Linked Lists

Lecture 05

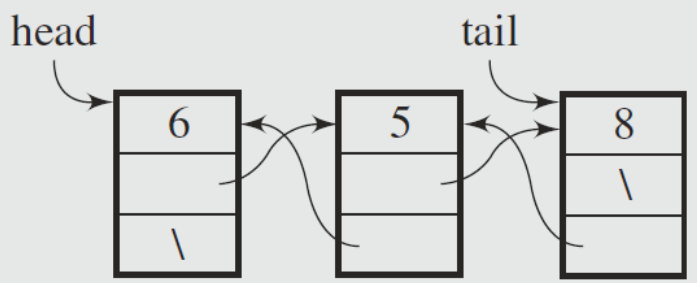
Instructor: Dr. Cong Pu, Ph.D.

cong.pu@okstate.edu

Adapted partially from Data Structures and Algorithms in Java, M.T. Goodrich, R.Tamassia and M. H. Goldwasser, Sixth Edition,Wiley; Data Structures and Algorithms in C++, Adam Drozdek, 4th Edition, Cengage Learning

Doubly Linked Lists

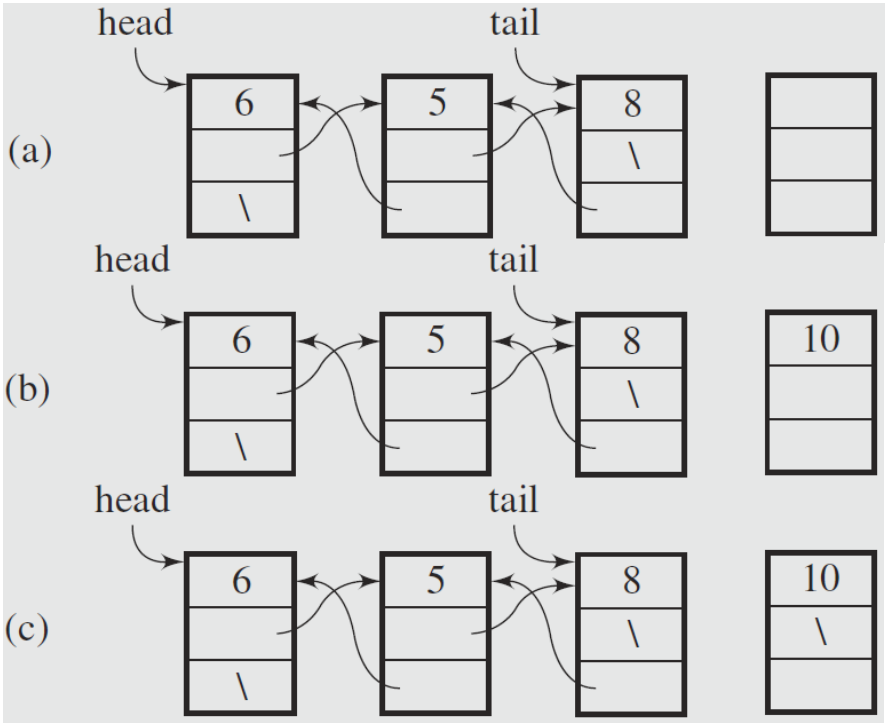
- Singly linked lists
  - difficulty in deleting a node from the end of a singly linked list
  - continually scan to the node just before the end in order to delete correctly
- To address this problem,
  - redefine the node structure and add a second pointer that points to the predecessor
  - ***doubly linked lists***



functions for processing doubly linked lists are more complicated

- maintaining one more pointer

Doubly Linked Lists (cont.)



