CSE 326: Data Structures
More Heaps

Hannah Tang and Brian Tjaden
Summer Quarter 2002

Outline

- Extra heap operations
- *d*-heaps
- Leftist heaps
- Skew heaps

Other Priority Queue Operations

- decreaseKey
  - given an object in the queue, reduce its priority value
- increaseKey
  - given an object in the queue, increase its priority value
- remove
  - remove a given object from the priority queue
- buildHeap
  - given a set of items, build a heap

DecreaseKey, IncreaseKey, and Remove

```
void decreaseKey(int obj, double decrease) {
  // Position of object ≤ size
  temp = Heap[obj] - decrease;
  newPos = percolateUp(obj, temp);
  Heap[newPos] = temp;
}
```

```
void remove(int obj) {
  // Position of object ≤ size
  percolateUp(obj,
          NEG_INF_VAL);
  deleteMin();
}
```

```
void increaseKey(int obj, double increase) {
 // Position of object ≤ size
  temp = Heap[obj] + increase;
  newPos = percolateDown(obj, temp);
  Heap[newPos] = temp;
}
```

BuildHeap naïvely

runtime:

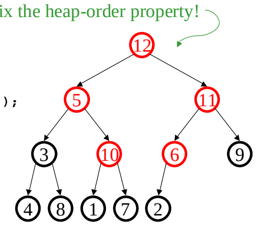
BuildHeap

Floyd's Method. Thank you, Floyd.

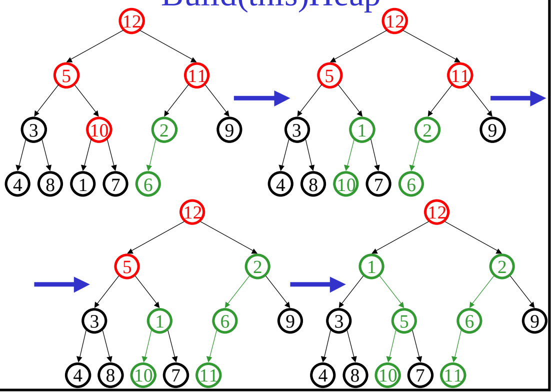
12	5	11	3	10	6	9	4	8	1	7	2

pretend it's a heap and fix the heap-order property!

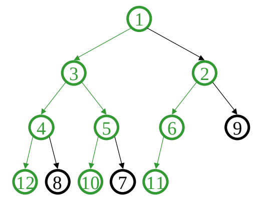
```
void buildHeap() {
  for(i=size/2; i>0; i--)
    percolateDown(i,Heap[i]);
}
```



Build(this)Heap



Finally…



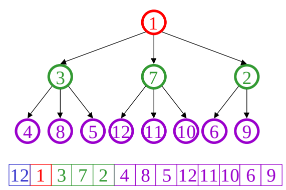
runtime:

Thinking about Heaps

- Observations
  - finding a child/parent index is a multiply/divide by two
  - operations jump widely through the heap
  - each operation looks at only two new nodes
  - inserts are at least as common as deleteMins
- Realities
  - division and multiplication by powers of two are **fast**
  - looking at one new piece of data sucks in a cache line
  - with **huge** data sets, disk accesses dominate

Solution: *d*-Heaps

- Each node has *d* children
- Still representable by array
- Good choices for *d*:
  - optimize performance based on # of inserts/removes
  - 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 block



One More Operation

- Merge two heaps. Ideas?

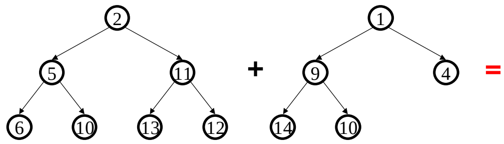
New Operation: Merge

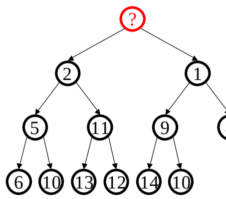
Given two heaps, merge them into one heap

- first attempt: insert each element of the smaller heap into the larger.
  runtime:
- second attempt: concatenate heaps' arrays and run buildHeap.
  runtime:

How about O(log n) time?

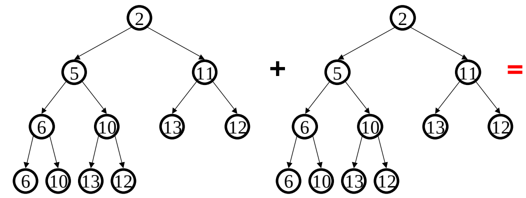
Idea: Hang a New Tree





Now, just percolate down!

Idea: Hang a New Tree



Problem?

