Linked Lists

Instructor: Dr. Sunho Lim (Ph.D., Assistant Professor)

Lecture 03

sunho.lim@ttu.edu

Adapted partially from Data Structures and Algorithms in C++, Adam Drozdek, 4th Edition, Cengage Learning; and Algorithms and Data Structures, Douglas Wilhelm Harder, Mmath

Introduction

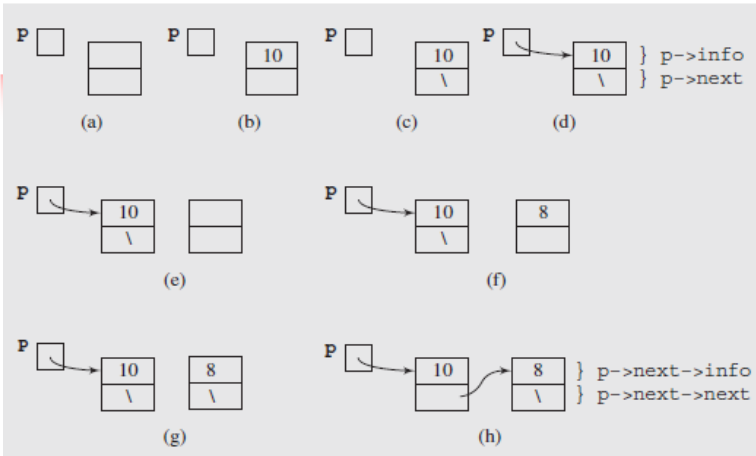
- Limitation of arrays,
  - the size of the array must be known at the time the code is compiled
  - the elements of the array are required potentially extensive shifting when inserting a new element
- ***linked lists***, collections of
  - independent memory locations (***nodes***) that store data
  - links to other nodes
    - the addresses of the nodes
    - follow the links to move between nodes
- Utilize **pointer** to implement linked lists,
  - providing great flexibility

Singly Linked Lists

```
class IntSLLNode {
public:
    IntSLLNode() {
        next = 0;
    }
    IntSLLNode(int i, IntSLLNode *in = 0){
        info = i; next = in;
    }
    int info;
    IntSLLNode *next;
}
```

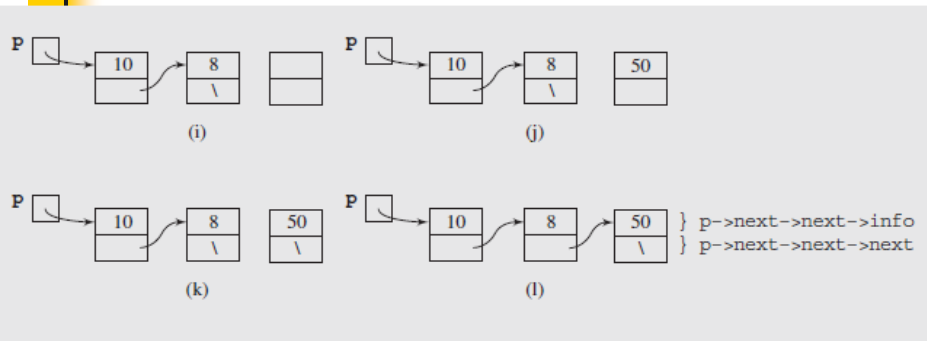
- A node consists of two data members,
  - **Info** - store the node's information content
  - **next** - point to the next node in the list

Singly Linked Lists (cont.)



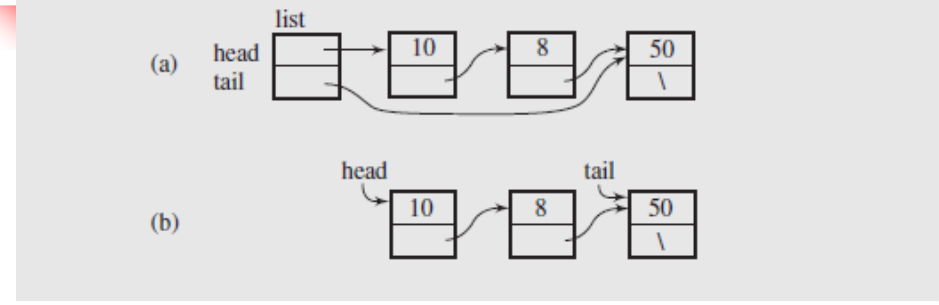
- Each node is composed of …
  - a datum and a link (the address) to the next node in the sequence
- Use the **single variable *p*** (e.g., IntSLLNode *p) to access the entire list;
- Has a **null pointer** (\ ) in the last node in the list

Singly Linked Lists (cont.)