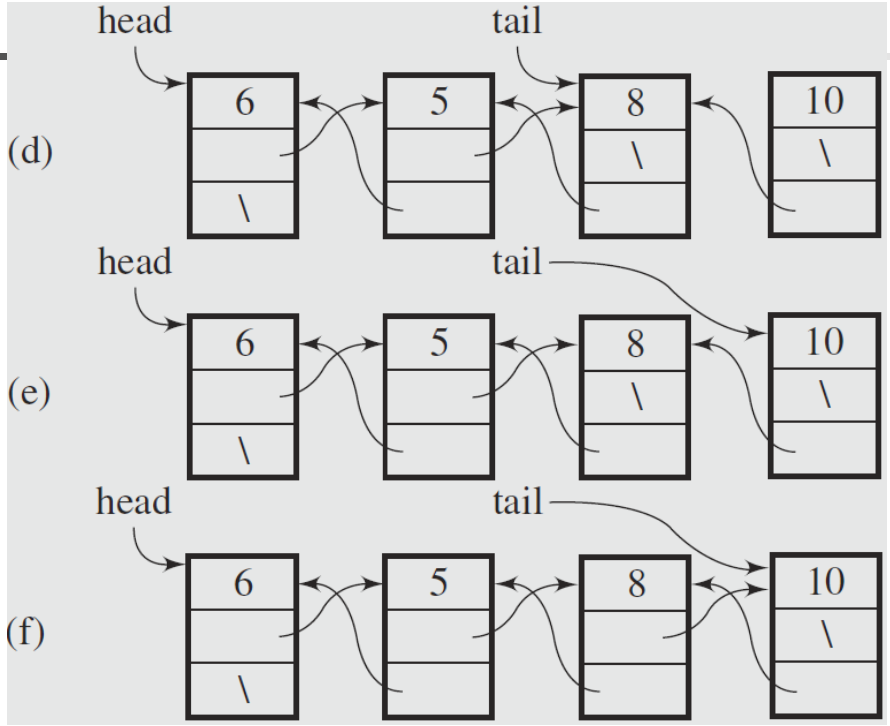

```
DLLNode *p;

p = new DLLNode(10);

P->next = 0;
```

- **Insertion:** at the end of a list
  - create a new node and initialize the data member
  - since being inserted at the end of the list, set its *next* member to ***null***

Doubly Linked Lists (cont.)



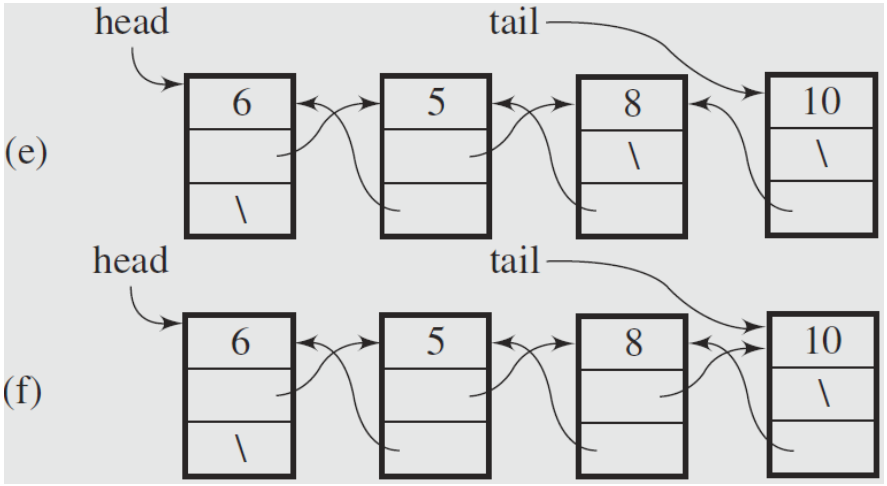

```
p->prev = tail;

tail->next = p;

tail = p;
```

- **Insertion** (cont.): at the end of a list
  - set the *prev* member to *tail* to link it back to the former end of the list
  - set the *next* member of the previous node to point to the new node
  - set the *tail* pointer now to point to this new node

Doubly Linked Lists (cont.)