Leftist Heaps

- Idea:

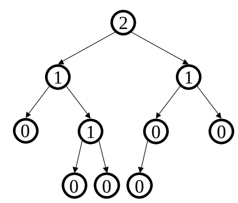
  make it so that all the work you have to do in maintaining a heap is in one small part

- Leftist heap:
  - almost all nodes are on the left
  - all the merging work is on the right

Random Definition: Null Path Length

the *null path length (npl)* of a node is the number of nodes between it and a null in the tree

- npl(null) = -1
- npl(leaf) = 0
- npl(single-child node) = 0



another way of looking at it:
npl is the height of the perfect subtree rooted at this node

Leftist Heap Properties

- Heap-order property
  - parent's priority value is ≤ to childrens' priority values
  - result: minimum element is at the root
- Leftist property
  - *null path length* of left subtree is ≥ *npl* of right subtree
  - result: tree is at least as "heavy" on the left as the right

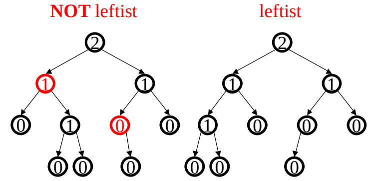
Are leftist trees complete?

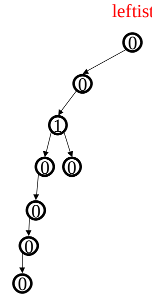
Leftist tree examples

NOT leftist

leftist

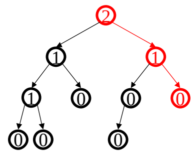
leftist





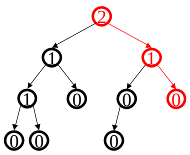
every subtree of a leftist tree is leftist, comrade!

Right Path in a Leftist Tree is Short



- Theorem: If the right path has length at least **r**, the tree has at least $2^r - 1$ nodes
- Proof by induction?
- So, a leftist tree with at least **n** nodes has a right path of at most **log n** nodes

Right Path in a Leftist Tree is Short



Proof by induction

Basis: $r = 1$.

Tree has at least one node: $2^1 - 1 = 1$

Inductive step:

Assume true for $r' < r$, and prove it's true for r.

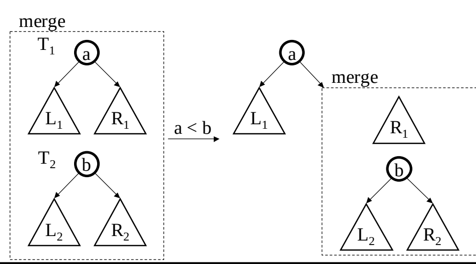
The right subtree has a right path of at least $r - 1$ nodes, so it has at least $2^{r-1} - 1$ nodes. The left subtree must also have a right path of at least $r - 1$ (otherwise, there is a null path of $r - 3$, less than the right subtree). So the left subtree has $2^{r-1} - 1$ nodes. All told then, there are at least:

$$2^{r-1} - 1 + 2^{r-1} - 1 + 1 = 2^r - 1$$

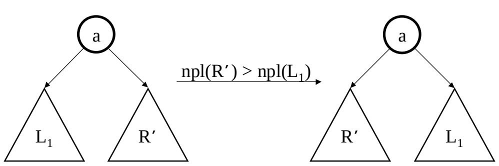
Whew!

Merging Two Leftist Heaps

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



Merge Continued

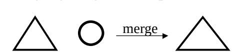


$R' = \text{Merge}(R_1, T_2)$

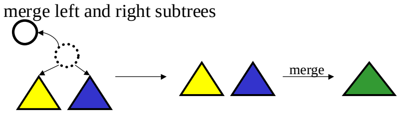
runtime:

Operations on Leftist Heaps

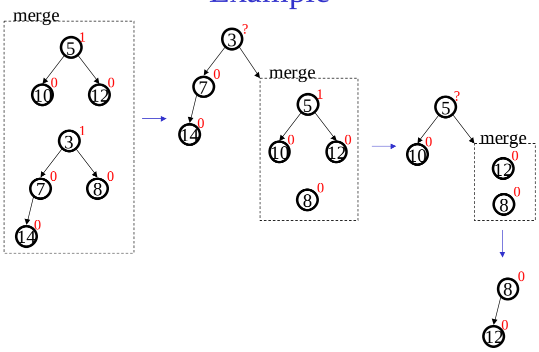
- **merge** with two trees of total size *n*: O(log *n*)
- **insert** with heap size *n*: O(log *n*)
  - pretend node is a size 1 leftist heap
  - insert by merging original heap with one node heap



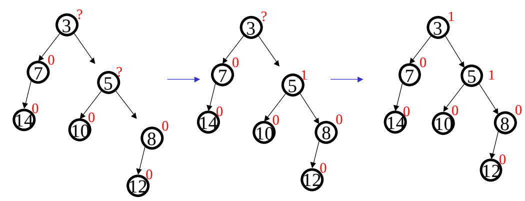
- **deleteMin** with heap size *n*: O(log *n*)
  - remove and return root
  - merge left and right subtrees



Example

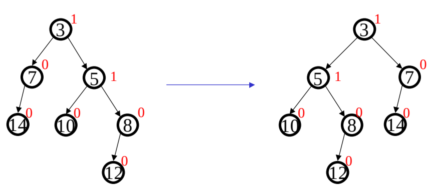


Sewing Up the Example



Done?

