Discussion Notes, week of 10/11/04 – 10/15/05

1. Announcements
   a. Project 1 Difficulties and Resolutions
   b. HW2 is due Thurs. Oct 14
   c. Project 2 will be out on Thurs. Oct 14

2. Heaps
   a. Example, building a heap
   b. Heapsort

3. Project 1 Solution
   a. Breadth First Search vs. Depth First Search
   b. 4x4 example
Heap Functions

void buildHeap(Item heap[], int n)
{
    for (unsigned int i = n/2; i > 0; --i)
        fixDown(heap, n, i);
}

void fixDown(Item heap[], int heapsize, int k)
{
    while(2*k <= heapsize) {
        int j = 2*k;
        if (j < heapsize && heap[j] < heap[j + 1]) ++j;
        if (heap[k] >= heap[j]) break;
        exch(heap[k], heap[j]);
        k=j;
    }
}

void fixUp(Item heap[], int k)
{
    while (k > 1 && heap[k/2] < heap[k]) {
        exch(heap[k], heap[k/2]);
        k /= 2;
    }
}

void insert(Item item)
{
    pq[++N] = item;
    fixUp(pq, N);
}

void getmax()
{
    exch(pq[1], pq[N]);
    fixDown(pq, N-1, 1);
    return pq[N--];
}

void sortHeap(Item heap[], int n)
{
    buildHeap(heap, n);
    for (unsigned int i = n; i >= 2; --i) {
        exch(heap[i], heap[1]);
        fixDown(heap, i-1, 1);
    }
}
Heapify Example

Input Array

<table>
<thead>
<tr>
<th>4</th>
<th>3</th>
<th>7</th>
<th>13</th>
<th>1</th>
<th>20</th>
<th>12</th>
<th>16</th>
<th>2</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>7</td>
<td>13</td>
<td><strong>18</strong></td>
<td>20</td>
<td>12</td>
<td>16</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>7</td>
<td><strong>16</strong></td>
<td>18</td>
<td>20</td>
<td>12</td>
<td><strong>13</strong></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td><strong>20</strong></td>
<td>16</td>
<td>18</td>
<td>7</td>
<td>12</td>
<td>13</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td><strong>18</strong></td>
<td>20</td>
<td>16</td>
<td><strong>3</strong></td>
<td>7</td>
<td>12</td>
<td>13</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td><strong>20</strong></td>
<td>18</td>
<td><strong>4</strong></td>
<td>16</td>
<td>3</td>
<td>7</td>
<td>12</td>
<td>13</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>18</td>
<td><strong>12</strong></td>
<td>16</td>
<td>3</td>
<td>7</td>
<td><strong>4</strong></td>
<td>13</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Heapified Array

| 20 | 18 | 12 | 16 | 3 | 7 | 4 | 13 | 2 | 1 |

Draw diagrams as necessary. I know you can build them fairly quickly, much faster than I can draw them in Word, so I didn’t bother to include them with the solution.

Insertion Example

Insert 16

Add 16 to end

| 20 | 18 | 12 | 16 | 3 | 7 | 4 | 13 | 2 | 1 | 16 |

Call FixUp()

| 20 | 18 | 12 | 16 | **16** | 7 | 4 | 13 | 2 | 1 | **3** |

New Heap with 16 inserted

| 20 | 18 | 12 | 16 | **16** | 7 | 4 | 13 | 2 | 1 | **3** |
Sorting Example

20 18 12 16 16 7 4 13 2 1 3

Swap last element with root element

3 18 12 16 16 7 4 13 2 1 20

Call FixDown on arraysize-1

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>3</td>
<td>12</td>
<td>16</td>
<td>16</td>
<td>7</td>
<td>4</td>
<td>13</td>
<td>2</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>18</td>
<td>16</td>
<td>12</td>
<td>3</td>
<td>16</td>
<td>7</td>
<td>4</td>
<td>13</td>
<td>2</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>18</td>
<td>16</td>
<td>12</td>
<td>13</td>
<td>16</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

repeat until entire array is covered

1  2  3  4  7 12 13 16 16 18 20

Show steps as necessary
Depth First Search (stack)

picks a direction and goes with it, based on direction preference (S, E, N, W preference in project 1)
creates an extremely inefficient path for all but the simplest cases

DFS – Path length of 7

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Crystals</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Start</td>
<td>1 6</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2 5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3 4</td>
</tr>
</tbody>
</table>

Breadth First Search (queue)

fans out in “circular” pattern
covers tiles more evenly
finds ideal path

BFS – Path length of 3

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>Crystals</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1 2</td>
</tr>
<tr>
<td>1</td>
<td>Start</td>
<td>1 2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Keys to project 1:
Need some sort of extra data structure:
Keep track of data about tiles, either mirroring the map or just a list
- remember the “parent tile” for each tile – easy way to retrace steps
- tracking depth properly made tracing your steps and path length easier
- having this keeps track of what tiles you’ve visited as well
Or, keep a stack of the solution path, popping off incorrect tiles
- harder to maintain, but can work
- also need to track what tiles you’ve covered separately

Start early! Most problems were encountered because people waited to the last minute!