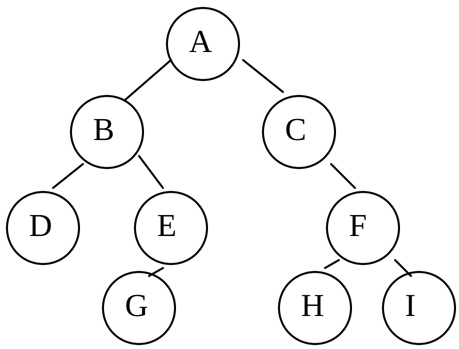
Data Structures

Trees

What is a binary tree?

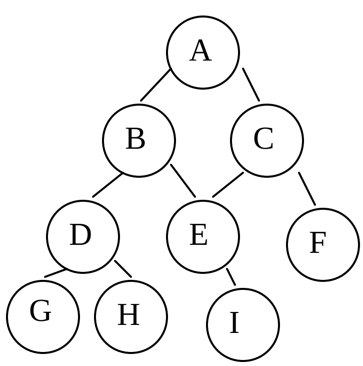
- ***Binary tree*** **-** a finite set of elements that is either *empty* or partitioned into three disjoint sets, called the ***root***, and the ***left*** and ***right subtrees***.

A sample binary tree

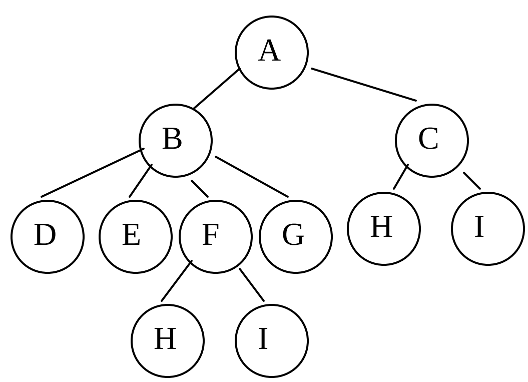


- A is B's father
- B is C's brother
- B & C are A's left and right sons respectively
- C is H's ancestor (grandfather)
- H is C's descendent (grandson)

These are ***NOT*** binary trees – why?



Not a binary tree – The circuit makes it a ***graph***



Not a binary tree – The number of subtrees makes it a ***general tree***

Some Definitions for Binary Trees

- ***Leaf*** – a node with empty left and right subtrees
- ***Strictly binary tree*** – all of the non-leaf nodes have both left and right subtrees.
- ***A complete binary tree*** of ***depth d*** is a strictly binary tree where all of the leaves are at level d. (A complete binary tree of depth d has $2^d - 1$ nodes).
- In an ***almost complete binary tree***:
  - every leaf is at level d or at d-1.
  - every node with a right descendent at level d has a left descendent at level d.

Operations on Binary Trees

For pointer p (pointing to the root of binary tree or subtree):

- Info(p) – returns node contents
- Left(p) – returns pointer for left son of p.
- Right(p) – returns pointer for right son of p.
- Father(p) – returns pointer for father of p.
- Brother(p) – returns pointer for brother of p.

Operations on Binary Trees (continued)

<u>Boolean functions</u>

- Isleft – TRUE is a left son; FALSE if not.
- Isright – TRUE is a right son; FALSE if not.

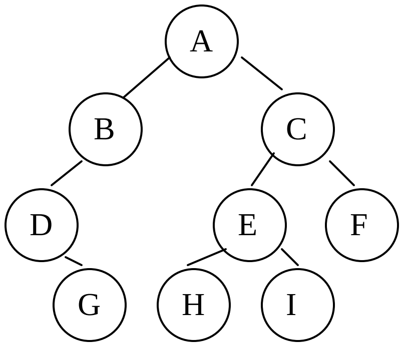
<u>In constructing a tree we need the following operations:</u>

- Maketree – creates a new binary tree with a single node and returns a pointer for it.
- Setleft(p, x) – creates a left son for p with info field x
- Setright(p, x) – creates a right son for p with info field x

Traversing A Tree

- ***<u>Preorder</u>*** – first the root, then the left subtree and lastly the right subtree.
- ***<u>Inorder</u>*** – first the left subtree, then the root and lastly the right subtree.
- ***<u>Postorder</u>*** – first the left subtree, then the right subtree and lastly, the root.

Example of Tree Traversal

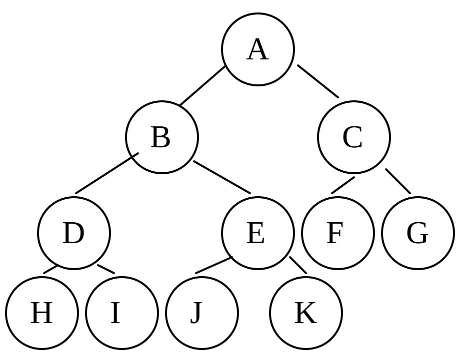


Preorder – ABDGCEHIF

Inorder – DGBAHEICF

Postorder - GDBHIEFCA

Example of Tree Traversal



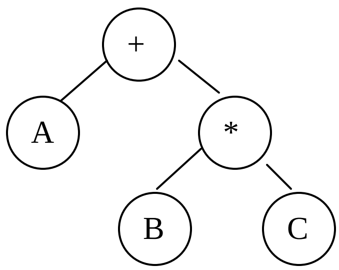
Preorder – ABDHIEJKCFG

Inorder – HDIBJEKAFCG

Postorder - HIDJKEBFGCA

Example of Tree Traversal

We can use the tree to convert infix, prefix, and postfix expression to each

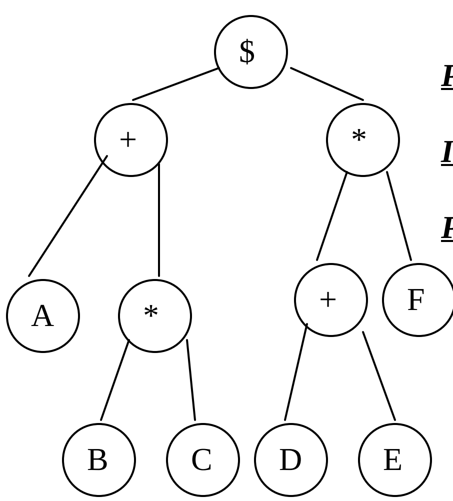


Preorder – + A * B C

Inorder – A + B * C

Postorder - A B C * +

Example of Tree Traversal



Preorder – $+ A * BC *+ DEF

Inorder – (A+B*C)$((D+E)*F)

