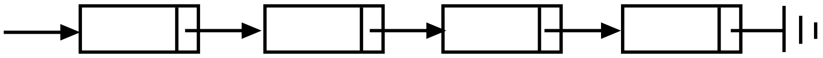
CSE 2320 Notes 8: Linked Lists

(Last updated 9/29/06 8:46 PM)

CLRS, 10.2-10.3

LINKED LISTS

1. Singly-linked (forward) lists.



Links may be:

Pointers

Subscripts

Disk addresses

Web URLs (a "logical" address vs. a "physical" address in the other three cases)

If the nodes have a key (i.e. a dictionary), should the list be ordered or unordered?

ASSUMPTION: Uniform access probabilities – equal likelihood for accessing each of n keys

expected probes	hit	miss
unordered	$\frac{n+1}{2}$	n
ordered	$\frac{n+1}{2}$	$\frac{n+1}{2}$

Most applications have many more hits than misses.

Many applications, however, need *ordered retrieval* (SUCCESSOR, PREDECESSOR).

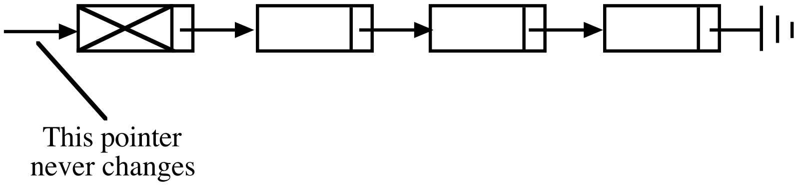
2. Keeping linked list code simple and efficient.

   a. *Header* – dummy node at beginning of list (even if no other nodes).

      Avoids “first node special” cases:

      DELETE

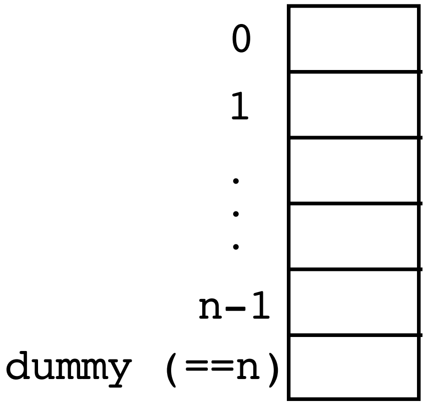
      UNION



      Can be wasteful if an application needs large number of very short lists.

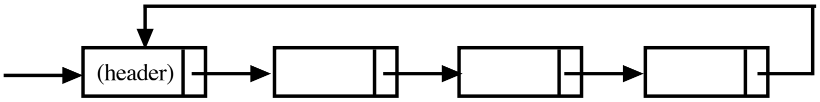
   b. *Sentinel* – dummy element at end of unordered table, unordered list, or tree.
      (Book uses term “sentinel” for both headers and sentinels.)

      Avoids checking for “end” of data structure.

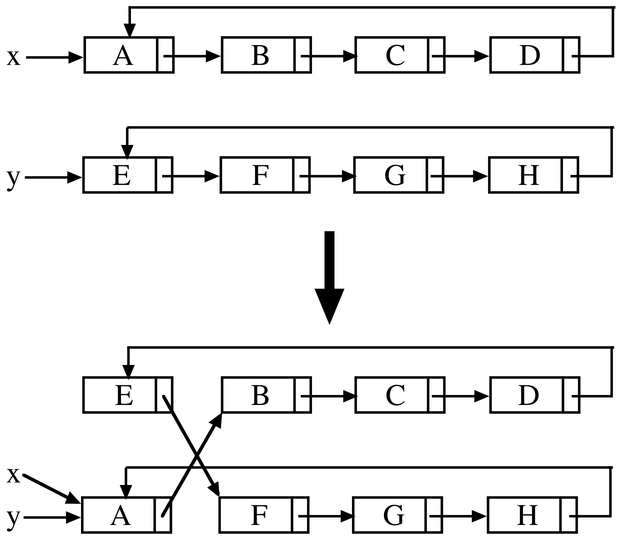


```
A[dummy] = key;
for (i = 0;
     A[i] != key;
     i++)
  ;
if (i == dummy)
  < not actually in table >
else
   < really in table >
```

3. Circular lists – can achieve $\Theta(1)$ time in special cases.



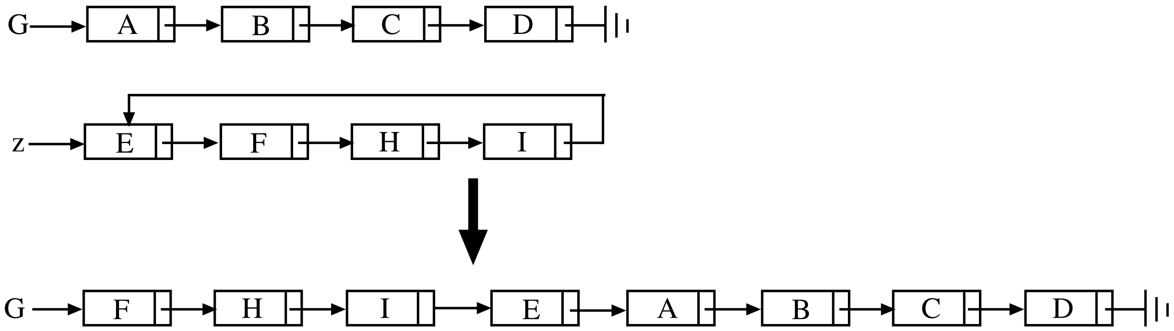
Example 1: Concatenate strings (sequences) stored as linked lists.



```
temp = (*x);
(*x) = (*y);
(*y) = temp;
x = y;
```