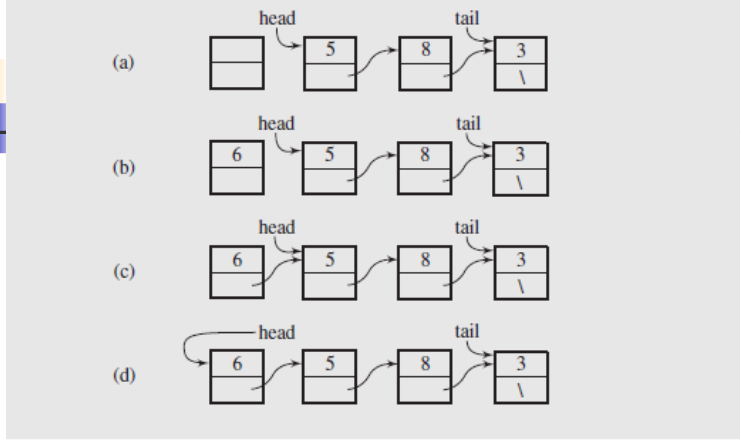
- A disadvantage of single-linked lists:
  - the longer the list, the **longer the chain** of next pointers that need to be followed to a given node
  - reduce flexibility, and is prone to errors
- An alternative, **use an additional pointer** to the end of the list

Singly Linked Lists (cont.)



- Uses two classes:
  - **IntSLLNode**, define the nodes of the list
  - **IntSLList**, define two pointers,
    - **head** and **tail** (e.g., IntSLLNode *head, *tail)

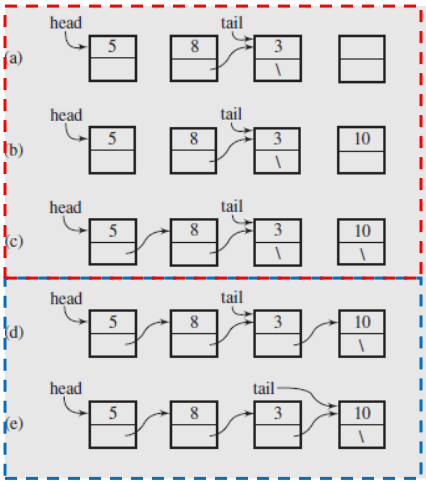
Singly Linked Lists (cont.)



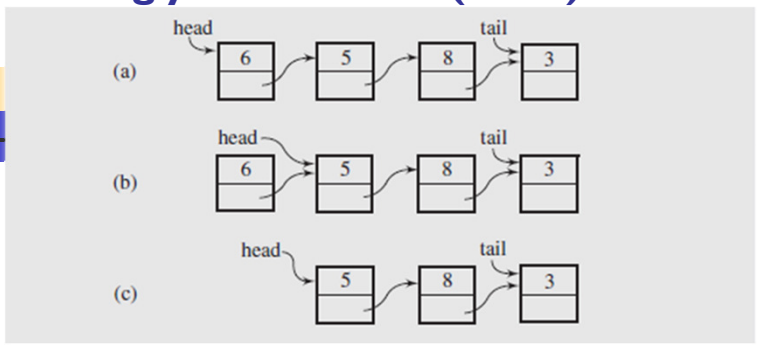
- **Insertion:** a node at the beginning of a list
  - create a new node (figure 3-4a)
  - initialize the info member of the node (figure 3-4b)
  - initialize the next member to point to the first node in the list, which is the current value of head (figure 3-4c)
  - update the head to point to the new node (figure 3-4d)

Singly Linked Lists (cont.)

- **Insertion:** a node at the end of a list
  - create the new node and initialize the info member of the node (figures 3-5a and 3-5b)
  - initialize the next member to null, since the node is at the end of the list (figure 3-5c)
  - set the next member of the current last node to point to the new node (figure 3-5d)
  - Since the new node is now the end of the list, update the tail pointer to point to it (figure 3-5e)
  - if the list is initially empty, both head and tail would be set to point to the new node



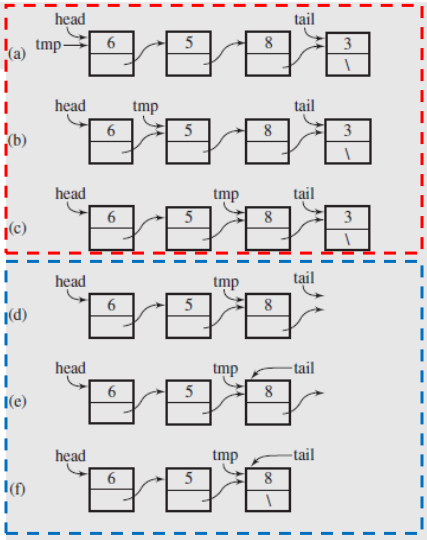
Singly Linked Lists (cont.)



