STEM Support

https://infomcgillstem.wixsite.com/stemsupportmcgill

MSSG = McGill Space systems group

http://www.mcgillspace.com/#!/

Tea and snacks provided, it's FREE.

THURSDAY NOVEMBER 9TH 5:30-7:30PM

Where: Rutherford Physics BOARD ROOM
What: Come chat about diversity in the Canadian Space Program AND MORE!

ALL Students and Staff welcome!
RECALL: \textit{min} Heap (definition)

Complete binary tree with (unique) comparable elements, such that each node’s element is less than its children’s element(s).
Heap index relations

parent = child / 2
left = 2*parent
right = 2*parent + 1
buildHeap()
add()

removeMin()
downHeap(1, size)
How to build a heap? (slight variation)

```java
buildHeap()
{
  // assume that an array already contains size elements
  for (k = 2; k <= size; k++)
    upHeap( k )
}
```
How to build a heap? (slight variation)

```java
buildHeap(){
    // assume that an array already contains size elements
    for (k = 2; k <= size; k++)
        upHeap( k )
}

upHeap(k){
    i = k
    while (i > 1) and ( heap[i] < heap[i / 2] ){
        swapElement(i, i/2)
        i = i/2
    }
}
```
Recall last lecture:  Worse case of buildHeap
Thus, worst case: $\text{buildHeap}$ is $\Theta(n \log_2 n)$

Next, I will show you a $\Theta(n)$ algorithm for building a heap.
How to build a heap? (fast)

Half the nodes of a heap are leaves.
(Each leaf is a heap with one node.)

The last non-leaf node has index $\frac{\text{size}}{2}$. 
How to build a heap? (fast)

```c
buildHeapFast()
{
    // assume that heap[ ] array contains size elements
    for (k = size/2; k >= 1; k--)
        downHeap( k, size )
}
```
k = 3
\[ k = 3 \]

downHeap(3, 6)
\begin{align*}
\text{downHeap}(3, 6) \\
k = 3
\end{align*}
downHeap(2, 6)

k = 2
k = 2
k = 1

downHeap( 1, 6 )

1  2  3  4  5  6

---------------
w  p  f  x  r  t
k = 1

1 2 3 4 5 6

f p w x r t

k = 1
k = 1
buildHeapFast(list){
    copy list into a heap array
    for (k = size/2; k >= 1; k--)
        downHeap(k, size)
}

Claim: this algorithm is $\Theta(n)$.

What is the intuition for why this algorithm is so fast?
We tend to draw binary trees like this:

But the number of nodes doubles at each level. So we should draw trees like this:
buildheap algorithms

last lecture

Most nodes swap $\sim h$ times in worst case.

today

Few nodes swap $\sim h$ times in worst case.
How to show buildHeapFast is \( \Theta(n) \)?

The worst case number of swaps needed to downHeap node \( i \) is the height of that node.

\[
t(n) = \sum_{i=1}^{n} \text{height of node } i
\]

\( \frac{1}{2} \) of the nodes do no swaps.
\( \frac{1}{4} \) of the nodes do at most one swap.
\( \frac{1}{8} \) of the nodes do at most two swaps....
Let’s do the calculation for a tree that whose last level is full.
Worse case of buildHeapFast?

How many elements at level \( l \)? \((l \in 0, \ldots, h)\)

What is the height of each level \( l \) node?
Worse case of buildHeapFast?

level $l$ has $2^l$ elements, $l \in 0, \ldots, h$

level $l$ nodes have height $h - l$.

$$t(n) = \sum_{i=1}^{n} \text{height of node } i$$

= ?
Worse case of buildHeapFast?

level $l$ has $2^l$ elements, $l \in 0,\ldots, h$

level $l$ nodes have height $h - l$.

$$t(n) = \sum_{i=1}^{n} \text{height of node } i$$

$$= \sum_{i=0}^{h} (h - l) \ 2^l$$
\[ t_{\text{worst case}}(h) = \sum_{l=0}^{h} (h - l) \ 2^l \]

\[ = h \sum_{l=0}^{h} 2^l - \sum_{l=0}^{h} l \ 2^l \]

