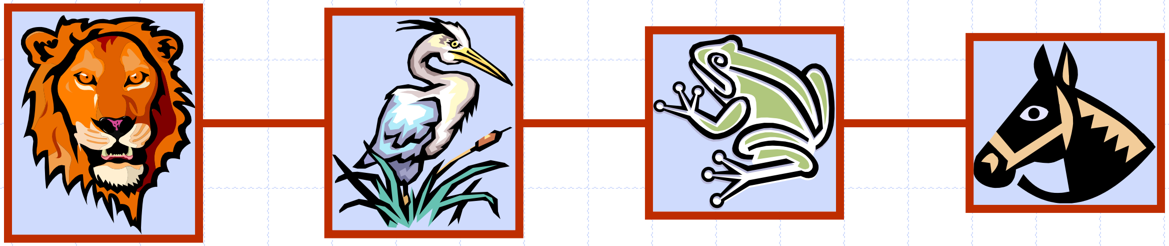
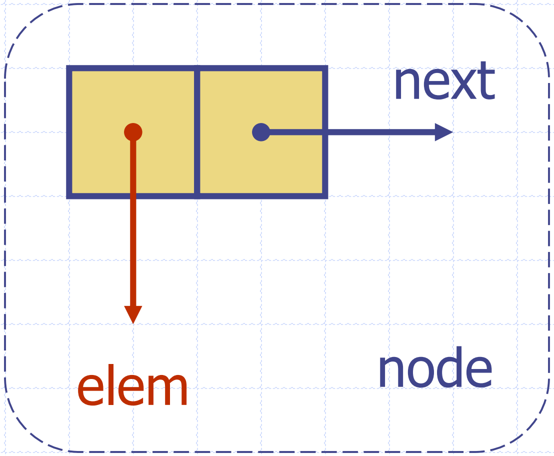
Linked Lists

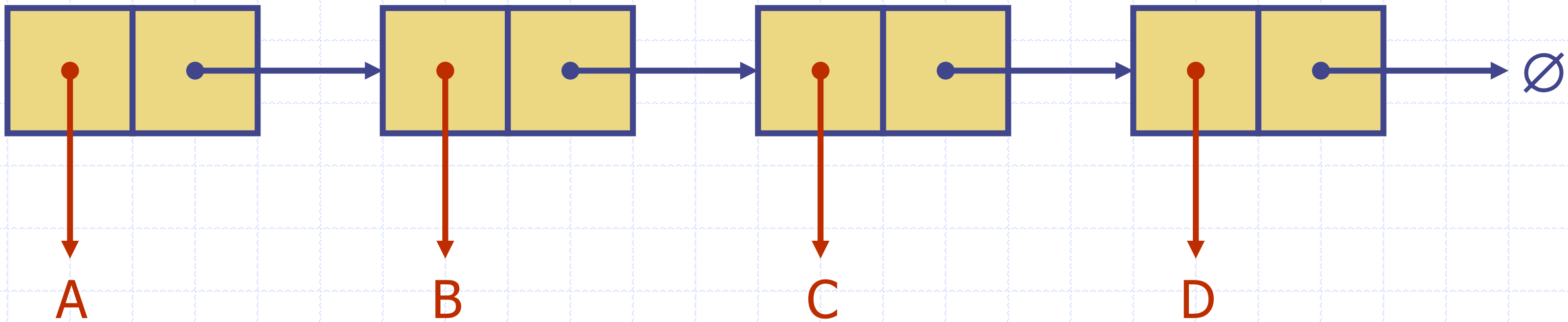
Sections 3.2 – 3.4



Singly Linked List

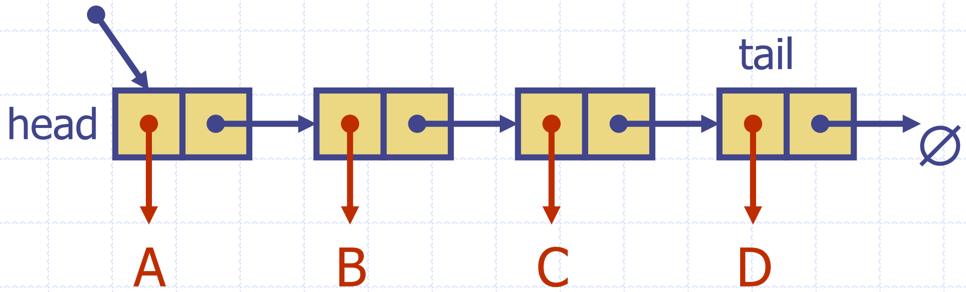
- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element or pointer to element
  - link to the next node