- **Deletion**: at the beginning of the list
  - returning the value stored in the node
  - releasing the memory occupied by the node
  - first retrieve the value stored in the first node (head → info)
  - use a temporary pointer to point to the node, and set head to point to head → next
  - delete the former first node, releasing its memory
  - Note that,
    - when a single node is in the list, requiring that **head and tail** be set to null to indicate the list is now **empty**

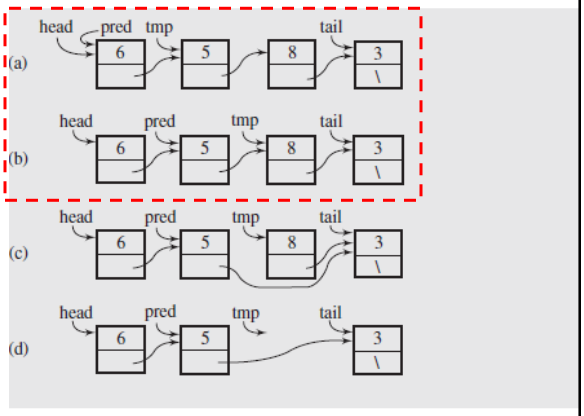
Singly Linked Lists (cont.)

- **Deletion** (cont.): at the end of a list
  - back the tail pointer to the previous node in the list
  - Since this cannot be done directly, need a temporary pointer to traverse the list until tmp → next = tail
  - Once have located that node, retrieve the value contained in tail → info, delete that node, and set tail = tmp



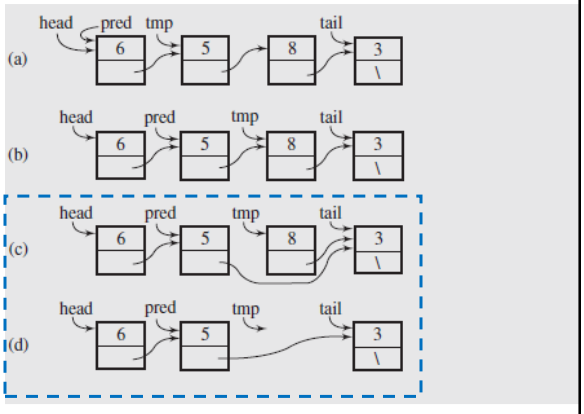
Singly Linked Lists (cont.)

- **Deletion** (cont.): at the middle of a list
  - locate the specific node, then link around it by linking the previous node to the following node
  - need to keep track of the previous node, and need to keep track of the node containing the target value
  - require two extra pointers, **pred** and **tmp**, initialized to the first and second nodes in the list
  - traverse the list until tmp → info matches the target value



Singly Linked Lists (cont.)

- **Deletion** (cont.)
  - set pred → next = tmp → next, "bypasses" the target node, allowing it to be deleted
  - several cases to consider
    - removing a node from an empty list or trying to delete a value that isn't in the list
    - deleting the only node in the list
    - removing the first or last node from a list with at least two nodes

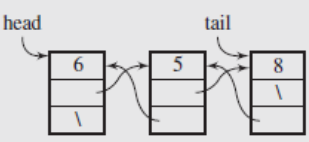


Singly Linked Lists (cont.)

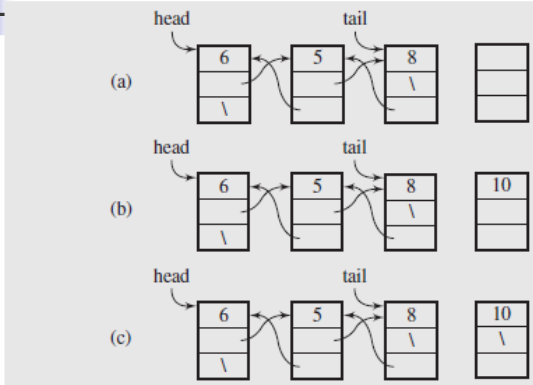
- **Searching**
  - scan a linked list to find a particular data member
  - no modification to the list,
    - use a single temporary pointer
  - traverse the list until
    - the info member of the node tmp points to matches the target, or
    - tmp → next is null
      - reached the end of the list and the search fails

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 previous node
  - ***doubly linked lists***



