### Heaps & Heapsort

| Charles Babbage (1864) | Analytic Engine (schematic) |

### Heaps, heap sort and priority queues

- **Priority Queue**: data structure that maintains a set $S$ of elements.
- Each element $v$ in $S$ has a key $key(v)$ that denotes the priority of $v$.
- Priority Queue provides support for:
  - adding, deleting elements,
  - selection / extraction of smallest (Min prioQ) or largest (Max prioQ) key element,
  - changing key value.
Applications

**E.g. used in managing real time events where we want to get the earliest next event and events are added / deleted on the fly.**

**Sorting**
- build a prioQ
- Iteratively extract the smallest element

PrioQs can be implemented using **heaps**

---

**Heaps**

**Heap:** array representation of a complete binary tree
- every level is completely filled except the bottom level: filled from left to right
- Can compute the index of parent and children: **WHY?**
  - parent(i) = floor((i-1)/2)
  - leftChild(i) = 2i+1
  - rightChild(i) = 2(i+1)

Max Heap property:
- for all nodes i>0: \( A[\text{parent}(i)] \geq A[i] \)
Max heaps have the max at the root

Min heaps have the min at the root
Heapify(A,i,n)

To create a heap at index i, assuming left(i) and right(i) are heaps, bubble A[i] down: swap with max child until heap property holds

heapify(A,i,n):
# precondition
# n is the size of the heap
# tree left(i) and tree right(i) are heaps

......

# postcondition: tree A[i] is a heap

Swapping Down

Swapping down enforces (max) heap property at the swap location:

new

y

x

\[ \text{new} < x \text{ and } y < x : \quad x > y \text{ and } x > \text{new} \]

\[ \text{swap}(x, \text{new}) \]

Are we done now?

NO! When we have swapped we need to carry on checking whether new is in heap position. We stop when that is the case.
Heap Extract

Heap extract:
Delete (and return) root

Step 1: replace root with last array element to keep completeness
Step 2: reinstate the heap property
Which element does not necessarily have the heap property?

How can it be fixed? Complexity?
heapify the root $O(\log n)$

Swap down: swap with maximum (maxheap), minimum (minheap) child as necessary, until in place.
Sometimes called bubble down

Correctness based on the fact that we started with a heap, so the children of the root are heaps

Heap Insert

Step 1: put a new value into first open position (maintaining completeness), i.e. at the end of the array, but now we potentially violated the heap property, so:

Step 2: bubble up

- Re-enforcing the heap property

- Swap with parent, if new value > parent, until in the right place.

- The heap property holds for the tree below the new value, when swapping up. WHY? We only compared the new element to the parent, not to the sibling!
Swapping up

Swapping up enforces heap property for the sub tree below the new, inserted value:

if (new > x) swap(x,new)  
\[ x > y, \text{ therefore } new > y \]

Building a heap

heapify performs at most \(\log n\) swaps

why? what is \(n\)?

buildheap: builds a heap out of an array:

- the leaves are all heaps WHY?
- heapify backwards starting at last internal node

WHY backwards?  
WHY last internal node?  
which node is that?
LERT'S DO THE BUILDHEAP!

[4, 8, 7, 2, 14, 1]

[8, 7, 14, 2]

Suggestions? ...

Complexity buildheap
Complexity buildheap

Initial thought: $O(n \log n)$, but

- half of the heaps are height 0
- quarter are height 1
- only one is height $\log n$

It turns out that $O(n \log n)$ is not tight!
complexity buildheap

max # swaps, see a pattern?
(What kind of growth function do you expect?)

<table>
<thead>
<tr>
<th>height</th>
<th>max # swaps</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2*1+2 = 4</td>
</tr>
<tr>
<td>3</td>
<td>2*4+3 = 11</td>
</tr>
</tbody>
</table>

complexity buildheap

<table>
<thead>
<tr>
<th>height</th>
<th>max # swaps</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 = 2^0-2</td>
</tr>
<tr>
<td>1</td>
<td>1 = 2^1-3</td>
</tr>
<tr>
<td>2</td>
<td>2*1+2 = 4  = 2^2-4</td>
</tr>
<tr>
<td>3</td>
<td>2*4+3 = 11 = 2^3-5</td>
</tr>
</tbody>
</table>
Conjecture:
height = \( h \)
max #swaps = \( 2^{h+1} \cdot (h+2) \)

Proof: induction
base?
step:
height = \( (h+1) \)
max #swaps:
\[ 2 \cdot (2^{h+1} \cdot (h+2) \cdot (h+1)) = 2^{h+2} \cdot 2h + 2h + 2 \]
\[ = 2^{h+2} \cdot 4 + 2h + 1 \]
\[ = 2^{h+2} \cdot (h+3) \]
\[ = 2^{h+2} \cdot ((h+1)+2) \]

\( n \) nodes \( \rightarrow \Theta(n) \) swaps

\[ T(n) = 2 \cdot T(n/2) + \lg n \]

Master theorem \( \Theta(n^{\lg_2 2}) = \Theta(n) \)
Heapsort, complexity

heapsort(A):
  buildheap(A)  # O(n)
  for i = n-1 downto 1:  # O( n )
    # put max at end array
    n=n-1
    # max is removed from heap
    # reinstate heap property  # * ( lg n )
    heapify:
    heapExtract:
    buildheap:
    heapsort:
    space: in place:
How not to heapExtract, heapInsert

# These "snail" implementations are NOT preserving the algorithm
# complexity of extractMin: log n and insert: log n and are therefore
# INCORRECT! from a complexity point of view (even though they are
# functionally correct). Remember one of the goals of our course:
# implementing the algorithms maintaining the analyzed complexity
# What are their complexities?

def snailExtractMin(A):
    n = len(A)
    if n == 0:
        return None
    min = A[0]
    A[0]=A[n-1]
    A.pop()
    buildHeap(A)  # O(n)
    return min

def snailInsert(A,v):
    A.append(v)
    buildHeap(A)  # O(n)

Priority Queues

heaps can be used to implement priority queues:
- each value associated with a key
- max priority queue S has operations that maintain
  the heap property of S
  - max(S) returning max element
  - Extract-max(S) extracting and returning max
    element
  - increase key(S,x,k) increasing the key value of x
  - insert(S,x)
    - put x at end of S
    - bubble x up in place