- **Insertion** (cont.): at the end of a list
  - set the *next* member of the previous node to point to the new node
    - assumed that the predecessor exists
    - what if it is an *empty linked list*?
      - new node is the only node
      - no predecessor
      - set *head* to point to the new node

Doubly Linked Lists (cont.)

```
class DLLNode {
public:
    DLLNode() {
        next = prev = 0;
    }
    DLLNode(int el, DLLNode *n = 0, DLLNode *p = 0) {
        info = el; next = n; prev = p;
    }
    int info;
    DLLNode *next, *prev;
};
```

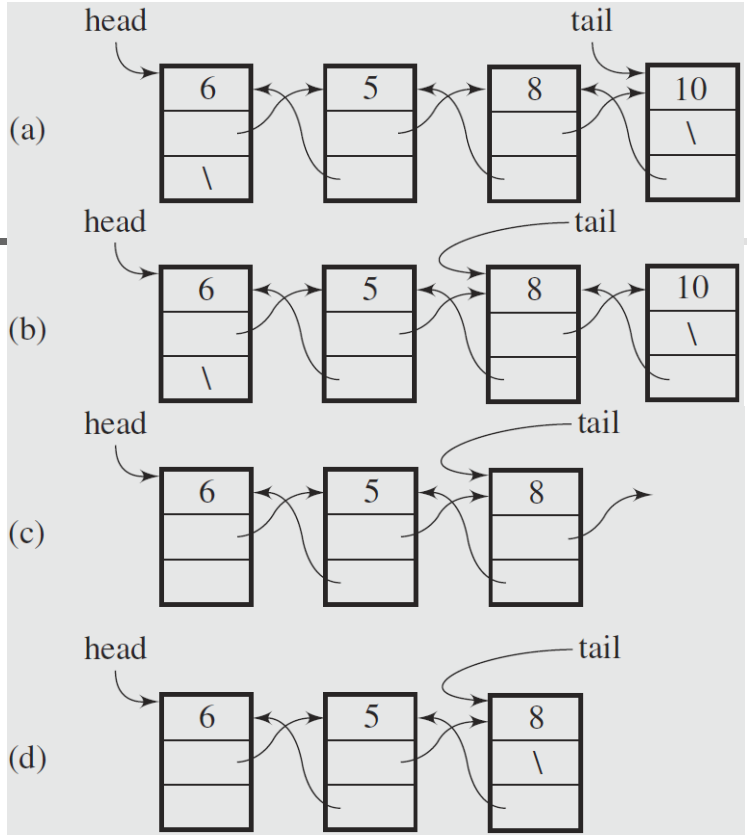
- **Insertion** (cont.): at the end of a list
  - create a new node and initialize the data member
  - since being inserted at the end of the list, set its *next* member to ***null***
  - set the *prev* member to *tail* to link it back to the former end of the list
  - set the *next* member of the previous node to point to the new node
  - set the *tail* pointer now to point to this new node

Doubly Linked Lists (cont.)

- **Insertion** (cont.): at the end of a list
  - create a new node and initialize the data member
  - since being inserted at the end of the list, set its *next* member to ***null***
  - set the *prev* member to *tail* to link it back to the former end of the list
  - set the *next* member of the previous node to point to the new node
  - set the *tail* pointer

```
void DoublyLinkedList::addToDLLTail(int el) {
    if (tail != 0) {
        tail = new DLLNode(el,0,tail);
        tail->prev->next = tail;
    }
    else head = tail = new DLLNode(el);
}
```

Doubly Linked Lists (cont.)




```
tail = tail->prev;

delete tail->next;

tail->next = 0;
```

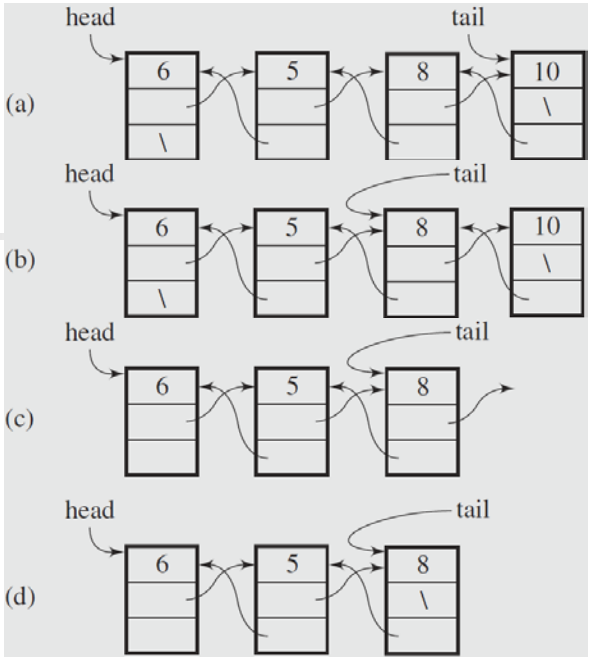
- **Deletion**: at the end of a list
  - a direct link to the predecessor of last node in the list
    - **no need to traverse** the list to find the predecessor
  - retrieve the *data* member from the node, then set *tail* to the node's predecessor
  - the predecessor becomes the last node

Doubly Linked Lists (cont.)

