Data Structures

(Feodor F. Dragan)
Department of Mathematics and Computer Science
Kent State University

Fall, 2000

CHAPTER 21

A Priority Queue: The Binary Heap

Contents

We will learn:

- The basic properties of the binary heap;
- How the ***insert*** and ***deleteMin*** operations can be performed in logarithmic time;
- A linear time heap construction algorithm;
- C++ implementations;
- ***HeapSort*** algorithm with *O(N logN)* worst case run time, w/o using extra memory; and
- The use of heaps to implement external sorting.

Basic Ideas

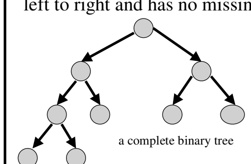
- Operations: *findMin*, *deleteMin*, *insert* an element.
- A linked list (or array) requires that some operations use linear time
  - linked list unsorted: *insert* in front in O(1), but *findMin* and *deleteMin* in O(N);
  - linked list sorted: *findMin* and *deleteMin* in O(1), but *insert* in O(N).
- An unbalanced binary search tree does not have a good worst case (O(N) worst, O(logN) average)
- A balanced binary search tree (O(logN) worst) requires lots of work
- The priority queue has properties that are a compromise between a queue and a binary search tree
  - Can be implemented by using a simple array (like a queue)
  - Supports *insert* and *deleteMin* in O(logN) worst case time (like the balanced binary search tree)
  - Supports *insert* in constant average time and *findMin* in constant worst case time (like the queue).

Binary Heap

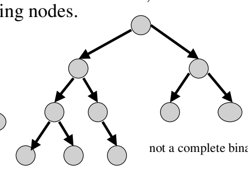
- The **binary heap** is the classic method used to implement **priority queues**. It has a structure property and an ordering property (like balanced BST).

Structure Property

- The **heap** is a complete binary search tree, allowing representation by a simple array and guaranteeing logarithmic depth.
- A **complete binary tree** is a binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right and has no missing nodes.



a complete binary tree

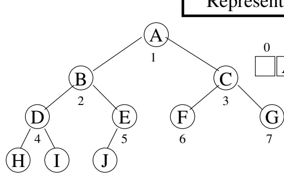


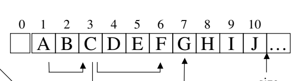
not a complete binary tree

Structure Property

- Height $H \le \lfloor \log N \rfloor$
  - $2^H \le$ number of nodes, N $\le 2^{H+1} - 1$
  - Way? (by induction)

Representation

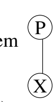




The parent is in position $\lfloor i/2 \rfloor$
The left child is in position *2i*
The right child is in position *2i+1*
➔ Left and right pointers are not needed,
➔ Vector gives a random access

Using an array to store a tree is called *implicit representation.*

Heap-Order Property



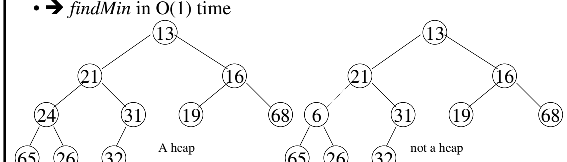
$P \le X$

- The heap-order property states that, in a heap, the item in the parent is never larger than the item in a node.

$heap[i] \le heap[2i]$ and $heap[i] \le heap[2i+1]$

(In binary search trees we had $heap[2i] \le heap[i] \le heap[2i+1]$)

- ➔ by this property, min element is in the root
- ➔ *findMin* in O(1) time



- A **max heap** supports access of the maximum instead of the minimum. Minor changes can be used to implement max heap.

C++ Implementation

The *BinaryHeap* class interface.

```
#include "vector.h"
// BinaryHeap class.
// CONSTRUCTION: with no parameters or vector containing items.
//
// ****************** PUBLIC PERATIONS **************
// void insert( x )      --> Insert x
// void deleteMin( )     --> Remove smallest item
// void deleteMin( min ) --> Remove and send back smallest item
// Comparable findMin( ) --> Return smallest item
// bool isEmpty( )       --> Return true if empty; else false
// void makeEmpty( )     --> Remove all items
// ****************** ERRORS *************************
// Throws UnderflowException as warranted.
template <class Comparable>
class BinaryHeap
{ public:
    BinaryHeap( );
    BinaryHeap( const vector<Comparable> & v );
    bool isEmpty( ) const;
    const Comparable & findMin( ) const;
    void insert( const Comparable & x );
    void deleteMin( );
    void deleteMin( Comparable & minItem );
    void makeEmpty( );
  private:
    int   theSize;  // Number of elements in heap
    vector<Comparable> array;    // The heap array

    void buildHeap( );
    void percolateDown( int hole );
  };
#include "BinaryHeap.cpp"
```


