Use of stack for temporary calculations
Pointers in C
- **Linked List**
  - **FIFO**
- **Linked structures**
  - **FSM**
  - **Trees**

```
short n;  // value -32768 to +32767
short m;  // value -32768 to +32767
short *p; // address 0x0000 to 0xFFFF

char c;   // value -128 to +127
char d;   // value -128 to +127
char *s;  // address 0x0000 to 0xFFFF
char name[8] = "valvano";
```

Pointer assignments

```
   p = &n;       // p points to n
   s = &c;       // s points to c
```

Pointer dereferencing

```
   *p = 5000;    // n = 5000
   *s = 60;      // c = 60
   m = *p;       // m = n (which is 5000)
   d = *s;       // d = c (which is 60)
```

More pointer assignments

```
      s = name;     // s points to name
or    s = &name[0]; // s points to name
```

Fixed offset pointer dereferencing

```
   c = *s;        // c = 'V'
   d = s[1];      // d = 'a'
```

Static Linked list circular output pattern

Structure defines the format of each entry

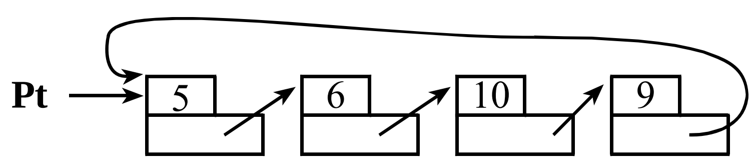
Putting the **`const`** causes it to be stored in ROM
- lots of ROM
- fixed values
- initialized when code burned into ROM

No **`const`** causes it to be stored in RAM
- Just some RAM
- variable values/pointers
- initialized at run time each time system is powered up
  - must have an initialization copy in ROM

```
const struct node {
  unsigned char data;   // output value
  const struct State *next; // links
};
typedef const struct node nodeType;
nodeType *Pt;
```

Linked list definition

```
nodeType LL[4]={
  {5, &LL[1]},
  {6, &LL[2]},
  {10,&LL[3]},
  {9, &LL[0]}};
```



Pointer initialization

```
      Pt = LL;
or    Pt = &LL[0];
```

Output all four values to port T

```
void OutputAll(void){
nodeType *p;
  p = Pt;
  do{
    PTT = p->data; // fetch value from list
    p = p->next;
  while(p != Pt);
}
```

Output one value to port T each interrupt

Pointer initialization

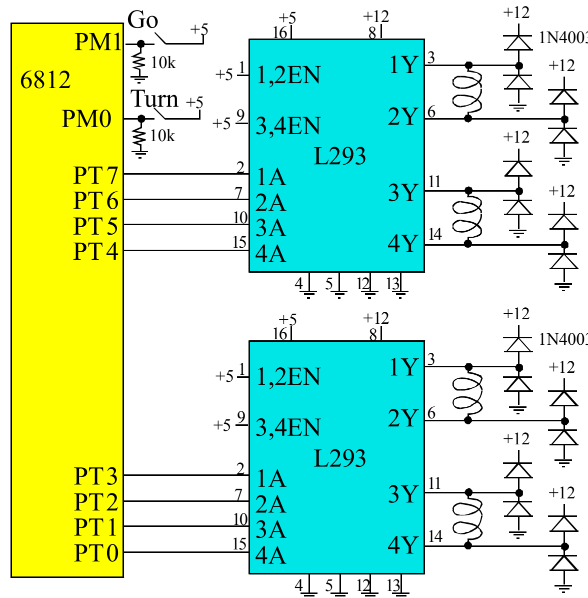
```
  Pt = LL;
```

Execute ISR every 1 ms

```
void interrupt 8 OC0ISR(void){
  PTT = Pt->data; // fetch value from list
  Pt = Pt->next;
  TC0 = TC0 + 1000;  // 1000 means 1ms
  TFLG1 = 0x01;      // acknowledge
}
```