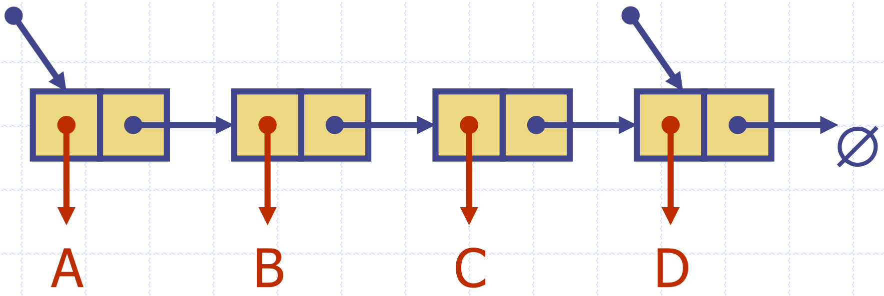


(Singly) Linked List

- The first node of a linked list is called the head.
- The last node of a linked list is called the tail.
- When storing a linked list, we keep a pointer to the head.

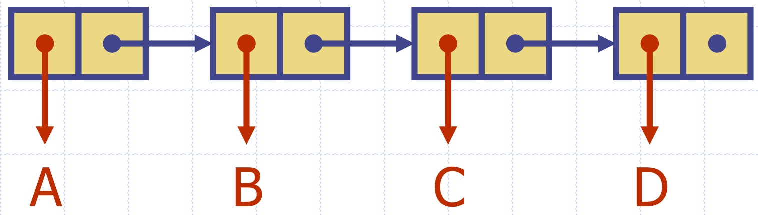


- Sometimes we also keep a pointer to the tail.



Linked List

- Traversing (moving from one node to another) in a linked list is called link hopping or pointer hopping.
- Sometimes null pointers are denoted with a pointer to a symbol ∅, and sometimes with just a dot.



- The tail is easily identified as the node with the null next pointer.
- A linked list, like an array, maintains its elements in a certain order. Unlike an array, it has no predetermined fixed size.

Linked List Implementation

```
class StringNode {
private:
    string elem;
    StringNode *next;

    friend class StringLinkedList;
}
```

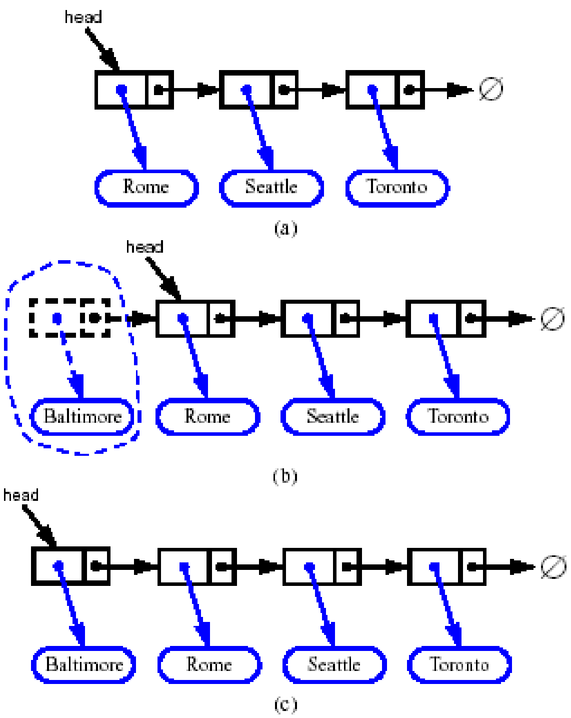
```
class StringLinkedList {
public:
    StringLinkedList();
    ~StringLinkedList();
    bool empty() const;
    const string& front() const;
    void addFront(const string& e);
    void removeFront();
private:
    StringNode* head;
}
```