Finally…



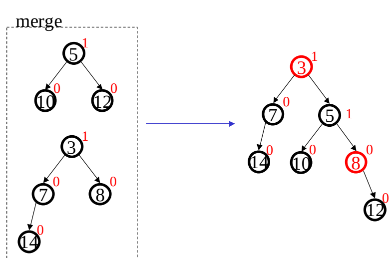
Recursive **merge** for leftist heaps

```
LeftistHeapNode merge(LeftistHeapNode h1, LeftistHeapNode h2) {
  if (h1 == null) return h2;
  if (h2 == null) return h1;
  if (h1.priority() < h2.priority()) return merge1(h1,h2);
  else return merge1(h2,h1);
}

LeftistHeapNode merge1(LeftistHeapNode h1, LeftistHeapNode h2) {
  if (h1.left == null) h1.left = h2;  // h1 has a single node
  else {
    h1.right = merge(h1.right, h2);
    if (h1.left.npl() < h1.right.npl()) swapChildren(h1);
    h1.npl = h1.right.npl() + 1;
  }
  return h1;
}
```

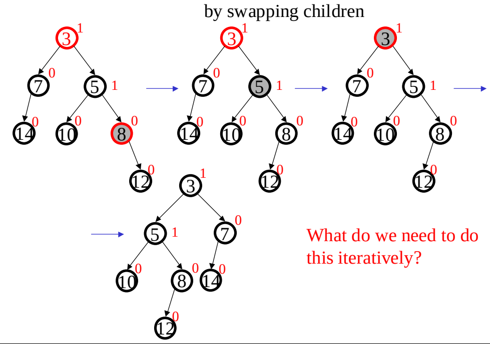
Iterative Leftist Merging

downward pass: merge right paths in sorted order



Iterative Leftist Merging

upward pass: fix right path (leftist heap property) by swapping children



What do we need to do this iteratively?

Random Definition: Amortized Time

am·or·tize
To write off an expenditure for (office equipment, for example) by prorating over a certain period.

time
A nonspatial continuum in which events occur in apparently irreversible succession from the past through the present to the future.

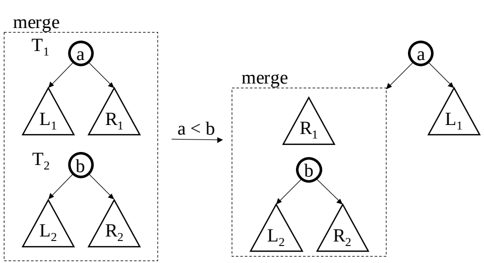
am·or·tized time
Running time limit resulting from writing off expensive runs of an algorithm over multiple cheap runs of the algorithm, usually resulting in a lower overall running time than indicated by the worst possible case.

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

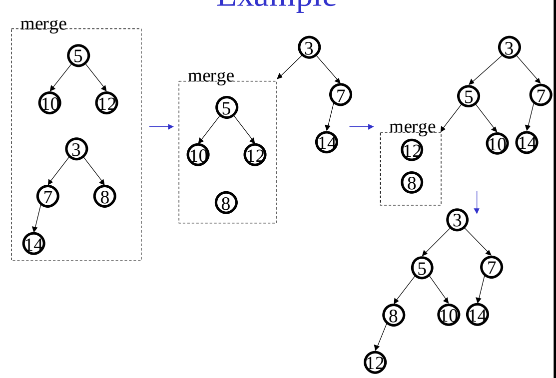
Skew Heaps

- Problems with leftist heaps
  - extra storage for *npl*
  - two pass merge (with stack!)
  - extra complexity/logic to maintain and check *npl*
- Solution: skew heaps
  - blind adjusting version of leftist heaps
  - amortized time for merge, insert, and deleteMin is O(log *n*)
  - worst case time for all three is O(*n*)
  - merge *always* switches children when fixing right path
  - iterative method has only one pass

Merging Two Skew Heaps



Example



Skew Heap Code

```
SkewHeapNode merge(heap1, heap2) {
  case {
     heap1 == NULL: return heap2;
     heap2 == NULL: return heap1;
     heap1.findMin() < heap2.findMin():
          temp = heap1.right;
          heap1.right = heap1.left;
          heap1.left = merge(heap2, temp);
          return heap1;
     otherwise:
          return merge(heap2, heap1);
  }
}
```

Comparing Heaps

- Binary Heaps
- Leftist Heaps
- *d*-Heaps
- Skew Heaps
- Binomial Queues