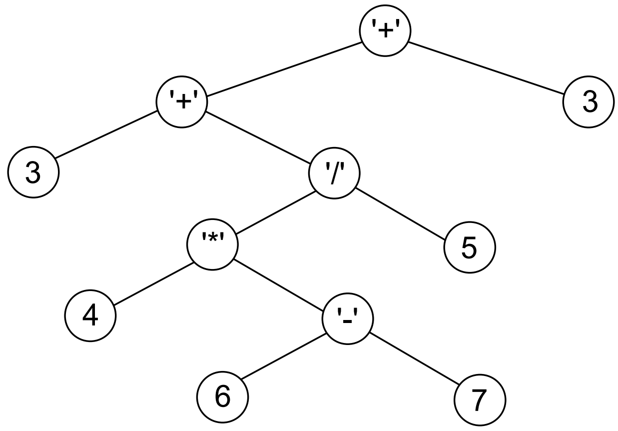
Representation of arithmetic expression using a binary tree



Binary tree representation 3+4(6-7)/5+3*

```
#define operand 0
#define operator 1

struct infoRecord {
    char dataType;
    union
    {
      //all members occupy the same physical space in memory
      //Union - http://www.cplusplus.com/doc/tutorial/other_data_types/
      char opr;
      double val;
    };
}

double evalExprTree(treeNode<infoRecord> *tree)
/* Precondition: the expression tree is nonempty and has no
   syntax error.
   The algorithm is based on postorder traversal. */
{
  if (tree->info.dataType == operand)
     return tree->info.val;
  else
  {
     double d1 = evalExprTree(tree->left);
     double d2= evalExprTree(tree->right);
     char symb = tree->info.opr;

     // compute the result
     return evaluate(symb, d1, d2);
  }
}
```

Huffman Code

- we have an alphabet of *n* symbols
- by assigning a bit string code to each symbol of the alphabet, a long message can be encoded by concatenating the individual codes of the symbols making up the message

For example,

Symbol	Code
A	00
B	01
C	10
D	11

The code for the message ABCACADA would be `0001100010001100`, which requires 16 bits.

To reduce the length of the encoded message, we can use variable-length code

- the code for one symbol cannot be a prefix of the code for another symbol.

For example,

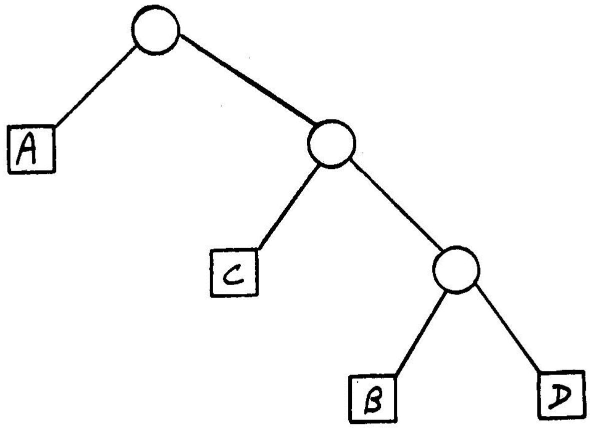
Symbol	Code
A	0
B	110
C	10
D	111

The code for the message ABCACADA would become `01101001001110`, which requires 14 bits.

If the frequency of the characters within a message is known <u>a priori</u>, then an optimal encoding that minimizes the total length of the encoded message can be determined using the Huffman algorithm. (assign shortest code to most frequent symbol)

A binary tree is used to construct/decode Huffman code. The binary tree is known as the Huffman tree.

To decode Huffman code, we interpret a zero as a left branch and a one as a right branch.



Huffman tree of the above example.

Algorithm to construct the Huffman tree

Inputs to the algorithm: the set of symbols and their weights.

```
struct record
{
    int weight;            // or frequency
    char symbol;
};
```