Constructors for the *BinaryHeap* class.

```
// Construct the binary heap.
template <class Comparable>
BinaryHeap<Comparable>::BinaryHeap( ) : array( 11 ), theSize( 0 ) { }

// Construct the binary heap.
// v is a vector containing the initial items.
template <class Comparable>
BinaryHeap<Comparable>::BinaryHeap( const vector<Comparable> & v )
  : array( v.size( ) + 1 ), theSize( v.size( ) )
{ for( int i = 0; i < v.size( ); i++ )
      array[ i + 1 ] = v[ i ];
  buildHeap( ); }
```

The *findMin* routine.

```
// Find the smallest item in the priority queue.
// Return the smallest item, or throw UnderflowException if empty.
template <class Comparable>
const Comparable & BinaryHeap<Comparable>::findMin( ) const
{ if( isEmpty( ) )
      throw UnderflowException( );
  return array[ 1 ];  }
```

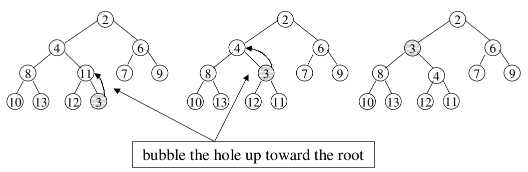
isEmpty and makeEmpty.

```
// Test if the priority queue is logically empty.
template <class Comparable>
bool BinaryHeap<Comparable>::isEmpty( ) const
{ return theSize == 0; }

// Make the priority queue logically empty.
template <class Comparable>
void BinaryHeap<Comparable>::makeEmpty( )
{ theSize = 0; }
```


The *insert* operation.

• Insertion is implementing by creating a hole at the next available location and then percolating it up until the new item can be placed in it without introducing a heap-order violation with the hole's parent.



$$insert(x): O(\text{height}) = O(\log n)$$

• Insertion takes constant time on average but logarithmic in the worst case.

The *insert* member function.

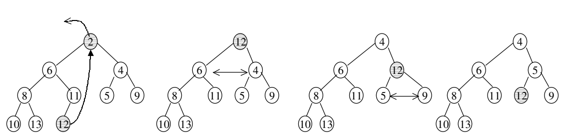
```
// Insert item x into the priority queue, maintaining heap order.
// Duplicates are allowed.
template <class Comparable>
void BinaryHeap<Comparable>::insert( const Comparable & x )
{
    array[ 0 ] = x;   // initialize sentinel
    if( theSize + 1 == array.size( ) )
        array.resize( array.size( ) * 2 + 1 );

      // Percolate up
    int hole = ++theSize;
    for( ; x < array[ hole / 2 ]; hole /= 2 )
        array[ hole ] = array[ hole / 2 ];
    array[ hole ] = x;
}
```


The *deleteMin* operation.

• Deletion of the minimum involves placing the former last item in a hole that is created at the root.

• The hole is percolated down the tree through minimum children until the item can be placed without violating the heap-order property.



• The *deleteMin* operation is logarithmic in both the worst case and average case.

The *deleteMin* member function.

```
// Remove the smallest item from the priority queue.  Throw UnderflowException if empty.
template <class Comparable>
void BinaryHeap<Comparable>::deleteMin( )
{  if( isEmpty( ) ) throw UnderflowException( );
   array[ 1 ] = array[ theSize-- ];
   percolateDown( 1 ); }

// Remove the smallest item from the priority queue and place it in minItem.
template <class Comparable>
void BinaryHeap<Comparable>::deleteMin( Comparable & minItem )
{  minItem = findMin( );
   array[ 1 ] = array[ theSize-- ];
   percolateDown( 1 ); }
```

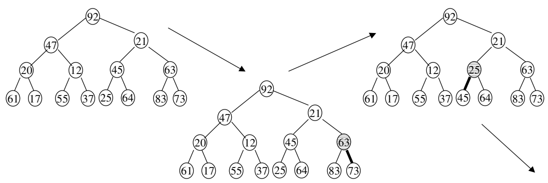
The *percolateDown* method.

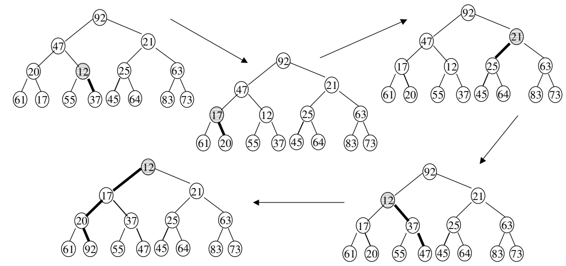
```
// Internal method to percolate down in the heap. Hole is the index at which the percolate begins.
template <class Comparable>
void BinaryHeap<Comparable>::percolateDown( int hole )
{  int child;
   Comparable tmp = array[ hole ];
   for( ; hole * 2 <= theSize; hole = child )
   {  child = hole * 2;
      if( child != theSize && array[ child + 1 ] < array[ child ] )  child++;
      if( array[ child ] < tmp ) array[ hole ] = array[ child ];
      else break;  }
   array[ hole ] = tmp;
}
```

The *buildHeap* linear time operation.