Postorder - ABC *+ DE +F *$

The **`nodetree`** Class for Array Implementation of Trees

```
#include    <iostream.h>
#include    <stdlib.h>

const int   numnodes = 500;
struct nodetype   {
      int   info;
      int   left, right, father;
};
```

```
class nodetree    {
public:
      nodetree(void);
      int   getnode(void);
      void  freenode(int p);
      int   maketree(int x);
      void  setleft(int p, int x);
      void  setright(int p, int x);
private:
      void  error(char *message);
      struct nodetype   node[numnodes];
        int       avail;
};
```

```
nodetree::nodetree(void)
{
      int   i;

      avail = 0;
      for  (i = 1;  i <= numnodes -1;  i++)
            node[i].left = i+1;
      node[numnodes].left = -1;
}
```

```
int   nodetree::getnode(void)
{
      int   newavail;
      if (avail == -1)  error("Tree overflow");
      newavail = avail;
      avail = node[avail].left;
      return(newavail);
}
void  nodetree::freenode(int p)
{
      node[p].left = avail;
      avail = p;
}
```

```
int   nodetree::maketree(int x)
{
      int   p;

      p = getnode();
      node[p].info = x;
      node[p].left = -1;
      node[p].right = -1;
      node[p].father = -1;
      return(p);
}
```

```
void  nodetree::setleft(int p, int x)
{
      int   q;

      if (p == -1)
            error("Void insertion");
      if (node[p].left != -1)
            error("Invalid insertion");

      q = maketree(x);
      node[p].left = q;
      node[q].father = p;
}
```

```
void  nodetree::setright(int p, int x)
{
      int   q;

      if (p == -1)
            error("Void insertion");
      if (node[p].right != -1)
            error("Invalid insertion");

      q = maketree(x);
      node[p].right = q;
      node[q].father = p;
}
```

```
void  nodetree::error(char *message)
{
      cerr << message << endl;
      exit(1);
}
```

Pointer Implementation of Trees

```
#include  <iostream.h>
#include  <stdlib.h>

struct nodetype     {
     int       info;
     struct    nodetype  *left, *right;
};
typedef   struct nodetype     *nodeptr;
```

```
nodeptr   getnode(void);
void      freenode(nodeptr p);
nodeptr   maketree(int x);
void      setleft(nodeptr p, int x);
void      setright(nodeptr p, int x);
void      error(char *message);
```

```
nodeptr     getnode(void)
{
      nodeptr     p;
      p = new struct nodetype;
      return(p);
}

void  freenode(nodeptr p)
{
      delete p;
}
```

```
nodeptr     maketree(int x)
{
      nodeptr     p;

      p = getnode();
      p ->info = x;
      p -> left = NULL;
      p -> right = NULL;
      return(p);
}
```

```
void  setleft(nodeptr p, int x)
{
      nodeptr     q;

      if (p == NULL)
            error("Void insertion");
      if (p -> left != NULL)  {
            cerr << p -> info << endl;
            error("Invalid insertion");
      }
      q = maketree(x);
      p -> left = q;
}
```

```
void  setright(nodeptr p, int x)
{
      nodeptr     q;

      if (p == NULL)
            error("Void insertion");
      if (p -> right != NULL){
            cerr << p -> info << endl;
            error("Invalid insertion");
        }
      q = maketree(x);
      p -> right = q;
}
```

```
void  error(char *message)
{
      cout << message << endl;
        exit(1);
}
```

The father field is not necessary when traversing downward and therefore, it is rarely used.

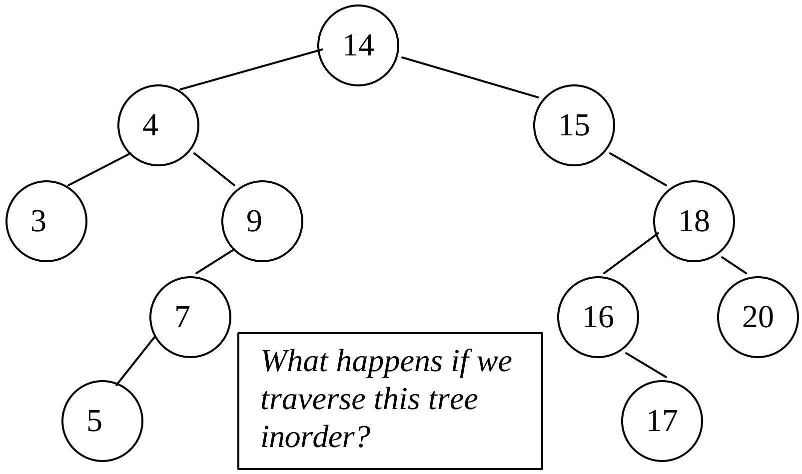
A Program to Find Duplicate Numbers

```
#include    "trees.h"
#include    <fstream.h>

int   main(void)
{
      ifstream    datfile;
      nodeptr     tree, p, q;
      int         number;

      datfile.open("datfile.dat");
      datfile >> number;
```

```
        tree = maketree(number);
        while (!datfile.eof())  {
              datfile >> number;
              p = q = tree;
              while (number != p -> info
                          && q != NULL)     {
                    p = q;
                    if (number < p -> info)
                          q = p -> left;
                    else
                          q = p -> right;

              }
```

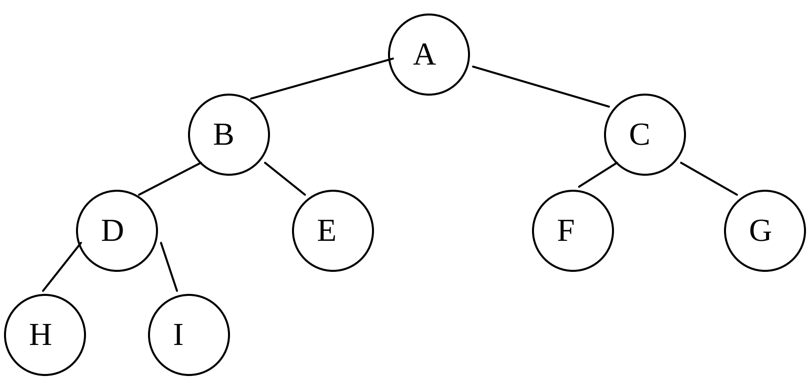
```
              if (number == p -> info)
                    cout << number
                          << " is a duplicate"
                          << endl;
              else if (number < p -> info)
                    setleft(p, number);
              else
                    setright(p, number);
        }

        cout << "That\'s all, folks!!";
        return(0);
  }
```

