CSC 181F Lecture Notes

These lecture notes are provided for the personal use of students taking CSC181F in the Fall term 1999/2000 at the University of Toronto

Copying for purposes other than this use and all forms of distribution are expressly prohibited.

Reading Assignment

`K.N. King          Sections 17.1, 17.3, 17.4`

Dynamic Storage Allocation

- Dynamic storage allocation is a mechanism that allows the programmer to create new storage for data during the execution of a program
- Dynamic storage allocation is used to build complicated data structures like lists, trees, and graphs whose size and shape is determined during program execution
- Think of using dynamic storage allocation for problems where you have to deal with arbitrarily large amounts of data or where the structure of the data isn't known before program execution
- **Use of dynamic storage allocation is an important programming technique because it frees the programmer from the restrictions imposed by statically sized data structures.**

Dynamic Storage Allocation

- Storage can be allocated for any type of data (i.e any *type-name*).
- *Pointers* are used to access dynamically allocated storage.
- Storage must be allocated and deallocated under programmer control
- Items in storage can be linked together with pointers in *arbitrary* ways
- Items are accessed using pointers, any item not pointed to by some pointer is permanently inaccessible

Storage Allocation Functions

#include < stdlib.h >
void *malloc(size_t size)
void *calloc(size_t nobj , size_t size)
void *realloc(**void** * p, size_t size)

- malloc allocates *size* bytes of storage and returns a pointer to the storage. calloc allocates storage for an array of *nobj* elements where each element requires *size* bytes of storage
- realloc changes (smaller or larger) the size of the block of storage pointed to by *p* to be *size* bytes. Returns a pointer to the new storage. Copies old block of storage to new block of storage up to min(oldSize, newSize)
- Since malloc *et.al.* return a value of type **void *** you should use a type cast to convert the type of the pointer returned to what you need.

- malloc doesn't initialize memory but calloc does initializes the memory by setting all bits to 0.
- When it is not possible to locate a block of memory large enough, the storage allocation functions return NULL.
  WARNING: Always check if the return value is NULL.

  ```
  p = (char *) malloc(n+1);
  if (p == NULL) ...
  ```

- **WARNING:** Be sure that a pointer passed to realloc came from previous call of malloc, calloc, or realloc
- WARN realloc *may* move the data being reallocated to a different part of memory. This will **invalidate** all existing pointers to that data. The **ONLY** pointer you can trust is the one returned by malloc.

Deallocating Storage

void free(**void** * p)

---

free releases the storage block pointed to by *p*.
That storage must have been allocated using malloc, calloc or realloc or CHAOS will ensue.

Dangling pointers are pointers that point to storage elements that have already been freed

It is a logic error in your program if you try to use a dangling pointer

HOW TO Use Storage Allocation

- Always use **sizeof** to calculate the amount of storage reqested from the storage allocation functions.
- Never pass any pointer to free that wasn't obtained directly from one of the storage allocation functions. CHAOS will ensue if you get this wrong.
- Safe Storage Allocation

```
#define MALLOC( size, type , pointer ) \
  { void * mTemp ; \
    mTemp = malloc( (size) ) ; \
    assert ( mTemp != NULL ) ; \
    pointer = ( type ) mTemp ; \
  }
```

Storage Allocation Examples

```
int * A ;
int nWords = 100 ;
A = (int * ) malloc(nWords * sizeof ( int ) ) ;
```

```
typedef struct {
    int X , Y ;
} Point ;
typedef struct Point * PointPtr ;
#define PointArraySize ( 360 )
PointPtr P ;
P = ( PointPtr ) calloc( PointArraySize , sizeof ( Point ) ) ;
P[100] -> X = 10 ;
P[100] -> Y = 20 ;
```


NULL Pointer Dereferencing

- It is an ERROR to apply the pointer dereferencing operator (*) to any pointer that has the value NULL.
  The following property should hold for correct programs.

