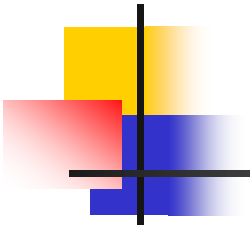
Linked Lists

Lecture 03

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

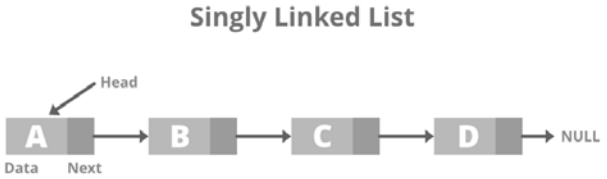


Introduction

- Limitations 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
  - ***nodes*** storing data and links to other nodes
    - 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

Singly linked list: A node has a link only to its successor in this sequence.



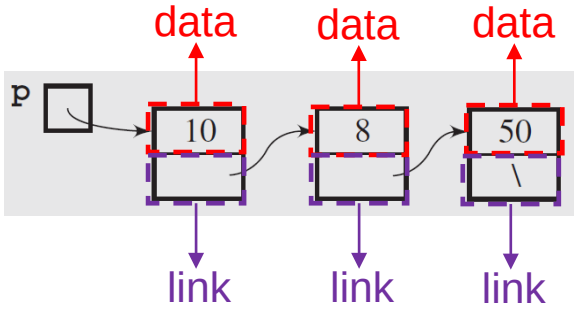

```
class IntSLLNode {
public:
    IntSLLNode() {
        next = 0;
    }
    IntSLLNode(int i, IntSLLNode *in = 0){
        info = i; next = in;
    }
    int info;
    IntSLLNode *next;
}
```

two constructors

Self-referential objects: IntSLLNode is defined in terms of itself

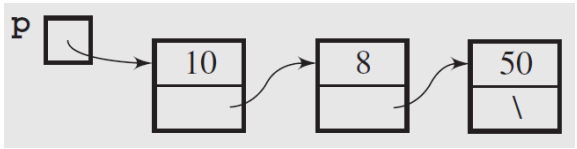
- A node consists of ***two*** data members,
  - **info** - store the node's information content
  - **next** - point to the next node in the list

Singly Linked Lists (cont.)