The Tree Produced by **dup**



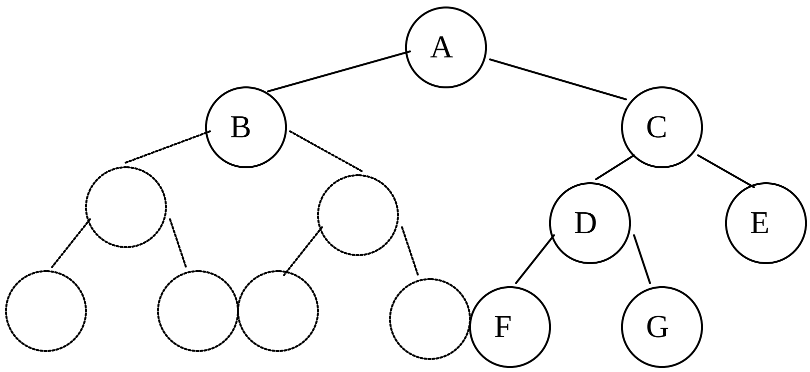
What happens if we traverse this tree inorder?

Array Representation of Binary Trees



0	1	2	3	4	5	6	7	8
A	B	C	D	E	F	G	H	I

What If the Tree Isn't Almost Complete?



A	B	C			D	E	
1	2	3	4	5	6	7	8

Rewriting the trees operations `atree.h`

```
#include  <iostream.h>
#include  <stdlib.h>

const int numnodes = 500;
enum boolean  { False, True};

struct    nodetype  {
     int  info;
     int  used;
};
```

```
struct nodetype     node[numnodes];
void maketree(int x);
void setleft(int p, int x);
void setright(int p, int x);
void error(char *message);
```

```
void  maketree(int x)
{
      int   p;

      for  (p = 1;  p < numnodes; p++)
            node[p].used = False;
      node[0].info = x;
      node[0].used = True;
}
```

```
void  setleft(int p, int x)
{
      int   q;

      q = 2*p+1;  //q = left(p)
      if  (q >= numnodes)
            error("Array overflow");
      else  {
            node[q].info = x;
            node[q].used = True;
        }
}
```

```
void  setright(int p, int x)
{
      int   q;

      q = 2*(p+1);      // q = right(p)
      if (q >= numnodes)
            error("Array overflow");
      else  {
            node[q].info = x;
            node[q].used = True;
        }
}
```

Rewriting **dup.cpp**

```
#include    "atrees.h"
#include    <fstream.h>

int   main(void)
{
      ifstream    datfile;
      int         p, q;
      int         number;

      datfile.open("datfile.dat");
      datfile >> number;
      maketree(number);
```

```
      while (!datfile.eof())  {
            datfile >> number;

            p = q = 0;
            while (q <= numnodes
                  && node[q].used == True
                  && number != node[p].info)    {
                  p = q;
                  if (number < node[p].info)
                        q = 2*p + 1;
                  else
                        q = 2*(p +1);
            }
```

```
            if (number == node[p].info)
                  cout << number
                        << " is a duplicate“
                        << endl;
            else if (number < node[p].info)
                  setleft(p, number);
            else
                  setright(p, number);
      }

      cout << "That\'s all, folks!!";
      return(0);
}
```

Preorder Traversal

```
void  pretrav(nodeptr   tree)
{
      if (tree != NULL) {
            //Visit the root
            cout << tree -> info << endl;
            //Traverse left subtree
            pretrav(tree -> left);
            // Traverse right subtree
           pretrav(tree -> right);
      }
}
```

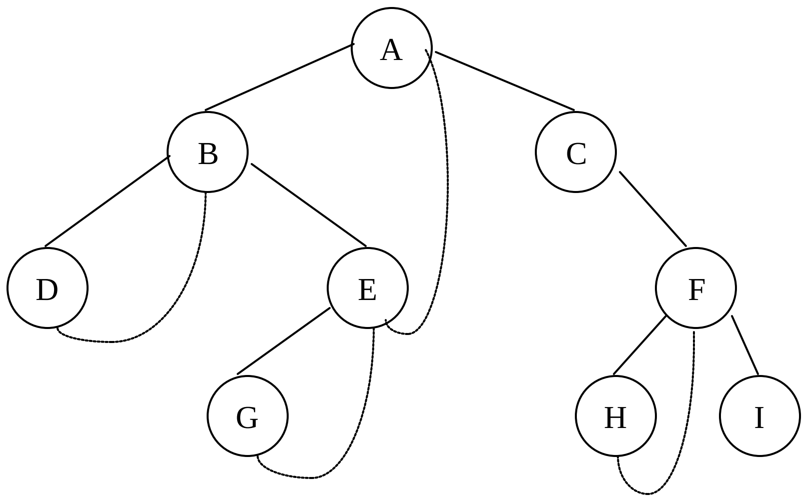
Inorder Traversal

```
void  intrav(nodeptr    tree)
{
      if (tree != NULL) {
            //Traverse left subtree
            intrav(tree -> left);
            //Visit the root
            cout << tree -> info << endl;
            // Traverse right subtree
           intrav(tree -> right);
      }
}
```

Postorder Traversal

```
void  posttrav(nodeptr  tree)
{
      if (tree != NULL) {
            //Traverse left subtree
            posttrav(tree -> left);
            // Traverse right subtree
            posttrav(tree -> right);
            //Visit the root
            cout << tree -> info << endl;
      }
}
```