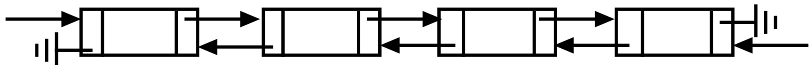
Example 2: Free storage list – avoids `malloc`/`free` overhead

Including unneeded circular list in a garbage list:

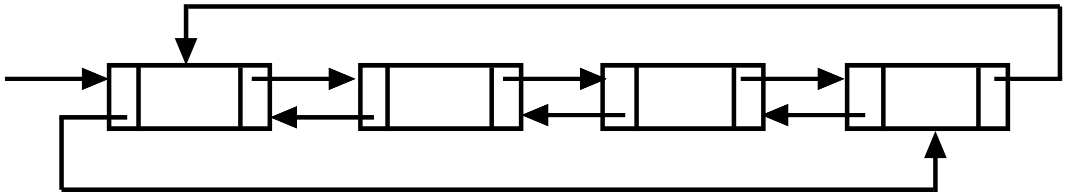


```
work = z->next;
z->next = G;
G = work;
```

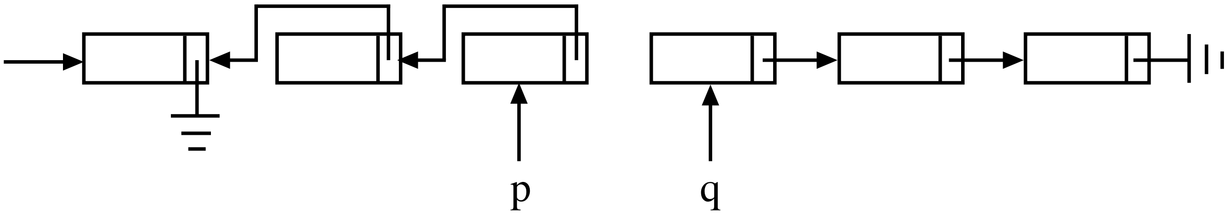
4. Doubly-linked lists.



Can also have circular doubly-linked.



Example 1: Flexibility to go both ways, but can also use the following clever solution if concurrent access is not needed:



Example 2: Student Database

- Each student record is in a number of linked lists: ethnicity, major, place-of-birth, previous colleges, etc. to allow production of reports.
- Regardless of how a record is reached, it may be necessary to remove from one list and insert in another (e.g. change of major). Trade-off:
  - If double linking is used, the "predecessor" is immediately available but more space is used.
  - Without double linking, the "predecessor" is found by traversing the list. Suitability depends on length of lists.
- Insert node that `x` points to after node that `p` points to:

```
q=p->next;
x->next=q;
x->prev=p;
p->next=x;
q->prev=x;
```

- Remove node that `x` points to:

```
p=x->prev;
q=x->next;
p->next=q;
q->prev=p;
```

Example 3: Maintain the following abstraction for *n* elements, 0 . . . *n*-1:

Specification: (could be used for handles in minHeap in Notes 5)

- Initially all elements are *free*, but may become *allocated*.
- A particular free element may be requested and it becomes allocated. (`allocate()`)
- A particular allocated element may be requested and it becomes free. (`freeup()`)
- A request to find and allocate <u>*any*</u> free element may be made. (`allocateAny()`)
- All operations are to be supported in $O(1)$ time (except initialization).

Implementation:

- An array with *n*+1 elements is used. Element *n* acts as a header for a circular, doubly-linked list. Initialization:

`n=4`

	0	1	2	3	4
prev	4	0	1	2	3
next	1	2	3	4	0

- `allocate(int x)` is just deletion of `x` from a doubly-linked list:

```
p=prev[x];
q=next[x];
next[p]=q;
prev[q]=p;
prev[x]=next[x]=(-1);
```

- `freeup(int x)` inserts the freed element `x` after the header.

```
q=next[n];
next[x]=q;
prev[x]=n;
next[n]=x;
prev[q]=x;
```

- `allocateAny()` deletes the successor of the header:

```
p=next[n];
allocate(p);
return p;
```

- Possible errors? (see `circularFree.cpp`)

CLRS Problem ***10-1: Comparisons among lists***

	unsorted, singly linked	sorted, singly linked	unsorted, doubly linked	sorted, doubly linked
SEARCH(L, k)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
INSERT(L, x)	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
DELETE(L, x)	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
SUCCESSOR(L, x)	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
PREDECESSOR(L, x)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
MINIMUM(L)	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
MAXIMUM(L)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$