- The ***buildHeap operation*** takes a complete tree that does not have heap order and reinstates it.
- It can be done in linear time by applying a percolate down routine to nodes in reverse level order.

```
// Establish heap-order property from an arbitrary arrangement of items. Runs in linear time.
template <class Comparable>
void BinaryHeap<Comparable>::buildHeap( )
{  for( int i = theSize / 2; i > 0; i-- )
      percolateDown( i ); }
```





- The linear time bound can be shown by computing the sum of the heights of all the nodes in the heap.
- The bound follows from the following Theorem.

Theorem: For the perfect binary tree of height H containing $N = 2^{H+1} - 1$ nodes, the sum of the heights of the nodes is *N-H-1*.

Proof: (by induction) …

C++ Implementation of the *priority_queue*

The *priority_queue* class interface.

```
template <class Object, class Container, class Compare>
class priority_queue
{public:
   priority_queue( );

   int size( ) const;
   bool empty( ) const;

   const Object & top( ) const;

   void push( const Object & x );
   void pop( );

  private:
   Container theItems;
   Compare   lessThan;
};

#include "queue.cpp"
```

Constant-time methods.

```
template <class Object, class Container, class Compare>
priority_queue<Object,Container,Compare>::priority_queue( )
  : theItems( 1 ), lessThan( Compare( ) ) { }

template <class Object, class Container, class Compare>
int priority_queue<Object,Container,Compare>::size( ) const
{   return theItems.size( ) - 1; }

template <class Object, class Container, class Compare>
bool priority_queue<Object,Container,Compare>::empty( ) const
{ return size( ) == 0; }

template <class Object, class Container, class Compare>
const Object & priority_queue<Object,Container,Compare>
::top( ) const
{  if( empty( ) ) throw UnderflowException( "Cannot top an
                                              empty priority queue" );
   return theItems[ 1 ]; }
```


Push and *pop* class mutators.

```
//template <class Object, class Container, class Compare>
void priority_queue<Object,Container,Compare>::push( const Object & x )
{  theItems.push_back( x );
   theItems[ 0 ] = x;   // initialize sentinel

     // Percolate up
   int hole = size( );
   for( ; lessThan( theItems[ hole / 2 ], x ); hole /= 2 )
      theItems[ hole ] = theItems[ hole / 2 ];
   theItems[ hole ] = x;
}

template <class Object, class Container, class Compare>
void priority_queue<Object,Container,Compare>::pop( )
{  if( empty( ) ) throw UnderflowException( "Cannot pop an empty priority queue" );
   int hole = 1;
   int child;
   Object tmp = theItems.back( ); theItems.pop_back( );
   int theSize = size( );
   for( ; hole * 2 <= theSize; hole = child )
   {  child = hole * 2;
      if( child != theSize && lessThan( theItems[ child ], theItems[ child + 1 ] ) ) child++;
      if( lessThan( tmp, theItems[ child ] ) ) theItems[ hole ] = theItems[ child ];
      else break;   }
   if( !empty( ) ) theItems[ hole ] = tmp;
}
```


A Test Program

```
#include "functional.h"
#include "vector.h"
#include "pr_queue.h"
#include <iostream>

// Test program
int main( )
{  int numItems = 10000;
   priority_queue<int, vector<int>, greater<int> > h;  // min heap
   priority_queue<int, vector<int>, less<int> > g;      // max heap
   int i = 37;
   int x, y;
   for( i = 37; i != 0; i = ( i + 37 ) % numItems )
      {  h.push( i );
         g.push( -i );
      }
      for( i = 1; i < numItems; i++ )
      {   x = h.top( );
          y = g.top( );
          h.pop( ); g.pop( );
          if( x != i || x + y != 0 )
             cout << "Oops! " << i << " " << x << " " << y << endl;
      }
return 0;
}
```

Extension:

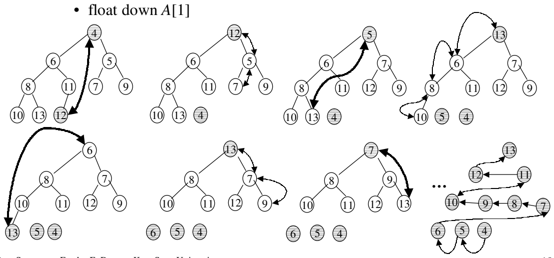
- the *decreaseKey* operation is supported by the binary heap (priority_queue).
- It is useful in implementing graph algorithms (e.g., Dijkstra's algorithm for shortest paths)

HeapSort

- A priority queue can be used to sort in O(N logN) time. An algorithm based on this idea is *HeapSort.*
- One way is by
  - Inserting every item into a binary heap and
  - Extracting every item by calling *deleteMin* N times, thus sorting the result.
- A better way is by
  - Tossing each item into a binary heap,
  - Applying *buildHeap*, and
  - Calling *deleteMin* N times, with the item exiting the heap in sorted order.
- If we use a max heap we obtain items in increasing order.