  At every point where a pointer variable is dereferenced, the pointer variable **does not** have the value NULL

  If you cannot make an informal argument that this property holds at every pointer dereference than you *have* an ERROR in your program.
- **WARNING: The NULL pointer is lurking everywhere waiting to cause your program to crash**

Dangling Pointers

- A dangling pointer is a pointer that points at some block of storage that has been freed.
  It is an ERROR to use a dangling pointer since it points to GARBAGE. The following property should hold for correct programs.

  At every point where a pointer variable is dereferenced, the pointer variable **can not be** a dangling pointer.

  If you cannot make an informal argument that this property holds at every pointer dereference than you *have* an ERROR in your program.

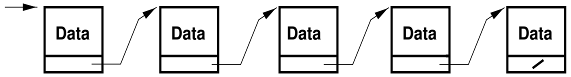
Dangling Pointer Example

```
typedef struct {
    float X ;
    ...
} bigStruct ;
typedef struct bigStruct * bigPtr ;
bigPtr P, Q ;
P = ( bigPtr ) malloc( sizeof ( bigStruct ) ) ;
...
Q = P ;
...
free( P ) ;
...
/* Q is a dangling pointer */
Q -> X = 25.7 ;      /* ERROR */
```

Memory Leaks

- A program has a *memory leak* if it allocates storage that is never freed.
- Memory leaks are **not** an issue for short lived programs (e.g. programming assignments) that do something and then stop. the operating system cleans up memory when a program ceases execution.
- Memory leaks are an issue for long running programs like operating systems, web browsers, word processors, and X window systems because the leaks cause the program to grow slowly over time until it becomes to large to continue running.
- **IF** you are writing a long lived program you should take care to avoid memory leaks. This is *hard*, it takes a lot of really careful memory management.

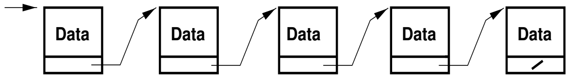
List Data Structures



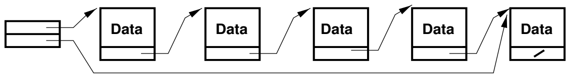
- A list is a sequence of nodes that are chained together using pointers.
- By convention the pointer value **NULL** is used to mark the end of a list.
- Lists allow you to deal with arbitrary amounts of data
- List processing algorithms are designed to processes lists from beginning to end. Random access to items in a list requires searching which can be *very* inefficient.
- *Recursive functions* are often the best way to design list processing algorithms.
- List items are called *nodes, cells,* or *elements.* The explicit data that specifies "next" are called *pointers* or *links*.
- A Pointer variable which contains a pointer value to the first node on a list is called *header*

Types of Lists

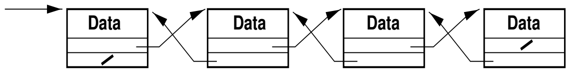
- *Singly linked lists* A sequence of nodes linked by a single pointer.



- *Singly linked list with header* A singly linked list that is accessed via a header block that contains pointers to the first and last node in the list.



- *Doubly linked list* A sequence of nodes where each node contains a pointer to the next node in the list and a pointer to the *previous* node in the list



Singly Linked List Declarations

```
/* singly linked lists */
typedef struct listNode * ListPtr ;
struct listNode {
    /* Declare Data Here */
    int value ;      /* list of integers */
    listPtr next ;   /* link to next node */
} ;
typedef struct listNode LISTNODE ;
#define LISTNODESIZE ( sizeof( LISTNODE ) )
```

HOW TO Think About Lists

- Lists are inherently a **linear** data structure.
- For singly linked lists, the only access to the list is via a pointer to the head of the list.
- Algorithms that process lists *must* be designed to work through the nodes in a list in order.
- List processing algorithms **must** be designed to handle the empty list correctly.
- Absolutely arbitrary data items can be stored in lists,
  e.g. structures, arrays, lists, trees ... Data can be *ordered* or *unordered*.
- Lists allow you to write programs that can deal with arbitrarily large amounts of data (limited only be the size of memory).

Operations on Lists

- *Append* - Add a new node to the end of a list.
- *Delete* - Remove a node from the list.
- *Trace* - Chain through the nodes in a list, performing some processing on each node.
- *Search* - Chain through a list looking for a node that contains a particular value.
- *Concatenate* - Append one list to the end of another list.