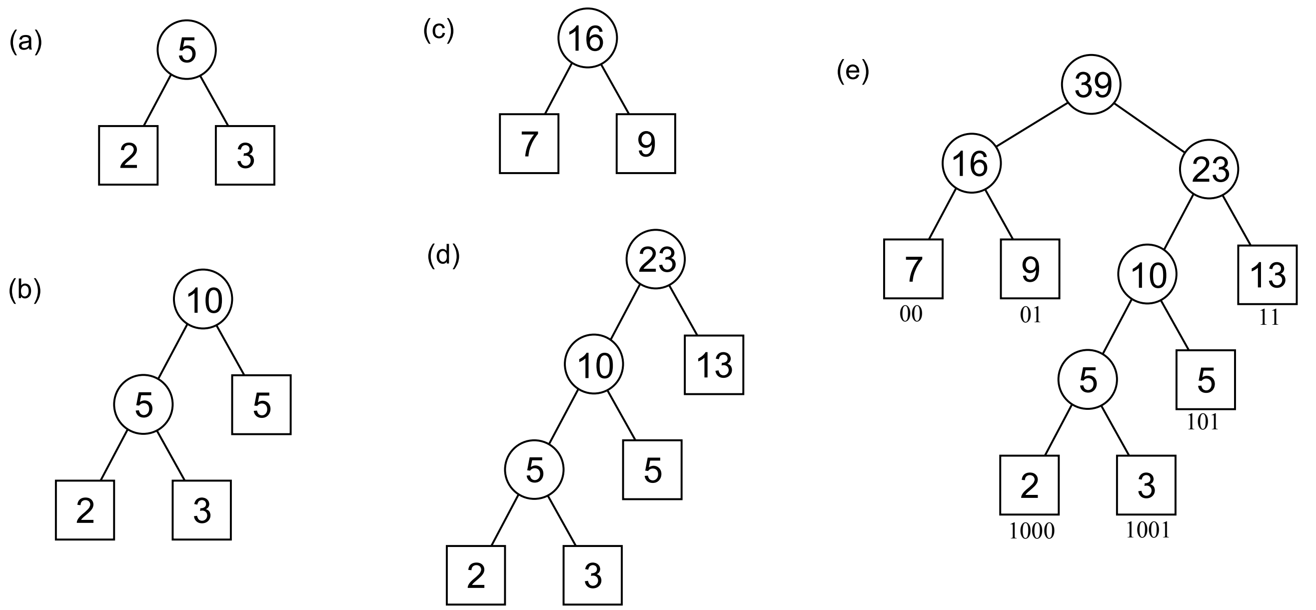
Suppose there are *n* symbols, *n* binary trees corresponding to the *n* symbols are created, each consists of one node. The binary trees are maintained in a (priority) list L.

```
//L is a list of n trees ordered by weight of the tree
treeNode<record>* huffmanTree(PriorityList& L)
{
   //pseudo code
   int n = L.length();
   for(int i = 1; i < n; i++)
   {
      treeNode<record>* t = new treeNode<record>;

      t->left = L.least(); /* retrieve & remove the tree
                              in L with smallest weight */
      t->right = L.least();
      t->info.weight = t->left->info.weight
                     + t->right->info.weight;

      L.insert(t);
   }
   return L.least();
}
```

For example, there are 6 symbols with frequencies
$(q_0,2), (q_1,3), (q_2,5), (q_3,7), (q_4,9), (q_5,13)$



Huffman codes for the 6 symbols

symbol	code
q_0	1000
q_1	1001
q_2	101
q_3	00
q_4	01
q_5	11

Priority queue

- The element to be deleted is the one with the highest priority.

Linear array implementation of priority queue

Case 1: Unordered list

insertion: Insert at the rear which takes O(1) time

deletion: Search for the highest priority element and move the record at the rear to fill the hole. This takes O(n) time.

Case 2: Ordered list

insertion: Insert into an ordered list takes O(n) time.

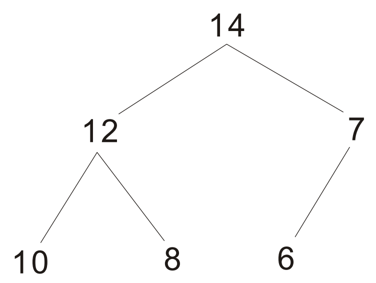
deletion: The highest priority element is at the front or the rear. Removing the front/rear element takes O(1) time.

<u>Implement a priority queue using a Heap</u>

A max tree is a tree in which the key value in each node is no smaller than the key values in its children (if any).

Similarly, a min tree is a tree in which the key value in each node is no larger than the key values in its children (if any).

A max heap (descending heap) is an almost complete binary tree that is also a max tree.



an example max heap

A min heap (ascending heap) is an almost complete binary tree that is also a min tree.

