CMSC 341

Binary Heaps

Priority Queues

Priority Queues

Priority: some property of an object that allows it to be prioritized with respect to other objects of the same type

Min Priority Queue: homogeneous collection of Comparables with the following operations (duplicates are allowed). Smaller value means higher priority.

- `void insert (const Comparable &x)`
- `void deleteMin( )`
- `void deleteMin ( Comparable & min)`
- `const Comparable &findMin( ) const`
- `Construct from a set of initial values`
- `bool isEmpty( ) const`
- `bool isFull( ) const`
- `void makeEmpty( )`

Priority Queue Applications

Printer management:

  the shorter document on the printer queue, the higher its priority.

Jobs queue within an operating system:

  users' tasks are given priorities. System priority high.

Simulations

  the time an event "happens" is its priority

Sorting (heap sort)

  an elements "value" is its priority

Possible Implementations

Use a sorted list. Sorted by priority upon insertion.

- findMin( )　　--> list.front( )
- insert( )　　--> list.insert( )
- deleteMin( )　　--> list.erase(list.begin( ))

Use ordinary BST

- findMin( )　　--> tree.findMin( )
- insert( )　　--> tree.insert( )
- deleteMin( )　　--> tree.delete(tree.findMin( ))

Use balanced BST

- guaranteed O(lg n) for Red-Black

Min Binary Heap

A min binary heap is a complete binary tree with the further property that at every node neither child is smaller than the value in that node (or equivalently, both children are at least as large as that node).

This property is called a ***partial ordering.***

As a result of this partial ordering, every path from the root to a leaf visits nodes in a non-decreasing order.

What other properties of the Min Binary Heap result from this property?

Min Binary Heap Performance

Performance (n is the number of elements in the heap)

– construction	O(n)
– findMin	O(1)
– insert	O(lg n)
– deleteMin	O(lg n)

Heap efficiency results, in part, from the implementation

- conceptually a complete binary tree
- implementation in an array/vector (in level order) with the root at index 1

Min Binary Heap Properties

For a node at index i

- its left child is at index 2i
- its right child is at index 2i+1
- its parent is at index $\lfloor i/2 \rfloor$

No pointer storage

Fast computation of 2i and $\lfloor i/2 \rfloor$ by bit shifting

$i << 1 = 2i$

$i >> 1 = \lfloor i/2 \rfloor$

Min BinaryHeap Definition

```
template <typename Comparable>
class MinBinaryHeap {
public:
   explicit MinBinaryHeap(int capacity = BIG);
   explicit MinBinaryHeap(vector<Comparable>& items);
   bool isEmpty() const;
   const Comparable& findMin() const;
   void insert (const Comparable& x);
   void deleteMin();
   void deleteMin(Comparable& min_item);
   void makeEmpty();
private:
   int currentSize;
   vector< Comparable > array;
   void buildHeap();
   void percolateDown(int hole);
};
```

MinBinaryHeap Implementation

```
template <typename Comparable>
const Comparable& MinBinaryHeap::findMin( ) const
{
  if ( isEmpty( ) ) throw Underflow( );
  return array[1];
}
```

Insert Operation

Must maintain

- CBT property (heap shape):
  - easy, just insert new element at "the end" of the array
- Min heap order
  - could be wrong after insertion if new element is smaller than its ancestors
  - continuously swap the new element with its parent until parent is not greater than it
    - called *sift up* or *percolate up*

Performance of insert is O(lg n) in the worst case because the height of a CBT is O(lg n)

MinBinaryHeap Insert (cont'd)

```
template <typename Comparable>
void MinBinaryHeap<Comparable>::
insert(const Comparable & x)
{
  if (currentSize == array.size( ) -1)
     array.resize( array.size( ) * 2)

  int hole = ++currentSize;

  // percolate up
  for (; hole > 1 && x < array[hole/2]; hole /= 2)
     array[hole] = array[hole/2];

  // put x in hole
  array[hole] = x;
}
```

Deletion Operation

Steps

- remove min element (the root)
- maintain heap shape
- maintain min heap order

To maintain heap shape, actual node removed is “last one” in the array

- replace root value with value from last node and delete last node
- sift-down the new root value
  - continually exchange value with the smaller child until no child is smaller

MinBinaryHeap Deletion(cont)

```
template <typename Comparable>
void MinBinaryHeap<Comparable>::
deleteMin(Comparable& minItem)
{
  if ( isEmpty( ) ) throw Underflow( );

  minItem = array[1];
  array[1] = array[currentSize--];
  percolateDown(1);
}
```

MinBinaryHeap PercolateDown (cont'd)

```
template <typename Comparable>
void MinBinaryHeap<Comparable>::percolateDown(int hole)
{
  int child;
  Comparable tmp = array[hole];
  for (; hole * 2 <= currentSize; hole = child)
  {
     child= hole * 2;
     if (child != currentSize
     && array[child + 1] < array[ child ] )
          child++;
     if (array [child] < tmp)
          array[ hole ] = array[ child ];
     else break;
  }
  array[hole] = tmp;
}
```

Constructing a Min Binary Heap

A BH can be constructed in O(n) time.