A Threaded Binary Tree



Threaded Trees

```
#include <iostream.h>
#include <stdlib.h>
enum boolean {false, true};
struct nodetype {
      int info;
      struct nodetype *left, *right;
      boolean thread; // p-> is NULL or a thread
};
typedef struct nodetype *NodePtr;
void        intrav(NodePtr tree);
NodePtr     getnode(int x);
void        setleft(NodePtr tree, int x);
void        setright(NodePtr tree, int x);
void        error(char *message);
```

```
void intrav(NodePtr tree)
{
      NodePtr p, q;
      boolean RightThread;
      // Set p to the root and go as far
      // down to the left as possible
      p = tree;
      do {
            q = NULL;
            while (p != NULL) {
                  q = p;
                  p = p -> left;
            }
```

```
            if (q != NULL) {
                  cout << q -> info << endl;
                  p = q -> right;
                  while (q->thread && p != NULL)  {
                        cout << p -> info
                              << endl;
                        q = p;
                        p = p -> right;
                  }
            }
      } while (q != NULL);
}
```

```
NodePtr getnode(int x)
{
      NodePtr p;

      p = new struct nodetype;
      p -> info = x;
      p -> left = p-> right = NULL;
      p-> thread = true;

      return(p);
}
```

```
void setleft(NodePtr tree, int x)
{
      NodePtr p;
      if (tree == NULL)
            error("Void insertion");

      if (tree -> left != NULL)
            error("Invalid insertion");

      p = getnode(x);
      tree -> left = p;
      p -> right = tree;
      p -> thread = true;
}
```

```
void setright(NodePtr tree, int x)
{
      NodePtr p, q;
      if (tree == NULL)
            error("Void insertion");

      if (!tree -> thread)
            error("Invalid insertion");

      p = getnode(x);
      q = tree -> right;
      tree -> right = p;
      tree -> thread = false;
      p -> right = q;
      p -> thread = true;
}
```

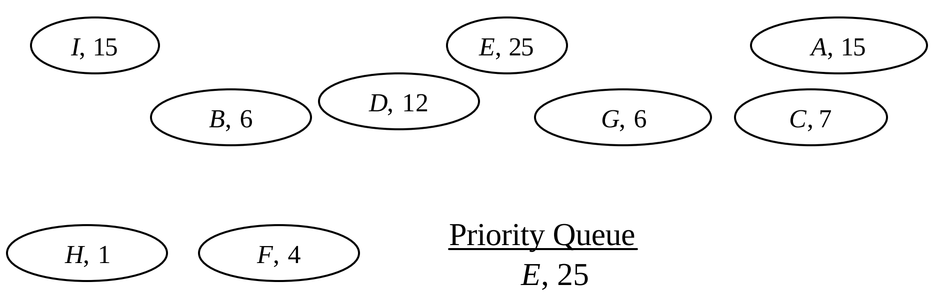
Huffman's Algorithm and Huffman Trees

- One problem that comes up repeatedly in computer science is how to represent data in the most compressed form.
- Imagine that we have a long message that we wish to transmit – how do we represent the characters in the message so they take up the least space?
- We would want the frequency of a character's appearances to be inversely proportional to the length of its representation.

Huffman's Algorithm

- Huffman's algorithm places the characters on a priority queue, removing the two least frequently appearing characters (or combination of characters), merging them and placing this node on the priority tree.
- The node is linked to the two nodes from which it came.