A heap can be represented efficiently using an array.
In this part of the notes, I shall only show the `insert` and `delete` functions.

```
template<class Type>
class heap
{
private:
   Type *store;
   int maxSize;
   int size;

public:

   heap(int capacity=100)
   {
      maxSize = capacity;
      size = 0;
      store = new Type[maxSize];
   }

   void insert(Type x)
   {
      if (size >= maxSize)
      {  cout << "Heap overflow\n" << endl;
         exit(0);
      }

      //i refers to the vacant node under consideration
      int i = size;
      size++;

      while (i > 0 &&
             comparePriority(x, store[(i-1)/2]) > 0)
      {
         // priority of x is higher than priority of
         // store[(i-1)/2]

         //move element from parent node to node i
         store[i] = store[(i-1)/2];
         i = (i-1)/2; //update i to point to its parent
      }
      store[i] = x;
   }
```

```
    Type delete()
    {
       if (size == 0)
       {  cout << "Heap underflow\n" << endl;
          exit(0);
       }

       Type x = store[0];         //element to be removed
       Type k = store[size-1];    //element to be relocated
       size--;

       bool done = false;
       int i = 0;   // i refers to the empty node
       int j = 1;   // j is the left child of i

       while (j < size && !done)
       {
          if (j < size-1)
             if (comparePriority(store[j], store[j+1]) < 0)
                j++;

          if (comparePriority(k, store[j]) >= 0)
             done = true;
          else
          {  //move larger child to parent node
             store[i] = store[j];
             i = j;
             j = 2*i + 1;
          }
       }
       store[i] = k;
       return x;
    }

    // other functions of the class
};

int comparePriority(const Type& a, const Type& b)
{
    // if a < b : return -1
    // if a == b: return 0
    // if a > b : return 1
}
```

Both the insert and delete operations on a heap require $O(\log n)$ time.

Implementing binary tree as a class in C++

To avoid the tedious details, only the implementation of some selected member functions will be given.

A binary tree is a container (i.e. it is used to hold a collection of items).

We need to provide one or more types of iterator such that the external user can use it to traverse the elements in the tree one at a time.

The implementation of the iterator class given below only serves to illustrate the conceptual idea.