Easy (number of nodes) Difficult (sum of node depths)
\[ t_{\text{worst case}}(h) = \sum_{l=0}^{h} (h - l) \ 2^l \]

\[ = h \sum_{l=0}^{h} 2^l - \sum_{l=0}^{h} l \ 2^l \]

\[ = h(2^{h+1} - 1) - (h - 1)2^{h+1} - 2 \]
\[
\sum_{l=0}^{h} l \ 2^l = \sum_{l=0}^{h} l (2^{l+1} - 2^l) \quad \text{(trick)}
\]

\[
= \sum_{l=0}^{h} l \ 2^{l+1} - \sum_{l=0}^{h} l \ 2^l
\]

\[
= \sum_{l=0}^{h} l \ 2^{l+1} - \sum_{l=0}^{h-1} (l + 1) \ 2^{l+1} \quad \text{Second term index goes to h-1 only}
\]

\[
= h \ 2^{h+1} + 2 \sum_{l=0}^{h-1} (l - (l + 1)) \ 2^l
\]

\[
= h \ 2^{h+1} - 2 \sum_{l=0}^{h-1} 2^l
\]

\[
= h \ 2^{h+1} - 2(2^h - 1)
\]

\[
= (h - 1)2^{h+1} + 2
\]
\[ t_{\text{worst case}}(h) = \sum_{l=0}^{h} (h - l) 2^l \]

\[ = h \sum_{l=0}^{h} 2^l - \sum_{l=0}^{h} l 2^l \]

\[ = h(2^{h+1} - 1) - (h - 1)2^{h+1} - 2 \quad \text{from above} \]

\[ = 2^{h+1} - h - 2 \]

Since \( n = 2^{h+1} - 1 \), we get:

\[ t_{\text{worst case}}(n) = n - \log(n + 1) \]
Summary: buildheap algorithms

last lecture

$O(n \log_2 n)$

height

today

$h$

$O(n)$
Heapsort

Given a list with size elements:

Build a heap.

Repeatedly call removeMin() and put the removed elements into a list.
“in place” Heapsort

Given an array \( \text{heap}[\ ] \) with size elements:

heapsort()
{
    buildheap()
    for \( i = 1 \) to size{
        swapElements( heap[1], heap[size + 1 - i])
        downHeap( 1, size - i )
    }
    return reverse(heap)
}
a d b e l u k f w | w d b e l u k f | a
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>d</td>
<td>b</td>
<td>e</td>
<td>l</td>
<td>u</td>
<td>k</td>
<td>f</td>
<td>w</td>
</tr>
<tr>
<td>b</td>
<td>d</td>
<td>k</td>
<td>e</td>
<td>l</td>
<td>u</td>
<td>w</td>
<td>f</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>a</td>
<td>d</td>
<td>b</td>
<td>e</td>
<td>l</td>
<td>u</td>
<td>k</td>
<td>f</td>
<td>w</td>
</tr>
<tr>
<td>b</td>
<td>d</td>
<td>k</td>
<td>e</td>
<td>l</td>
<td>u</td>
<td>w</td>
<td>f</td>
<td>a</td>
</tr>
<tr>
<td>f</td>
<td>d</td>
<td>k</td>
<td>e</td>
<td>l</td>
<td>u</td>
<td>w</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>d</td>
<td>f</td>
<td>k</td>
<td>e</td>
<td>l</td>
<td>u</td>
<td>w</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>d</td>
<td>e</td>
<td>k</td>
<td>f</td>
<td>l</td>
<td>u</td>
<td>w</td>
<td>b</td>
<td>a</td>
</tr>
</tbody>
</table>
1 2 3 4 5 6 7 8 9

<table>
<thead>
<tr>
<th>a</th>
<th>d</th>
<th>b</th>
<th>e</th>
<th>l</th>
<th>u</th>
<th>k</th>
<th>f</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>d</td>
<td>k</td>
<td>e</td>
<td>l</td>
<td>u</td>
<td>w</td>
<td>f</td>
<td>a</td>
</tr>
<tr>
<td>d</td>
<td>e</td>
<td>k</td>
<td>f</td>
<td>l</td>
<td>u</td>
<td>w</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>e</td>
<td>f</td>
<td>k</td>
<td>w</td>
<td>l</td>
<td>u</td>
<td>d</td>
<td>b</td>
<td>a</td>
</tr>
</tbody>
</table>
heapsort(list){
    buildheap(list)
    for i = 1 to size{
        swapElements( heap[1], heap[size + 1 - i])
        downHeap( 1, size - i)
    }
    return reverse(heap)
}