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

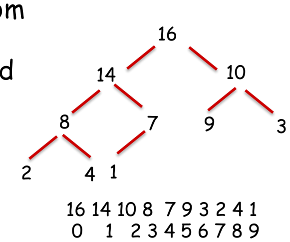
E.g. used in managing real time events where we want to get the earliest next event and events are added / deleted on the fly.

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[parent(i)] >= 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

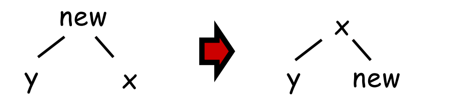
```
heapify(A,i,n):
# precondition
# n is the size of the heap
# tree left(i) and tree right(i) are heaps

    …….

# postcondition:  tree A[i] is a heap
```

Swapping Down

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



new<x and y<x:
    swap(x,new)

x>y and x>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.

6

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

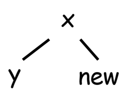
Step 1: put a new value into first open position (maintaining completeness), i.e. at the end of the array, but now we potentially violated the heap property, so:

Step 2: bubble up

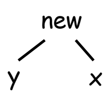
- **Re-enforcing the heap property**
- Swap with parent, if new value > parent, until in the right place.
- The heap property holds for the tree below the new value, when swapping up. **WHY? We only compared the new element to the parent, not to the sibling!**

Swapping up

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



if (new > x) swap(x,new)



x>y, therefore new > y

9

Building a heap

heapify performs at most lg n swaps

why? what is n?

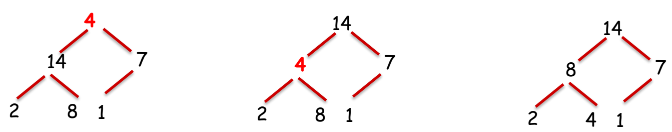
buildheap: builds a heap out of an array:

- 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]



Complexity buildheap

Suggestions? ...

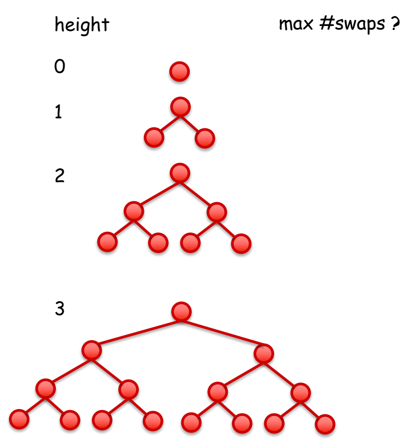
Complexity buildheap

Initial thought: O(n lgn), but

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

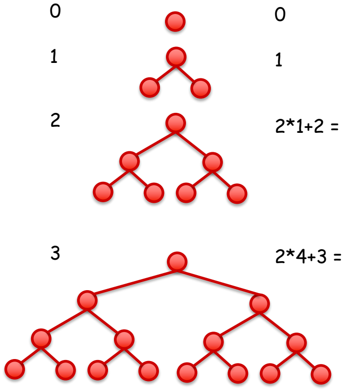
It turns out that O(n lgn) is not tight!

complexity buildheap



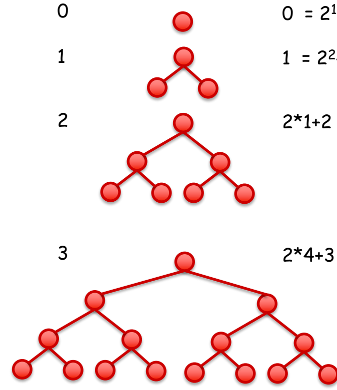
complexity buildheap

height	max #swaps, see a pattern? (What kind of growth function do you expect ?)
0	0
1	1
2	2*1+2 = 4
3	2*4+3 = 11



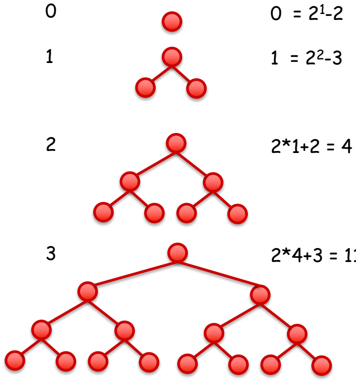
complexity buildheap

height	max #swaps
0	$0 = 2^1-2$
1	$1 = 2^2-3$
2	$2*1+2 = 4 = 2^3-4$
3	$2*4+3 = 11 = 2^4-5$



complexity buildheap

height	max #swaps
0	0 = 2^1-2
1	1 = 2^2-3
2	2*1+2 = 4 = 2^3-4
3	2*4+3 = 11 = 2^4-5



Conjecture:
 height = h
 max #swaps = 2^{h+1}-(h+2)

Proof: induction
 base?
 step:
  height = (h+1)
  max #swaps:
   $2*(2^{h+1}-(h+2))+(h+1)$
  $= 2^{h+2}-2h-4+h+1$
  $= 2^{h+2}-(h+3)$
  $= 2^{(h+1)+1}-((h+1)+2)$

n nodes → Θ(n) swaps

See it the Master theorem way

T(n) = 2*T(n/2)+ lg n

Master theorem $\Theta(n^{\lg_2 2}) = \Theta(n)$

18

Heapsort, complexity

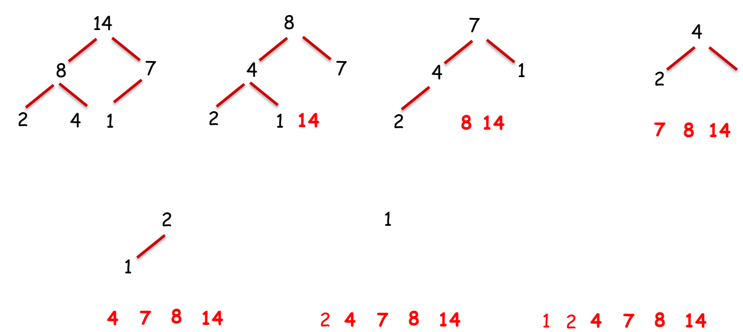
```
heapsort(A):
  buildheap(A)     # O( n )
  for i = n-1 downto 1 :            #   O( ( n )
    # put max at end array

    # max is removed from heap
    n=n-1

    # reinstate heap property    #       * ( lg n)  )
```

- heapify: Θ(lgn)
- heapExtract: Θ(lg n)
- buildheap: Θ(n)
- heapsort: Θ(n lg n)
- space: in place: Θ(n)

DO THE HEAPSORT, DO IT, DO IT!



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)
```

21

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