Different implementation methods are used in the C++ STL.

```
//binaryTree.h
#ifndef BINARY_TREE_H
#define BINARY_TREE_H

#include <stack>  //required by the iterator
#include <iostream>
#include <iomanip>

using namespace std;


template<class Type>
struct treeNode
{
   Type info;
   treeNode<Type> *left, *right;
};



template<class Type>
class treeIterator
{
protected:
   treeNode<Type> *current;
   stack<treeNode<Type>*> S;

public:
   treeIterator(treeNode<Type> *p);
   treeIterator(const treeIterator<Type>& other);
   Type& operator*();
   treeIterator<Type>& operator++(); //pre-increment operator
   bool operator==(const treeIterator<Type>& other);
   bool operator!=(const treeIterator<Type>& other);
};
```

```
template<class Type>
class binaryTree
{
protected:
   treeNode<Type> *root; //point to root node of the tree

public:
   binaryTree();
   binartTree(const binaryTree<Type>& other);
   ~binaryTree();
   const binaryTree<Type>& operator=(const binaryTree<Type>&
                                     other);
   bool empty();
   void initialize();
   void destroy();

   int height();     //note that the public member functions
   int nodeCount();  //should not require any private member
   void print();     //variables as input parameters

   void preorderTraversal(void (*visit)(treeNode<Type>*));
   //(*visit) is a function pointer, i.e. pass a function by
   //pointer as an input parameter to another function
   //function pointer: http://www.cplusplus.com/doc/tutorial/pointers/

   void inorderTraversal(void (*visit)(treeNode<Type>*));
   void postorderTraversal(void (*visit)(treeNode<Type>*));

   void insert(Type& item) = 0;
   void remove(Type& item) = 0;
   //insert and remove are defined as pure virtual functions,
   //their implementation details will be discussed later.

   treeIterator<Type> begin();
   treeIterator<Type> end();

private:
   void destroy(treeNode<Type> *p);
   int height(treeNode<Type> *p);
   void printTree(treeNode<Type> *p, int indent);
   treeNode<Type>* copyTree(const treeNode<Type>* other);

   void preorder(treeNode<Type> *p,
                 void (*visit)(treeNode<Type>*));
   void inorder(treeNode<Type> *p,
                void (*visit)(treeNode<Type>*));
   void postorder(treeNode<Type> *p,
                  void (*visit)(treeNode<Type>*));

};
```

```
//----- Member functions of binaryTree

template<class Type>
bool binaryTree<Type>::empty()
{
   return root == NULL;
}


template<class Type>
void binaryTree<Type>::initialize()
{
   root = NULL;
}


template<class Type>
int binaryTree<Type>::height()
{
   return height(root);  //call the private function
}


template<class Type>
void binaryTree<Type>::print()
{
   printTree(root, 3);  //initially, indent = 3
}


template<class Type>
const binaryTree<Type>& binaryTree::operator=
                              (const binaryTree<Type>& other)
{
   if (this != &other)
   {
      destroy(); //clear the contents of *this
      root = copyTree(other.root);
   }
   return *this;
}
```

```
template<class Type>
void binaryTree<Type>::destroy()  //public function
{
   destroy(root);  //call the private function
   root = NULL;
}

template<class Type>
void binaryTree<Type>::destroy(treeNode<Type> *p) //private fn
{
   if (p != NULL)
   {
      destroy(p->left);
      destroy(p->right);
      delete p;
   }
}

// You can notice that the function to destroy a binary tree
// is based on postorder traversal.

template<class Type>
void binaryTree::
     postorderTraversal(void (*visit)(treeNode<Type>*))
{
   postorder(root, visit);
}


template<class Type>
void binaryTree::postorder(treeNode<Type> *p,
                           void (*visit)(treeNode<Type>*))
{
   if (p != NULL)
   {
      postorder(p->left, visit);
      postorder(p->right, visit);
      (*visit)(p); //call function (*visit) to process the node
   }
}

/* Remark:
   In the textbook, the function (*visit) is defined as:
   void (*visit)(Type&);

   A disadvantage of passing Type& as input parameter to the
   (*visit) function is that the allowed processing on the
   node is very limited.
*/
```

```
/* Examples to illustrate the uses of function parameter.

   template<class Type>
   void deleteNode(treeNode<Type> *p)  //non-member function
   {
      if (p != NULL)
         delete p;
   }

   template<class Type>
   void printNode(treeNode<Type> *p)  //non-member function
   {
      if (p != NULL)
         cout << p->info << " ";
   }

   void myApplication()
   {
      binaryTree<char> bt;

      // codes to create bt
      // ...

      bt.postorderTraversal(printNode);
      //print the postorder traversal sequence

      bt.postorderTraversal(deleteNode);
      bt.initialize();
      //same effect as bt.destroy()
   }
*/


template<class Type>
treeIterator<Type> binaryTree::begin()
{
   //return an (inorder) iterator that refers to the first
   //element
   return treeIterator<Type>(root);
}

template<class Type>
treeIterator<Type> binaryTree::end()
{
   //return an iterator that refers to the element pass the last
   //element
   return treeIterator<Type>(NULL);
}
```

```
//----- member functions of treeIterator
//Iterator loops over the collection one-by-one, so recursive
//traversal is not suitable here. The treeIterator should
//support non-recursive (inorder) traversal of the underlying
//binary tree.

//reference the non-recursive inorder traversal algorithm

template<class Type>
treeIterator<Type>::treeIterator(treeNode<Type> *p)
{
   //constructor
   current = NULL;

   while (p != NULL)
   {
      S.push(p);
      p = p->left;
   }
   if (!S.empty())
   {
      current = S.top();
      S.pop();
   }
}

template<class Type>
treeIterator<Type>& treeIterator<Type>::operator++()
{
   //advance current to point to its inorder successor
   if (current != NULL)
   {
      current = current->right;
      while(current != NULL);
      {
         S.push(current);
         current = current->left;
      }
      if (!S.empty())
      {
         current = S.top();
         S.pop();
      }
   }
   else
      cerr << "Error: treeIterator gets out of bound" << endl;

   return *this;
}
```

```
template<class Type>
treeIterator<Type>::treeIterator(const treeIterator<Type>& other)
{
   current = other.current;
   S = other.S;
}


template<class Type>
Type& treeIterator<Type>::operator*()
{
   return current->info;
}


template<class Type>
bool treeIterator<Type>::
                operator==(const treeIterator<Type>& other)
{
   //determine if *this and other are referring to the same
   //tree node
   return current == other.current;
}


template<class Type>
bool treeIterator<Type>::
                operator!=(const treeIterator<Type>& other)
{
   return current != other.current;
}


#endif  //end of file binaryTree.h
```