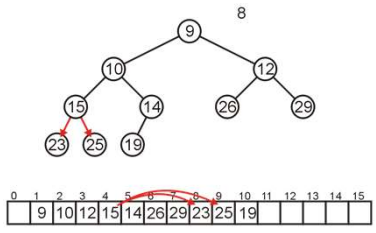
- Swap 15 and 10



Array Implementation: Pop

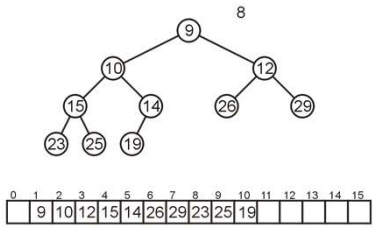
Compare Node 4 with its children: Nodes 8 and 9

- 15 < 23 and 15 < 25 so stop



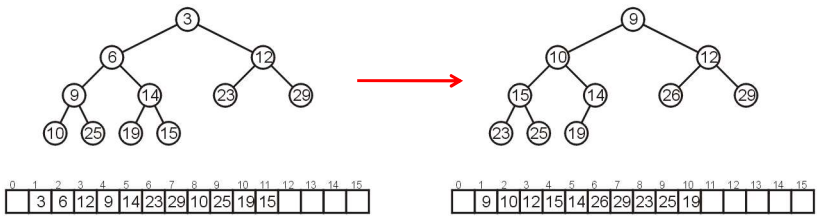
Array Implementation: Pop

The result is a properly formed binary min-heap



Array Implementation: Pop

After all our modifications, the final heap is



Run-time Analysis

Accessing the top object is $\Theta(1)$

Popping the top object is $\Theta(\ln(n))$

- We copy something that is already in the lowest depth—it will likely be moved back to the lowest depth

How about push?

Run-time Analysis

If we are inserting an object less than the root (at the front), then the run time will be $\Theta(\ln(n))$

If we insert at the back (greater than any object) then the run time will be $\Theta(1)$

How about an arbitrary insertion?

- **An average run time of $\Theta(1)$**

Run-time comparison

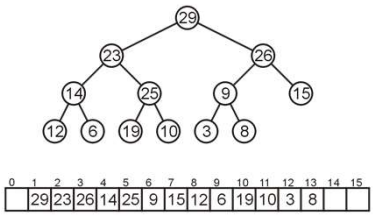
This table shows us the time complexity of known data structures for this data structure

Implementation	Insertion	Deletion	Find Min
Unordered array	1	N	N
Unordered linked list	1	N	N
Ordered array	N	1	1
Ordered linked list	N	1	1
Binary Search Tree	lnN(average)	lnN(average)	lnN(average)
AVL Tree	lnN	lnN	lnN
Binary Heap	**1(average)**	**lnN**	**1**

Binary Max Heaps

A binary max-heap is identical to a binary min-heap except that the parent is always larger than either of the children

For example, the same data as before stored as a max-heap yields



Implementation of binary Heap

- Please read the textbook 1 for more details.

```
/* Place in implementation file */
struct HeapStruct
{
    int Capacity;
    int Size;
    ElementType *Elements;
};
```

Heap Sort

Given an array with n numbers, process of heap sort algorithm is following:

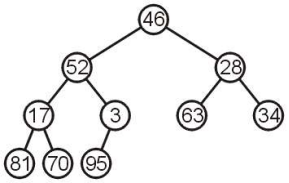
1. Convert the array into a heap
2. Pop n items from the heap

In-place Heapification

Now, consider this unsorted array:

46	52	28	17	3	63	34	81	70	95

This array represents the following complete tree:



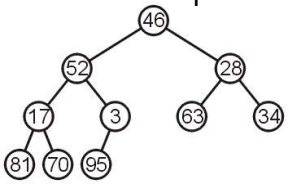
This is neither a min-heap, max-heap, or binary search tree

In-place Heapification

Now, consider this unsorted array:

46	52	28	17	3	63	34	81	70	95

Additionally, because arrays start at 0 (we started at entry 1 for binary heaps) , we need different formulas for the children and parent



The formulas are now:

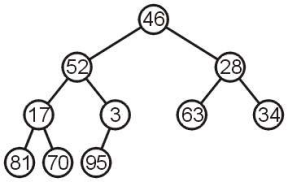
Children	`2*k + 1`	`2*k + 2`
Parent	`(k + 1)/2 - 1`	

In-place Heapification

Can we convert this complete tree into a max heap?

Restriction:

- The operation must be done in-place

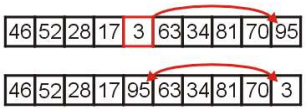


Answer: Floyd Algorithm

- The basic idea:
  - Start with an array of all n elements
  - Start traversing backwards – e.g. from the bottom of the tree to the top
  - Call percolate_Down(...) per each node

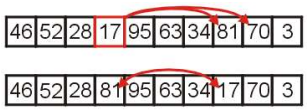
Example Heap Sort

We compare 3 with its child and swap them



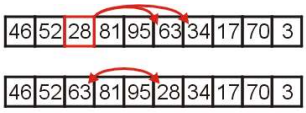
Example Heap Sort

We compare 17 with its two children and swap it with the maximum child (70)



Example Heap Sort

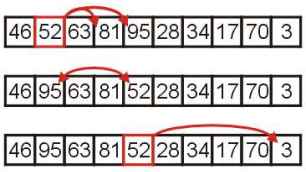
We compare 28 with its two children, 63 and 34, and swap it with the largest child



