CSE 332 Autumn 2023
Lecture 7: Priority Queues & Recurrences

Nathan Brunelle

http://www.cs.uw.edu/332
**Thinking through implementations**

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Worst case time to insert</th>
<th>Worst case time to deleteMin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Array</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Unsorted Linked List</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Sorted Circular Array</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Sorted Linked List</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Binary Heap</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
</tbody>
</table>

Note: Assume we know the maximum size of the PQ in advance
Heap – Priority Queue Data Structure

• Idea: We need to keep some ordering, but it doesn’t need to be perfectly sorted
• $\Theta(\log n)$ worst case for deleteMin and insert
Heap Insert

```plaintext
insert(item)
    put item in the “next open” spot (keep tree complete)
    while (item.priority < parent(item).priority)
        swap item with parent
    swap item with parent

```
Heap deleteMin

deleMin() {
    min = root
    br = bottom-right item
    move br to the root
    while (br > either of its children) {
        swap br with its smallest child
    }
    return min
}
Percolate Up and Down

• Goal: restore the “Heap Property”

• Percolate Up:
  • Take a node that may be smaller than a parent, repeatedly swap with a parent until it is larger than its parent

• Percolate Down:
  • Take a node that may be larger than one of its children, repeatedly swap with smallest child until both children are larger

• Worst case running time of each:
  • \( \Theta(\log n) \)
Representing a Heap

- Every complete binary tree with the same number of nodes uses the same positions and edges
- Use an array to represent the heap
- Index of root: 1
- Parent of node $i$: $\left\lfloor \frac{i}{2} \right\rfloor$
- Left child of node $i$: $2 \cdot i$
- Right child of node $i$: $2 \cdot i + 1$
- Location of the leaves: last half
Other Operations

• Increase Key
  • Given the index of an item in the PQ, subtract from its priority value
  • Update the priority, then percolate [up or down?]

• Decrease Key
  • Given the index of an item in the PQ, add to its priority value
  • Update the priority, then percolate [up or down?]

• Remove
  • Given the item at the given index from the PQ
  • Change its priority to $-\infty$
  • deleteMin
Building a Heap From “Scratch”

- Suppose we had $n$ items and wanted to “heapify” them.
Floyd’s buildHeap method

- Working towards the root, one row at a time, percolate down

```java
buildHeap()
    { for(int i = size; i>0; i--){ percolateDown(i); } }
```
Floyd’s buildHeap method

• Suppose we had $n$ items and wanted to “heapify” them

```
buildHeap()
{
    for(int i = size; i>0; i--){
        percolateDown(i);
    }
}
```
Floyd’s buildHeap method

- Suppose we had \( n \) items and wanted to “heapify” them
Floyd’s buildHeap method

• Suppose we had \( n \) items and wanted to “heapify” them
Floyd’s buildHeap method

• Suppose we had $n$ items and wanted to “heapify” them
Floyd's buildHeap method

• Suppose we had \( n \) items and wanted to “heapify” them

```
buildHeap()
{
    for(int i = size; i>0; i--){
        percolateDown(i);
    }
}
```
Floyd’s buildHeap method

• Suppose we had \( n \) items and wanted to “heapify” them

```
buildHeap()
{
    for(int i = size; i > 0; i--){
        percolateDown(i);
    }
}
```
How long did this take?

- Worst case running time of buildHeap:
  - No node can percolate down more than the height of its subtree
    - When \( i \) is a leaf: 0
    - When \( i \) is second-from-last level: 1
    - When \( i \) is third-from-last level: 2
  - Overall Running time:
    - \( \frac{n}{2} \) of the items are leaves
      - 0 swaps total
    - \( \frac{n}{4} \) of the items are at second-from-last level
      - \( \frac{n}{4} \) total swaps
    - \( \frac{n}{8} \) of the items are at third-from-last level
      - \( \frac{n}{8} \times 2 \) total swaps
      - \( \frac{n}{16} \times 3 \) total swaps
    - This sum converges to \( 2n \in \Theta(n) \)

```java
buildHeap()
{
    for(int i = size; i>0; i--){
        percolateDown(i);
    }
}
```
End-of-Yarn Finding