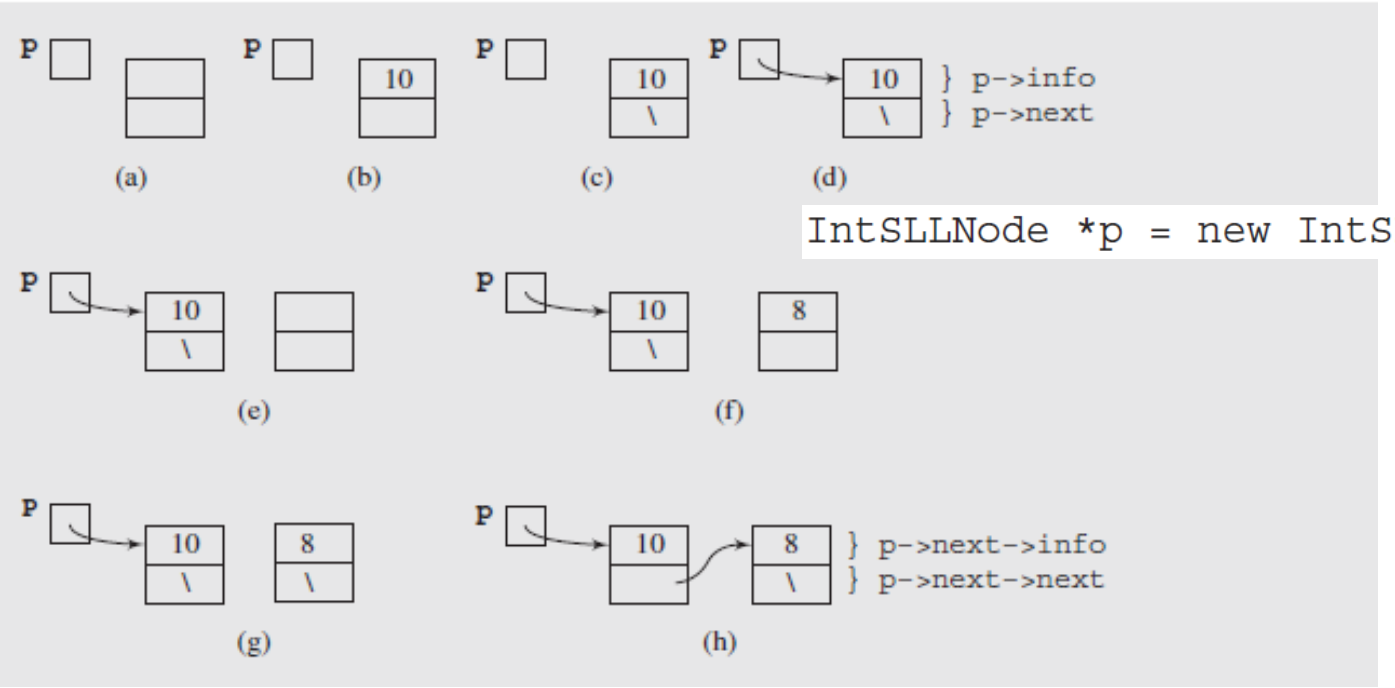


note: the next of last node is a null pointer.

- Each node is composed of …
  - a data 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
- links have "*direction*"
- Let's create the linked list

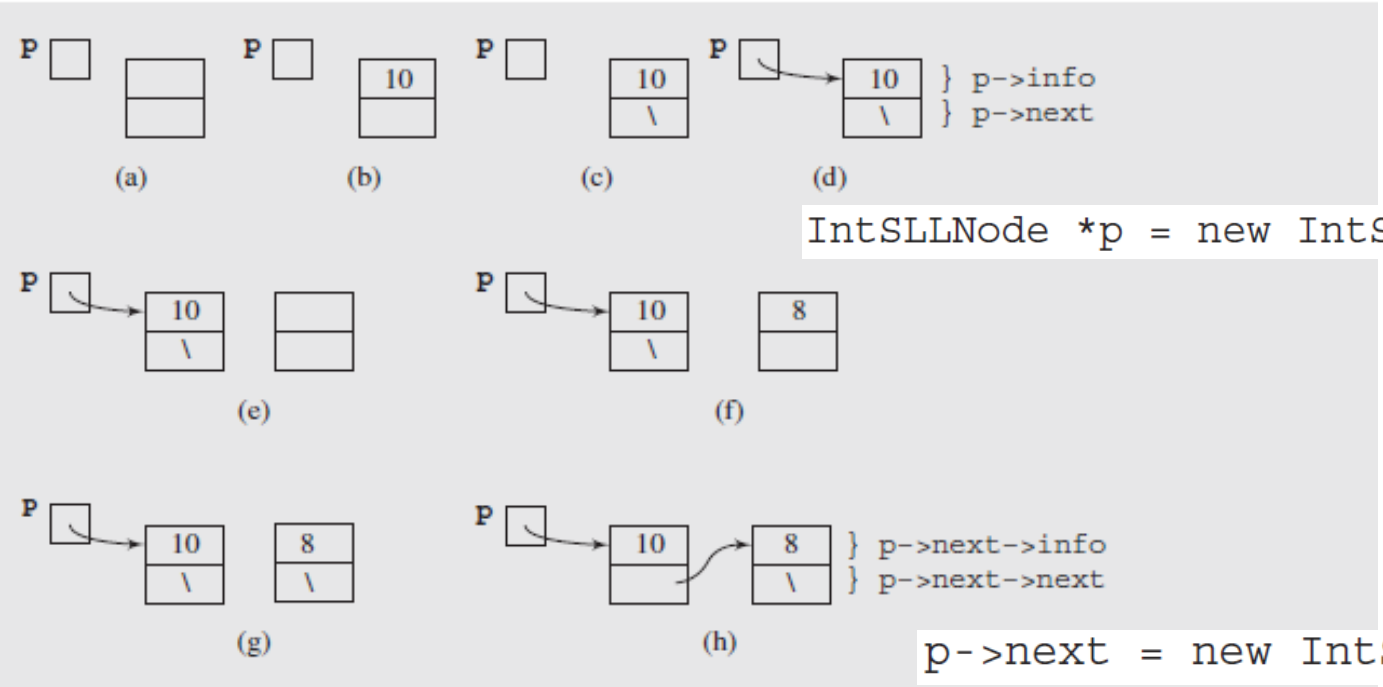


Singly Linked Lists (cont.)