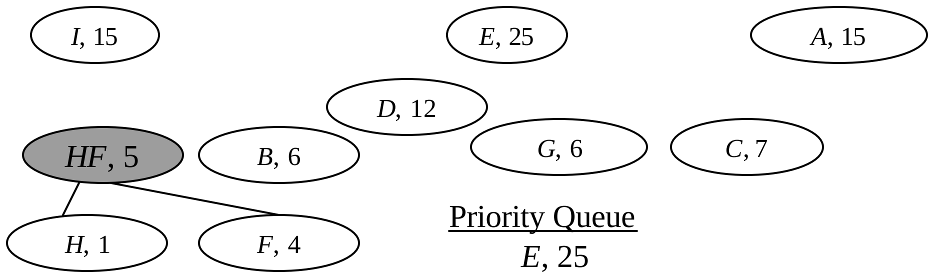
Huffman Tree



<u>Priority Queue</u>

E, 25
I, 15
A, 15
D, 12
C, 7
G, 6
B, 6
F, 4
H, 1

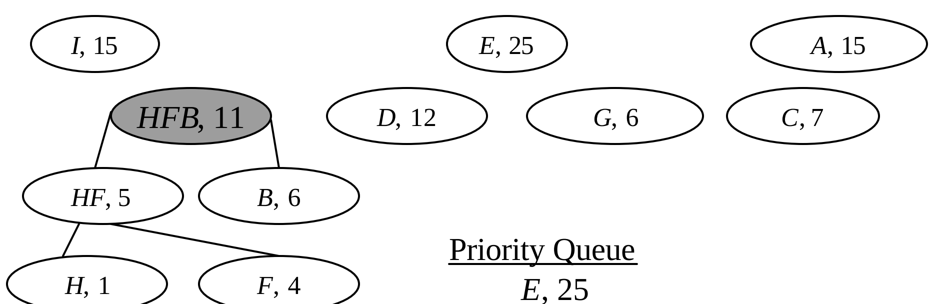
Huffman Tree



<u>Priority Queue</u>

E, 25
I, 15
A, 15
D, 12
C, 7
G, 6
B, 6
HF, 5

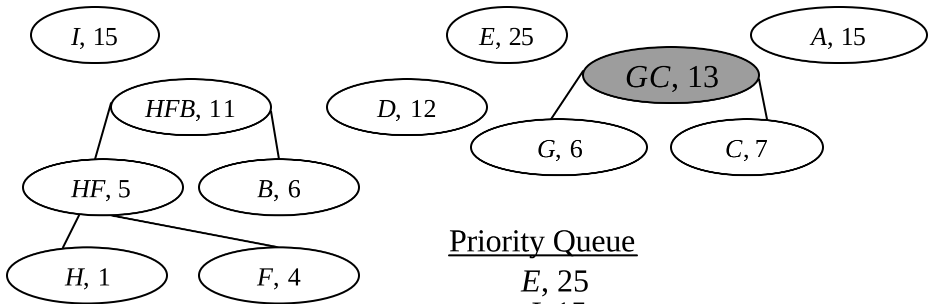
Huffman Tree



<u>Priority Queue</u>

E, 25
I, 15
A, 15
D, 12
HFB, 11
C, 7
G, 6

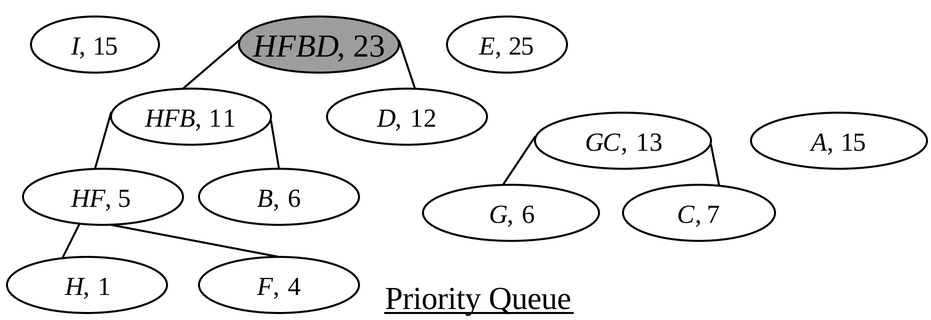
Huffman Tree



<u>Priority Queue</u>

E, 25
I, 15
A, 15
GC, 13
D, 12
HFB, 11

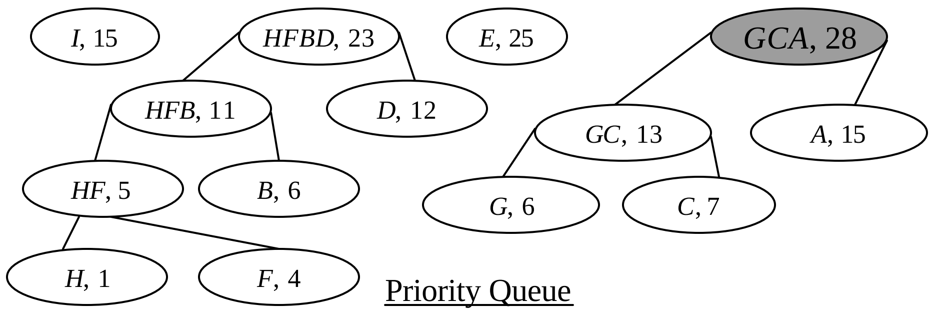
Huffman Tree



<u>Priority Queue</u>

E, 25
HFBD, 23
I, 15
A, 15
GC, 13

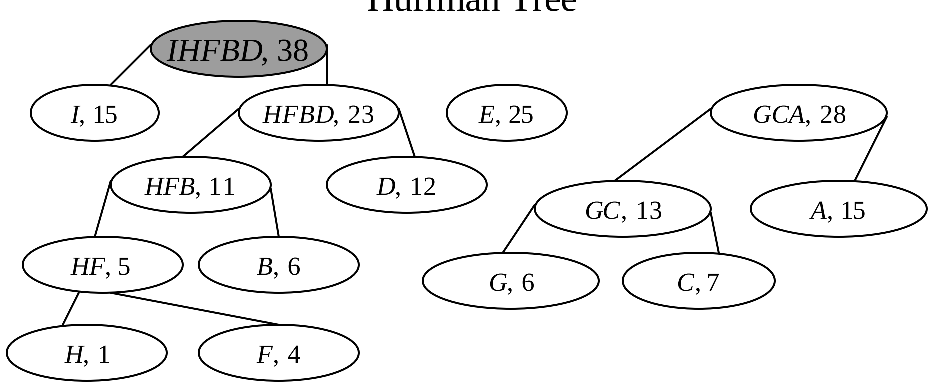
Huffman Tree



<u>Priority Queue</u>

GCA, 28
E, 25
HFBD, 23
I, 15

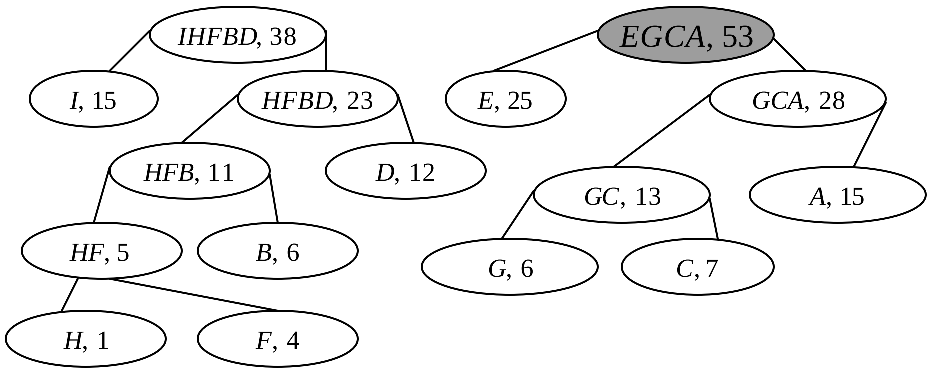
Huffman Tree



<u>Priority Queue</u>

IHFBD, 38
GCA, 28
E, 25

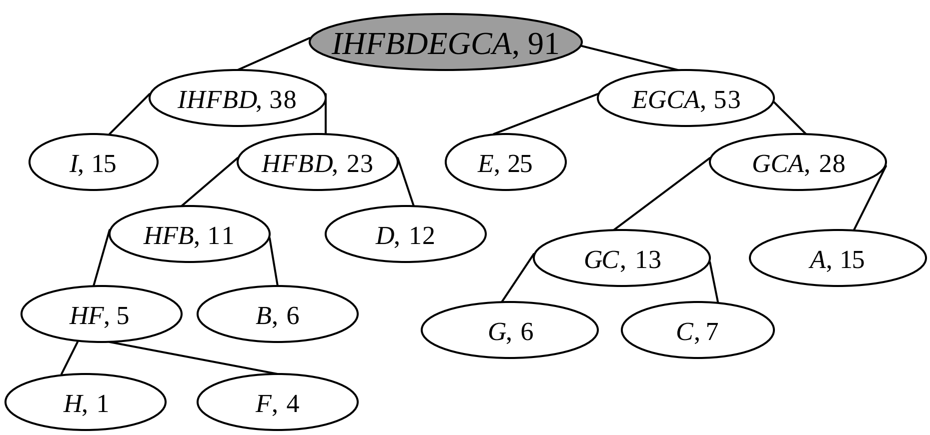
Huffman Tree



<u>Priority Queue</u>

EGCA, 53
IHFBD, 38

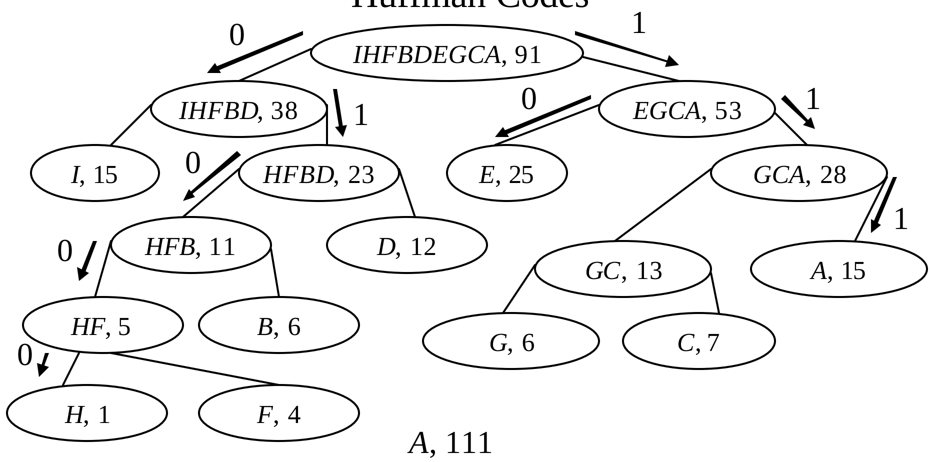
Huffman Tree



<u>Priority Queue</u>

IHFBDEGCA, 91

Huffman Codes



A, 111
E, 10
H, 01000

Huffman's Algorithm

```
#include      <iostream.h>
#include      <fstream.h>
#include      <stdlib.h>
#include      <string.h>

// There can be as many as 50 symbols; Huffman codes
//  can be up to 50 bits in length
const int     MaxSymbols = 50, MaxBits = 50;

// There will be 2*n -1 nodes for n symbols
const int     MaxNodes = 99;
const int     FileNameLen = 20;

enum boolean  {false, true};
```

```
// Each node on the Huffman tree
struct nodetype     {
  char symbols[MaxSymbols];
  int  freq;
  struct nodetype   *father;
  boolean    isLeft;
};
typedef       struct nodetype     *NodePtr;
```

```
// The table in which we will store the data includes:
// the symbol, its frequency, its code and a pointer
// to its leaf in the Huffman tree
class tableclass    {
public:
  friend     PriorityQueue;
  tableclass(void);
  NodePtr    getnode(int i);
  void       display(void);
  inline int tablesize(void)     {return(numentries);}
private:
  char       symbol[MaxSymbols];
  int        freq[MaxSymbols];
  char       code   [MaxSymbols][MaxBits];
  NodePtr    treeptr[MaxSymbols];
  int        numentries;
};
```

```
// tableclass() -   A constructor that opens the data
   file and reads
//             in the symbols and their frequencies.
tableclass::tableclass(void)
{
   ifstream   infile;
   char       filename[FileNameLen];
   int        i;

   // Initially everything is empty since not every
   // entry may be used
   for  (i = 0;  i < MaxSymbols;  i++)   {
        symbol[i] = '\0';
        freq[i] = 0;
        code[i][0] = '\0';
   }
```

```
   // Open the file
   cout <<"File name?";
   cin >> filename;

   infile.open(filename);
   if (!infile)      {
        cerr << "Could not open " << filename << endl;
        exit(1);
   }

   // Read the entries
   for (i = 0;  !infile.eof();  i++)
        infile >> symbol[i] >> freq[i];
   //Keep the number of entries and close the file
   numentries = (symbol[i] == '\0')? i-1 : i;
   infile.close();
}
```

```
// getnode() -      Get a node, fill it with data from
//     the table so it can be placed on the priority
//     queue and eventually on the Huffman tree
NodePtr      tableclass::getnode(int i)
{
   NodePtr    p;

   p = new struct nodetype;
   p -> symbols[0] = symbol[i];
   p -> symbols[1] = '\0';
   p -> freq = freq[i];
   p -> father = NULL;
   p -> isLeft = false;
         treeptr[i] = p;
   return(p);
}
```

```
// display() - Display the table's contents
void   tableclass::display(void)
{
   int  i;

   cout << "There are " << numentries << " entries\n";
   for  (i = 0;  i < numentries;  i++)   {
        cout << symbol[i] << '\t' << freq[i]
        << "\t\"" << code[i] << '\"' << endl;
   }
}
```

```
// The priority queue on which the nodes are placed and
//  eventually removed so the tree can be built.
