Recall Queues

- FIFO: First-In, First-Out
- Some contexts where this seems right?
- Some contexts where some things should be allowed to skip ahead in the line?

Queues that Allow Line Jumping

- Need a new ADT
- Operations: Insert an Item, Remove the “Best” Item

Priority Queue ADT

1. **PQueue data**: collection of data with priority
2. **PQueue operations**
   - insert
   - deleteMin
3. **PQueue property**: for two elements in the queue, x and y, if x has a lower priority value than y, x will be deleted before y

Applications of the Priority Queue

- Select print jobs in order of decreasing length
- Forward packets on routers in order of urgency
- Select most frequent symbols for compression
- Sort numbers, picking minimum first
- Anything greedy

Potential Implementations

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>deleteMin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted list (Array)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Unsorted list (Linked-List)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Sorted list (Array)</td>
<td>O(n)</td>
<td>O(1)*</td>
</tr>
<tr>
<td>Sorted list (Linked-List)</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
Recall From Lists, Queues, Stacks

- Use an ADT that corresponds to your needs
- The right ADT is efficient, while an overly general ADT provides functionality you aren’t using, but are paying for anyways
- Heaps provide $O(\log n)$ worst case for both insert and deleteMin, $O(1)$ average insert

Binary Heap Properties

1. Structure Property
2. Ordering Property

Tree Review

- root($T$):
- leaves($T$):
- children($B$):
- parent($H$):
- siblings($E$):
- ancestors($F$):
- descendants($G$):
- subtree($C$):

More Tree Terminology

- depth($B$):
- height($G$):
- degree($B$):
- branching factor($T$):

Brief interlude: Some Definitions:

A **Perfect** binary tree – A binary tree with all leaf nodes at the same depth. All internal nodes have 2 children.

- height $h$
- $2^{h-1}$ – 1 nodes
- $2^h - 1$ non-leaves
- $2^h$ leaves

Heap **Structure** Property

- A binary heap is a **complete** binary tree. **Complete binary tree** – binary tree that is completely filled, with the possible exception of the bottom level, which is filled left to right.

Examples:
Representing Complete Binary Trees in an Array

From node i:
- left child:
- right child:
- parent:

Implicit (array) implementation:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<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>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

Why this approach to storage?

Heap Order Property

Heap order property: For every non-root node X, the value in the parent of X is less than (or equal to) the value in X.

Heap Operations

- findMin:
- insert(val): percolate up.
- deleteMin: percolate down.

Heap – Insert(val)

Basic Idea:
1. Put val at “next” leaf position
2. Percolate up by repeatedly exchanging node until no longer needed

Insert: percolate up
Insert Code (optimized)

```java
void insert(Object o) {
    assert(!isFull());
    size++;
    newPos = percolateUp(size, o);
    Heap[newPos] = o;
}
```

```java
int percolateUp(int hole, Object val) {
    while (hole > 1 && val < Heap[hole/2]) {
        Heap[hole] = Heap[hole/2];
        hole /= 2;
    }
    return hole;
}
```

DeleteMin: percolate down

```java
Object deleteMin() {
    assert(!isEmpty());
    returnVal = Heap[1];
    size--;
    newPos = percolateDown(1, Heap[size+1]);
    Heap[newPos] = Heap[size + 1];
    return returnVal;
}
```

```java
int percolateDown(int hole, Object val) {
    while (2*hole <= size) {
        left = 2*hole;
        right = left + 1;
        if (right <= size && Heap[right] < Heap[left]) {
            target = right;
        } else {
            target = left;
        }
        if (Heap[target] < val) {
            Heap[hole] = Heap[target];
            hole = target;
        } else {
            break;
        }
    }
    return hole;
}
```

Heap – Deletemin

Basic Idea:
1. Remove root (that is always the min!)
2. Put “last” leaf node at root
3. Find smallest child of node
4. Swap node with its smallest child if needed.
5. Repeat steps 3 & 4 until no swaps needed.

CSE 326: Data Structures

Binary Heaps
Building a Heap

- Adding the items one at a time is $O(n \log n)$ in the worst case
- I promised $O(n)$ for today

Working on Heaps

- What are the two properties of a heap?
  - Structure Property
  - Order Property
- How do we work on heaps?
  - Fix the structure
  - Fix the order

Buildheap pseudocode

```java
private void buildHeap() {
    for (int i = currentSize/2; i > 0; i-- )
        percolateDown( i );
}
```

BuildHeap: Floyd’s Method

Add elements arbitrarily to form a complete tree.
Pretend it’s a heap and fix the heap-order property!
More Priority Queue Operations

- **decreaseKey**
  - given a pointer to an object in the queue, reduce its priority value
  
  Solution: change priority and ______________________

- **increaseKey**
  - given a pointer to an object in the queue, increase its priority value
  
  Why do we need a pointer? Why not simply data value?
  Solution: change priority and ______________________
More Priority Queue Operations