- `IntSLLNode *p = new IntSLLNode(10);`
  - create the first node and make *p* point to this first node
  - four steps:
    - a new *IntSLLNode* and *p* are created (Figure a)
    - set *info* of this new node to 10 (Figure b)
    - set *next* of this new node to null (Figure c)
    - make *p* point to this new node (Figure d)

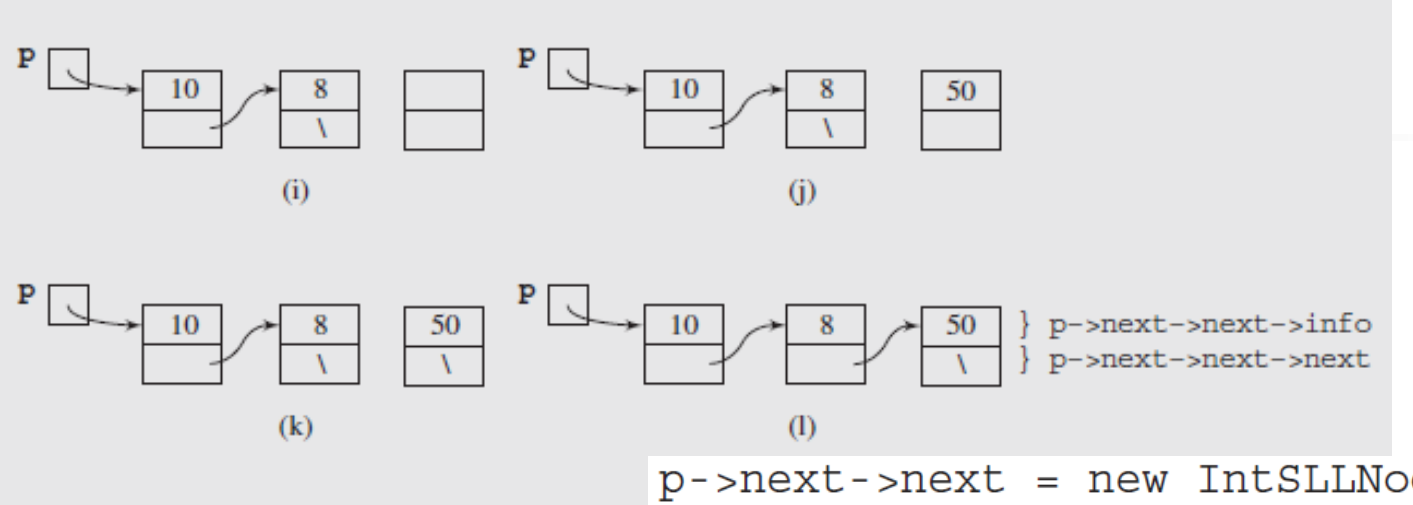
Singly Linked Lists (cont.)



(the *next* member of the node pointed to by *p*)

- `p->next = new IntSLLNode(8);`
  - create the second node and make first node point to second node
  - four steps:
    - a new *IntSLLNode* is created (Figure e)
    - set *info* of this new node to 8 (Figure f)
    - set *next* of this new node to null (Figure g)
    - make the first node point to this second node (Figure h)

Singly Linked Lists (cont.)