NULL & Dangling Pointers

- **WARNING: BEWARE of NULL pointers, they are lurking EVERYWHERE trying to cause bugs in your programs**
- **It is a logic error in your program if you try to use a pointer with the value NULL to access data**
- You should be able to establish the following condition:
  At every place where a pointer is dereferenced
  *(e.g. constructs of the form * pointerVariable and pointerVar −>)*
  the pointerVar can NOT have the value NULL
- Dangling pointers are pointers that point to storage elements that have already been freed
- **It is a logic error in your program if you try to use a dangling pointer to access data.**

Recursion and List Processing

Recursion is often a very good way to design and express algorithms that process lists.

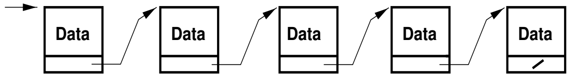
Basis:	The empty list (NULL list pointer)
Decomposition:	Process node pointed at by list pointer Process rest of list recursively.
Composition	Combine result for node pointed at with result for rest of list.

Generic Recursive List Processing Model

```
type-name Func ( ListPtr inputList )
{
    if ( inputList == NULL ) {
        /* Return value for NULL list */
        return ...
    }
    else {
        ListPtr restResult ;
        /* Process node pointed to by inputList */
        ...
        /* Recursively process the rest of the list */
        restResult = Func( inputList-> next ) ;
        /* Combine result for this node and restResult */
        return ...
    }
}
```


Generic Interative List Processing Model

- Use a temporary pointer variable to point to the node being processed in the list.
- Initialize this pointer to point at the first node in the list.
- Stop processing when the temporary pointer has the value NULL. Conveniently NULL == **false** .
- Use the *next* link in each node to advance the temporary pointer from one node to the next.



Examples - Iterative List Processing

```
ListPtr P, Q ;
/* Assume P points at a list */
/* Process list pointed at by P */
Q = P ;
WHILE( Q ) {
    /* process node pointed */
    /* at by Q */
    /* advance to next node */
    Q = Q -> next ;
}
```

```
for ( Q = P ; Q ; Q = Q-> next ){

    /* process node pointed at */
    /* by Q here             */

}
```


Creating Lists

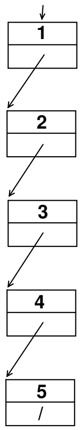
- Lists are created one node at a time.
  - Use malloc to allocate storage for the node.
  - Use the size of the list node as the argument to malloc.
  - **Good Style:** Always check that malloc returned a non-NULL pointer.
  - Add the node to the list at an apropriate place.
- **Good Technique:** If there are any pointers in a node that are not given a value immediately after allocation initialize those pointers to NULL. This action turns uninitialized pointer dereference errors into NULL pointer dereference errors which are more likely to be detected.

Example:

```
ListPtr P ;
P = ( ListPtr ) malloc( LISTNODESIZE ) ;
assert ( P ) ;
P-> value =     /* some value */     ;
P-> next = NULL ;
```

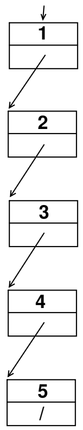
List Processing Examples

```
ListPtr P , Q ;
int listSum ;
/* ASSUME P is set to point at list here */

/* Compute sum of nodes in list */
for ( listSum = 0 , Q = P ; Q ; Q = Q-> next )
    listSum + = Q-> value ;

/* traverse and print list */
Q = P ;
while ( Q ) {
    printf("%d\n", Q-> value ) ;
    Q = Q-> next ;
}
```



List Processing Examples

```
/* Destroy entire list */
ListPtr Q;
while ( P ) {
    Q = P-> next ;
    free( P ) ;
    P = Q ;
}
/* Reverse Order of Nodes in List */
ListPtr Q = NULL, R ;
while ( P ) {
    R = P-> next ;
    P-> next = Q ;
    Q = P ;
    P = R ;
}
```



