Computer Systems 1
Lecture 14

Linked Lists

Dr. John T. O'Donnell
School of Computing Science
University of Glasgow

Topics

Review of pointers

- p := &x   p is a pointer to x

```
    lea    R5,x[R0]     ; R5 := &x
```

- y := *p   y is the value that p points to

```
    load   R6,0[R5]     ; R6 := *R5
```

Nodes

- A linked list consists of a linear chain of *nodes*
- A node is a record with two fields
  - *value* is a word containing useful information, the *content* of the node. May be an integer, character, or even a pointer to something else.
  - *next* is a word containing a pointer to the next node in the list
- The last node in the list has a special value *nil* in the *next* field
- *nil* is represented by 0 (so you can't have a pointer to memory location 0, but normally that's where the program will be so you wouldn't want that anyway)

Accessing the fields of a node

- Suppose p is a pointer to a node
- load R1,p[R0] ; R1 := p
- load R2,0[R1] ; R2 := (*p).value
- load R3,1[R1] ; R3 := (*p).next

Representing a linked list

	a	mem[a]	mem[a+1]
	0		
	1		
	2	37	6
	3	42	0
	4		
p	5	24	2
	6	97	3
	7		
	8		

p = 5, and the list p = [24, 37, 97, 42]

Representing a linked list

	a	mem[a]	mem[a+1]
	0		
	1		
	2	37	6
	3	42	0
	4		
p	5	24	2
	6	97	3
	7		
	8		

p = 5, and the list p = [24, 37, 97, 42]

Representing a linked list

	a	mem[a]	mem[a+1]
	0		
	1		
	2	37	6
	3	42	0
	4		
p	5	24	2
	6	97	3
	7		
	8		

p = 5, and the list p = [24, 37, 97, 42]

Representing a linked list

	a	mem[a]	mem[a+1]
	0		
	1		
	2	37	6
	3	42	0
	4		
p	5	24	2
	6	97	3
	7		
	8		

p = 5, and the list p = [24, 37, 97, 42]

Basic operations on lists

- Three key operations:
  - Is a list p empty?
  - What's the value in a node?
  - What's the next node
- The following code assumes that all the pointer variables (p, q) are in memory, so they must be loaded and stored
- In practice, we often keep the pointers in registers so you don't need all those loads and stores

Is list p empty?

- Nil is 0, so the list that p points at is empty iff p=0
- Generally it is unsafe to perform an action on a list p unless p actually points to a node, so this test is commonly needed

```
    load   R1,p[R0]
    cmpeq  R2,R1,R0
    jumpt  R2,pIsEmpty[R0]
; No, p is not empty
 ...
 ...
pIsEmpty
; Yes, p is empty
```

Get value in node that p points at: x := *p.value

- x := *p.value
- This is safe to do only if p is not empty
- The value field of a node is at offset 0 in the node record

```
load   R1,p[R0]     ; R1 := p
load   R2,0[R1]     ; R2 := *p.value
store  R2,x[R0]     ; x := *p.value
```

Get pointer to next node in a list: q := *p.next

- q := *p.next
- This is safe to do only if p is not empty
- The next field of a node is at offset 1 in the node record

```
load   R1,p[R0]     ; R1 := p
load   R2,1[R1]     ; R2 := *p.next
store  R2,q[R0]     ; q := *p.next
```

Traversing a list p

A while loop is the best looping construct for traversing a list

```
ListSum (p)
  { sum := 0;
    while p /= nil do
      { x := (*p).value;
        sum := sum + x;
        p := (*p).next;
      }
```

Search a list p for a value x

Again, the best looping construct is a while loop

```
ListSearch (p, x)
  { found := False;
    while p /= nil && not found do
      { found := x = (*p).value;
        p := (*p).next;
      }
    return found;
  }
```

This is a good example of the proper use of a while loop

- The condition checks for end of data, and also for early completion
- There is no break statement or goto
- The loop works even if the original list p is nil

cons — constructing a list be consing a value to the front

- Suppose p = [23, 81, 62]
- q := cons (56, p)
- After computing q, we have
  - q = [56, 23, 81, 62]   q is the same as p but with 56 attached to the front
  - p = [23, 81, 62]   p is unchanged

Implementing cons

```
cons (x, p)
  { q := newnode ();
    (*q).value := x;
    (*q).next := p;
    return q;
  }
```