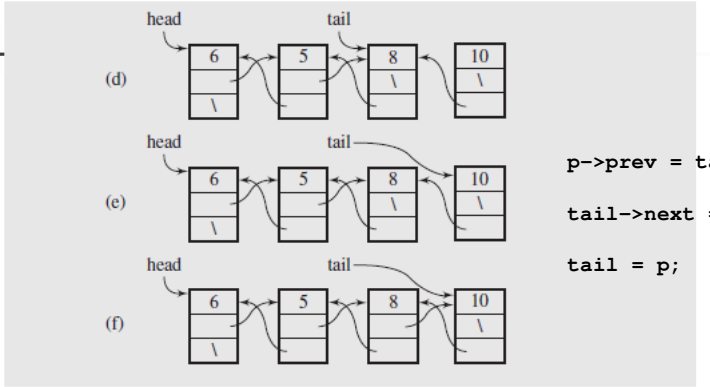
Doubly Linked Lists (cont.)




```
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

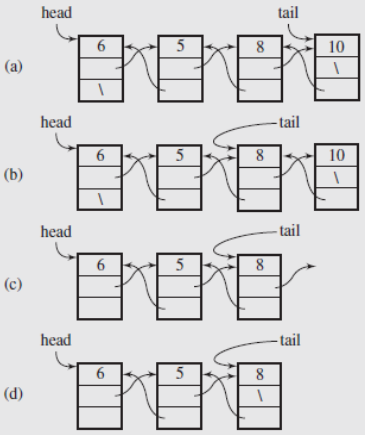
Doubly Linked Lists (cont.)




```
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 tail pointer now to point to this new node
  - set the next member of the previous node to point to the new node

Doubly Linked Lists (cont.)




```
tail = tail->prev;

delete tail->next;

tail->next = 0;
```

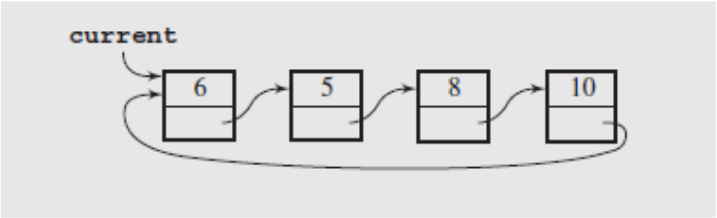
- **Deletion**: at the end of a list
  - a direct link to the previous node in the list
    - **no need to traverse** the list to find the previous node
  - retrieve the data member from the node, then set tail to the node's predecessor

Doubly Linked Lists (cont.)

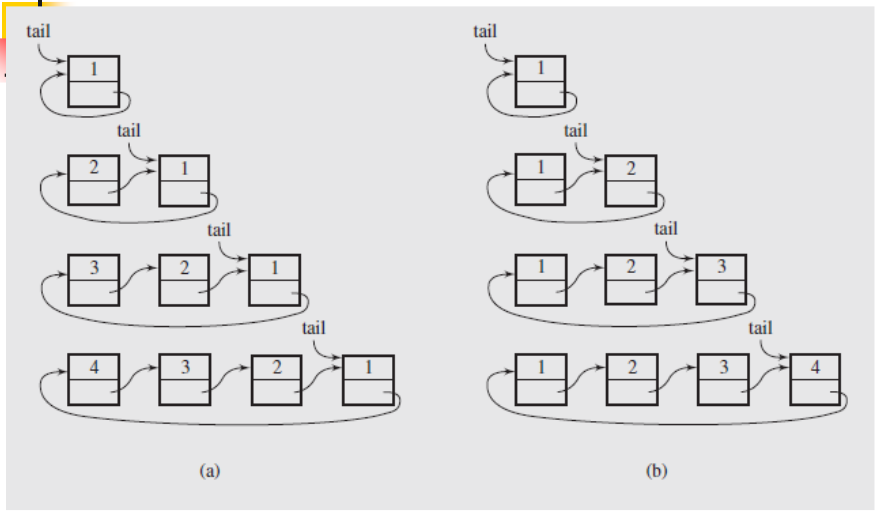
- **Deletion** (cont.)
  - special cases
    - If the node being deleted is the only node in the list,
      - head and tail need to be set to null
    - if the list is empty,
      - an attempt to delete a node should be handled and reported to the user
      - e.g., isEmpty();

Circular Lists

- The nodes form a **ring**
  - require only one permanent pointer (usually referred to as tail)



Circular Lists (cont.)



- **Insertions**: at the front and the end of this type of list

Circular Lists (cont.)

- **Insertions** (cont.): at the front and the end of this type of list

```
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, 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 linked? form two rings
  - going forward through **the next pointers**, and
  - going backwards through **the prev pointers**