- **Deletion** (cont.)
  - special cases
    - if the list is empty,
      - an attempt to delete a node should be handled and reported to the user
      - e.g., isEmpty();
    - If the node being deleted is the only node in the list,
      - *head* and *tail* need to be set to ***null***

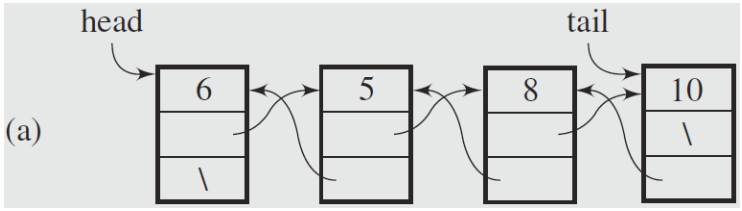
Doubly Linked Lists (cont.)

- **Deletion**: at the end of a list
  - a direct link to the predecessor of last node in the list
    - **no need to traverse** the list to find the predecessor
  - retrieve the *data* member from the node, then set *tail* to the node's predecessor
  - the predecessor becomes the last node




```
int DoublyLinkedList::deleteFromDLLTail() {
    int el = tail->info;
    if (head == tail) { // if only one DLLNode on the list;
        delete head;
        head = tail = 0;
    }
    else {              // if more than one DLLNode in the list;
        tail = tail->prev;
        delete tail->next;
        tail->next = 0;
    }
    return el;
}
```

Doubly Linked Lists (cont.)




```
head = head->next;

delete head->prev;

head->prev = 0;
```

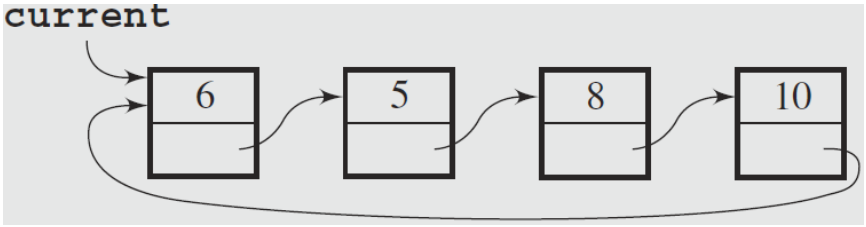
Deletion: at the beginning of a list

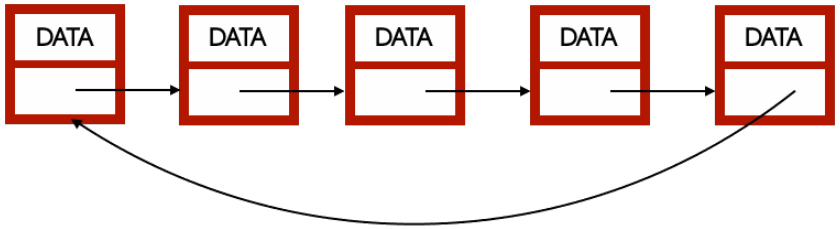
- a direct link to the successor of first node in the list
- retrieve the *data* member from the first node, then set *head* to the node's successor
- the successor becomes the first node

```
int DoublyLinkedList::deleteFromDLLHead() {
    int el = head->info;
    if (head == tail) { // if only one DLLNode on the list;
        delete head;
        head = tail = 0;
    }
    else {              // if more than one DLLNode in the list;
        head = head->next;
        delete head->prev;
        head->prev = 0;
    }
    return el;
}
```