- No change is made to p, or to the node p points to
- A new node is allocated and set to point to p
- A pointer to the new node is returned
- A function like cons — which produces a new result but does not modify its arguments — is called a *pure function*

Getting a new node from avail list

```
if avail = nil
  then { error "fatal error: out of heap" }
  else { newnode := avail;
         avail := (*avail).next;
         return newnode;
       }
```

Inserting a node with x where p points

```
r := newnode ();
(*r).value := x;
(*r).next := (*p).next;
(*p).next := r;
```

- Notice that we can insert x *after* the node that p points to
- But we cannot insert x *before* that node
- It's common, in list algorithms, to have two pointers moving along through the list, one lagging an element behind the other, to make insertion possible

List header

- Suppose we have a list p and a value x
- We want to insert x into the list p at an arbitrary point
- Another pointer q points to the insertion position
- A slightly awkward problem: the code to insert x at the front of the list is slightly different from the code to insert x after some element (*q)
  - If somewhere in the middle, we can insert x *after* the node that q points to
  - The insertion algorithm will change (*q).next
  - But if we need to insert x at the beginning of the list, we cannot do that; instead the pointer p needs to be changed
- Solution: don't use an ordinary variable for p; make a *header node* whose next field points to the list

Deleting a node

- Need a pointer p into the list; the node *after* p will be deleted
- Just change (*p).next to skip over the next node, and point to the one after
- The node being deleted should be returned to the avail list, so it can be reused

Code for deleting a node

If p points to a node, delete the node after that, assuming it exists

```
delete (p)
  { if p /= nil
    then { q := (*p).next;
           if q /= nil
             then { (p*).next := (*q).next;
                    (*q).next := avail;
                    avail := q;
                  }
         }
  }
```

- We can't delete the node p points to, we can only delete the following node, which q points at
- If you know that p cannot be nil, the first test can be omitted
- We do need to check whether q = nil; if it is, there's no node to delete
- It doesn't matter whether (*q).next is nil

Space leaks

- If you return a deleted node to the avail list, it can be reused
- If you don't, this node becomes inaccessible: it doesn't hold useful data, yet it can't be allocated
- this is a bug in the program
- Over time, as a program runs, more and more nodes may become inaccessible: a space leak

Memory management

- It's a bug if you delete a node that contains useful data
- It's a bug if you don't delete a node that doesn't contain useful data
- With complicated data structures, this can be difficult
- A common solution is garbage collection
  - The program doesn't explicitly return nodes to the avail list
  - Periodically, the *garbage collector* traverses all data structures and marks the nodes it finds
  - Then the GC adds all unmarked nodes to the avail list

Sharing and side effects

- Suppose p = [6, 2, 19, 37, 41]
- Traverse a few elements, and set q to point to the 19 node
- Now q = [19, 37, 41] and p is unchanged
- Then delete the second element of q. The result is
  - q = [19, 41]
  - p = [6, 2, 19, 41] Modifying q has also modified p
- This is called a side effect
- Sometimes you want this to happen, sometimes not, so it's important to be careful about it!

Comparing lists and arrays

- Lists and arrays are two different kinds of data structure that contain a sequence of data values
- How do you decide which to use?
- Consider the properties of lists and arrays, and the needs of your program
- And there are many other data structures to choose from, which you'll encounter as you learn computer science

Accessing elements

- Direct access to an element
  - Array: gives direct access ("random access") to element with arbitrary index i
  - List: gives direct access only to an element you have a pointer to; random access is inefficient
- Traversal
  - Array: initialise i to 0; repeatedly set `i := i+1`; terminate when $i \geq n$ (that's the purpose of a **for** loop)
  - List: initialize p to point to the list; repeatedly set `p := (*p).next`; terminate when `p = nil`

Usage of memory

- Memory needed per element
  - Array: need just the memory required for the element itself (typically a word)
  - List: need a node for each element, which also requires space for the next pointer (typically a word)
  - So typically, an array with n elements needs n words, while a list requires $2 \times n$ words
- Flexibility
  - An array has fixed size and needs to be allocated fully
  - A list has variable size and needs only enough memory to hold its nodes

More general data structures

- We can put several pointer fields in each node, and produce an enormous variety of data structures, tailored for the needs of an application program
- Just a few examples
  - Doubly linked list: each node contains two pointers, one to the previous node and one to the next. Allows traversal both directions.
  - Circular list: there is no “last” node where next=nil; instead, *every* node points to the next node, and the list loops back to itself. There is no “first” or “last” node.