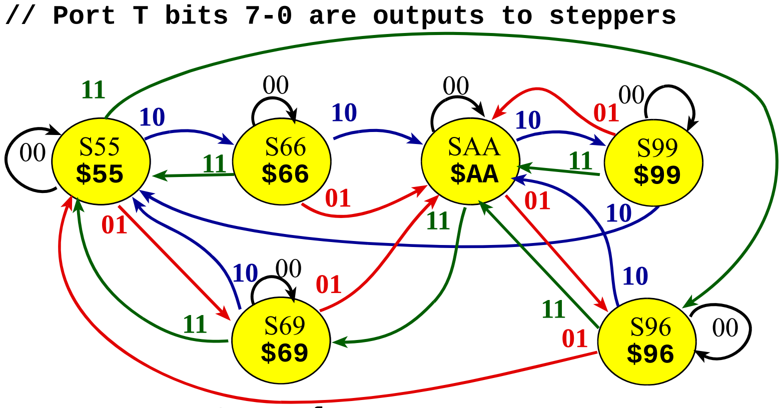
Stepper motor controller

Inputs: Go and Turn
Outputs: two 4-wire bipolar stepper motors



Bipolar stepper motor interface using an L293 driver

```
// Port M bits 1-0 are inputs
// =00 Stop
// =10 Go    (55,66,AA,99)
// =01 RTurn(55,69,AA,96)
// =11 LTurn(55,96,AA,69)
// Port T bits 7-0 are outputs to steppers
```




```
const struct State {
  unsigned char out;        // command
  const struct State *next[4];};
typedef const struct State StateType;
StateType *Pt;
#define S55 &fsm[0]
#define S66 &fsm[1]
#define SAA &fsm[2]
#define S99 &fsm[3]
#define S69 &fsm[4]
#define S96 &fsm[5]
```

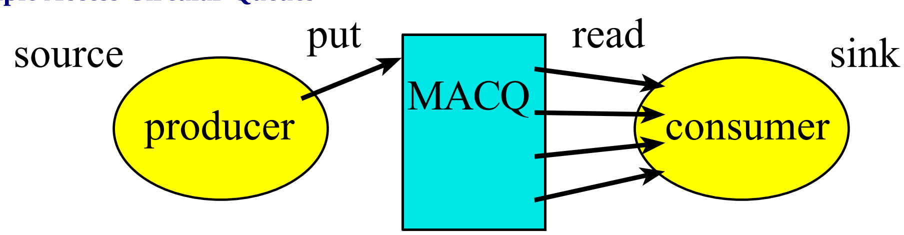
```
StateType fsm[6]={
  {0x55,{S55,S69,S66,S96}},  // S55
  {0x66,{S66,SAA,SAA,S55}},  // S66
  {0xAA,{SAA,S99,S99,S69}},  // SAA
  {0x99,{S99,SAA,S55,SAA}},  // S99
  {0x69,{S69,SAA,S55,S55}},  // S69
  {0x96,{S96,S55,SAA,SAA}}}; // S96
```

This stepper motor FSM has two input signals four outputs.

```
void main(void){
unsigned char Input;
  Timer_Init();
  DDRT = 0x0ff;
  DDRM = 0;
  Pt = S55;     // initial state
  while(1){     // never quit
    PTT = Pt->out;     // stepper drivers
    Timer_Wait(2000); // 0.25ms wait
    Input = PTM&0x03;
    Pt = Pt->next[Input];
  }
}
```

Rewrite this to run in background

10.7. Multiple Access Circular Queues



used for data flow problems source to sink
digital filters and digital controllers
fixed length
order preserving
MACQ is always full

source process (producer)

places information into the MACQ
oldest data is discarded when new data is entered

sink process (consumer)

can read any data
MACQ is not changed by the read operation.

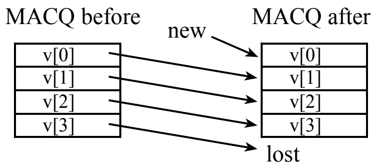
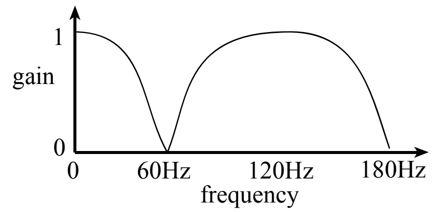


Figure 10.7. A multiple access circular queue stores the most recent set of measurements.

Perform a 60Hz notch filter on a measured signal.

v[0] v[1] v[2] and v[3] are the most recent data
sampled at 360 Hz.



filtered output = $\frac{v[0]+v[3]}{2}$

```
unsigned char v[4];
unsigned char samp(void){
  v[3] = v[2];
  v[2] = v[1];
  v[1] = v[0];

  v[0] = Ad_In(2);


  return (v[0]+v[3])/2;
}
```

```
     org $0800
v    rmb  4
     org $F000
samp movb v+2,v+3
     movb v+1,v+2
     movb v,v+1
     ldaa #2
     jsr  AD_In
     staa v
     adda v+3 9-bit
     rora     (v[0]+v[3])/2
     rts
```

10.9. Trees

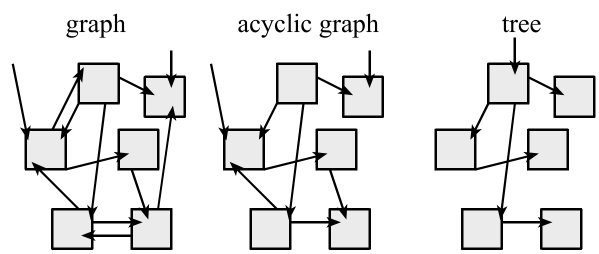


Figure 10.11. Graphs and trees have nodes and are linked with pointers.

