Heaps

17.1 Priority Queue

The priority queue ADT supports:

- insertItem(e): Insert new item e
- removeMin(): Remove and return item with minimum key (Error if priority queue is empty)
- standard isEmpty() and size, maybe peeks

Other possible methods include decrease-key, increase-key, and delete. Applications include selection, and the event queue in discrete event simulation.

There are several inefficient implementations:

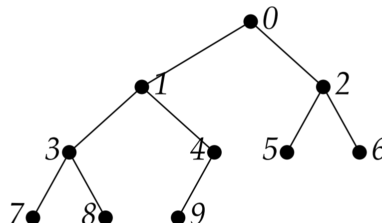
	insert	removeMin
unsorted linked list	$O(1)$	$O(n)$
sorted linked list or array	$O(n)$	$O(1)$
binary search tree	$O(n)$; average $O(\log n)$	

17.2 Heap

In ***level numbering*** in binary trees, the nodes are numbered such that:

for a node numbered x, its children are 2x+1 and 2x+2

Thus a node's parent is at (x-1)/2 round down, and the root is 0.

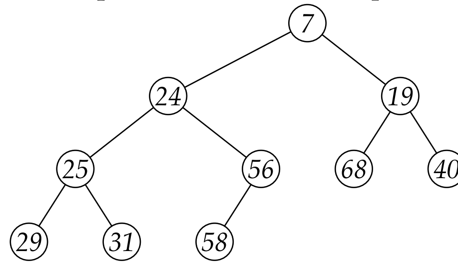


One can store a binary tree in an array/vector by storing each value at the position given by level numbering. But this is wasteful storage, unless nearly balanced.

We can change the definition of ***complete binary tree*** as a binary tree where each level except the last is complete, and in the last level nodes are added left to right. With this definition, a ***heap*** is a complete binary tree, normally stored as a vector, with values stored at nodes such that:

> ***heap-order*** *property: for each node, its value is smaller than or equal to its children's*

So the minimum is on top. A ***heap*** is the standard implementation of a priority queue.



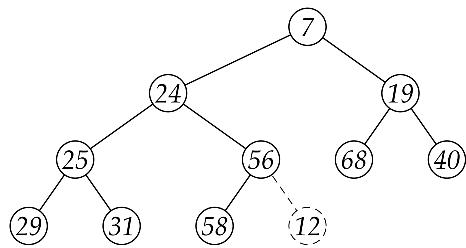
17.3 Heap Operations

The idea for ***insertion*** is to *Add as last leaf, then bubble up value until heap-order property re-established.*

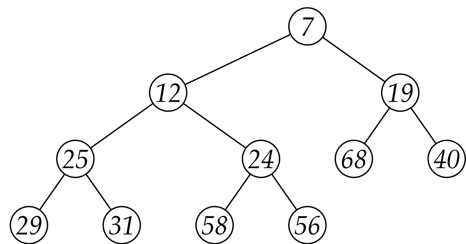
```
Algorithm: Insert(v)
    add v as next leaf
    while v<parent(v) {
        swapElements(v,parent(v))
        v=parent(v)
    }
```

Use "hole" to reduce data movement.

Here is an example of Insertion: inserting value 12:



Before

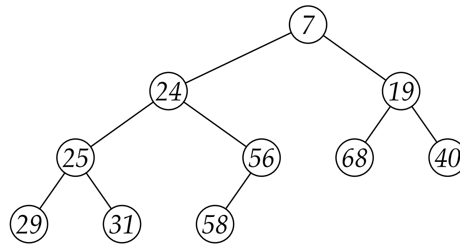


After

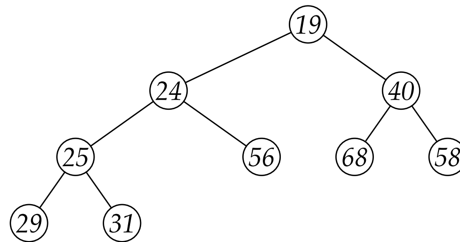
The idea for ***removeMin*** is to *Replace with value from last leaf, delete last leaf, and bubble down value until heap-order property re-established.*

```
Algorithm: RemoveMin()
    temp = value of root
    swap root value with last leaf
    delete last leaf
    v = root
    while v > any child(v) {
        swapElements(v, smaller child(v))
        v= smaller child(v)
    }
    return temp
```

Here is an example of RemoveMin:



Before



After

Variations of heaps include

- d-heaps; each node has d children
- support of merge operation: leftist heaps, skew heaps, binomial queues

17.4 Heap Sort

Any priority queue can be used to sort:

```
Insert all values into priority queue
Repeatedly removeMin()
```

It is clear that inserting n values into a heap takes at most $O(n \log n)$ time. Possibly surprising is that we can create a heap in linear time. Here is one approach: work up the tree level by level, correcting as you go. That is, at each level, you push the value down until it is correct, swapping with the smaller child.