Suppose we are given an array of objects in an arbitrary order. Since it’s an array with no holes, it’s already a CBT. It can be put into heap order in O(n) time.

- Create the array and store n elements in it in arbitrary order. O(n) to copy all the objects.
- Heapify the array starting in the “middle” and working your way up towards the root

```
for (int index = ⌊n/2⌋ ; index > 0; index--)
    percolateDown( index );
```

Constructing a Min BinaryHeap (cont’d)

```
template <typename Comparable>
MinBinaryHeap( const vector<Comparable>& items )
:array(items.size() + 1), currentSize((items.size( ))
{
  for (int i = 0; i < items.size( ); i++)
      array[ i + 1 ] = items[ i ] ;
  buildHeap( );
}

template <typename Comparable>
void MinBinaryHeap<Comparable>:: buildHeap( )
{
  for(int i = currentSize/2; i > 0; i--)
      percolateDown( i );
}
```

Performance of Construction

A CBT has 2^{h-1} nodes on level h-1.

On level h-l, at most 1 swap is needed per node.

On level h-2, at most 2 swaps are needed.

…

On level 0, at most h swaps are needed.

Number of swaps = S

$= 2^h*0 + 2^{h-1}*1 + 2^{h-2}*2 + \ldots + 2^0*h$

$$= \sum_{i=0}^{h} 2^i (h-i) = h\sum_{i=0}^{h} 2^i - \sum_{i=0}^{h} i2^i$$

$= h(2^{h+1}-1) - ((h-1)2^{h+1}+2)$

$= 2^{h+1}(h-(h-1))-h-2$

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

Performance of Construction (cont)

But 2^{h+1}-h-2 $= O(2^h)$

But $n = 1 + 2 + 4 + \ldots + 2^h = \sum_{i=0}^{h} 2^i$

Therefore, $n = O(2^h)$

So $S = O(n)$

A heap of n nodes can be built in O(n) time.

Heap Sort

Given n values we can sort them in place in O(n log n) time

- Insert values into array -- O(n)
- heapify -- O(n)
- repeatedly delete min -- O(lg n), n times

Using a min heap, this code sorts in reverse (high down to low) order.

With a max heap, it sorts in normal (low up to high) order.

Given an unsorted array A[ ] of size n

```
for (i = n-1; i >= 1; i--)
{
        x = findMin( );
        deleteMin( );
        A[i+1] = x;
}
```

Limitations

MinBinary heaps support `insert`, `findMin`, `deleteMin`, and `construct` efficiently.

They do not efficiently support the `meld` or `merge` operation in which 2 BHs are merged into one. If H_1 and H_2 are of size n_1 and n_2, then the merge is in $O(n_1 + n_2)$.

Leftist Min Heap

Supports

– findMin	-- O(1)
– deleteMin	-- O(lg n)
– insert	-- O(lg n)
– construct	-- O(n)
– merge	-- O(lg n)

Leftist Tree

The ***null path length, npl(X),*** of a node, X, is defined as the length of the shortest path from X to a node without two children (a ***non-full*** node).

Note that npl(NULL) = -1.

A Leftist Tree is a binary tree in which at each node X, the null path length of X’s right child is not larger than the null path length of the X’s left child .
I.E. the length of the path from X’s right child to its <u>nearest non-full node</u> is not larger than the length of the path from X’s left child to its <u>nearest non-full node</u>.

An important property of leftist trees:

– At every node, the shortest path to a non-full node is along the rightmost path.

  “Proof”: Suppose this was not true. Then, at some node the path on the left would be shorter than the path on the right, violating the leftist tree definition.

Leftist Min Heap

A ***leftist min heap*** is a leftist tree in which the values in the nodes obey heap order (the tree is partially ordered).

Since a LMH is not necessarily a CBT we do not implement it in an array. An explicit tree implementation is used.

