2110206 Assembly language programming

Data structure

P. Chongstitvatana
Department of Computer Engineering
Chulalongkorn University
February 2007

- list
- tree
- binary tree
- trie

List

contain unordered elements

```
(B A C)  (3 5 4)
```

each element can be another list

```
((B A) C)  ((4 6) 3)
```

Why?

List stores a "collection" of items. You can "enumerate" items in a list. So, an item can be "added" to a list, an item can be "searched" whether it is a member of the list. Here is some example of use, a list of name in a telephone-book, a shopping list.

```
((prabhas 2186982) (somchai 2186981) (som 2222))
( meat milk bread jam rice banana)
```

List and array

List is different from an array that an array can only store elements of the same kind. However, an array can be accessed directly, every item takes the same amount of time to be accessed, but a list is accessed sequentially, the last element takes the longest time to be accessed (however, there is some implementation method to improve this).

List is also flexible in that it can grow (an array can not grow, its maximum size is fixed at the time the array is created, also, there is a way to overcome this).

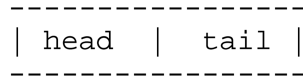
Representation

How a list is stored in a computer?

Our "universal" list is built from a two-slot "cell". Both slots contain links to another cell (a link is an address of a cell). The first slot is called "head", the second slot is called "tail". A normal cell is called a "dot-pair". Data is also stored in a special kind of cell called "atom".

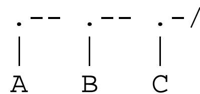
The first slot of an atom stored the type of data; the second slot stored a value. To distinguish between a dot-pair and an atom, the value in the first slot of a dot-pair is greater than 10. The type of an atom is 0..9, so there is no confusion whether a cell is a dot-pair or an atom, just look at the value of the first slot. (we assume that a pointer to a cell will have a value greater than 10, usually it is an address therefore it is not a small number).

An atom can be many types. For example, let 0 be the type integer, the second slot of an integer atom stored its value. Let 1 be the type string, the second slot stored a pointer to that string.

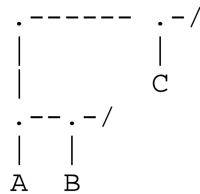


A special value, NIL, in the tail slot indicates the end of a list. To simplify the drawing I will use a diagram like this:

```
an atom        A, 1

a dot-pair     .--tail
               |
               head

a NIL          -/

(A B C)
```

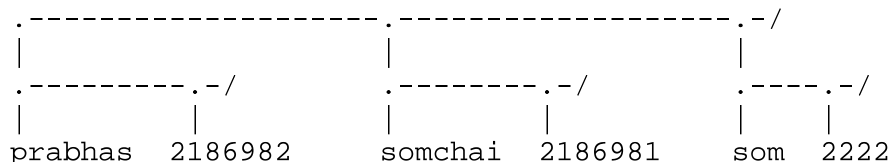



```
((A B) C)
```



The beginning of a list will be positioned on the top left of the diagram.

```
((prabhas 2186982) (somchai 2186981) (som 2222))
```



List operations

To "enumerate" a list, we need operators that "take apart" a list. The function "head" gets the head slot, the function "tail" gets the tail slot. To determine the type of a cell, the function

"isatom" checks whether the cell is an atom. Here is how we "print" a list on a screen, assuming a "print" function that prints a data properly.

<head, tail, isatom>

```
printList(e)
  if e == NIL stop
  if isatom(e)  print e
  else
    printList head(e)
    printList tail(e)
```

Please notice that the operation on list is recursively defined by nature.

<cons, atom>

To build a list, we need a function "cons" (constructor) that takes two items and connects them using a dot-pair. Here is a definition:

```
if e is a list.  cons(head(e),tail(e))  is  e.
```

To make a data into an atom, we use a function "atom". Now we can build a list,

```
cons(atom(A),cons(atom(B)),cons(atom(C),NIL))) is (A B C)
cons(cons(atom(A),cons(atom(B),NIL)),atom(C))  is ((A B) C)
```

assuming A, B, C are data of the proper kind. Here is how to make a copy of a list. Remember that copy() is a function, it returns a value, that is, a list.

```
copy e
  if e == NIL return NIL
  if isatom(e) return atom(e)
  else return cons(copy(head(e)),copy(tail(e)))
```

Other simple function is to count the length (the number of elements) of a list. The first definition is for a simple list (we do not count into the element which is a list.

```
count e
  if e == NIL return 0
  else return 1 + count(tail(e))

a = cons(atom(A),cons(atom(B)),cons(atom(C)),NIL)))
(a is (A B C))
```

count(a) will return 3. Now to get "into" the element which is a list, we can recursively enumerate it.

```
count2 e
  if e == NIL return 0
  if isatom(head(e)) return 1 + count2(tail(e))
  else return count2(head(e)) + count2(tail(e))
```

let b is ((A B) C), count(b) will return 2, count2(b) will return 3

Assembly language

A concrete representation of list in our S2 machine is defined as follows. A cell has two words (we shall align a cell on an even address). For sake of demonstration, a data will be a 24-bit integer (so that it can reside in a head slot easily).