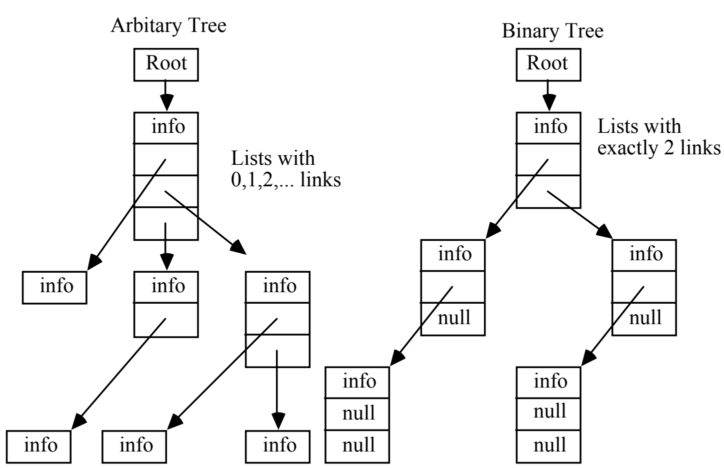


Figure 10.12. A tree can be constructed with only down arrows, and there is a unique path to each node.

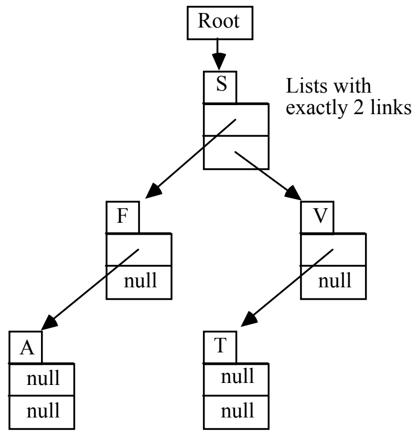


Figure 10.13. A binary tree is constructed so that earlier elements are to the left and later ones to the right.

```
Value equ  0  name of the node
Data  equ  1  data for this node
Left  equ  2  pointer to son
Right equ  4  pointer to son
ROOT  fdb  WS Pointer to top
NULL  equ  0  undefined address
WS    fcb 'S',1 name,data
      fdb  WF   Left son
      fdb  WV   Right son
WV    fcb 'V',2 name,data
      fdb  WT   WT is a left son
      fdb  NULL no right son
WT    fcb 'T',3 name,data
      fdb  NULL no children
      fdb  NULL no right son
WF    fcb 'F',4 name,data
      fdb  WA   WA is a left son
      fdb  NULL no right son
WA    fcb 'A',5 name,data
      fdb  NULL no children
      fdb  NULL
```

```
#define NULL 0
const struct Node{
  unsigned char Value;
  unsigned char Data;
  const struct Node *Left;
  const struct Node *Right;};
typedef const struct Node NodeType;
typedef NodeType * NodePtr;
#define Root WS
#define WS &Tree[0]
#define WV &Tree[1]
#define WT &Tree[2]
#define WF &Tree[3]
#define WA &Tree[4]
NodeType Tree[5]={
{ 'S',1, WF, WV},
{ 'V',2, WT, NULL},
{ 'T',3, NULL, NULL},
{ 'F',4, WA, NULL},
{ 'A',5, NULL, NULL}};
```

Program 10.20. Definition of a simple binary tree.

```
*Inputs:  Reg A = look up letter
*Outputs: Reg A=0 if not found,
*               =data if found
Look  ldx  Root      current word
loop  cpx  #NULL
      beq  fail
      cmpa Value,x   Match
      beq  found     Skip if found
      blo  golft
      ldx  Right,x   letter>value
      bra  loop
golft ldx  Left,x    letter<value
      bra  loop
fail  clra           not in tree
      bra  exit
found ldaa Data,x    return value
exit  rts
```

```
int Look(unsigned char letter){
  NodePtr pt = Root;  /* top */
  while(pt!=NULL){ // done if null
    if(pt->Value == letter){
      return(pt->Data); /* good */
    }
    if(pt->Value < letter){
      pt = pt->Right;
    }
    else{
      pt = pt->Left;
    }
  }
  return NULL; /* not in tree */
}
```

Program 10.21. Binary tree search functions.