Operations

– findMin	-- return root value, same as MBH
– deleteMin	-- implemented using meld operation
– insert	-- implemented using meld operation
– construct	-- implemented using meld operation

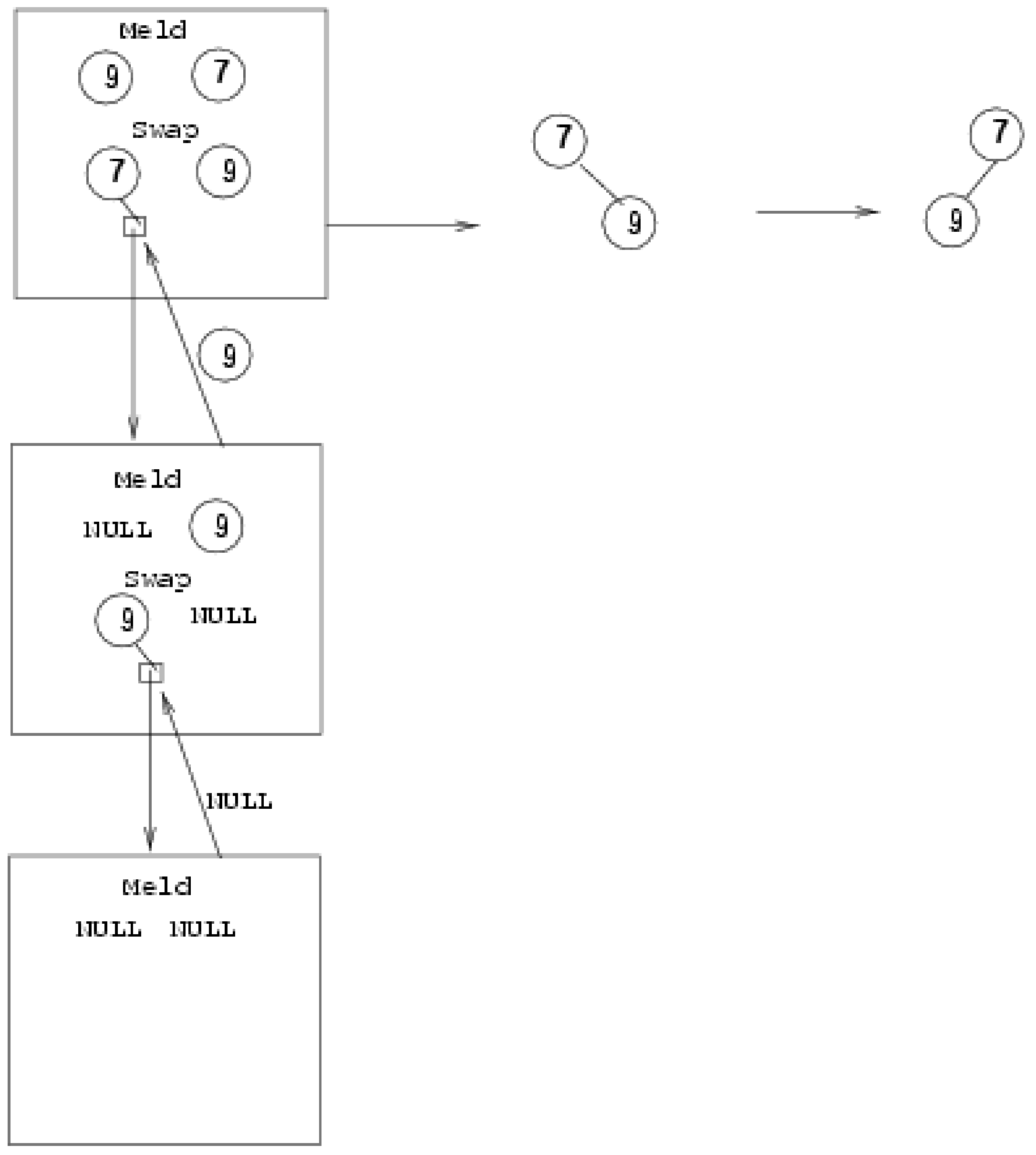
Meld

```
Meld (H1, H2)   // pseudo-code
{
    // rename the LHs so that H1 has the smaller root (or is the only LH)
    if (!root( H1 ) || (root_value( H1 ) > root_value( H2 ) )
            swap (H1, H2)

    // if H1 exists, meld H1's right subtree with H2
    // and replace H1's right subtree with the result of the meld
    if ( root( H1 ) != NULL )
        right( H1 ) = Meld( right( H1 ), H2 )

    // if the null-path-length of H1's left subtree is shorter
    // than the null-path length of H1's right subtree, swap subtrees
    if (left_length( H1 ) <  right_length( H1 )
            swap( left( H1 ),  right( H1 );
}
```