Circular Lists

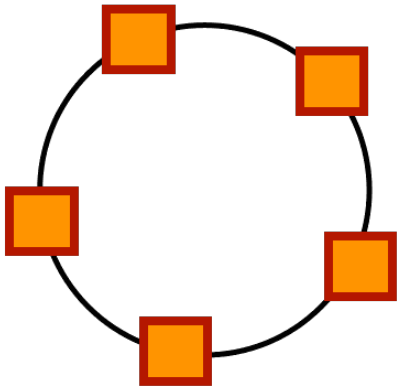
- The nodes form a **ring**
  - the list is finite and each node has a successor
  - in implementation require only one permanent pointer (usually referred to as *tail*)



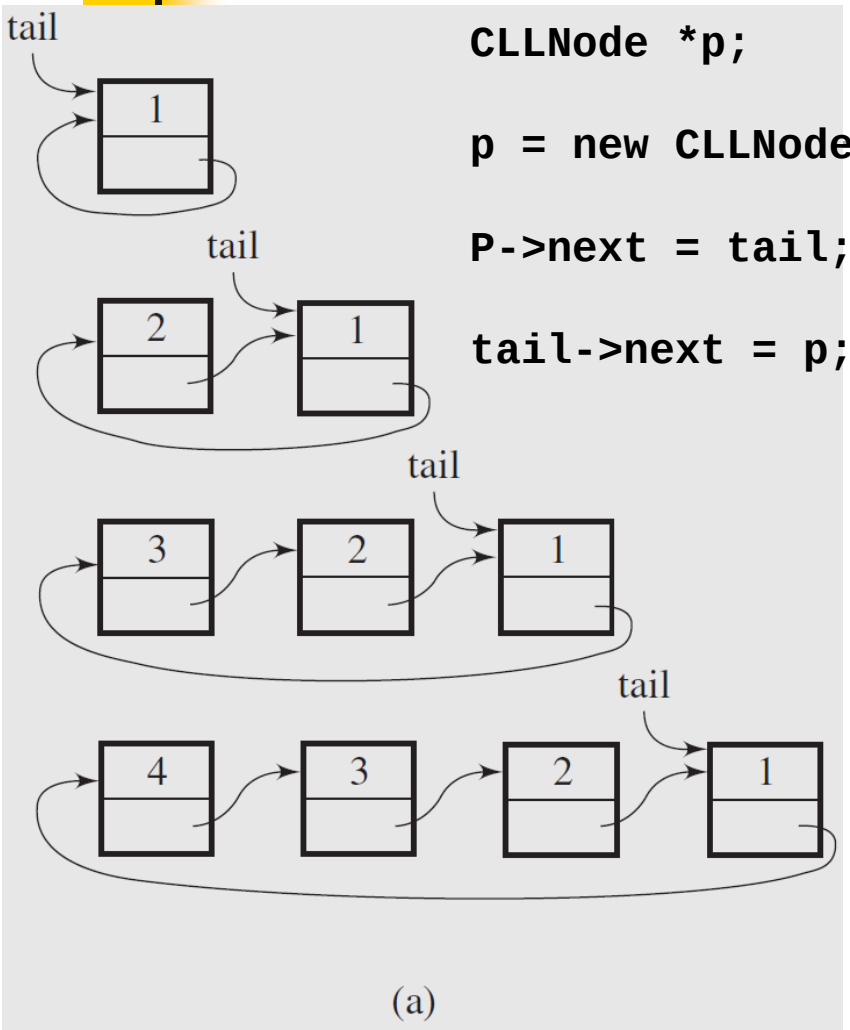


Example:

- several people request to use the same resource for same amount of time
  - each people has a fair share of the resource
  - all people form a circular list
    - after one person used the resource, we move to the next person



Circular Lists (cont.)