Example - Input to a List

```
/* read data and add to */
/* list in order */
ListPtr P = NULL, Q = NULL ;
while ( ! feof(stdin) ) {
    if ( Q == NULL ) {
        /* first node in list */
        Q = (ListPtr)malloc(LISTNODESIZE) ;
        assert ( Q ) ;
        P = Q ;
    } else {
        Q-> next = (ListPtr)malloc(LISTNODESIZE) ;
        Q = Q -> next ;
        assert ( Q ) ;
        }
    Q -> next = NULL ;
    scanf ("%d", Q-> value) ;
}
```

```
/* read data and add to */
/* list in REVERSE order */
ListPtr P = NULL, Q = NULL ;
while ( ! feof(stdin) ) {
    Q = (ListPtr)malloc(LISTNODESIZE) ;
    assert ( Q ) ;
    Q-> next = P ;
    scanf ("%d", Q-> value) ;
    P = Q ;
}
```


Example - Append to Integer List

```
/* Add value K to end of list P */
ListPtr P, Q, R ;
/* Assume P gets value here */
R = (ListPtr) malloc( LISTNODESIZE );
assert( R );
R-> value = K ;
R-> next = NULL ;
if ( P == NULL )
    P = R ;  /* List was empty */
else {       /* find end of list */
    Q = P ;
    while ( Q-> next != NULL )
        Q = Q-> next ;
    Q -> next = R ;
}
```

```
ListPtr Append( ListPtr P , int K ) {
    if ( P == NULL ) {
        ListPtr Q ;
        Q = (ListPtr) malloc( LISTNODESIZE ) ;
        assert( Q );
        Q-> value = K ;
        Q-> next = NULL ;
        return Q;
    } else {
        P-> next = Append( P-> next, K) ;
        return P;
    }
}
```

Example - Copy a List

```
/* Set Q to copy of list P */
ListPtr P, Q, R = NULL ;
/* Assume P gets value here */
while ( P != NULL ) {
    if ( R == NULL ) {
        /* First Node */
        R = (ListPtr) malloc( LISTNODESIZE );
        assert( R ) ;
        Q = R ;
    } else {
        R-> next = (ListPtr) malloc( LISTNODESIZE ) ;
        R = R-> next ;
        assert( R ) ;
    }
    * R = * P ; /* Copy node */
    R-> next = NULL ;
    P = P-> next ;
}
```

```
/* Return Copy of List P */
ListPtr Copy( ListPtr P ) {
    if ( P == NULL )
        return NULL ;
    else {
        ListPtr Q ;
        Q = (ListPtr) malloc( LISTSIZE );
        assert( Q ) ;
        * Q := * P        /* copy node */
        Q-> next = Copy( P-> next ) ;
        return Q
    }
}
```


Insert in List

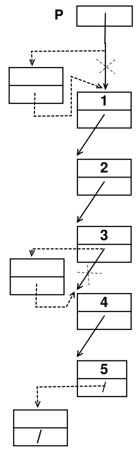
Cases:

Empty list

Insert at head

Insert in Middle

Insert at end



Iterative Ordered List Insert

```
void insert( ListPtr * P , int V ){
    /* NOTE P is LISTNODE * *        */
    /* insert node for V in ordered list P */
    assert ( P ) ;
    ListPtr Q = * P ;
    ListPtr newPtr = (ListPtr) malloc( sizeof(LISTNODE)) ;
    assert (newPtr ) ;
    newPtr -> value = V ;
    if ( Q == NULL || Q -> value > V ){
        /* empty list or insert at head */
        * P = newPtr ;
        newPtr -> next = Q ;
    }else {
        ListPtr nextPtr = Q -> next ;
        while ( TRUE ){
            assert( Q ) ;
            if ( nextPtr == NULL || nextPtr -> value > V ){
                /* insert at middle or end */
                Q -> next = newPtr ;
                newPtr -> next = nextPtr ;
                break ;
            } ;
            Q = nextPtr ;
            nextPtr = Q -> next ;
        }
    }
}
```