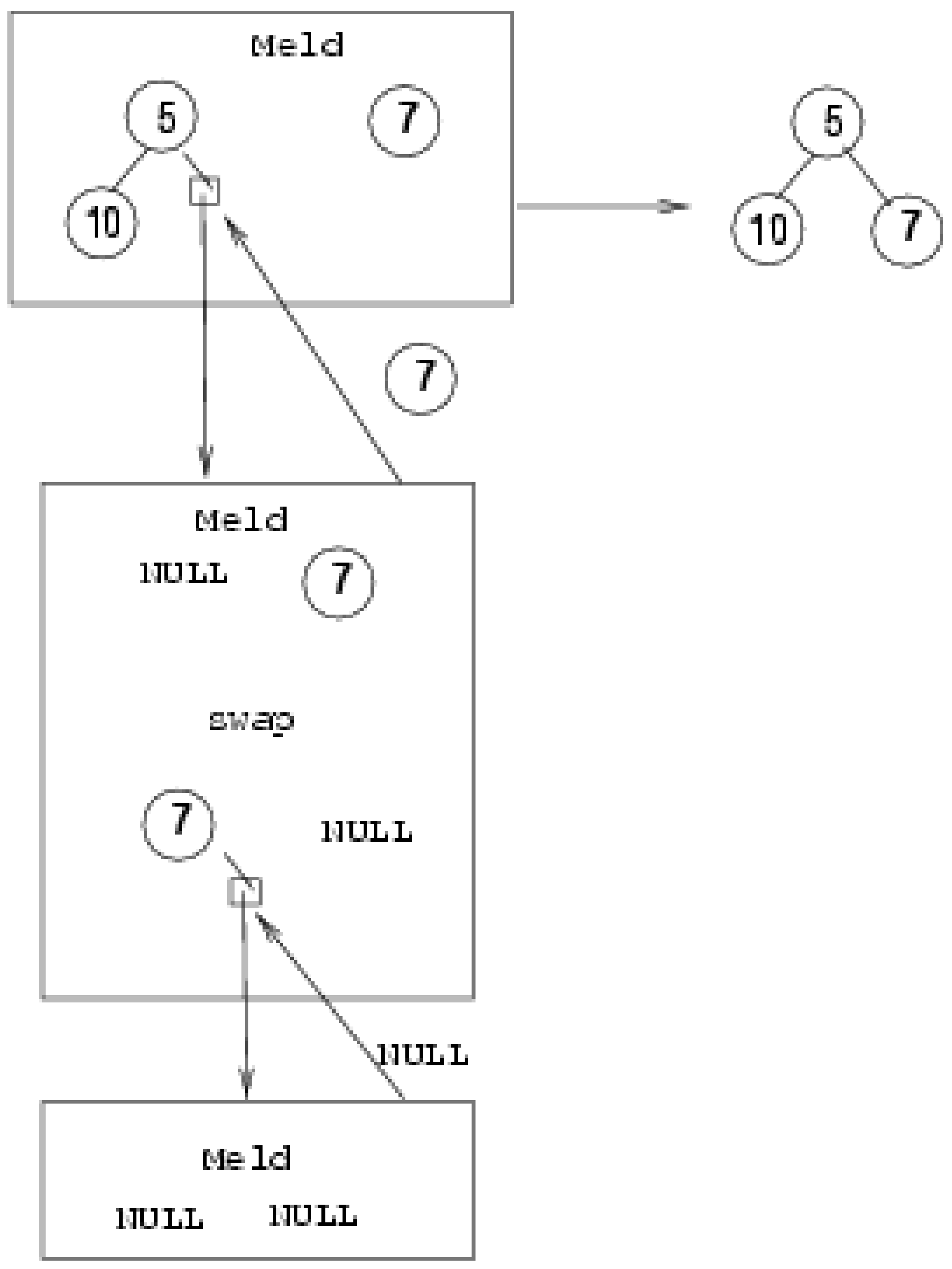
Meld (cont)