Linked List Implementation

```
StringLinkedList::StringLinkedList()
    : head(NULL) { }

StringLinkedList::~StringLinkedList()
    { while (!empty())  removeFront(); }

bool StringLinkedList::empty() const
    { return head == NULL; }

const string& front() const
    { return head->elem; }
```

Inserting at the Head

1. Allocate a new node
2. Insert new element
3. Have new node point to old head
4. Update head to point to new node

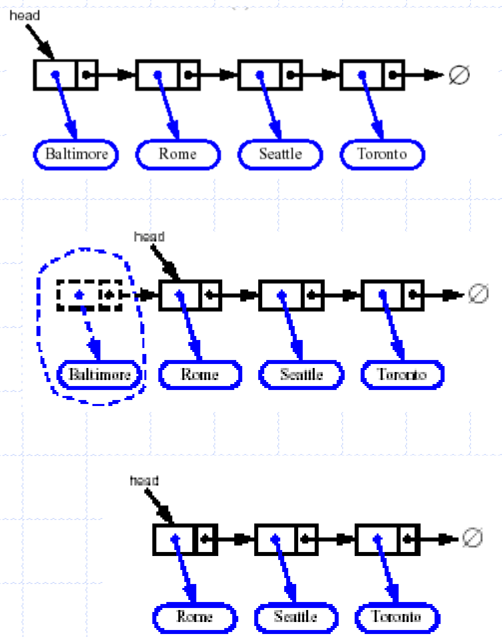


Inserting at the Head

```
void StringLinkedList::addFront(const string& e) {
    StringNode* v = new StringNode;
    v->elem = e;
    v->next = head;
    head = v;
}
```

Removing at the Head

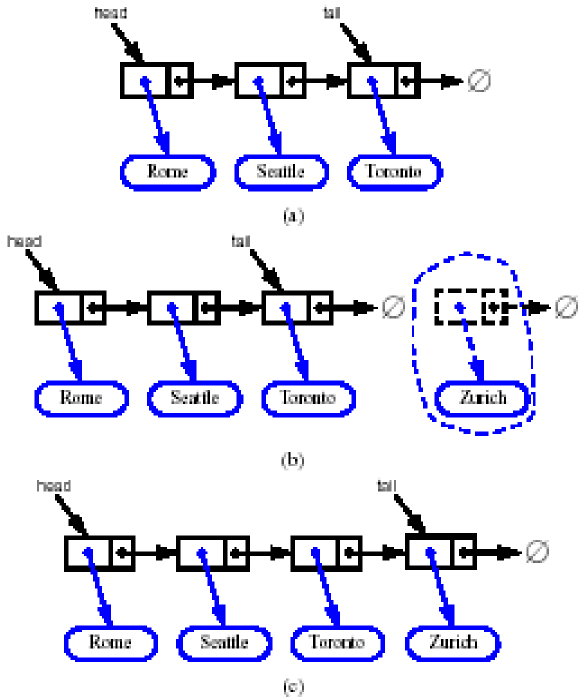
1. Update head to point to next node in the list
2. Delete the former first node




```
void StringLinkedList::removeFront()  {
    StringNode* old = head;
    head = old->next;
    delete old;
}
```