INSPECT for non-NULL Pointer Property

```
void insert( ListPtr * P , int V ){
    /* NOTE P is LISTNODE * *        */
    /* insert node for V in ordered list P */
    assert ( P ) ;
    ListPtr Q = * P ;
    ListPtr newPtr = (ListPtr) malloc( sizeof(LISTNODE)) ;
    assert (newPtr ) ;
    newPtr -> value = V ;
    if ( Q == NULL || Q -> value > V ){
        /* empty list or insert at head */
        * P = newPtr ;
        newPtr -> next = Q ;
    }else {
        ListPtr nextPtr = Q -> next ;
        while ( TRUE ){
            assert( Q != NULL ) ;
            if ( nextPtr == NULL || nextPtr -> value > V ){
                /* insert at middle or end */
                Q -> next = newPtr ;
                newPtr -> next = nextPtr ;
                break ;
            } ;
            Q = nextPtr ;
            nextPtr = Q -> next ;
        }
    }
}
```

Recursive Ordered List Insert

```
void insert( ListPtr * P , int V){
    /* NOTE P is LISTNODE * * */
    assert ( P ) ;
    /* insert node for V in ordered list P */
    if ( * P == NULL || ( * P )-> value > V ){
            ListPtr Q = * P ;
            * P = (ListPtr) malloc( LISTNODESIZE ) ;
            assert ( * P ) ;
            (* P)-> value = V ;
            (* P)-> next = Q ;
    }else {
            insert( & P-> next, V ) ;
    }
}
```


Example - Apply

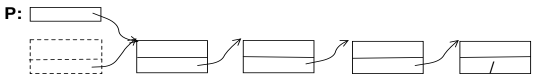
Apply a function F to every node in a list.

Return list containing result.

```
ListPtr apply( ListPtr P , int F ( int val ) ){
    if ( P == NULL )
        return NULL ;
    else {
        ListPtr newNode ;
        newNode = (ListPtr) malloc( LISTNODESIZE ) ;
        assert ( newNode) ;
        newNode-> value = (* F)( P-> value ) ;
        newNode-> next = apply( P-> next , F ) ;
        return newNode ;
    }
}
```


Technique - Dummy List Head

- In processing a list, the first node in the list often needs to be treated differently since it is pointed to by a pointer that is not in a list node Algorithms for processing lists can often be simplified by adding a dummy node at the start of the list. This eliminates the need to treat the first list node as a special case.
- Before Processing:
  Create dummy list node
  Set next link in dummy node to point at list
- After processing:
  Restore original list pointer from dummy node next link
  This is necessary for situations where the head of the list is changed

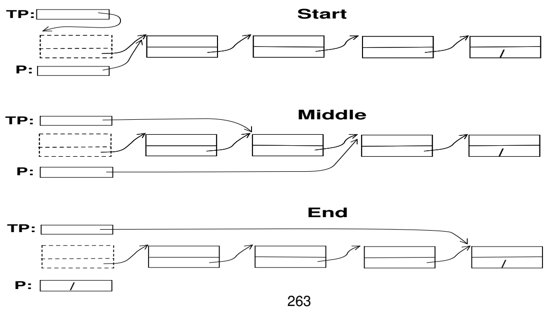


Example - Input using Dummy List Head

```
ListPtr P , Q ;
LISTNODE dummy ;
/* Set up dummy list head */
Q = & dummy ;
while ( ! feof(stdin) ) {
    Q-> next = (ListPtr)malloc(LISTNODESIZE) ;
    Q = Q-> next ;
    assert ( Q != NULL ) ;
    scanf ("%d", Q-> value ) ;
}
/* Finish off List */
Q-> next = NULL ;
P = dummy. next
```

Technique - Trailing Pointers

- For many list processing algorithms, it is convenient to keep pointers to the list node being processed and the immediately preceding node in the list. The pointer to the preceding node is called a *trailing pointer*.
- Dummy list heads are the recommended technique for setting up trailing pointers