There are five functions: head, tail, isatom, atom, cons. Assume input parameters are in r1 and r2 (for cons).

```
;; register convention
;; r31  system stack
;; r30  display
;; r29  link register
;; r28  return value

.symbol
  NIL -1

.code 200
:head
  ld r28 @0 r1
  ret r29

:tail
  ld r28 @1 r1
  ret r29

:isatom
  ld r28 @0 r1
  lt r28 r28 #10 ;; value < 10 is atom
  ret r29

:atom            ;; input: data in r1
  trap 5         ;; get a cell
  st @0 r28 r0   ;; set head slot to type int
  st @1 r28 r1   ;; set tail to be data
  ret r29

:cons            ;; cons r1 r2
  trap 5         ;; get a cell
  st @0 r28 r1
  st @1 r28 r2
  ret r29
```

```
;; count the number of element in a list

:count            ;; input in r1, use r3
  push r31 r29    ;; save link, recursive call
  push r31 r3
  eq r3 r1 #NIL
  jf r3 count1
  mv r28 #0       ;; return 0
  jmp count_ret
:count1
  jal r29 tail    ;; call tail(e)
  mv r1 r28
  jal r29 count    ;; call count(tail(e))
  add r28 r28 #1   ;; return 1 + count(tail(e))
:count_ret
  pop r31 r3
  pop r31 r29
  ret r29
```

Now we try the program.

```
.code 0
:main
  jal r29 make
  mv r1 r28       ;; r1 = (1 2 3)
  jal r29 count   ;; count it
  mv r30 r28
  trap 6          ;; print list
  mv r30 r28
  trap 1          ;; print result
  trap 0

;; make a list of (1 2 3), use r1 r2, return r28
:make
  push r31 r29
  mv r1 #3
  jal r29 atom
  mv r1 r28
  mv r2 #NIL
  jal r29 cons    ;; cons(atom(3),NIL)
  mv r2 r28       ;; r2 = (3)
  mv r1 #2
  jal r29 atom
  mv r1 r28
  jal r29 cons    ;; cons(atom(2),r2)
  mv r2 r28       ;; r2 = (2 3)
  mv r1 #1
  jal r29 atom
  mv r1 r28
  jal r29 cons    ;; cons(atom(1),r2)
  pop r31 r29
  ret r29
.end
```

Binary tree

Our universal list can be used to build a tree. A list has unordered elements. Some operation needs to enumerate the list, hence it will take an average time around n/2 where n is the number of elements in the list. For example, to find x in a list e, we do

```
member(x,e)
  if e == NIL return NO
  if x == head(e) return YES
  return member(x,tail(e))
```

To improve the efficiency of access, we can arrange a "structure" of elements. A tree is a kind of structure, let we make a three-element node where one node stored the element, the other two nodes point to a left tree and a right tree, where all elements in the left tree is smaller than x, and all elements in the right tree is larger than x. Then, to find x in a tree t, we do

```
member(x,t)
  if t == NIL return NO
  if x == head(t) return YES
  if x < head(t) return member(x,left(t))
  return member(x,right(t))
```

We can build a three-element node from our list as follows.

```
.---.-----.-/
|   |     |
x   left  right
```

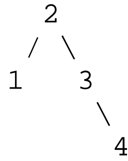
In a "printed" version, it is, (x left right). So, the access function; left, right are:

```
left(t)
  head(tail(t))

right(t)
  head(tail(tail(t)))
```

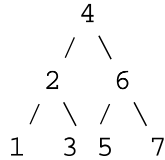
For example, if we have {1, 2, 3, 4} and take 2 as the root node, the tree will look like, `(2 (1 () ()) (3 () (4 () ())))` where `()` is a NIL.

```
.---.-----------.-/
|   |           |
2   .--.--.-/   .--.--.-/
    |  /  /     |  /  |
    1           3     .--.--.-/
                      |  /  /
                      4
```

This tree is called "binary tree", as you can see it like this.



You can imagine a "complete" binary where all nodes are occupied, like this.



The way we test a membership of a binary tree is called, binary search. On average, it took the time log(n) to search a tree of n elements.