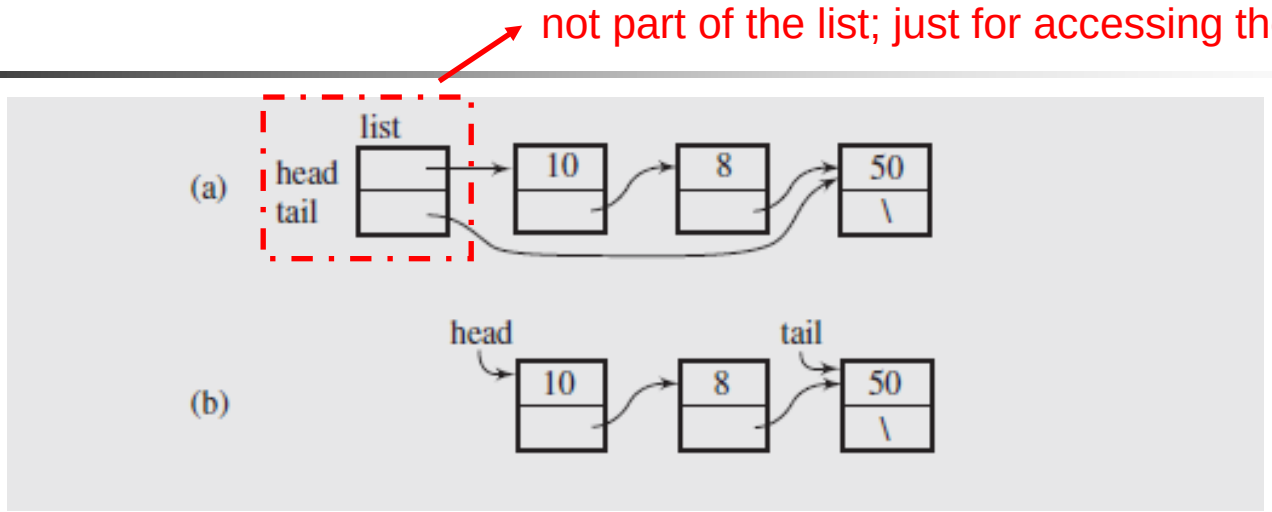



```
p->next->next = new IntSLLNode(50);
```

(the *next* member of the second node in the list)

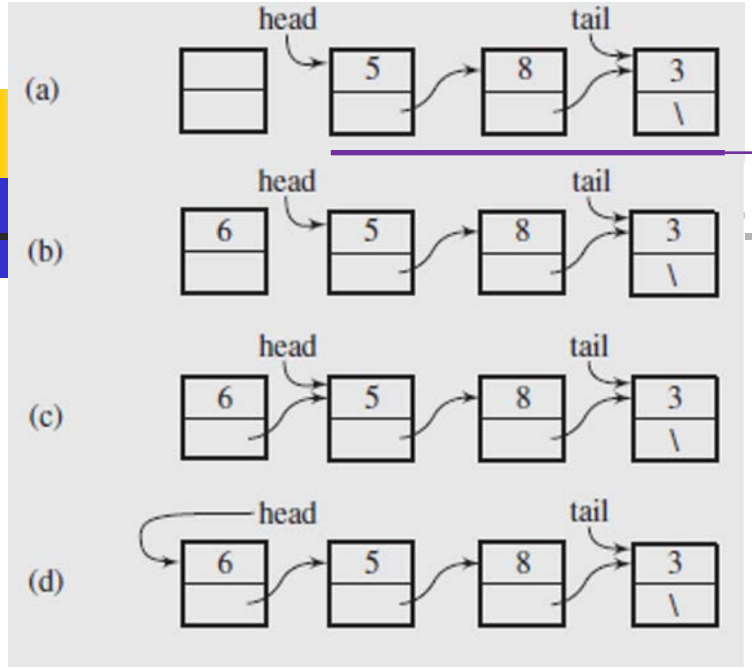
- A *disadvantage* of single-linked lists:
  - the ***longer the list***, the ***longer the chain of next pointers*** that need to be followed to a given node
  - reduce flexibility, and is prone to errors
- An alternative, **use an additional pointer** to the end of the list
  - keep two pointers: one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

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.)