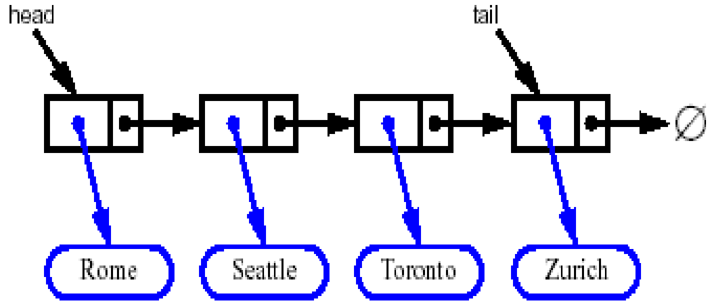
Inserting at the Tail

1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node



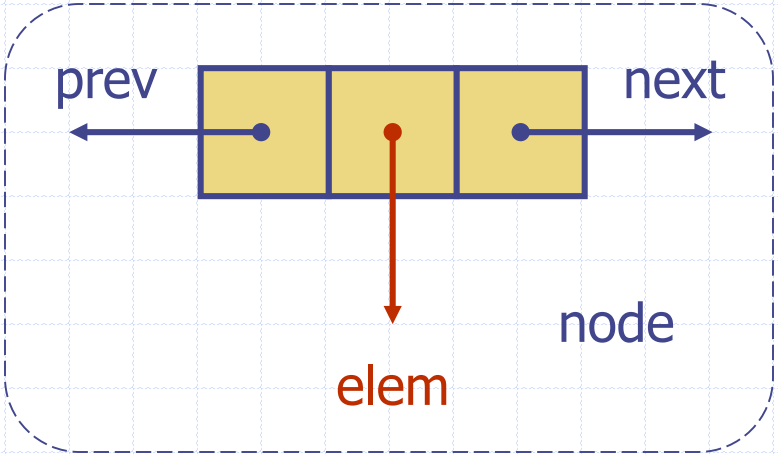
Removing at the Tail

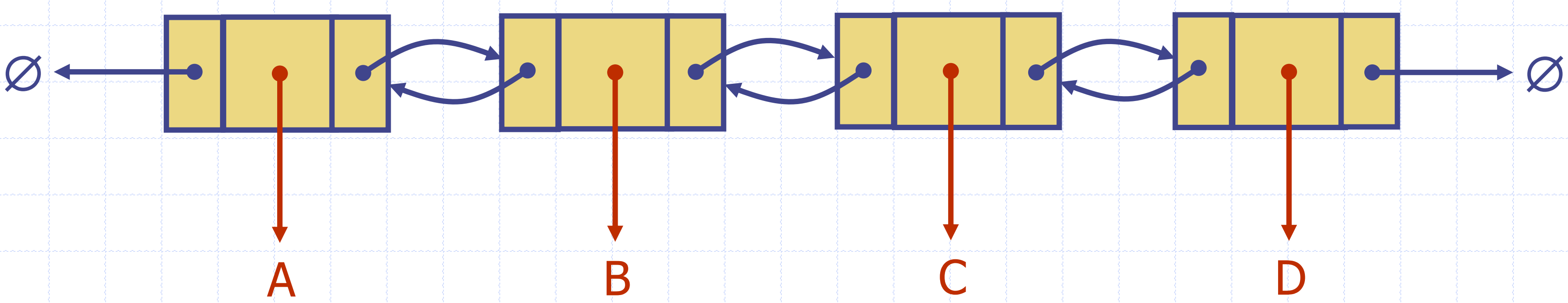
- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node



Doubly Linked Lists

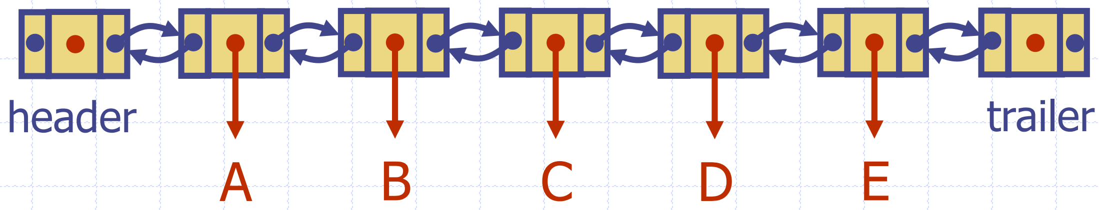
- We get much more flexibility by adding a predecessor link to each node, at the cost of almost doubling the overhead.