- **Remove**(objPtr)
  - given a pointer to an object in the queue, remove the object from the queue
  
  **Solution:** set priority to negative infinity, percolate up to root and deleteMin

- **FindMax**

Facts about Heaps

**Observations:**
- Finding a child/parent index is a multiply/divide by two
- Operations jump widely through the heap
- Each percolate step looks at only two new nodes
- Inserts are at least as common as deleteMins

**Realities:**
- Division/multiplication by powers of two are equally fast
- Looking at only two new pieces of data: bad for cache!
- With huge data sets, disk accesses dominate

A Solution: \(d\)-Heaps

- Each node has \(d\) children
- Still representable by array
- Good choices for \(d\):
  - (choose a power of two for efficiency)
  - fit one set of children in a cache line
  - fit one set of children on a memory page/disk

One More Operation

- Merge two heaps

- Add the items from one into another?
  - \(O(n \log n)\)

- Start over and build it from scratch?
  - \(O(n)\)

CSE 326: Data Structures

**Priority Queues**

Leftist Heaps & Skew Heaps
New Heap Operation: Merge

Given two heaps, merge them into one heap
- first attempt: insert each element of the smaller heap into the larger.
  *runtime:*
- second attempt: concatenate binary heaps’ arrays and run buildHeap.
  *runtime:*

Leftist Heaps

Idea:
Focus all heap maintenance work in one small part of the heap

Leftist heaps:
1. Most nodes are on the left
2. All the merging work is done on the right

Definition: Null Path Length

null path length (npl) of a node $x$ = the number of nodes between $x$ and a null in its subtree

$\text{npl}(x) = \min \text{distance to a descendant with 0 or 1 children}$

- $\text{npl}(\text{null}) = -1$
- $\text{npl}(\text{leaf}) = 0$
- $\text{npl}(\text{single-child node}) = 0$

Equivalent definitions:
1. $\text{npl}(x)$ is the height of the largest complete subtree rooted at $x$
2. $\text{npl}(x) = 1 + \min \{\text{npl}(\text{left}(x)), \text{npl}(\text{right}(x))\}$

Leftist Heap Properties

- Heap-order property
  - parent’s priority value is $\leq$ to children’s priority values
  - result: minimum element is at the root
- Leftist property
  - For every node $x$, $\text{npl}(\text{left}(x)) \geq \text{npl}(\text{right}(x))$
  - result: tree is at least as “heavy” on the left as the right

Are leftist trees...
- complete?
- balanced?

Right Path in a Leftist Tree is Short (#1)

Claim: The right path is as short as any in the tree.

Proof: (By contradiction)
Pick a shorter path: $D_1 < D_2$
Say it diverges from right path at $x$

$\text{npl}(\text{L}) \leq D_1 - 1$ because of the path of length $D_1 - 1$ to null

$\text{npl}(\text{R}) \geq D_2 - 1$ because every node on right path is leftist

Leftist property at $x$ violated!
Right Path in a Leftist Tree is Short (#2)

Claim: If the right path has \( r \) nodes, then the tree has at least \( 2^r - 1 \) nodes.

Proof: (By induction)

Base case: \( r=1 \). Tree has at least \( 2^1 - 1 = 1 \) node.

Inductive step: assume true for \( r' < r \). Prove for tree with right path at least \( r \).

1. Right subtree: right path of \( r-1 \) nodes
   \[ 2^{r-1} - 1 \] right subtree nodes (by induction)

2. Left subtree: also right path of length at least \( r-1 \) (by previous slide)
   \[ 2^{r-1} - 1 \] left subtree nodes (by induction)

Total tree size: \( 2^{r-1} - 1 + 2^{r-1} - 1 + 1 = 2^r - 1 \)

Why do we have the leftist property?

Because it guarantees that:
- the right path is really short compared to the number of nodes in the tree
- A leftist tree of \( N \) nodes, has a right path of at most \( \log(N+1) \) nodes

Idea – perform all work on the right path

Merge two heaps (basic idea)

- Put the smaller root as the new root,
- Hang its left subtree on the left.
- Recursively merge its right subtree and the other tree.

Merging Two Leftist Heaps

- \( \text{merge}(T_1, T_2) \) returns one leftist heap containing all elements of the two (distinct) leftist heaps \( T_1 \) and \( T_2 \)

Let’s do an example, but first...

Other Heap Operations

- insert ?
- deleteMin ?
Operations on Leftist Heaps

- **merge** with two trees of total size $n$: $O(\log n)$
- **insert** with heap size $n$: $O(\log n)$
  - pretend node is a size 1 leftist heap
  - insert by merging original heap with one node
- **deleteMin** with heap size $n$: $O(\log n)$
  - remove and return root
  - merge left and right subtrees

Leftist Merge Example

Sewing Up the Example

Finally…

Leftist Heaps: Summary

**Good**
- 
- 

**Bad**
- 
- 

Random Definition: Amortized Time

*Amortized time:* Running time limit resulting from “writing off” expensive runs of an algorithm over multiple cheap runs of the algorithm, usually resulting in a lower overall running time than indicated by the worst possible case.

