G205
Fundamentals of Computer Engineering
CLASS 9, Mon. Oct. 6 2003
Stefano Basagni
Fall 2003
M-W, 9:50am-11:30am, 410 Ell
Sets

- Collection of objects
- As important as in math
- Dynamic sets: Change over time
- Basic techniques for representing and manipulating finite dynamic sets
- Best way of implementing a dynamic set depends on the operations to be performed on the set
Elements of a Dynamic Set

- Each element is seen as an object with different fields.
- Often one field is identified as the key.
- Non-key fields are satellite data unused in the set implementation.
- Often a total ordering is assumed among the keys of a set.
Operations on Dynamic Sets

Two categories
- Modifying operations: Change the set
- Queries: Return information about the set

Modifying operations
- Insert(S, x): Insert (element pointed by) x in S
- Delete(S, x): Remove (element pointed by) x from S
Query Operations

- **Search**(S,k): Returns a pointer x to an element in S such that key[x]=k, or NIL
- **Minimum**(S): Returns a pointer x to the element of S with the smallest k
- **Maximum**(S): Similar to Minimum(S)
- **Successor**(S,x): Returns a pointer to the next larger element in S, or NIL if x is the maximum
- **Predecessor**(S,x): Similar to Successor(S,x)
Stacks and Queues

- Simple data structures for representing dynamic sets that use pointers
- Delete operation is prespecified
  - Stack: Delete the most recently inserted element (implements LIFO)
  - Queue: Delete the element in the set for the longest time (implements FIFO)
Stacks

- Implementation of a stack with at most \( n \) elements with an array \( S[1...n] \)
- \( \text{top}[S] \) maintains the index of the most recently inserted element in the array
- The stack consist of \( S[1...\text{top}[S]] \)
- When \( \text{top}[S] \) is 0, the stack is empty
- We do not worry here with stack overflows (\( \text{top}[S] > n \))
Stack Operations

Stack-Empty(S)
return top[S] = 0

Push(S, x) // Insert
  top[S] = top[S] + 1
  S[top[S]] = x

Pop(S) // Delete
  if Stack-Empty(S) then error “underflow”
  else top[S] = top[S] – 1
  return S[top[S]+1]
Queues

- Implementation of a queue with at most $n-1$ elements with an array $Q[1...n]$
- $\text{head}[Q]$ maintains the index to the head of the queue (the element first to be removed)
- $\text{tail}[Q]$ indexes the next location a new element is inserted
- When $\text{head}[Q]=\text{tail}[Q]$ the queue is empty
- When $\text{head}[Q]=\text{tail}[Q]+1$ the queue is full
  (Addresses are “wrapped around”)

10/6/2003
Queues Operations

Enqueue(Q,x) // Insert
   Q[tail[Q]] = x
   if tail[Q]=n then tail[Q]=1
   else tail[Q]=tail[Q]+1

Dequeue(Q) // Delete
   x=Q[head[Q]]
   if head[Q]=n then head[Q]=1
   else head[Q]=head[Q]+1
   return x
Linked Lists

- Objects are arranged in linear order
- Order is determined by a pointer (not by an index)
- Support all operations on dynamic sets
- Doubly-Linked List implementation: key, prev, and next fields
  - Head of the list has no prev element
  - Tail of the list has no next element
- head[L] points to the first element in the list
- If head[L] is NIL, the list is empty
Different Linked Lists

- Doubly linked lists
- Singly linked lists: No prev pointer
- Circular list
  - The prev pointer of the head of the list points to the tail
  - The next pointer of the tail of the list points to the head
- Lists can be sorted or unsorted
Searching a Linked List

- Finds the first element in the list with a given key
- Linear search that returns a pointer: $\Theta(n)$

List-Search(L,k)

```plaintext
x = head[L]
while x ≠ NIL and key[x] ≠ k do
    x = next[x]
return x
```
Inserting Into a Linked List

Insertion at the front of the list: $O(1)$

List-Insert($L, x$)

next[$x$] = head[$L$]

if head[$L$] $\neq$ NIL

then prev[head[$L$]] = $x$

head[$L$] = $x$

prev[$x$] = NIL
Deleting from a Linked List

Use Search-List to retrieve the element’s pointer: $\Theta(n)$

\textbf{List-Delete}(L, x)

\begin{enumerate}
  \item if prev[x] $\neq$ NIL
    \begin{enumerate}
      \item then next[prev[x]] = next[x]
    \end{enumerate}
  \item else head[L] = next[x]
  \item if next[x] $\neq$ NIL
    \begin{enumerate}
      \item then prev[next[x]] = prev[x]
    \end{enumerate}
\end{enumerate}
Rooted Trees

- Each tree node is an object with a key field and pointers

**BINARY TREES:**
- Three pointers: left, right and p to the left child, to the right child and to the parent
- If \( p[x] \neq \text{NIL} \) then \( x \) is the root
- \( \text{root}[T] \) is the root of a tree \( T \)
- If \( \text{root}[T] = \text{NIL} \) then the tree is empty
Unbounded Branches Trees

- Left-child, right-sibling representation
- p is the pointer to the parent and root[T] points to the root
- Each node has only two other pointers:
  - left-child[x] points to the leftmost child of x
  - right-sibling[x] points to the sibling of x immediately to the right
Assignments

- Textbook, pages 196—217
- Updated information on the class web page:
  www.ece.neu.edu/courses/eceg205/2003fa