Sentinels

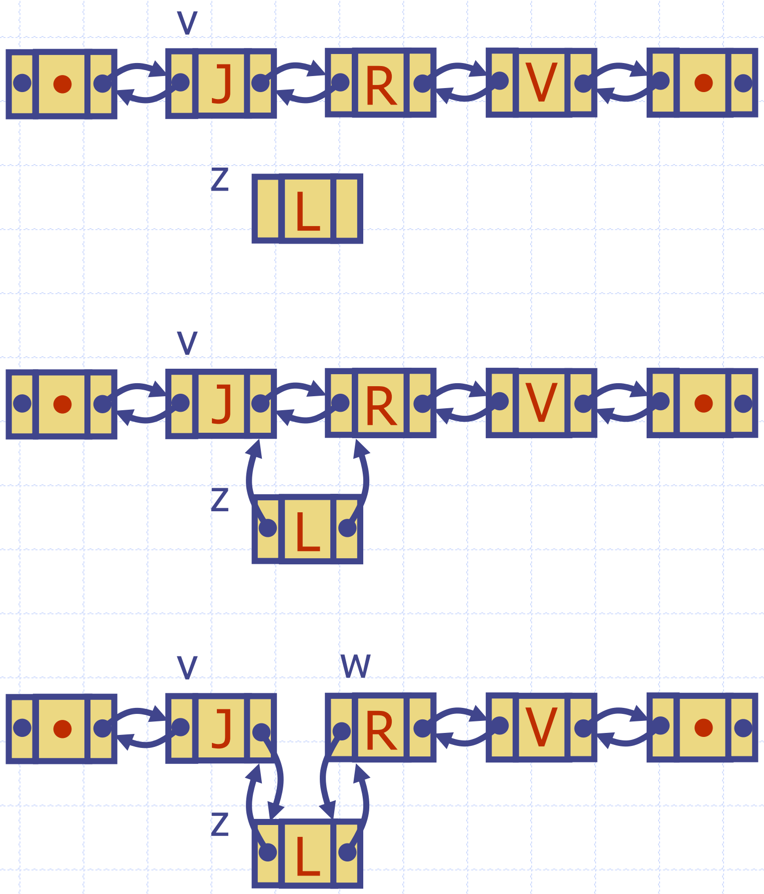
- Oftentimes we add sentinel (a.k.a. dummy) nodes to the beginning and end of a doubly linked list.
- These are called the header and the trailer.
- Sentinels simplify programming; with them, a real list node always has a non-null prev and next.



Inserting after a node v (Linking in)

1. Allocate a new node z
2. Insert new element
3. Make z's prev link point to v
4. Make z's next link point to w = v->next
5. Make w's prev link point to z
6. Make v's next link point to z

linking in



Inserting after a node v

- Text gives code for inserting before node v; this code is for inserting after.

```
void DLinkedList::add(DNode* v, const Elem& e) {
    DNode* z = new DNode;
    z->elem = e;
    z->prev = v;
    z->next = v->next;
    v->next->prev = z;
    v->next = z;
}
```

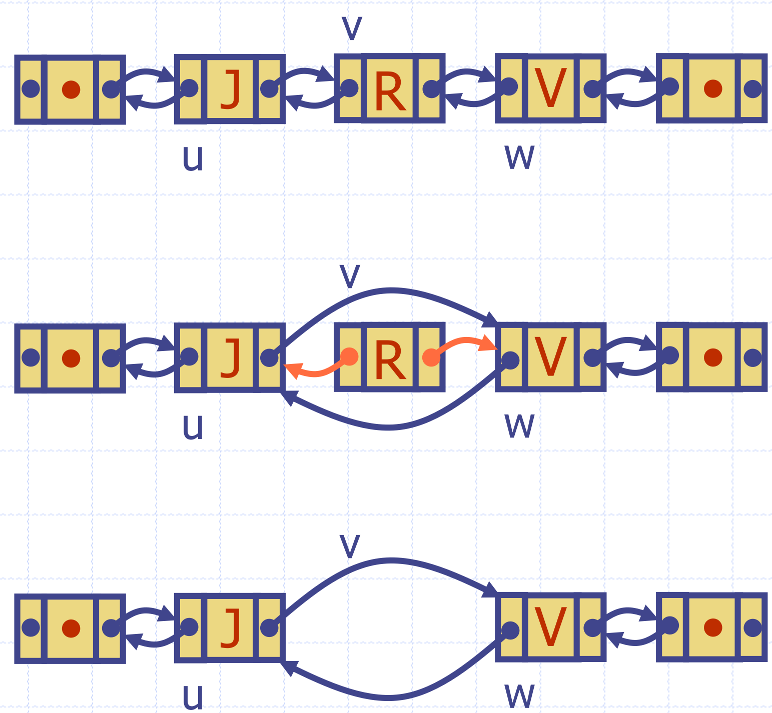
```
void DLinkedList::addFront(const Elem& e)
    { add(header, e); }

void DLinkedList::addBack(const Elem& e)
    { add(trailer->prev, e); }
```