HeapSort

- HeapSort
  - build heap (or for (i=1..n) do <u>insert</u> (A[1..i],A[i])
  - for (i=n..2) do
    - swap (A[1] $\leftrightarrow$ A[i])
    - heapsize = heapsize-1
    - float down A[1]



The HeapSort Routine

- By using empty parts of the array, we can perform the sort in place (better than *MergeSort*; what about *QuickSort ?*)

```
// Standard Heapsort
void heapsort( vector<Comparable> & a )
{
   for ( int i  = a.size( ) / 2;  i >=  0;  i-- )     // build heap
        percDown ( a, i, a.length( )  );

   for ( int j  = a.size( ) - 1;  j >  0;  j-- )
        {
            swap ( a[ 0 ], a[ j ] );                    // deleteMax
            percDown ( a, 0, j  );
        }
}
```


<u>External Sorting</u>

- External sorting is used when the amount of data is too large to fit in main memory.

Model for external sorting

- We assume that sorts are performed on tape. Only sequential access of the input is allowed.
- **The basic external sort uses repeated two-way merging**.
- Assume we have
  - two input tapes *A1, A2* and two output tapes *B1, B2*
  - *A1* has initial input of ***N*** records , memory can hold ***M*** records
- Then,
  - ***M*** records are read from *A1,* sorted and written to *B1* and *B2* alternatively
  - each group of sorted records is a ***run***
  - when done, we rewind all the tapes
- (***a pass***) A run from *B1* is merged with a run from *B2* and stored on *A1* and *A2* alternatively (few times). No more runs ➔ rewind all tapes.
- Do a pass using the A tapes → the B tapes way …

A1 | 81 94 11 96 12 35 17 99 28 58 41 75 15
A2 |
B1 |
B2 |

Initial tape configuration

A1 |
A2 |
B1 | 11 81 94 | 17 28 99 | 15 |
B2 | 12 35 96 | 41 58 75 | |

Distribution of runs to two tapes

As a result of a pass, the length of the runs doubles and eventually only a single run remains. We need $\lceil \log(N/M) \rceil$ passes over the input before we have one final giant run.

A1 | 11 12 35 81 94 96 | 15 |
A2 | 17 28 41 58 75 99 | |
B1 |
B2 |

After the first pass

A1 |
A2 |
B1 | 11 12 17 28 35 41 58 75 81 94 96 99 |
B2 | 15 |

After the second pass

A1 | 11 12 15 17 28 35 41 58 75 81 94 96 99

After the third pass

***K*-way merging** reduces the number of passes. The obvious implementation uses 2K tapes. ($\lceil \log_K(N/M) \rceil$ passes)

A1 | 81 94 11 96 12 35 17 99 28 58 41 75 15

Initial tape configuration (A2, A3, B1, B2, B3 are empty and not presented)

A1 |
A2 |
A3 |
B1 | 11 81 94 | 41 58 75 |
B2 | 12 35 96 | 15 |
B3 | 17 28 99 | |

Distribution of runs to three tapes

To find the smallest of the ***K*** elements, we use a priority queue (*deleteMin* and *insert* operations).

A1 | 11 12 17 28 35 81 94 96 99 |
A2 | 15 41 58 75 |
A3 |
B1 |
B2 |
B3 |

After the first pass

B1 | 11 12 15 17 28 35 41 58 75 81 94 96 99 |

After the second pass (A1, A2, A3, B2, B3 are not presented)

Polyphase Merge

- The polyphase merge implements a ***K***-way merge with ***K+1*** tapes only.
- The distribution of runs affects performance.
- The best distribution is related to the **Fibonacci numbers**.
- Consider the case ***K=2*** (for larger ***K***, ***K***th order Fibonacci numbers are used)
  - If the number of runs is a Fibonacci number *f(n)* , the best way to distribute them is to split them into two Fibonacci numbers *f(n-1)* and *f(n-2).*
  - Otherwise, the tape must be padded with dummy runs in order to increase the number of runs to a Fibonacci number.

$$f(n) = \begin{cases} n & \text{if } n = 0 \text{ or } n = 1 \\ f(n-1) + f(n-2) & \text{if } n \geq 2 \end{cases}$$

		After						
	# of runs	T3+T2	T1+T2	T1+T3	T2+T3	T1+T2	T1+T3	T2+T3
T1	0	13	5	0	3	1	0	1
T2	21	8	0	5	2	0	1	0
T3	13	0	8	3	0	2	1	0