Monday:
1st Midterm Exam

REMINDER FROM LAST WEEK:

The linked list data types in Standish chapter 2:

```
typedef char AirportCode[4];

typedef struct NodeTag {
            AirportCode  airport;
            struct NodeTag *link;
          } NodeType, *NodePointer;
```

Inserting a new second node

- e.g., have (DUS→ORD→SAN),
  want (DUS→BRU→ORD→SAN)
  or have (DUS), want (DUS→BRU)
  or have ( ), want (BRU)
  - Any other special cases?
- A strategy:

```
create new node to hold BRU – call it n;
if empty list – point "list" at n; return;
else   set n.link to "list".link;
       set "list".link to &n; return;
```

Code to insert new 2nd node

- Assume external variable for list:

```
NodePointer list;
```

- And assume list already initialized *and has at least one node (i.e., no special case of empty list)*, then:

```
void insertNewSecondNode(void) {
    NodePointer n;
    n = (NodePointer)malloc(sizeof(NodeType));
    strcpy(n->airport, "BRU");
    n->link = list->link;
    list->link = n;
}
```

Searching a list for some info

- Idea is to *return a pointer to the node that contains the info* we are searching for, or return NULL if the info is not in the list
- Strategy:

```
declare local node pointer - call it n;
point n at first node in list;
while (n points to non-null node) {
    if (n's referent has the info)
        return n;
    else advance n to n->link;
}
return NULL if get this far;
```

List traversal & other notes

- Search strategy typifies list traversal:

```
start by pointing to first node;
process that node;
change pointer to that node's link;
keep going until success (e.g., found info), or
   until end (i.e., pointing at NULL);
```

  - Same idea works for lots of list operations
    - e.g., print list – immediately applicable
    - To append, first must get to last node
    - To remove a node, must get to it first
- But also usually consider potential special cases
  - e.g., first node, last node, empty list, just one node, …

Strategy to delete last node

```
declare 2 local node pointers: current, previous;
  /* then handle special cases first */
just return (i.e., do nothing) if list is empty;
free(list) and return if just one node in list;
  /* otherwise traverse list to find second-to-last node */
point previous at first node;
point current at previous->link;
while (current->link does not point to null)
    advance both pointers;
  /* finally, set link of second-to-last, and destroy last */
set previous->link = NULL;
free (current);  /* Done. */
```

Code to delete last node (pt. 1)

```
void deleteLastNode(NodePointer *l) {
   /* note: pointer to pointer – allows changing original pointer */
   NodePointer previous, current;
   if (*l != NULL) {  /* case of empty list – do nothing */
       if ((*l)->link == NULL) {  /* list with 1 node */
          free(*l);
          *l = NULL;
       } else {   /* general case  (i.e., all other cases) */
          previous = *l;
          current = (*l)->link;

          /* continued next slide */
```

Code to delete last node (pt. 2)

```
          while (current->link != NULL) {
               /* i.e., not at last node yet */
               previous = current;
               current = current->link;
           }
           /* now previous points to next-to-last, so make it last */
           previous->link = NULL;

           /* current points to old last, so recycle the storage */
           free(current);

       }  /* end general case */
   }  /* end case of non-empty list */
}  /* end function */
```

btw: other linked structures

- More elaborate linked lists are often useful
  - e.g., nodes with 2 links: previous and next
    - Easy *reverse* traversal, insertion *before* a node, …
    - But 2 links to worry about for insert, remove, …
  - e.g., circular lists – last points to first (and first points to last for 2-way circular list)
  - Choice depends on problem and efficiency (more to come in later chapters; maybe upcoming project too)
- Trees – see figure 2.23 (p. 56) – more later
- Graphs – chapter 10 – not part of CS 12 though