If $M$ operations take total $O(M \log N)$ time, amortized time per operation is $O(\log N)$

Difference from average time:
Skew Heaps

Problems with leftist heaps
- extra storage for npl
- extra complexity/logic to maintain and check npl
- right side is “often” heavy and requires a switch

Solution: skew heaps
- “blindly” adjusting version of leftist heaps
- merge always switches children when fixing right path
- amortized time for: merge, insert, deleteMin = O(log n)
- however, worst case time for all three = O(n)

Merging Two Skew Heaps

Only one step per iteration, with children always switched

Example

Skew Heap Code

```c
void merge(heapl, heap2) {
    case {
        heapl == NULL: return heap2;
        heap2 == NULL: return heapl;
        heapl.findMin() < heap2.findMin():
            temp = heapl.right;
            heapl.right = heapl.left;
            heapl.left = merge(heap2, temp);
            return heapl;
        otherwise:
            return merge(heap2, heapl);
    }
}
```

Runtime Analysis:
Worst-case and Amortized

- No worst case guarantee on right path length!
- All operations rely on merge
  ⇒ worst case complexity of all ops =
- Probably won’t get to amortized analysis in this course, but see Chapter 11 if curious.
- Result: M merges take time M log n
  ⇒ amortized complexity of all ops =

Comparing Heaps

- Binary Heaps
- Leftist Heaps
- d-Heaps
- Skew Heaps

Still room for improvement! (Where?)
Yet Another Data Structure: Binomial Queues

- Structural property
  - Forest of binomial trees with at most one tree of any height

- Order property
  - Each binomial tree has the heap-order property

What’s a forest? What’s a binomial tree?

The Binomial Tree, $B_h$

- $B_h$ has height $h$ and exactly $2^h$ nodes
- $B_h$ is formed by making $B_{h-1}$ a child of another $B_{h-1}$
- Root has exactly $h$ children
- Number of nodes at depth $d$ is binomial coeff.
  - Hence the name; we will not use this last property

Binomial Queue with $n$ elements

Binomial Q with $n$ elements has a unique structural representation in terms of binomial trees!

Write $n$ in binary: $n = 1101$ (base 2) = 13 (base 10)

Properties of Binomial Queue

- At most one binomial tree of any height
- $n$ nodes $\Rightarrow$ binary representation is of size?
  $\Rightarrow$ deepest tree has height?
  $\Rightarrow$ number of trees is?

Define: height(forest $F$) = max$_T$ in $F$ { height($T$) }

Binomial Q with $n$ nodes has height $\Theta(\log n)$

Operations on Binomial Queue

- Will again define merge as the base operation
  - insert, deleteMin, buildBinomialQ will use merge
- Can we do increaseKey efficiently?
  decreaseKey?
- What about findMin?
Merging Two Binomial Queues

Essentially like adding two binary numbers!

1. Combine the two forests
2. For $k$ from 0 to maxheight {
   a. $m \leftarrow$ total number of $B_k$'s in the two BQs
   b. if $m=0$: continue;
   c. if $m=1$: continue;
   d. if $m=2$: combine the two $B_k$'s to form $B_{k+1}$
   e. if $m=3$: retain one $B_k$ and combine the other two to form a $B_{k+1}$
}

Claim: When this process ends, the forest has at most one tree of any height.

Example: Binomial Queue Merge

<table>
<thead>
<tr>
<th>H1:</th>
<th>H2:</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Diagram" /></td>
<td><img src="image2.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Example: Binomial Queue Merge

<table>
<thead>
<tr>
<th>H1:</th>
<th>H2:</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image3.png" alt="Diagram" /></td>
<td><img src="image4.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Example: Binomial Queue Merge

<table>
<thead>
<tr>
<th>H1:</th>
<th>H2:</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image5.png" alt="Diagram" /></td>
<td><img src="image6.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Example: Binomial Queue Merge

<table>
<thead>
<tr>
<th>H1:</th>
<th>H2:</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image7.png" alt="Diagram" /></td>
<td><img src="image8.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>
**Example: Binomial Queue Merge**

H1:

![Diagram](image1)

H2:

![Diagram](image2)

**Complexity of Merge**

Constant time for each height

Max number of heights is: $\log n$

$\Rightarrow$ worst case running time = $\Theta($

**Insert in a Binomial Queue**

Insert(x): Similar to leftist or skew heap

- **runtime**
  - Worst case complexity: same as merge
  - $O(\ )$
  - Average case complexity: $O(1)$

Why?? Hint: Think of adding 1 to 1101

**deleteMin in Binomial Queue**

Similar to leftist and skew heaps….

**deleteMin: Example**

BQ

find and delete smallest root

BQ

![Diagram](image3)

merge BQ (without the shaded part) and BQ'

Result:

**deleteMin: Example**

BQ

![Diagram](image4)

runtime: