Data Structures

Ray Seyfarth

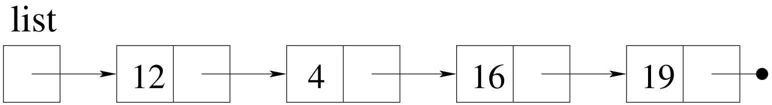
August 6, 2011

Data structures

- Data structures can implement an ordering to data
  - A stack where the items are ordered by time of insertion and the newest item is removed first
  - A queue where the items are ordered by time of insertion and the oldest item is removed first
  - A priority queue where items are ordered by priority
  - A binary tree where items are kept in order based on a key
- Some data structures implement a "dictionary"
  - Each item inserted has a "key", like a person's student id
  - Information is stored with the key
  - A hash table implements an efficient dictionary without maintaining an ordering of keys
  - A binary tree implements a dictionary keeping the keys in order

Outline

Linked lists



- A simple linked list is constructed of a sequence of structs
- Each struct has some data and a pointer to the next item on the list
- The filled circle means a pointer equal to NULL (0)
- There needs to be some memory cell containing the first pointer
- This list has no obvious order to the keys
- It could be ordered by insertion time in two ways: by inserting at the front or the end
- It is easier to insert at the front, though the value of `list` will change with each insertion

List node struct definition

```
        struc   node
n_value resq    1
n_next  resq    1
        align   8
        endstruc
```

- Using "`align 8`" insures that the size is a multiple of 8 bytes
- This is not needed here since, both `node` items are quad words
- It's "defensive programming" to insert it now in case the definition changes

Creating an empty list

- The only requirement will be to set the pointer to NULL
- Having a function makes it possible to change later with less impact on the rest of the program

```
newlist:
        xor     eax, eax
        ret
        ...
        call    newlist
        mov     [list], rax
```

Inserting a number into a list

- A new `node` will be allocated and placed at the start
- We must pass the list pointer into the function
- We also must receive a new pointer back to store in `list`
- In C we would use

```
list = insert ( list, k );
```

- In assembly we would insert k using

```
mov     rdi, [list] ; pass in the list pointer
mov     rsi, [k]
call    insert
mov     [list], rax ; we have a new list pointer
```

Insert code

```
insert:
.list   equ     0
.k      equ     8
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     [rsp+.list], rdi   ; save list pointer
        mov     [rsp+.k], rsi      ; and k on stack
        mov     edi, node_size
        call    malloc             ; rax will be node pointer
        mov     r8, [rsp+.list]    ; get list pointer
        mov     [rax+n_next], r8   ; save pointer in node
        mov     r9, [rsp+.k]       ; get k
        mov     [rax+n_value], r9  ; save k in node
        leave
        ret
```

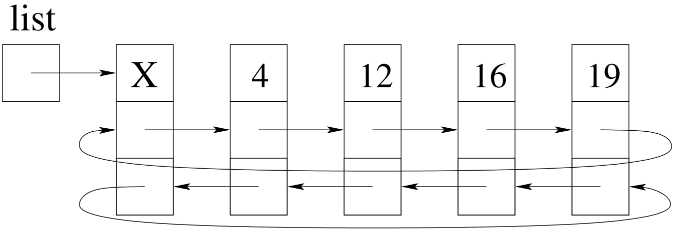
Traversing the list

```
print:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16           ; subtract multiples of 16
        mov     [rsp], rbx        ; save old value of rbx
        cmp     rdi, 0
        je      .done
        mov     rbx, rdi
.more   lea     rdi, [.print_fmt]
        mov     rsi, [rbx+n_value]
        xor     eax, eax
        call    printf
        mov     rbx, [rbx+n_next]
        cmp     rbx, 0
        jne     .more
.done   lea     rdi, [.newline]
        xor     eax, eax
        call    printf
        mov     rbx, [rsp]        ; restore rbx
        leave
```

Main program to build a list

```
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        call    newlist
        mov     [rsp+.list], rax  ; .list equal to 0, not shown
.more   lea     rdi, [.scanf_fmt] ; .scanf_fmt not shown
        lea     rsi, [rsp+.k]     ; .k equal to 8, not shown
        xor     eax, eax          ; no floating point value parameters
        call    scanf
        cmp     rax, 1            ; quit it scanf does not return 1
        jne     .done
        mov     rdi, [rsp+.list]  ; Get the list pointer
        mov     rsi, [rsp+.k]     ; Get k
        call    insert
        mov     [rsp+.list], rax  ; Save new list pointer
        mov     rdi, rax          ; Move the pointer to be a parameter
        call    print
        jmp     .more             ; Try to read another number
.done   leave
```

Doubly linked lists

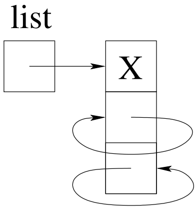


- This list uses forwards and backwards pointers to make a cycle
- Also the first node is not used, so an empty list will have one node and will be circular
- The first node is called a “head” node
- Using a head node and a circular list makes insertion trivial
- You can also insert and remove from either end easily

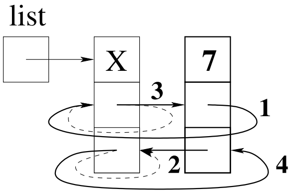
Doubly linked list node struct

```
        struc   node
n_value resq    1
n_next  resq    1
n_prev  resq    1
        align   8
        endstruc
```

- An “empty” list is still circular
- There are no special cases to consider



Inserting at the front of a doubly linked list



- The original links are dashed lines
- Make the new node point forward to the head cell's next
- Make the new node point backward to the head cell
- Make the head cell point forward to the new cell
- Make the new cell's next node point backward to the new cell

Insertion function

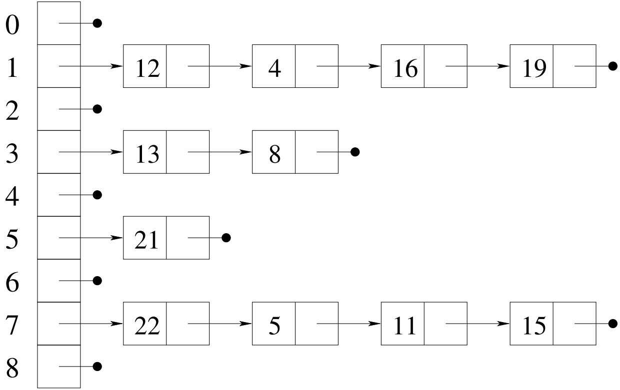
```
;       insert ( list, k );
insert: push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     [rsp+.list], rdi  ; save list pointer, .list equ 0
        mov     [rsp+.k], rsi     ; and k on stack, .k equ 8
        mov     edi, node_size
        call    malloc            ; rax will be node pointer
        mov     r8, [rsp+.list]   ; get list pointer
        mov     r9, [r8+n_next]   ; get head's next
        mov     [rax+n_next], r9  ; set new node's next
        mov     [rax+n_prev], r8  ; set new node's prev
        mov     [r8+n_next], rax  ; set head's next
        mov     [r9+n_prev], rax  ; set new node's next's prev
        mov     r9, [rsp+.k]      ; get k
        mov     [rax+n_value], r9 ; save k in node
        leave
        ret
```

List traversal

```
;       print ( list );
print:  push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     [rsp+.rbx], rbx    ; save rbx, .rbx equ 0
        mov     [rsp+.list], rdi   ; save list, .list equ 8
        mov     rbx, [rdi+n_next]  ; skip the nead node
        cmp     rbx, [rsp+.list]   ; is the list empty?
        je      .done
.more   lea     rdi, [.print_fmt]  ; .print_fmt not shown
        mov     rsi, [rbx+n_value]
        call    printf             ; print the node's value
        mov     rbx, [rbx+n_next]  ; advance to the next node
        cmp     rbx, [rsp+.list]   ; have we reached the head cell?
        jne     .more
.done   lea     rdi, [.newline]    ; .newline not shown
        call    printf
        mov     rbx, [rsp+.rbx]    ; restore rbx
        leave
        ret
```

Hash tables

- For each key, compute a hash value
- The hash value defines an index in an array to store the key
- Collisions occur when 2 different keys hash to the same index
- The simplest collision resolution is to use a linked list



A good hash function for integers

- A good hash function spreads the keys around
- Using $k \mod t$ where t is the table size is good
- It could be bad if the keys are related to the table size
- A good recommendation is to make t prime
- In this example, $t = 256$, so using `and` works

```
;       i = hash ( n );
hash    mov     rax, rdi
        and     rax, 0xff
        ret
```

A good hash function for strings

- The code below uses the characters of the string as coefficients of a polynomial
- The polynomial is evaluated at 191 (a prime)
- Then a mod is done with 100000 to get the hash value
- Assembly code is an exercise for the reader

```
int hash ( unsigned char *s )
{
    unsigned long h = 0;
    int i = 0;

    while ( s[i] ) {
        h = h*191 + s[i];
        i++;
    }
    return h % 100000;
}
```

Hash node structure and array of pointers

- The hash table has only 256 pointers
- Usually the array would be larger and a creation function needed

```
        segment .data
table   times 256 dq    0    ; All NULL pointers
        struc  node
n_value resq    1
n_next  resq    1            ; Singly linked list
        align   8
        endstruc
```

