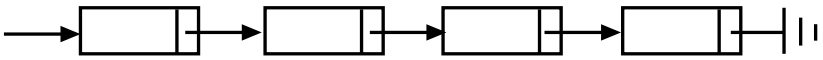
CSE 3318 Notes 9: Linked Lists

(Last updated 8/2/22 9:19 AM)

CLRS 10.2

(Aside: `https://patents.google.com/patent/US7028023B2/en`)

9.A. 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 (ascending) or unordered?

Ordered: If node y is the *physical successor* of x, then y is the *logical successor* of x.

Unordered: No relationship between adjacent nodes.

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

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

$$\frac{1+2+\cdots+n}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}$$

Most applications have many more hits than misses.

Many applications, however, need *ordered retrieval* (LOGICALSUCCESSOR, LOGICALPREDECESSOR).

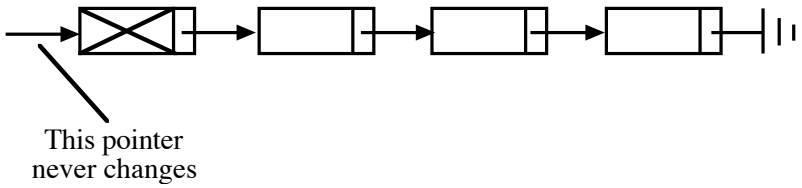
9.B. KEEPING LINKED LIST CODE SIMPLE AND EFFICIENT.

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

Avoids "first node special" cases:

DELETE

UNION

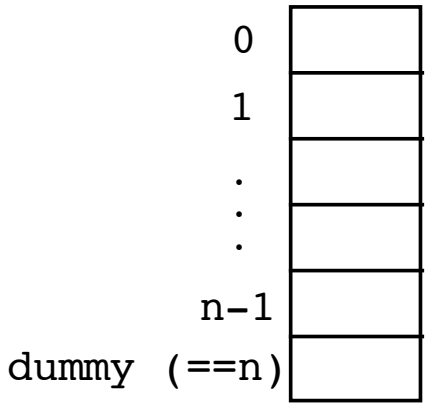


Can be wasteful if an application needs large number of very short lists.

b. *Sentinel* – dummy element at end of unordered table, unordered list, or tree.

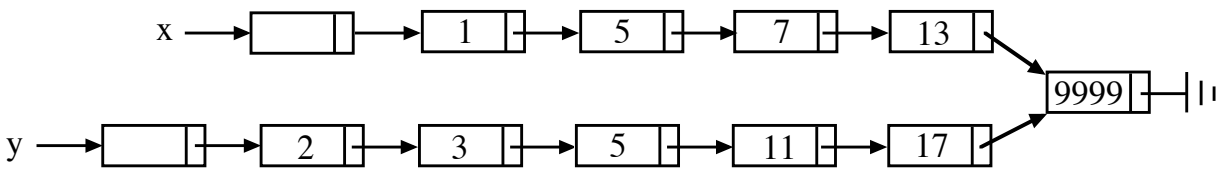
Avoids checking for "end" of data structure.

Linear search:



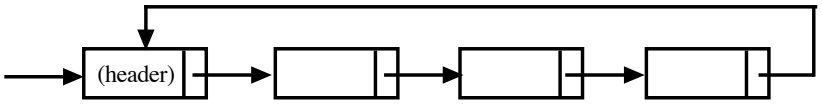

```
A[dummy] = key;
for (i = 0;
     A[i] != key; // No && i<n
     i++)
  ;
if (i == dummy)
  < not actually in table >
else
   < really in table >
```

Destructive union of ordered linked lists:

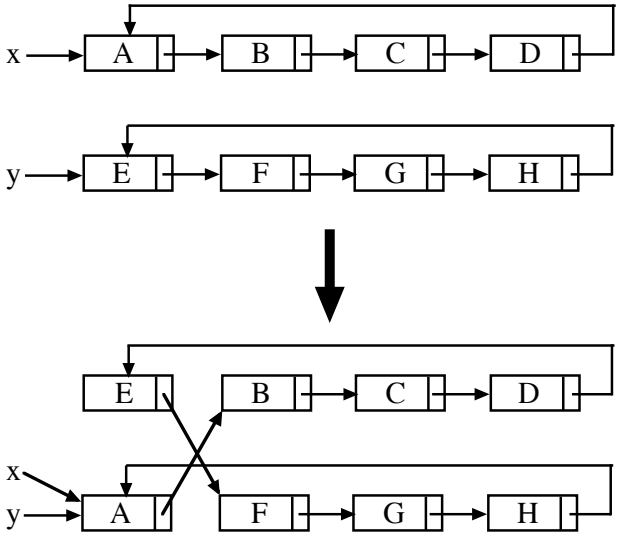


(https://ranger.uta.edu/~weems/NOTES3318/mergeDummy.c)

9.C. CIRCULAR LISTS – can achieve $\Theta(1)$ time in special cases.