Assembly language

We use the five functions: head, tail, isatom, atom, cons previously defined to construct the binary search.

```
;; ---- bsearch ------

;; let r1 = input

:left
  push r31 r29
  jal r29 tail
  mv r1 r28
  jal r29 head
  pop r31 r29
  ret r29

:right
  push r31 r29
  jal r29 tail
  mv r1 r28
  jal r29 tail
  mv r1 r28
  jal r29 head
  pop r31 r29
  ret r29

;; get data from head(t)
:value
  push r31 r29
  jal r29 head
  mv r1 r28
  jal r29 tail
  pop r31 r29
  ret r29
```

```
;;  member(x,t)
;;   if t == NIL return NO
;;   if x == head(t) return YES
;;   if x < head(t) return member(x,left(t))
;;   return member(x,right(t))

;; let r1 = x, r2 = list, r3 = test, r4 = temp
:member
  push r31 r29
  push r31 r1
  push r31 r2
  push r31 r3
  push r31 r4
  eq r3 r2 #NIL
  jf r3 mem2
  mv r28 #NO
  jmp mem_ret
:mem2
  mv r4 r1        ;; save x in r4
  mv r1 r2        ;; pass t
  jal r29 value   ;; r28 = value(head(t))
  eq r3 r4 r28    ;; x == head(t)?
  jf r3 mem3
  mv r28 #YES
  jmp mem_ret
:mem3
  lt r3 r4 r28    ;; x < head(t)?
  jf r3 mem4
  mv r1 r2        ;; pass t
  jal r29 left
  mv r1 r4        ;; pass x
  mv r2 r28       ;; pass left(t)
  jal r29 member
  jmp mem_ret
:mem4
  mv r1 r2        ;; pass t
  jal r29 right
  mv r1 r4        ;; pass x
  mv r2 r28       ;; pass right(t)
  jal r29 member
:mem_ret
  pop r31 r4
  pop r31 r3
  pop r31 r2
  pop r31 r1
  pop r31 r29
  ret r29

;; Now we try it.  Make a list of
;; (2 (1 () ())(3 () (4 () ()))
;; and call member(3,t).

.code 0
:main
```

```
  mv r31 #1000     ;; system stack at 1000
  jal r29 make_t
  mv r30 r28
  trap 6           ;; display t
  mv r1 #3
  mv r2 r28
  jal r29 member   ;; member(3,t)
  mv r30 r28
  trap 1           ;; display result
  trap 0

;; input r1 = data, r2 = left, r3 = right
;; r4 =temp, r5 = temp
;; return (data left right)
:make_node
  push r31 r29
  push r31 r1
  push r31 r2
  push r31 r3
  push r31 r4
  push r31 r5
  jal r29 atom
  mv r4 r28     ;; save atom(data)
  mv r5 r2      ;; save left
  mv r1 r3
  mv r2 #NIL
  jal r29 cons
  mv r1 r5
  mv r2 r28     ;; r28 = (right)
  jal r29 cons ;; r28 = (left right)
  mv r1 r4      ;; atom(data)
  mv r2 r28
  jal r29 cons ;; r28 = (data left right)
  pop r31 r5
  pop r31 r4
  pop r31 r3
  pop r31 r2
  pop r31 r1
  pop r31 r29
  ret r29

;; Make a list of (2 (1 () ()) (3 () (4 () ())))
;; use r1, r2, r3, r4
:make_t
  push r31 r29
  mv r1 #4
  mv r2 #NIL
  mv r3 #NIL
  jal r29 make_node
  mv r1 #3
  mv r2 #NIL
  mv r3 r28
  jal r29 make_node
  mv r4 r28     ;; save (3 () (4 () ()))
  mv r1 #1
```

```
  mv r2 #NIL
  mv r3 #NIL
  jal r29 make_node
  mv r1 #2
  mv r2 r28   ;; r2 = (1 () ())
  mv r3 r4
  jal r29 make_node
  pop r31 r29
  ret r29

.end
```

Trie

A symbol table can be realised as a Trie implemented by linked list. Each node contains a character and two links: choice link points to next alternative character, next link points to next character. There is also one attribute field in each node to store useful information.

a node is : char, choice, next, value

A Trie is built dynamically; it starts from an empty node. "search" scans input character by character (until end of word, terminated by 0). Each character is matched with Trie. First, it moves right (by next link) one step for the next character, if it is at the end of next link, the input character is inserted. Then, check each choice (choice link) until found matched. If no match, the input character is inserted at the end. "search" returns index to Trie node (the last node that matched). The "search" is defined to do "search and insert" and it works beautifully to compact the symbol table search into two while loops.

```
scan(i,c)          scan symbol table at i for character c
  if sym[i] == 0
    sym[i] = newcell(c)  insert a new cell with char c
  return sym[i]

search
  i = 0
  c = getc()
  while c != 0
    i = scan i+2 c     move next
    while sym[i] != c
      i = scan i+1 c   move right (choice)
    c = getc()
  return i             index to last matched node
```