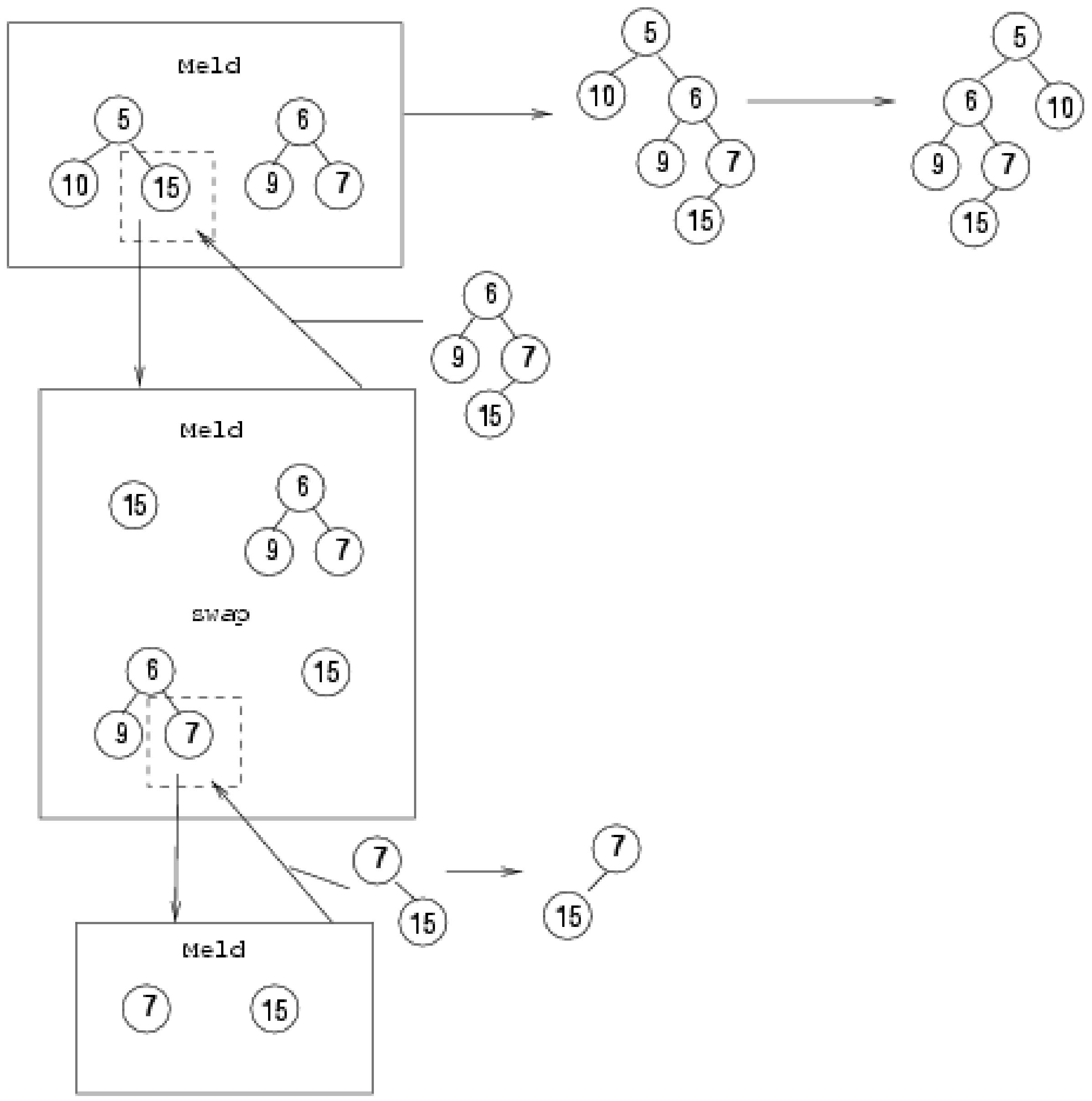
Performance: $O(\lg n)$

- the rightmost path of each tree has at most $\lfloor \lg(n+1) \rfloor$ nodes. So $O(\lg n)$ nodes will be involved.



Meld
9
7
Swap
7
9
9
Meld
NULL
9
Swap
9
NULL
NULL
Meld
NULL NULL
7
9
7
9


Meld
5
10
7
5
10
7
7
Meld
NULL
7
swap
7
NULL
NULL
Meld
NULL
NULL


Meld
Meld
swap
Meld

Min Leftist Heap Operations

Other operations implemented using Meld( )

- insert (item)
  - make item into a 1-node LH, X
  - Meld(*this, X)
- deleteMin
  - Meld(left subtree, right subtree)
- construct from N items
  - make N LHs from the N values, one element in each
  - meld each in
    - one at a time (simple, but slow)
    - use queue and build pairwise (complex but faster)

LH Construct

Algorithm:

```
make N leftist heaps, $H_1 \ldots H_N$ each with one data value
Instantiate Queue<LeftistHeap> Q;
for (i = 1; i <= N; i++)
    Q.Enqueue($H_i$);
Leftist Heap H = Q.Dequeue( );
while ( !Q.IsEmpty( ) )
    Q.Enqueue( meld( H, Q.Dequeue( ) ) );
    H = Q.Dequeue( );
```