class  PriorityQueue {
public:
   PriorityQueue(void);
   void        insert(NodePtr px);
   NodePtr     mindelete(void);
   void        init(tableclass &table);
   void        display(void);
   void        getcodes(tableclass &table);
   inline boolean finished(void)
        {return((front -> father == NULL)? true:
   false);}
private:
   NodePtr     front;
};
```

```
//PriorityQueue() - Initialize the pointer to the front
//            of the queue as NULL
PriorityQueue::PriorityQueue(void)
{
   front = NULL;
}

// mindelete() -     Remove the front item from the
//                   queue
NodePtr              PriorityQueue::mindelete(void)
{
   NodePtr     p;

   p = front;
   front = front -> father;
   return(p);
}
```

```
// insert() - Insert an item in its proper place on the
   priority queue
void   PriorityQueue::insert(NodePtr px)
{
   NodePtr    p, q;

   q = NULL;
   // Find its place on the queue
   for (p = front;  p != NULL && px ->freq > p -> freq;
        p = p -> father)
        q = p;

    // If q is Null, place it at the front
   if (q == NULL)    {
        px -> father = front;
        front = px;
   }
```

```
   // Otherwise place it after px
   else {
        px -> father = q -> father;
        q -> father = px;
   }
}

// display() -       Display the items on the priority
//      queue.  This is for debugging purposes.
void   PriorityQueue::display(void)
{
   NodePtr    p;

   for (p = front;  p != NULL;  p = p -> father)
        cout << p-> isLeft << '\t'<< p-> symbols
               << '\t' << p -> freq << endl;
}
```

```
// init() -   Place the initial items on the priority
   queue
void   PriorityQueue::init(tableclass &table)
{
   NodePtr    p;
   int  i;

   for (i = 0;  i < table.tablesize();  i++)    {
        p = table.getnode(i);
   insert(p);
   }
}
```

```
void   reverse(char s[]);

// getcodes() -      Go through the table and determine
//      the Huffman code for each item by working your
//      way up the Huffman tree.
void   PriorityQueue::getcodes(tableclass &table)
{
   int        i;
   char       s[MaxBits], t[2];
   NodePtr    p;
```

```
    for (i = 0;  i < table.tablesize();  i++)    {
        for (p = table.treeptr[i], s[0] = '\0';
              p -> father != NULL; p = p -> father)
              strcat(s,
                    (p -> isLeft == true)? "0" : "1");
        reverse(s);
        strcpy(table.code[i], s);
    }
}
```

```
// reverse() -       reverse string s in place,
//      taken with Kernighan and Ritchie page 62
void   reverse(char s[])
{
    int c, i, j;

    for  (i = 0, j = strlen(s)-1;  i < j;  i++, --j)    {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}


void   buildtree(PriorityQueue &pq);
```

```
//  main() - Build a huffman tree and derive the
//           Huffman codes
int    main(void)
{
   tableclass        table;
   PriorityQueue     pq;

   pq.init(table);
   buildtree(pq);
   pq.getcodes(table);
   table.display();

   return(0);
}
```

```
// buildtree() - Build the Huffman tree
void   buildtree(PriorityQueue &pq)
{
   NodePtr    p, q1, q2;

   // As long as there is more than one node on the
   //   priority queue
   while (!pq.finished())   {
              //Remove the first two items
       q1 = pq.mindelete();
       q2 = pq.mindelete();

       // Get a new node, place in it the concatenated
       // string add the frequencies and set the left
       // son's isLeft value to true, the right son's
       // to false
```

```
        p = new struct nodetype;
        strcpy(p -> symbols, q1 -> symbols);
        strcat(p -> symbols, q2 -> symbols);
        p -> freq = q1 -> freq + q2 -> freq;
        p -> father = NULL;
        p -> isLeft = false;
        q1 -> father = q2 -> father = p;
        q1 -> isLeft = true;
        q2 -> isLeft = false;
        pq.insert(p);
    }
}
```