1. Set aside the already-obtained “beginning”

2. If you see the end of the yarn, you’re done!

3. Separate the pile of yarn into 2 piles, note which connects to the beginning (call it pile A, the other pile B)

4. Count the number of strands crossing the piles

5. If the count is even, pile A contains the end, else pile B does
Analysis of Recursive Algorithms

• Overall structure of recursion:
  • Do some non-recursive “work”
  • Do one or more recursive calls on some portion of your input
  • Do some more non-recursive “work”
  • Repeat until you reach a base case

• Running time: \( T(n) = T(p_1) + T(p_2) + \cdots + T(p_x) + f(n) \)
  • The time it takes to run the algorithm on an input of size \( n \) is:
    • The sum of how long it takes to run the same algorithm on each smaller input
    • Plus the total amount of non-recursive work done at that step

• Usually:
  • \( T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n) \)
    • Called “divide and conquer”
  • \( T(n) = T(n - c) + f(n) \)
    • Called “chip and conquer”
How Efficient Is It?

- $T(n) = \text{count}(n) + T\left(\left\lfloor \frac{n}{2} \right\rfloor \right)$
- $T(n) = 5 + T\left(\left\lfloor \frac{n}{2} \right\rfloor \right)$
- Base case: $T(1) = 5$

$T(n)$ = “cost” of running the entire algorithm on an $n$ inch string

$\text{count}(n)$ = “cost” of counting the crossing strands (I arbitrarily picked 5)
Let’s Solve the Recurrence!

\[ T(1) = 5 \]

\[ T(n) = 5 + T\left(\frac{n}{2}\right) \]

\[ 5 + T\left(\frac{n}{4}\right) \]

\[ 5 + T\left(\frac{n}{8}\right) \]

\[ \vdots \]

\[ \sum_{i=1}^{\lceil \log_2 n \rceil} 5 = 5 \lceil \log_2 n \rceil \]

\[ T(n) = \Theta(\log n) \]

\[ T\left(\frac{n}{2}\right) = 5 + T\left(\frac{n}{4}\right) \]

\[ \lceil \log_2 n \rceil \]
Recursive Linear Search

search(value, list){
    if(list.isEmpty()){
        return false;
    }
    if (value == list[0]){
        return true;
    }
    list.remove(0);
    return search(value, list);
}
Unrolling Method

• Repeatedly substitute the recursive part of the recurrence
• $T(n) = T(n - 1) + c$
• $T(n) = T(n - 2) + c + c$
• $T(n) = T(n - 3) + c + c + c$
• ...
• $T(n) = c + c + c + \cdots + c$
  • How many $c$’s?
Recursive List Summation

\[
T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(1)
\]

\[
\sum(\text{list})\{
    \text{return}\ \sum\_\text{helper}(\text{list}, 0, \text{list.size});
\}

\sum\_\text{helper}(\text{list}, \text{low}, \text{high})\{
    \text{if}\ (\text{low} == \text{high})\{ \text{return}\ \text{0}; \}
    \text{if}\ (\text{low} == \text{high}-1)\{ \text{return}\ \text{list[low]}; \}
    \text{middle} = (\text{high} + \text{low})/2;
    \text{return}\ \sum\_\text{helper}(\text{list}, \text{low}, \text{middle}) + \sum\_\text{helper}(\text{list}, \text{middle}, \text{high});
\}
**Tree Method**

\[ T(n) = 2T\left(\frac{n}{2}\right) + c \]

- Red box represents a problem instance
- Blue value represents time spent at that level of recursion

\[ T(n) = \sum_{i=1}^{\log_2 n} 2^i \cdot c \]
Recursive List Summation

\[ T(n) = \sum_{i=1}^{\log_2 n} 2^i \cdot c \]

\[ = c \cdot \sum_{i=1}^{\log_2 n} 2^i \]

\[ = c \left( \frac{1 - 2^{\log_2 n}}{1 - 2} \right) \]
Binary Search

search(value, sortedArr){
    return helper(value, sortedArr, 0, sortedArr.length);
}
helper(value, arr, low, high){
    if (low == high){ return false; }
    mid = (high + low) / 2;
    if (arr[mid] == value){ return true; }
    if (arr[mid] < value){ return helper(value, arr, mid+1, high); }
    else { return helper(value, arr, low, mid); }
}