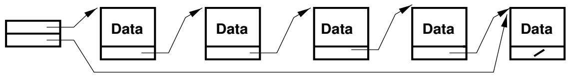


Iterative Ordered List Insert - Trailing Pointer

```
void insert( ListPtr P , int V ) {
    /* insert node for V in ordered list P */
    ListPtr trailP, currentP, newPtr ;
    newPtr = (ListPtr) malloc( LISTNODESIZE ) ;      /* Create new node for V */
    assert ( newPtr ) ;
    newPtr -> value = V ;
    /* Setup dummy head and trailing pointer */
    dummyHead. next = P ;
    trailP = & dummyHead ;
    currentP = trailP -> next ;
    while ( true ) {      /* Loop to do insertion */
        if ( currentP == NULL || currentP -> value > V ) {
            /* Insert here */
            trailP-> next = newPtr ;
            newPtr-> next = currentP ;
            break ;
        }
        trailP = currentP ;      /* Advance down list */
        currentP = trailP -> next ;
    }
    P = dummyHead. next ;
}
```


Singly Linked Lists with Header Blocks

- A singly linked list that is accessed via a header block that contains pointers to the first and last node in the list.



- Typical header block declaration

```
struct headerBlock {
    ListPtr head ;
    ListPtr tail ;
} ;
typedef struct headerBlock * headerPtr ;
```

- This added data structure makes access to the end of a list very efficient. **The header block must be updated whenever the first or last node in the list is changed.**

Append to Integer List with Header Block

```
/* Add value K to end of list P */
ListPtr P, Q, R ;
struct headerBlock HDR ;
/* Assume P gets value here */
/* Assume HDR points at P */
R = (ListPtr) malloc( LISTNODESIZE );
assert( R );
R-> value = K ;
R-> next = NULL ;
if ( HDR. head == NULL )  /* list was empty */
    HDR. head = HDR. tail = R ;
else {
    HDR. tail -> next = R ;
    HDR. tail = R ;
}
```

Doubly Linked Lists

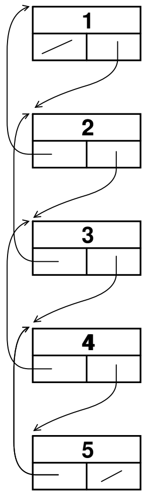
- Forward and backward pointers allow two way traversal
- Often useful if many insert, deletes or appends.
- Typical declaration:

```
struct dNode {
    int value ;
    struct dNode *forward, *back ;
} ;
typedef struct dNode * dPtr ;
```

- **Technique:** Use doubly linked lists when list must be traversed in both directions or when the end of the list must be quickly located.
  Don't use a doubly linked list when a singly linked list will do

Example - Create Double List

```
/* Create example list */
dPtr P , Q ;
int J ;
Q = NULL ;
for ( J = 1 ; J <= 5 ; j+ + ) {
    P = (dPtr) malloc(sizeof(struct dNode)) ;
    assert ( P ) ;
    P−> value = J ;
    if ( Q )
        Q−> forward = P ;
    P−> back = Q ;
    Q = P ;
}
p-¿forward = NULL ;
```



HOW TO Choose List Data Structures

- KISS
- For many cases a singly linked list is the best choice.
  Unless you know that lists will get long, searching a singly linked list from the head to find a node is a GOOD option compared to other alternatives.
- For cases where there will be a lot of appending to the end of a list, a singly linked list with a header block is a good solution.
- Use doubly linked lists **only** if you had an unavoidable need to traverse the list in both directions.
- Most programmers use a list structure that is *too complicated*. Use the **simplest** list structure that will do the job.
- KISS

Trees and Graphs

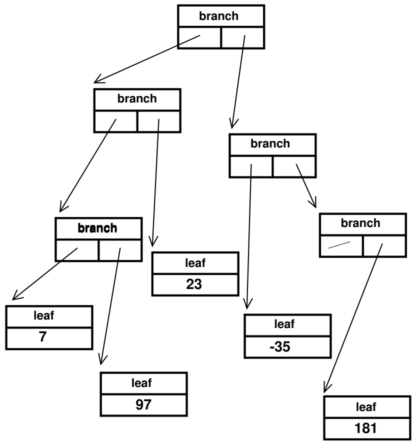
- A tree is a collection of nodes that are linked to a common root node. A graph is a collection of nodes that are linked together is some *arbitrary* way.
- Trees and graphs are used to represent data that has some *structure* that must be preserved during processing.
- The most common tree is the *binary* tree which has (at most) two branches from each node. *N-ary* trees are possible.
- *Recursive functions or procedures* are usually the best way to process trees and graphs.