In order to add and remove nodes at run time
tree must be defined in RAM.
first search for the word (the search should fail),
change the null pointer to point to the new list.

```
* Inputs : Reg Y => new word to be added
*     new word is already in memory formatted
*     fcb  'J',6
*     fdb  NULL
*     fdb  NULL
NEW   ldaa 0,Y     Reg A is the name of the new word
      bsr  LOOK
      tsta
      bne  ok      skip if already defined
      sty  0,X     Update link
OK    rts
```

Program 10.22. Program to add a node to a binary tree.

Figure 10.14 shows the binary tree as the nodes J, U, G are added to the dictionary.

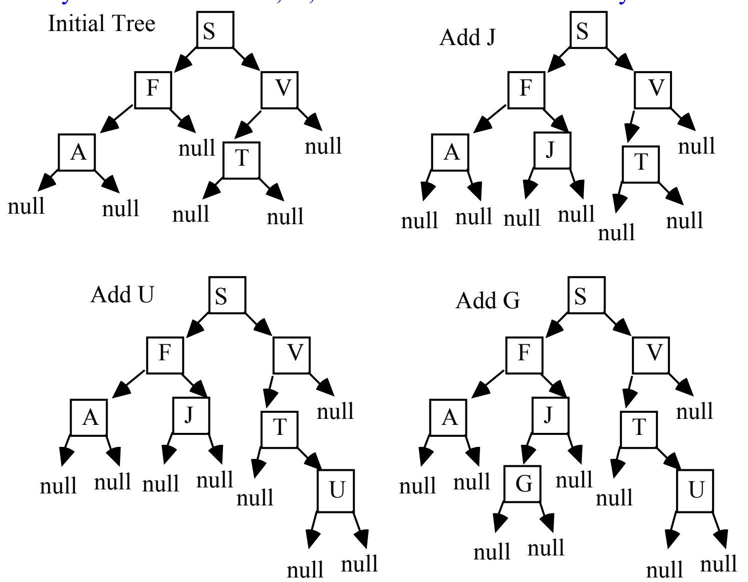


Figure 10.14. Nodes are added to a binary tree such that the alphabetical order is maintained.

The search time for a binary tree increases as the log2 of the size of the dictionary.

Expression evaluation

Polish notation is a prefix notation used in logic and arithmetic operations. The Polish logician Jan Łukasiewicz invented this notation around 1920 in order to simplify sentential logic. The following expression:

* 2 3

evaluates to 6. This more complex expression:

* + 1 2 + 3 4

can sometimes be written as

(* (+ 1 2) (+ 3 4))

and evaluates to 21. Lisp s-expressions employ Polish notation.

Reverse Polish notation (RPN) is a postfix notation, invented by Australian philosopher and computer scientist Charles Hamblin in the mid-1950s. Edsger Dijkstra invented the "shunting yard" algorithm, which converts from infix notation to RPN.

Reverse Polish Notation

- numbers are pushed on the stack,
- values of the variables are pushed on the stack,
- unary function: input popped and result pushed,
- binary function: both inputs popped and result pushed.

Regular expression	Reverse Polish Notation
3*M+N	3 M * N +
~(M|(N&P))	N P & M | ~
M*(5+P)-N/10	M 5 P + * N 10 / -
w-x+y+z-4	w x – y + z + 4 -

Table 8.4. Examples of Reverse Polish Notation.

P=(M+2)*(5+P)+3*N M 2 + 5 P + * 3 N * +

```
     org  $0800
dataStack rmb  10
P    rmb  1
M    rmb  1
N    rmb  1
     org  $4000
calc ldy  #dataStack+10
     movb M,1,-y
     movb #2,1,-y
     jsr  Add
     movb #5,1,-y
     movb P,1,-y
     jsr  Add
     jsr  Mult
     movb #3,1,-y
     movb N,1,-y
     jsr  Mult
     jsr  Add
     movb 1,y+,P
     rts
Add  ldaa 1,y+
     adda 1,y+
     staa 1,-y
     rts
Mult ldaa 1,y+
     ldab 1,y+
     mul
     stab 1,-y
     rts
```

```
     org  $0800
P    rmb  1
M    rmb  1
N    rmb  1
     org  $4000
calc

     ldaa M
     adda #2

     ldab #5
     addb P

     mul
     pshb
     ldaa #3
     mul
     addb 1,sp+
     stab P
     rts
```