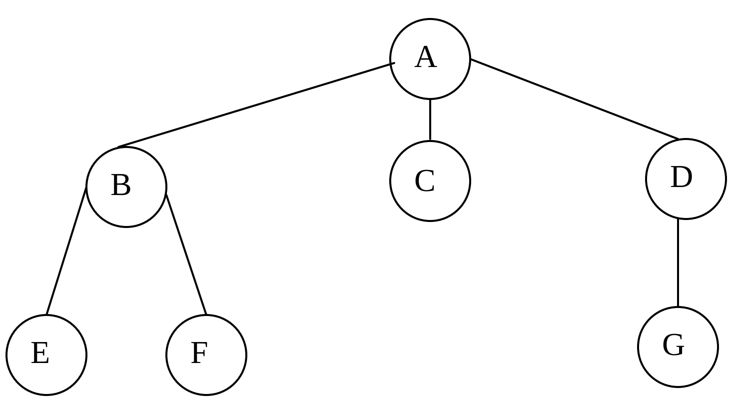
General Trees

- A (general) ***tree*** is a ***finite nonempty set*** of elements in which one element is called the *root* and the remaining elements are partitioned into *m* ³ *0* disjoint subsets, each of which is itself a tree. These elements are each called ***nodes***.
- As before, a node without subtrees is called a leaf. And the terms father, son, brother, ancestor, descendent, level and depth have the same meaning as they do with binary trees.

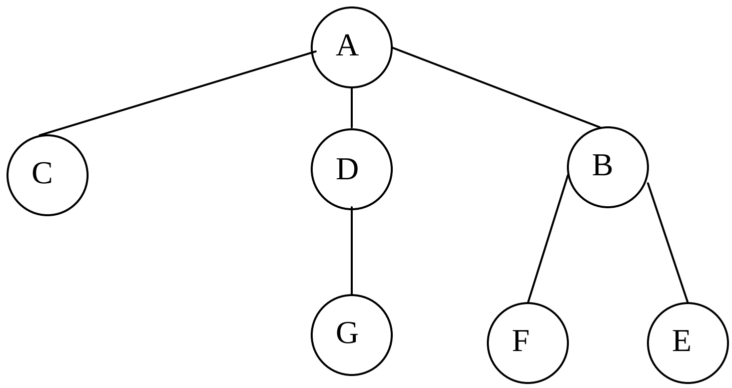
Ordered Trees

- An ***<u>ordered tree</u>*** is defined as a tree in which subtrees of each node form an ordered set, which we may called first, second, or last.
- We typically call these the ***<u>oldest</u>*** through ***<u>youngest</u>*** sons.
- A ***<u>forest</u>*** is an ordered set of ordered trees.

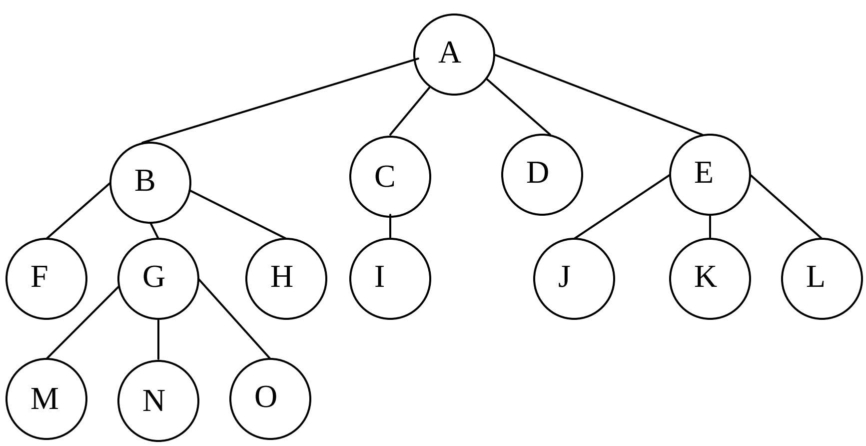
Examples of General Trees



Another Example of General Trees



Yet Another Example of General Trees



Implementing General Trees

We could write:

```
const int MaxSons = 20;

struct treenode{
  int               info;
  struct treenode   *father;
  struct treenode   *sons[MaxSons];
}
```

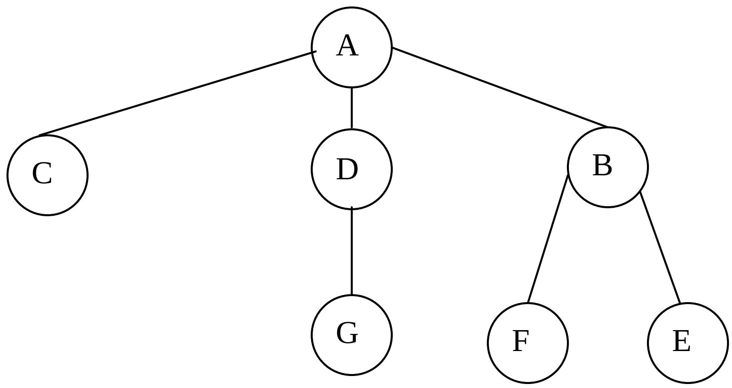
But this limits the number of sons or wastes memory.