Trees using Pointers

```
enum treeNodeType { branch , leaf } ;
struct treeNode {
    enum treeNodeType nodeKind ;
    int value ;
    struct  treeNode *left, *right ;
};
typedef struct treeNode * Tree ;
const Tree stump = NULL ;
Tree maple, elm ;
```


Example - Generic Tree Traversal

```
void trace ( Tree treep ) {
    if ( treep == stump )
        return ;
    if ( treep -> nodeKind
        == leaf ) {
         process leaf here */
    } else {
         process branches */
        ..
        trace( treep -> left ) ;
        ..
        trace( treep -> right ) ;
        ..
    }
}
```



Example - Tree Copy

```
Tree TreeCopy ( Tree treePtr ) {
    if ( treePtr == stump )
        return stump ;
    else {
        Tree twig ;
        twig = ( Tree ) malloc( sizeof( struct treeNode ) ) ;
        assert( twig ) ;
        twig-> value = treePtr-> value ;      /* copy node values */
        twig-> left = TreeCopy( treePtr-> left ) ;
        twig-> right = TreeCopy( treePtr-> right ) ;
        return twig ;
    }
}
```


Expression Trees

- Binary trees are an extremely convenient way to represent expressions in a program. The nodes in the tree represent operators and operands in the expression.
- Using trees makes it easy to describe the precedence of operators in the expression. Getting operator precedence right is esssential for any algorithm that processes expressions.
- Most expression algorithms are easy to design and implement if they work on expression trees as the primary data structure.

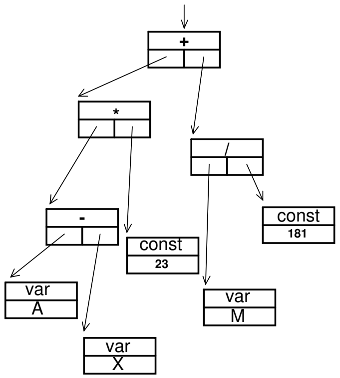
Example - Expression Tree Data Structure

- Randomly choosen example of an expression tree data structure

```
typedef enum { constant, variable, plus, minus, multiply, divide, power }
        treeNodeType ;
typedef struct treeNode * treePtr ;
typedef struct treeNode {
    treeNodeType nodeType ;     /* type of this node */
    unsigned constValue ;       /* value for constants */
    char * varName ;            /* Name of variable */
    treePtr leftOp ;            /* left operand for operator */
    treePtr rightOp ;           /* right operand for operator */
} treeNode ;
#define TREENODESIZE ( sizeof ( treeNode ) )
```


Example - Expression Tree Traversal

```
void trace ( Tree treePtr) {
    if ( treePtr == stump )
        return ;
    else {
        /* Preorder: process node here */

        trace( treePtr-> left )

        /* Inorder: process node here */

        trace( treePtr-> right )

        /* Postorder: process node here */
    }
}
```



Preorder: + * - A X 23 / M 181

Inorder: (A - X) * 23 + M / 181

Postorder: A X - 23 * M 181 / +

Example - Evaluate Constant Expression Tree

```
int evalTree( treePtr inTree ) {
    int leftValue, rightValue ;
    if ( inTree == NULL )
        return 0 ;
    if ( inTree -> nodeType == constant )
        return inTree -> constValue ;
    /* evaluate operands of operator */
    leftValue = evalTree( inTree -> leftOp ) ;
    rightValue = evalTree( inTree -> rightOp ) ;
    switch ( inTree -> nodeType ) {
        case plus :     return   leftValue + rightValue ;
        case minus :    return   leftValue - rightValue ;
        case multiply : return   leftValue * rightValue ;
        case divide :   return   leftValue / rightValue ;
        case power:     return   ( int ) pow( leftValue , rightValue ) ;
        default : assert ( false ) ;
    }
}
```