Deleting a node v (Linking out)

Let u be the node before v, and w be the node after.

linking out

1. Make w's prev link point to u
2. Make u's next link point to w
3. Delete node v



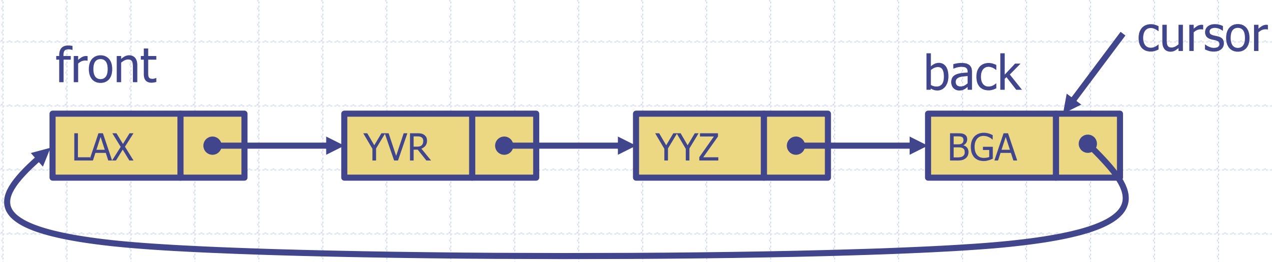
Deleting a node v

```
void DLinkedList::remove(DNode* v) {
    DNode* u = v->prev;
    DNode* w = v->next;
    u->next = w;
    w->prev = u;
    delete v;
}
```

```
void DLinkedList::removeFront()  {
    if (header->next == trailer)
        throw new RemoveFromEmptyListException("msg1");
    remove(header->next);
}

void DLinkedList::removeBack()  {
    if (trailer->prev == header)
        throw new RemoveFromEmptyListException("msg2");
    remove(trailer->prev);
}
```

Circularly Linked Lists

- For a circularly linked list, we use the same kind of nodes as a singly linked list.
- However, the "last" node of the list doesn't have a null next pointer, but rather a pointer to the "first" node.
- We keep a pointer to a node of the list, called the cursor.
- The node the cursor points to is called the back of the list; the next node is called the front.



Reversing a Doubly Linked List

- First approach: copy input list L into temporary list T in reverse order, then copy T back to L (without reversing).
- To get the reversed copy, repeatedly remove the first element of L and copy it to the front of T.
- To get the non-reversed copy, repeatedly remove the first element of T and copy it to the back of L.

```
void listReverse(DLinkedList& L)
{
    DLinkedList T;
    while (!L.empty()) {
        string s = L.front();
        L.removeFront();
        T.addFront(s);
    }
    while (!T.empty()) {
        string s = T.front();
        T.removeFront();
        L.addBack(s);
    }
}
```

Reversing a Doubly Linked List

- Second approach: return the reversed list; empty the input list in the process.

- How could you do this without emptying the input list? (Hint: it's much easier if this is a member function of DLinkedList)

```
DLinkedList* reverse(DLinkedList& L) {
    DLinkedList* T = new DLinkedList;
    while (!L.empty()) {
        string s = L.front();
        L.removeFront();
        T->addFront(s);
    }
    return T;
}
```