Function to find a key

```
;       p = find ( n );
;       p = 0 if not found
find:   push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     [rsp], rdi          ; save key
        call    hash
        mov     rax, [table+rax*8] ; get pointer
        mov     rdi, [rsp]          ; get key
        cmp     rax, 0              ; empty list?
        je      .done
.more   cmp     rdi, [rax+n_value] ; key match?
        je      .done
        mov     rax, [rax+n_next]  ; advance on the collision list
        cmp     rax, 0              ; end of list
        jne     .more
.done   leave
        ret
```

Function to insert a key

```
insert: push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     [rsp+.n], rdi      ; save n, .n equ 0
        call    find
        cmp     rax, 0             ; Is n already there?
        jne     .found
        mov     rdi, [rsp+.n]      ; compute hash(n)
        call    hash
        mov     [rsp+.h], rax      ; save hash value
        mov     rdi, node_size     ; allocate a node
        call    malloc
        mov     r9, [rsp+.h]       ; use r9 as index register
        mov     r8, [table+r9*8]   ; get old pointer from table
        mov     [rax+n_next], r8   ; make new node point to old
        mov     r8, [rsp+.n]       ; get n from the stack
        mov     [rax+n_value], r8  ; set the node value
        mov     [table+r9*8], rax  ; make new node first on its list
.found  leave
```

Testing the hash table

- Need to examine print function
- Need to examine main function
- Test the program

Binary trees

- A binary tree is a hierarchy of nodes
- There is a root node (or not, for an empty tree)
- Each node can have a left child and a right child
- The node structure has 2 pointers
- Either or both pointers could be NULL
- Binary trees are usually ordered like having all keys less the current key in the left subtree
- Such a tree is a "binary search tree"

```
        struc    node
n_value resq     1
n_left  resq     1
n_right resq     1
        align    8
        endstruc
```

A structure for the tree

- We could represent an empty tree as a NULL pointer
- This introduces special cases
- Instead we implement a `tree` struct
- It contains the root pointer which can be NULL
- It also contains the count of nodes in the tree
- After creating a tree, we use the same pointer for all function calls

```
        struc   tree
t_count resq    1
t_root  resq    1
        align   8
        endstruc
```

Creating a new tree

- The `new_tree` function allocates a `tree` struct and sets it up as an empty tree

```
new_tree:
        push    rbp
        mov     rbp, rsp
        mov     rdi, tree_size
        call    malloc
        xor     edi, edi
        mov     [rax+t_root], rdi
        mov     [rax+t_count], rdi
        leave
        ret
```

Finding a node in a tree: $p = find(t,n)$

```
find:   push    rbp
        mov     rbp, rsp
        mov     rdi, [rdi+t_root]
        xor     eax, eax
.more   cmp     rdi, 0
        je      .done
        cmp     rsi, [rdi+n_value]
        jl      .goleft
        jg      .goright
        mov     rax, rsi
        jmp     .done
.goleft:
        mov     rdi, [rdi+n_left]
        jmp     .more
.goright:
        mov     rdi, [rdi+n_right]
        jmp     .more
.done   leave
        ret
```

Inserting a node into a tree

- The code is too long for a slide
- First you check to see if the key is already in the tree
- If not, then you create a new node and set it value and set its two kids to NULL
- There is a special case for an empty tree
- If not empty, then we must traverse down the tree, going sometimes left and sometimes right to find the right place to insert the new node

Printing the keys in order

- We first call a non-recursive function with the tree object
- It calls a recursive function with the root node

```
;       print(t);
print:
        push    rbp
        mov     rbp, rsp
        mov     rdi, [rdi+t_root]
        call    rec_print
        segment .data
.print  db      0x0a, 0
        segment .text
        lea     rdi, [.print]
        call    printf
        leave
        ret
```

Recursive print function: rec_print(t)

```
rec_print: push    rbp
           mov     rbp, rsp
           sub     rsp, 16            ; make room to save t
           cmp     rdi, 0             ; return if t is NULL
           je      .done
           mov     [rsp+.t], rdi      ; save t, .t equ 0
           mov     rdi, [rdi+n_left] ; print the left sub-tree
           call    rec_print
           mov     rdi, [rsp+.t]      ; print the current node
           mov     rsi, [rdi+n_value]
           lea     rdi, [.print]      ; .print: format string
           call    printf
           mov     rdi, [rsp+.t]      ; print the right sub-tree
           mov     rdi, [rdi+n_right]
           call    rec_print
.done      leave
           ret
```