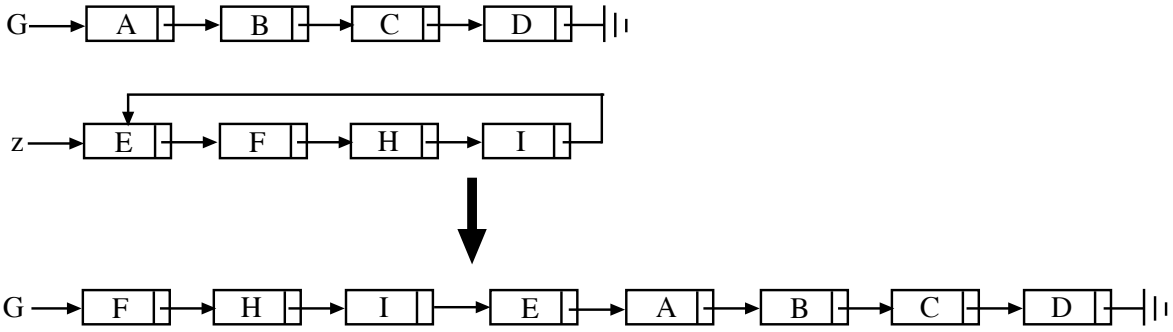


Example 1: Concatenate strings (sequences) stored as linked lists.



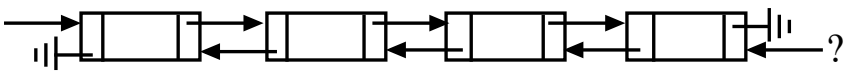
Example 2: Free storage (AKA garbage/recycling) list – avoids `malloc`/`free` overhead

Including unneeded circular list in a garbage list:

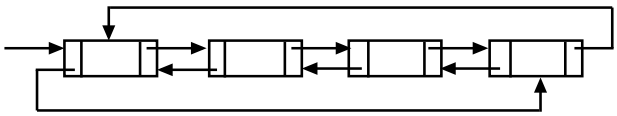


(See `https://ranger.uta.edu/~weems/NOTES3318/circular.c`)

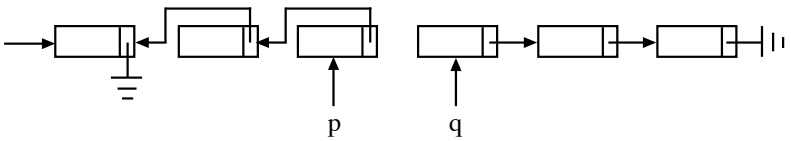
9.D. DOUBLY-LINKED LISTS



Can also have circular doubly-linked.

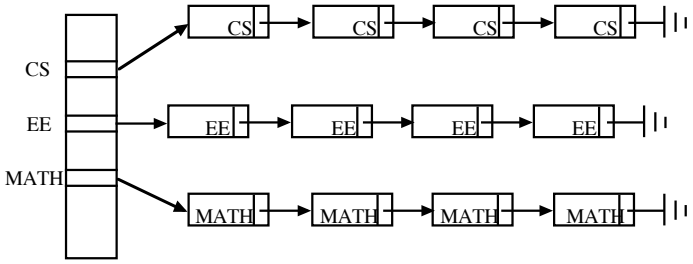


Example 1: Flexibility to go both ways, but can also use the following clever solution if concurrent access is not needed:



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 physical predecessor is immediately available but more space is used.
  - Without double linking, the physical predecessor is found by traversing the list. Suitability depends on length of lists.
- Insert node that `x` references after node that `p` references:

```
q=p.next;
x.next=q;
x.prev=p;
p.next=x;
q.prev=x;
```

- Remove node that `x` references: (Aside: `https://en.wikipedia.org/wiki/Dancing_Links`)

```
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 for maxHeap in Notes 05)

- 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 *any* 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 physical successor of the header:

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

- Error checking? `https://ranger.uta.edu/~weems/NOTES3318/circularFree.c`
- Aside: edge coloring
  `https://ranger.uta.edu/~weems/NOTES5311/misraGriesNew.c`

9.E. Comparisons Among List Implementations

L = pointer to first node of list, k = key, x = pointer to some node

SEARCH is an exact match search returning a pointer.
INSERT either inserts x at beginning (unordered) or at appropriate position (ordered).
DELETE removes the indicated node from list L.
PHYSICALSUCCESSOR returns pointer to the next node in a traversal along *next* links (not in CLRS).
PHYSICALPREDECESSOR returns pointer to the previous node in a traversal along *next* links (not in CLRS).
LOGICALSUCCESSOR returns pointer to the node whose key is the minimum among all nodes with larger keys (called SUCCESSOR in CLRS).
LOGICALPREDECESSOR returns pointer to the node whose key is the maximum among all nodes with smaller keys (called PREDECESSOR in CLRS).
MINIMUM returns pointer to the node whose key is the minimum among all nodes.
MAXIMUM returns pointer to the node whose key is the maximum among all nodes.

	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)	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
PHYSICALSUCCESSOR(L, x)	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
PHYSICALPREDECESSOR(L, x)	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
LOGICALSUCCESSOR(L, x)	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
LOGICALPREDECESSOR(L, x)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
MINIMUM(L)	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
MAXIMUM(L)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Which entry changes if the doubly-linked list is circular?