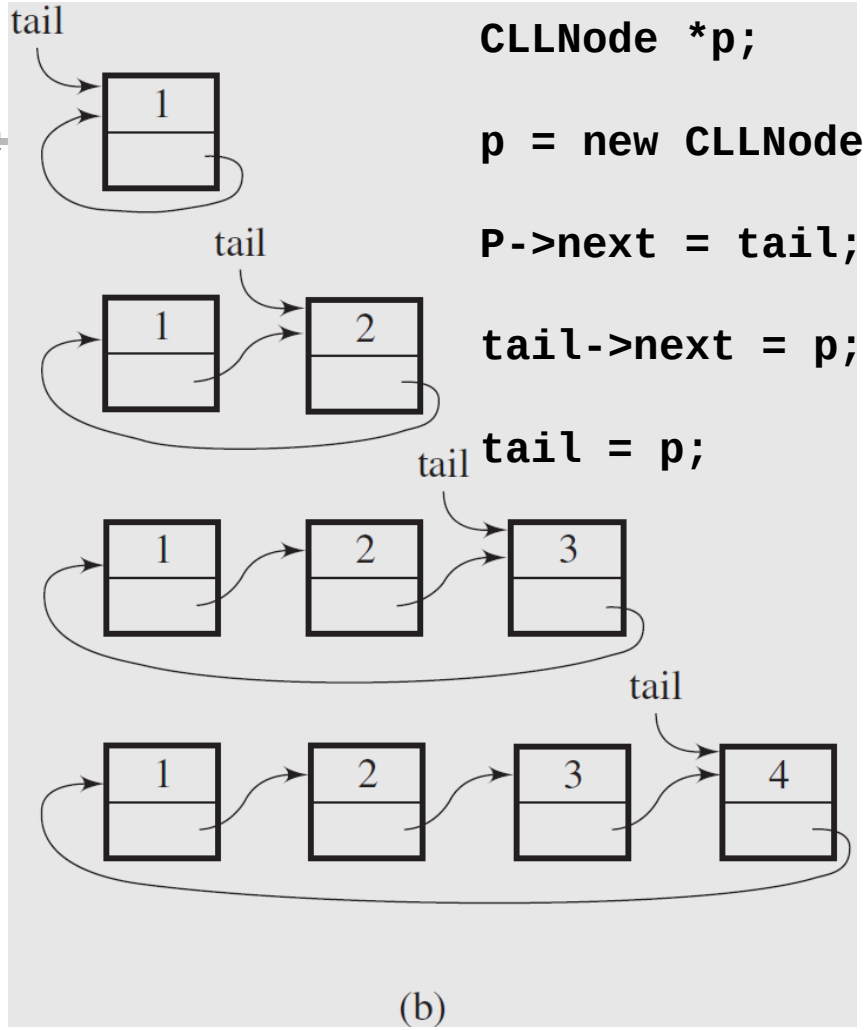



```
CLLNode *p;

p = new CLLNode(2);

P->next = tail;

tail->next = p;
```




```
CLLNode *p;

p = new CLLNode(2);

P->next = tail;

tail->next = p;

tail = p;
```

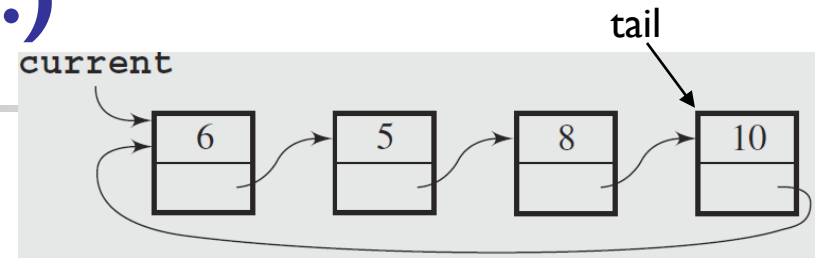
- **Insertions**: at the front and the end of circular lists

Circular Lists (cont.)

- **Insertions** (cont.): at the front and the end of circular lists

```
void addToTail(int el) {
   if (isEmpty()) {
    tail = new IntSLLNode(el);
    tail->next = tail;
   }
   else {
    tail->next = new IntSLLNode(el,tail->next);
    tail = tail->next;
   }
}
```

Circular Lists (cont.)



- A few problems:
  - in deleting last node, require a loop to locate the **predecessor** of the tail node, e.g., similar to singly linked lists
  - operations that require processing the list in **reverse** are going to be inefficient, e.g., directional
- Doubly circular linked list? form two rings
  - going forward through **the next pointers**, and
  - going backwards through **the prev pointers**