Example
The input words are: prabhas, peter, prapat, mon, mama, monkol.

```
Trie looks like this      .--choice
                          |
                          next
```

where x marks the value associated with the symbol.

```
p-r-a-b-h-a-s
| |   |     x
| |   p-a-t
| |       x
| e-t-e-r
|       x
|
m-o-n-k-o-l
  | x     x
  |
  a-m-a
      x
```

Assembly language

sym[] is an array. newcell() allocates a node of four words. A node is defined as follows:

```
--------------------------------
| char | choice | next | value |
--------------------------------
```

```
.symbol
  SYM  10000     ;; symbol table at 10000
  freecell 500   ;; global: freecell
  wp   501       ;; global: wp, point to WORD
  WORD 600       ;; input word

.code 200
;; allocate a new node, set char to c
;; input r1 = c,  no check for overflow!
:newnode
  push r31 r29
  push r31 r1
  ld r28 freecell
  st @SYM r28 r1       ;; store char
  add r1 r28 #4
  st freecell r1
  pop r31 r1
  pop r31 r29
  ret r29

;; scan(i,c)
;;    if sym[i] == 0
;;      sym[i] = newnode(c)
;;    return sym[i]

;; check if empty then insert c
;; r1 = i, r2 = c, r3 = test, r4 = temp
;; return index to current cell
:scan
  push r31 r29
  push r31 r1
  push r31 r2
  push r31 r3
```

```
  push r31 r4
  ld r28 @SYM r1
  eq r3 r28 #0
  jf r3 scan_ret
  mv r4 r1        ;; save i to r4
  mv r1 r2
  jal r29 newnode
  st @SYM r4 r28
:scan_ret
  pop r31 r4
  pop r31 r3
  pop r31 r2
  pop r31 r1
  pop r31 r29
  ret r29

;;  assume input word is at WORD, pointed to by wp
;;  return a char in r28
:getc
  push r31 r29
  push r31 r1
  ld r1 wp
  ld r28 @0 r1
  add r1 r1 #1
  st wp r1
  pop r31 r1
  pop r31 r29
  ret r29

;; search
;;   i = 0
;;   c = getc()
;;   while c != 0
;;     i = scan i+2 c     move next
;;     while sym[i] != c
;;       i = scan i+1 c   move right (choice)
;;     c = getc()
;;   return i             index to last matched node

;; let r5 = i, r6 = c, r3 = test, r4 = temp
:search
  push r31 r29
  push r31 r1
  push r31 r2
  push r31 r3
  push r31 r4
  push r31 r5
  push r31 r6
  mv r5 #0
  jal r29 getc
  mv r6 r28         ;; c = getc()
:while2
  ne r3 r6 #0       ;; while c != 0
  jf r3 search_ret
  add r1 r5 #2
```

```
  mv r2 r6
  jal r29 scan
  mv r5 r28         ;; i = scan i+2 c
:while1
  ld r4 @SYM r5
  ne r3 r4 r6       ;; while sym[i] != c
  jf r3 ewhile1
  add r1 r5 #1
  mv r2 r6
  jal r29 scan
  mv r5 r28         ;; i = scan i+1 c
  jmp while1
:ewhile1
  jal r29 getc
  mv r6 r28         ;; c = getc()
  jmp while2
:search_ret
  mv r28 r5
  pop r31 r6
  pop r31 r5
  pop r31 r4
  pop r31 r3
  pop r31 r2
  pop r31 r1
  pop r31 r29
  ret r29

;; r1 = i, r2 = val
:setvalue
  push r31 r1
  add r1 r1 #3
  st @SYM r1 r2
  pop r31 r1
  ret r29

;; r1 = i
:getvalue
  push r31 r1
  add r1 r1 #3
  ld r28 @SYM r1
  pop r31 r1
  ret r29

;; now try it.

.code 0
:main
  mv r31 #1000      ;; system stack at 1000
  mv r1  #WORD
  st wp r1          ;; set wp, point to WORD
  mv r1 #4
  st freecell r1    ;; freecell = 4
  jal r29 search
  mv r1 r28
  mv r2 #11
```

```
  jal r29 setvalue ;; insert 123, val 11
  jal r29 search
  mv r1 r28
  mv r2 #22
  jal r29 setvalue ;; insert 1356, val 22
  jal r29 search
  mv r1 r28
  jal r29 getvalue ;; search 123
  mv r30 r28       ;; print val
  trap 1
  trap 0

.data 600
  1 2 3 0
  1 3 5 6 0
  1 2 3 0
.end
```

End of lecture data structure