Analysis: Suppose the tree has depth k and $n = 2^{k+1} - 1$ nodes. An item that starts at depth j percolates down at most $k - j$ steps. So the total data movement is at most

$$\sum_{j=0}^{k} 2^j (k - j)$$

which is $O(n)$, it turns out.

Thus we get ***Heap-Sort***. Note that one can ***re-use*** the array/vector in which heap in stored: removeMin moves minimum to end, and so repeated application produces sorted list in the vector

A Heap-Sort Example is:

heap					
1	3	2	6	4	5

heap					
2	3	5	6	4	1

heap					
3	4	5	6	2	1

heap					
4	6	5	3	2	1

heap					
5	6	4	3	2	1

heap					
6	5	4	3	2	1

17.5 Application: Huffman Coding

The standard binary encoding of C characters takes $\lceil \log_2 C \rceil$ bits. In a variable-length code, the most frequent characters have the shortest representation. However, now we have to decode the encoded phrase: it is not clear where one character finishes and the next-one starts. In a ***prefix-free code***, no code is the prefix of another code. This guarantees unambiguous decoding: indeed, the ***greedy*** decoding algorithm works:

> *traverse the string until the part you have covered so far is a valid code; cut it off and continue.*

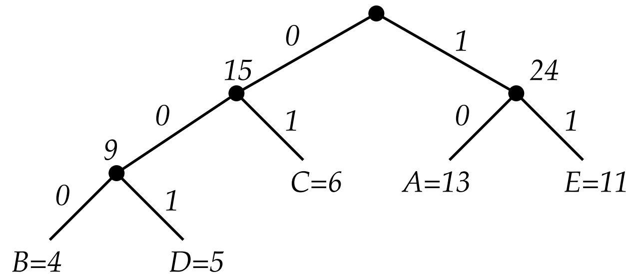
Huffman's algorithm constructs an optimal prefix-free code. The algorithm assumes we know the occurrence of each character:

```
Repeat
    merge the two (of the) rarest characters into a mega-character
    whose occurrence is the combined
Until only one mega-character left

Assign mega-character the code EmptyString
Repeat
    split a mega-character into its two parts assigning each of these
    the mega-characters code with either 0 or 1
```

The information can be organized in a trie: this is a special type of tree in which the links are labeled and the leaf corresponds to the sequence of labels one follows to get there.

For example if 39 chars are A=13, B=4, C=6, D=5 and E=11 we get the coding A=10, B=000, C=01, D=001, E=11.



Note that a priority queue is used to keep track of the frequencies of the letters.

17.6 Sample Code: Heap

```
// PriorityQueue.h - wdg 2008
#ifndef PRIORITY_QUEUE_H
#define PRIORITY_QUEUE_H

typedef double ItemType;
const ItemType ERROR=-1;

class PriorityQueue
{
 public:
  virtual void insertItem(ItemType)=0;
  virtual ItemType removeMin()=0;
  virtual bool isEmpty() const =0;
  virtual int getCount() const =0;
};

#endif
```

```
// Heap.h - wdg 2008
#ifndef HEAP_H
#define HEAP_H
#include "PriorityQueue.h"

class Heap : public PriorityQueue
```

```
{
 public:
   Heap();
   void insertItem(ItemType);
   ItemType removeMin();
   bool isEmpty() const;
   int getCount() const;

 private:
   int count;
   ItemType A[100];
};
#endif
```

```
// Heap.cpp - wdg 2008
#include "Heap.h"

  Heap::Heap( ) : count(0)
  {
  }

  void Heap::insertItem(ItemType newVal)
  {
      int bubble = count;
      int parent = (bubble-1)/2;  // round down
      while(  bubble>0 && newVal<A[parent] ) {
         A[bubble] = A[parent];
         bubble = parent;
         parent = (bubble-1)/2;
      }
      A[bubble] = newVal;
      count++;
  }

  ItemType Heap::removeMin( )
  {
    if (count==0)
      return ERROR;
    else {
      ItemType toReturn = A[0];
```

```
    count--;
    ItemType bubbleVal = A[count];
    int bubble = 0;
    int left = 1;
    int right = left+1;
    while(  ( left < count && bubbleVal>A[left]) ||
             ( right < count && bubbleVal>A[right] ) )
       {
       int smallerChild = left;
       if( right < count && A[right]<A[left] )
         smallerChild = right;
       A[bubble] = A[smallerChild];
       bubble = smallerChild;
       left = 2*smallerChild+1;
       right = left+1;
    } // endwhile
    A[bubble] = bubbleVal;
    return toReturn;
  }
}

bool Heap::isEmpty( ) const
{
  return (count==0);
}

int Heap::getCount( ) const
{
  return count;
}
```