How We Implement General Trees

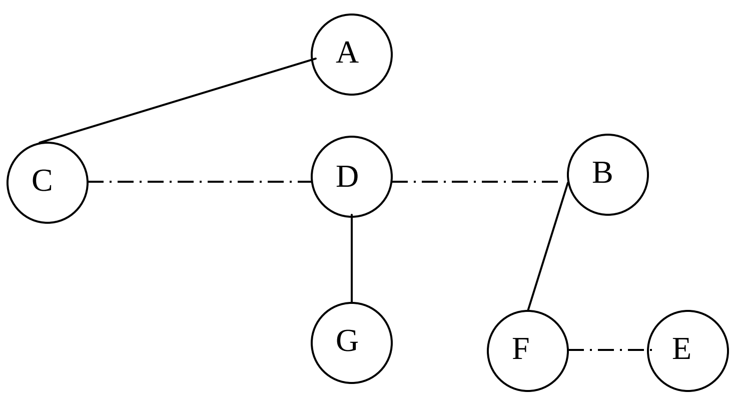
Instead, we write:

```
struct treenode     {
  int               info;
  struct treenode   *son;
  // Next younger brother
  struct treenode   *next;
}
typedef   struct treenode     *NodePtr;
```

Our Tree



What Our Implementation Looks Like



Traversing General Trees

- As with binary trees, there are three traversal methods for forests:
  - Preorder traversal
  - Inorder traversal
  - Postorder traversal

Preorder traversal

1. Visit the root of the first tree in the forest
2. Traverse preorder the forest formed by the first tree's subtrees.
3. Traverse preorder the remaining trees in the forest.

```
void      pretrav(NodePtr p)
{
    if (p != NULL)    {
          cout << p -> info << endl;
          pretrav(p -> son);
          pretrav(p -> next);
    }
}
```

Inorder traversal

1. Traverse inorder the forest formed by the first tree's subtrees.
2. Visit the root of the first tree in the forest
3. Traverse inorder the remaining trees in the forest.

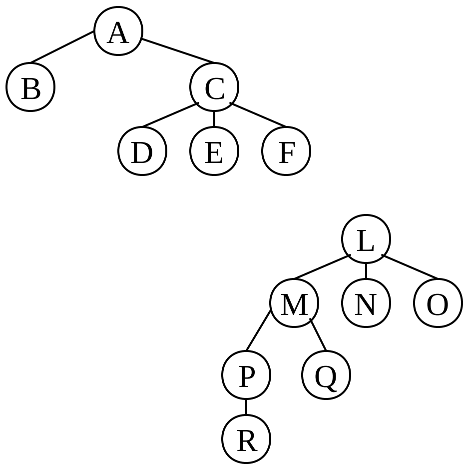
```
void      intrav(NodePtr p)
{
    if (p != NULL)    {
          intrav(p -> son);
          cout << p -> info << endl;
          intrav(p -> next);
    }
}
```

Postorder traversal

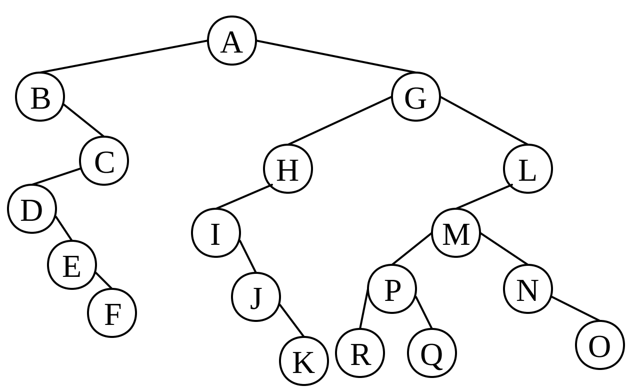
1. Traverse postorder the forest formed by the first tree's subtrees.
2. Traverse postorder the remaining trees in the forest.
3. Visit the root of the first tree in the forest

```
void      posttrav(NodePtr p)
{
    if (p != NULL)    {
          posttrav(p -> son);
          posttrav(p -> next);
          cout << p -> info << endl;
    }
}
```

A Forest



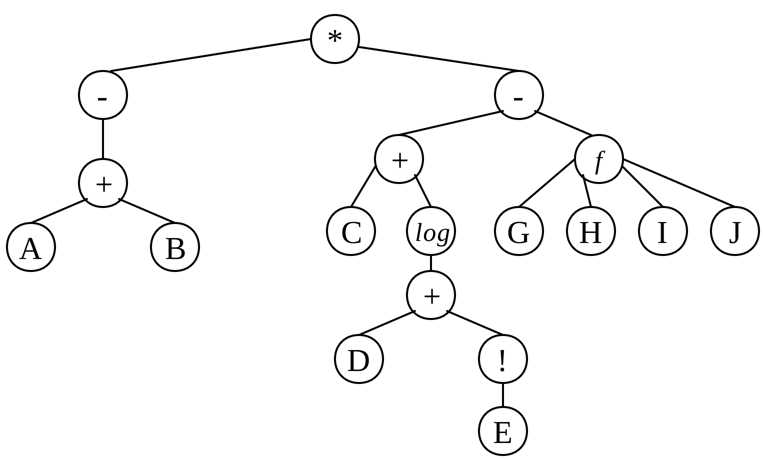
The Equivalent Binary Tree



Traversal Example

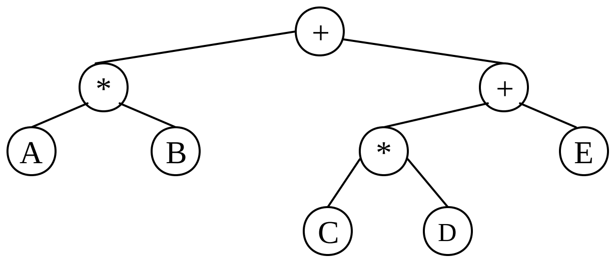
- For the forest shown:
  - Preorder traversal is ABCDEFGHIJKLMPRQNO
  - Inorder traversal is BDEFCAIJKHGRPQMNOL
  - Postorder traversal is FEDCBKJIHRQPONMLGA

Expression Trees



-(A+B)*(C + log(D + E!) - f(G, H, I, J))

Evaluating Expression Trees



Preorder: + * A B + C D E
Inorder: A * B C * D + E
Postorder: A B * C D * E + +

Game Trees

- General trees prove to be a useful strategy for planning moves in a game.
- Eliminating a potential move eliminates the entire subtree as a set of possible scenarios

Example of Game Tree: Nim