existing linked list
with three nodes

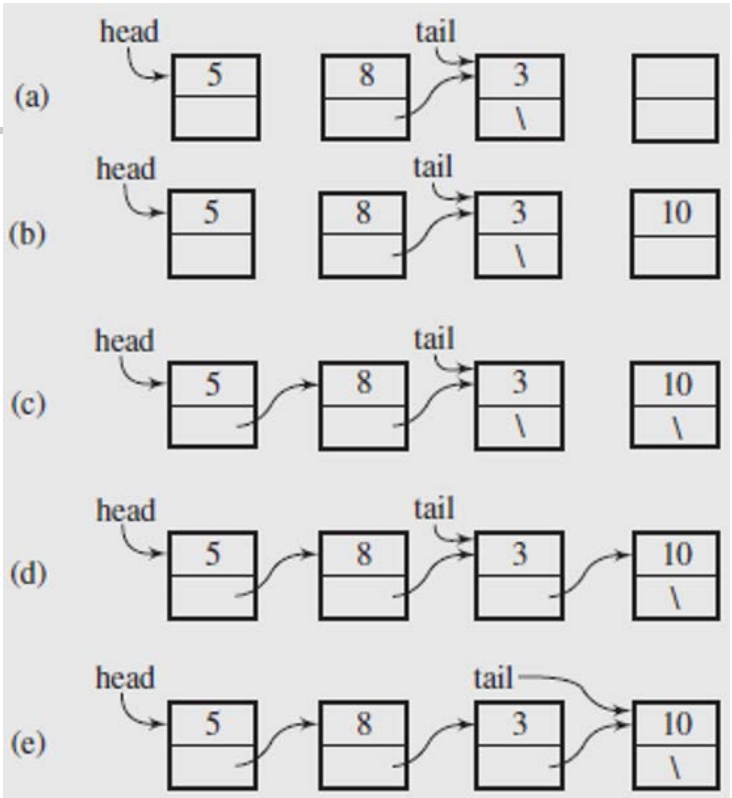
```
void IntSLList::addToHead(int el) {
    head = new IntSLLNode(el,head);
    if (tail == 0)
        tail = head;
}
```

```
head = new IntSLLNode(el,head);
```

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

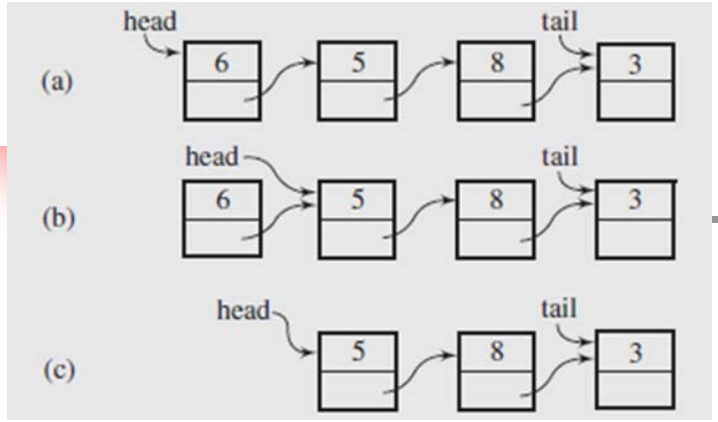
Singly Linked Lists (cont.)

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




```
void IntSLList::addToTail(int el) {
    if (tail != 0) {      // if list not empty;
        tail->next = new IntSLLNode(el);
        tail = tail->next;
    }
    else head = tail = new IntSLLNode(el);
}
```

Singly Linked Lists (cont.)




```
int IntSLList::deleteFromHead() {
    int el = head->info;
    IntSLLNode *tmp = head;
    if (head == tail)     // if only one node on the list;
        head = tail = 0;
    else head = head->next;
    delete tmp;
    return el;
}
```

- **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 (info) stored in the first node
  - use a temporary pointer to point to the first node
  - 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 list
    - *this cannot be done directly*
  - need a temporary pointer *tmp* 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

```
int IntSLList::deleteFromTail() {
    int el = tail->info;
    if (head == tail) {    // if only one node on the list;
        delete head;
        head = tail = 0;
    }
    else {                 // if more than one node in the list,
        IntSLLNode *tmp; // find the predecessor of tail;
        for (tmp = head; tmp->next != tail; tmp = tmp->next);
        delete tail;
        tail = tmp;        // the predecessor of tail becomes tail;
        tail->next = 0;
    }
    return el;
}
```