Abstract data type

- A stack is an abstract data type
  - The idea: define the type by the operations it supports, not by the code that implements it
  - This is useful because there may be different implementations of an ADT, and which implementation is best may depend on the application using it
- The stack ADT is defined by the operations it supports: push, pop
- There are several completely different ways to implement a stack
  - We have already seen how to implement a stack with an array
  - We can also do it with a linked list

Linked list implementation of stack

- A linked list gives easy access to the front of the list, and a stack gives easy access to the top of the stack.
- Represent Empty stack as nil
- Push x is implemented by stack := cons (x, stack)
- Pop x is implemented by stack := (*stack).next

Array representation of stack

- We can implement a stack using an array
- There is a variable *stLim* which gives the size of the array — this is the limit on the maximum number of elements that can be pushed
- There is a variable *stTop* that gives the current number of elements in the stack

Relationship between arrays and stacks

- Array
  - A container that holds many elements
  - Each element has an index (which is an integer)
  - You can access any element x[i]
  - You can access the elements in any order
- Stack
  - A container that holds many elements
  - You can only access the top element, and you don't need to know its index
  - You can (and must) access the elements in last in — first out order

Pushing x onto a stack

```
; push the x onto the stack
; stack[stTop] := R1; stTop := stTop + 1

push    load  R1,x[R0]          ; R1 := x
        load  R2,stTop[R0]      ; R2 := stTop
        store R1,stack[R2]      ; stack[stTop] := x
        lea   R3,1[R0]          ; R3 := constant 1
        add   R2,R2,R3          ; R2 := stTop + 1
        store R2,stTop[R0]      ; stTop := stTop + 1
```

Pop a stack, returning x

```
; pop the stack, store top element into x
; stTop := stTop - 1; x := stack[stTop]

pop     load  R2,stTop[R0]    ; R2 := stTop
        lea   R3,1[R0]        ; R3 := constant 1
        sub   R2,R1,R3        ; R2 := stTop - 1
        load  R1,stack[R2]    ; R1 := stack[stTop-1]
        store R1,x[R0]        ; x := stack[stTop-1]
        store R2,stTop[R0]    ; stTop := stTop - 1
```

Issues with simplest implementation

- It doesn't check for errors!
  - If push is called when stack is full, data will be written outside the array
  - If pop is called when stack is empty, a garbage result will be returned
- Either of these errors may cause the program to get wrong answers or to crash

Robust software

- Fragile software will respond to a minor problem by going haywire: might crash, or produce wrong answers
- Robust software checks for all errors and does something appropriate; a minor problem doesn't turn into a major one

Error checking and error handling

- Software should not assume everything is ok — it should check for errors
  - push (x) when the stack is full
  - x := pop () when the stack is empty
- If an error is detected, the error must be *handled*
- There are many approaches
  - Produce a message and terminate the program
  - Return an error code to the calling program and let it decide what to do
  - Throw an exception, which will interrupt the calling program, and invoke its error handler

For simplicity, we will terminate the program if an error occurs.

Error checking: push

If the stack is full, there is no space to store the new element, so push fails

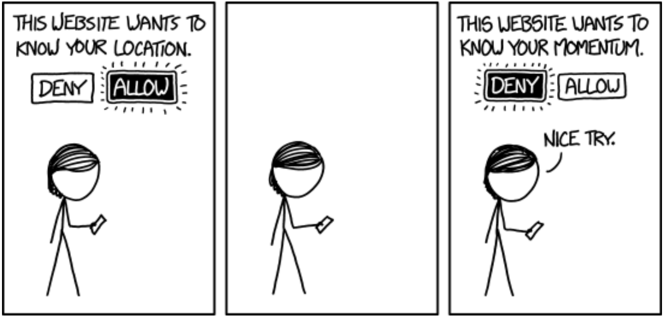
```
; push (v)
; if stTop >= stLim
;     then
;        terminate because the stack is full: cannot push
;     else
;        stack[stTop] := v
;        stTop := stTop + 1
;        return ()
```

Error checking: pop

If the stack is empty, there is no element to return, so pop fails

```
; v = pop ()
;   if stTop == 0
;     then
;       terminate because the stack is empty: cannot pop
;     else
;       stTop := stTop - 1
;       v := stack[stTop]
;       return (v)
```

Location sharing



https://xkcd.com/1473/