CS109A Notes for Lecture 3/8/96

Data Structures

1. *Linked list* = records with data field(s) and *next* field pointing to next element.
2. *Array* = array of limited size with cursor or pointer to last element.

Operations

Lookup, insert, delete (like the Dictionary ADT) are most common.

- Take $O(n)$ time on an n-element list.

Example: Here is insertion in an ML list.

- Does not create duplicate elements, so must check x is not already on list.

```
(1) fun insert(x,nil) = [x]
(2) |   insert(x, y::ys) =
(3)       if x<>y then y::insert(x,ys)
(4)       else y::ys;
```

Correctness proof:

- $S(n)$: If L is of length n, then $insert(x, L)$ returns a list with x and the elements of L, and nothing else.

Basis: $n = 0$. Then L has no elements, line (1)'s pattern matches, and a list with only x is returned.

Induction: Assume $S(n)$, $n \geq 0$. If L is of length $n+1$, line (1) doesn't match. Line (2) matches.

- If $x \neq y$, then by the inductive hypothesis, $insert(x, ys)$ returns a list with the elements of L except for y but with x included. Then line (3) returns a list with y, x, and the other elements of L, i.e., what $S(n+1)$ says should be returned.
- If $x = y$, then line (4) returns L. Since x is on L, again we return what $S(n+1)$ says should be returned.

- Note that we used the inductive hypothesis to talk about what happens on recursive calls, without having to imagine an arbitrarily large sequence of calls.

Implementation Variants

1. Sorting the list.
   - We can search only as far as x to test whether x is on the list (saves average factor of 2).
2. Allow duplicates.
   - Insert in $O(1)$.
   - Penalty is that lookup, delete may take longer because lists with duplicates get longer than number of elements.
3. *Sentinels*: Add x onto end of list before searching for x.
   - Suitable only for array representation.
   - Saves time testing for end of list at each step.

Stacks and Queues

- *Stack* = ADT with principal operations *push* and *pop*.

```
exception EmptyStack;

fun push(x,S) = x::S;

fun pop(nil) = raise EmptyStack
|   pop(x::xs) = xs;
```

- *Queue* = ADT with principal operations *enqueue* and *dequeue* (= *pop*).

```
exception EmptyQueue;

fun enqueue(x,Q) = Q@[x];

fun deqeue(nil) = raise EmptyQueue
|   dequeue(x::xs) = xs;
```

Use of Stack to Support Recursive Calls

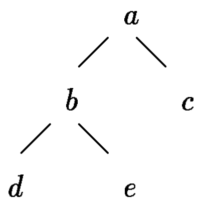
Here is the *preorder* function from Fig. 5.32, FCS.

```
     void preorder(TREE t)
 {
 (1) if (t != NULL) {
 (2)     printf("%c\n", t->nodeLabel);
 (3)     preorder(t->leftChild);
 (4)     preorder(t->rightChild);
     }
 }
```

The *run-time implementation* of such a function is essentially as follows.

- Keep a stack whose entries are pairs that tell us what we need to know about the state of a call to *preorder*:
    1. The value of t, a pointer to the root of the tree about which the call to *preorder* was made.
    2. The place in the execution of the function, essentially the line number being executed. Most important, when we make a recursive call, is it from line (3) or line (4)?
- When a new call is made at line (3) or (4), push the new value of t onto the stack with line number = 1.
    - □ When a call to *preorder* returns, pop the stack, exposing the value of t and the current line number from the previous call.

Example: Consider the tree:



Here is the sequence of stacks (top at the right) in which the pair (x, i) represents a stack entry for the call in which t is a pointer to node x and i is the line number being executed.

- Ignores calls on empty trees that immediately return.

$(a,1)$
$(a,3)(b,1)$
$(a,3)(b,3)(d,1)$
$(a,3)(b,3)$
$(a,3)(b,4)(e,1)$
$(a,3)(b,4)$
$(a,3)$
$(a,4)(c,1)$
$(a,4)$
ϵ