Example Heap Sort

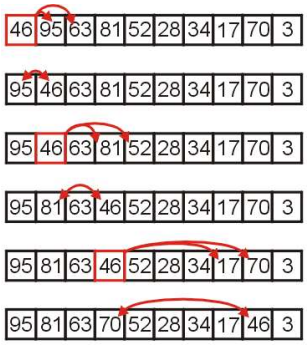
We compare 52 with its children, swap it with the largest

- Recursing, no further swaps are needed



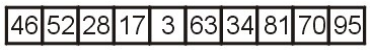
Example Heap Sort

Finally, we swap the root with its largest child, and recurse, swapping 46 again with 81, and then again with 70

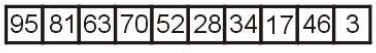


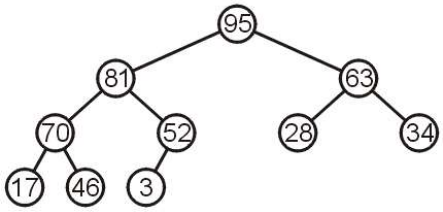
Heap Sort Example

We have now converted the unsorted array



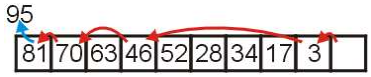
into a max-heap:



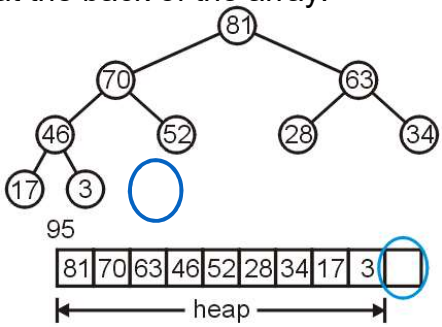


Heap Sort Example

Suppose we pop the maximum element of this heap

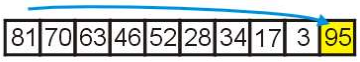


This leaves a gap at the back of the array:

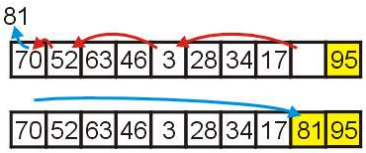


Heap Sort Example

This is the last entry in the array, so why not fill it with the largest element?



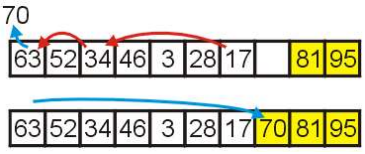
Repeat this process: pop the maximum element, and then insert it at the end of the array:



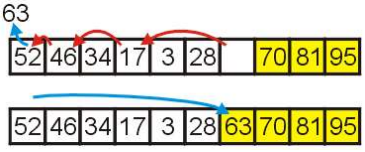
Heap Sort Example

Repeat this process

- Pop and append 70



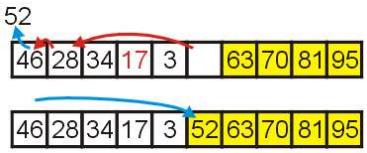
- Pop and append 63



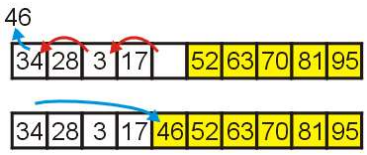
Heap Sort Example

We have the 4 largest elements in order

- Pop and append 52



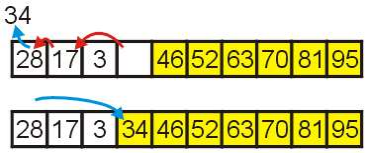
- Pop and append 46



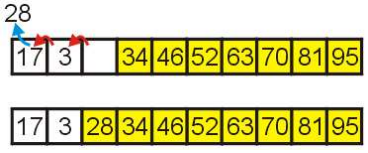
Heap Sort Example

Continuing...

- Pop and append 34

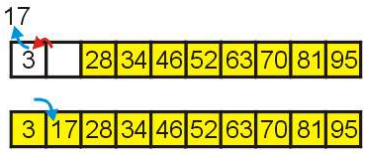


- Pop and append 28



Heap Sort Example

Finally, we can pop 17, insert it into the 2nd location, and the resulting array is sorted



Black Board Example

Sort the following 12 entries using heap sort

34, 15, 65, 59, 79, 42, 40, 80, 50, 61, 23, 46

Heap Sort

Heapification runs in $\Theta(n)$

Popping n items from a heap of size n, as we saw, runs in $\Theta(n \ln(n))$ time

- We are only making one additional copy into the blank left at the end of the array

Therefore, the total algorithm will run in $\Theta(n \ln(n))$ time

Run-time Summary

The following table summarizes the run-times of heap sort

Case	Run Time	Comments
Worst	$\Theta(n \ln(n))$	No worst case
Average	$\Theta(n \ln(n))$	
Best	$\Theta(n)$	All or most entries are the same

Summary of Heap Sort

We have seen our first in-place $\Theta(n \ln(n))$ sorting algorithm:

- Convert the unsorted list into a max-heap as complete array
- Pop the top n times and place that object into the vacancy at the end
- It requires $\Theta(1)$ additional memory—it is truly in-place

It is a nice algorithm; however, we had two other faster $n \ln(n)$ algorithms; however:

- Merge sort requires $\Theta(n)$ additional memory
- Quick sort requires $\Theta(\ln(n))$ additional memory

Next week

- Hashing (chapter 5 of textbook 1)
- Please do all homework