Doubly Linked List

Dr. Rajni Bala

Disadvantages of SLL

- Removing an element at the tail of a singly linked list is not easy. Indeed, it is time consuming to remove any node other than the head in a singly linked list, since we do not have a quick way of accessing the node immediately preceding the one we want to remove.
- There are many applications where we do not have quick access to such a predecessor node. For such applications, it would be nice to have a way of going both directions in a linked list.

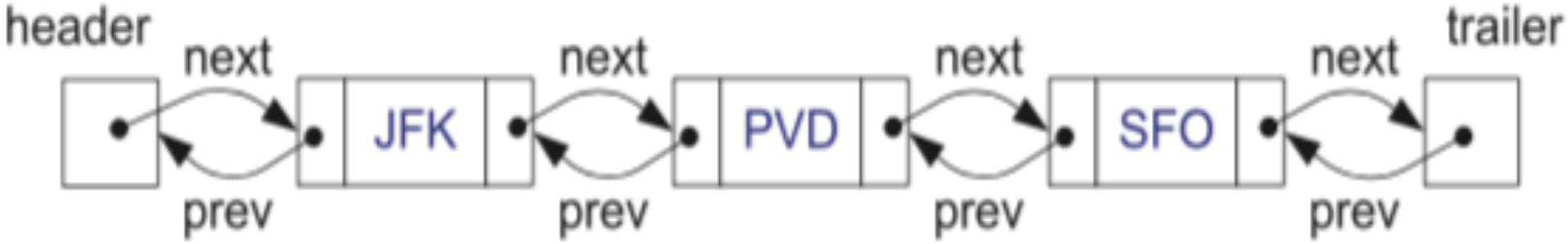
Doubly Linked List

- There is a type of linked list that allows us to go in both directions—forward and reverse—in a linked list. It is the ***doubly linked*** list.
- In addition to its element member, a node in a doubly linked list stores two pointers, a *next* link and a *prev* link, which point to the next node in the list and the previous node in the list.
- Such lists allow for a great variety of quick update operations, including efficient insertion and removal at any given position.

Doubly Linked List

- To simplify programming, it is convenient to add special nodes at both ends of a doubly linked list.
- a ***header*** node just before the head of the list, and a ***trailer*** node just after the tail of the list.
- These “dummy” or ***sentinel*** nodes do not store any elements. They provide quick access to the first and last nodes of the list. In particular, the header’s *next* pointer points to the first node of the list, and the *prev* pointer of the trailer node points to the last node of the list.

Doubly linked list



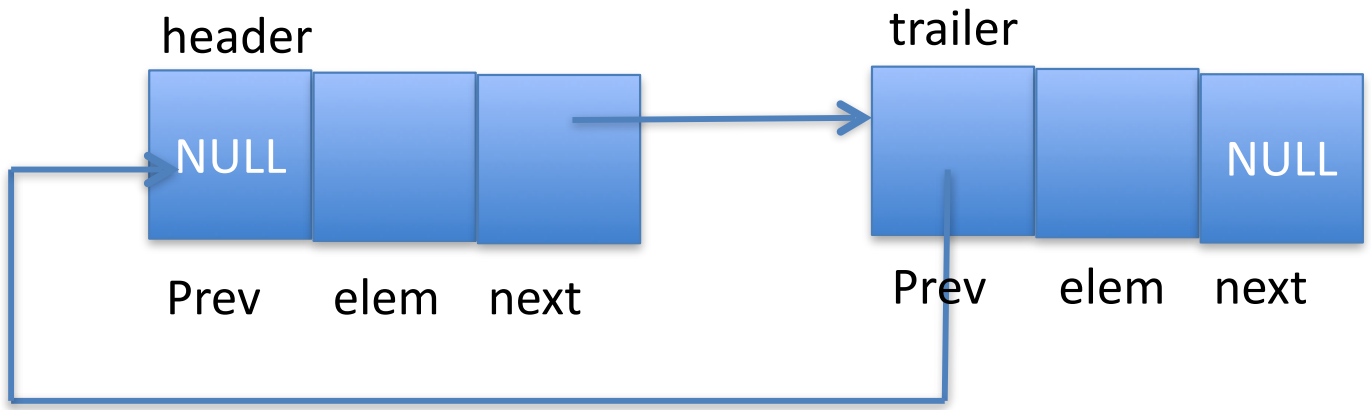
Doubly linked list node

```
typedef string Elem;                    // list element type
class DNode {                           // doubly linked list node
private:
  Elem elem;                            // node element value
  DNode* prev;                          // previous node in list
  DNode* next;                          // next node in list
  friend class DLinkedList;             // allow DLinkedList access
};
```

Code Fragment 3.22: C++ implementation of a doubly linked list node.

Doubly linked List

- Initially when the list will be created it will have 2 nodes header and trailer. Next of header will point to trailer. prev of trailer will point to header



C++ class for doubly linked list

```
class DLinkedList {                          // doubly linked list
public:
  DLinkedList();                             // constructor
  ~DLinkedList();                            // destructor
  bool empty() const;                        // is list empty?
  const Elem& front() const;                 // get front element
  const Elem& back() const;                  // get back element
  void addFront(const Elem& e);              // add to front of list
  void addBack(const Elem& e);               // add to back of list
  void removeFront();                        // remove from front
  void removeBack();                         // remove from back
private:                                     // local type definitions
  DNode* header;                             // list sentinels
  DNode* trailer;
protected:                                   // local utilities
  void add(DNode* v, const Elem& e);         // insert new node before v
  void remove(DNode* v);                     // remove node v
};
```

Code Fragment 3.23: Implementation of a doubly linked list class.

Class Constructor

```
DLinkedList::DLinkedList() {                    // constructor
  header = new DNode;                           // create sentinels
  trailer = new DNode;
  header->next = trailer;                       // have them point to each other
  trailer->prev = header;
}

DLinkedList::~DLinkedList() {                   // destructor
  while (!empty()) removeFront();               // remove all but sentinels
  delete header;                                // remove the sentinels
  delete trailer;
}
```

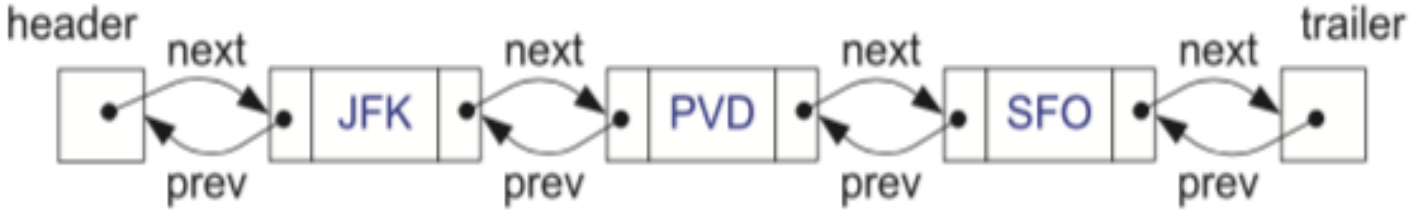
Code Fragment 3.24: Class constructor and destructor.

Accessor functions for DLL

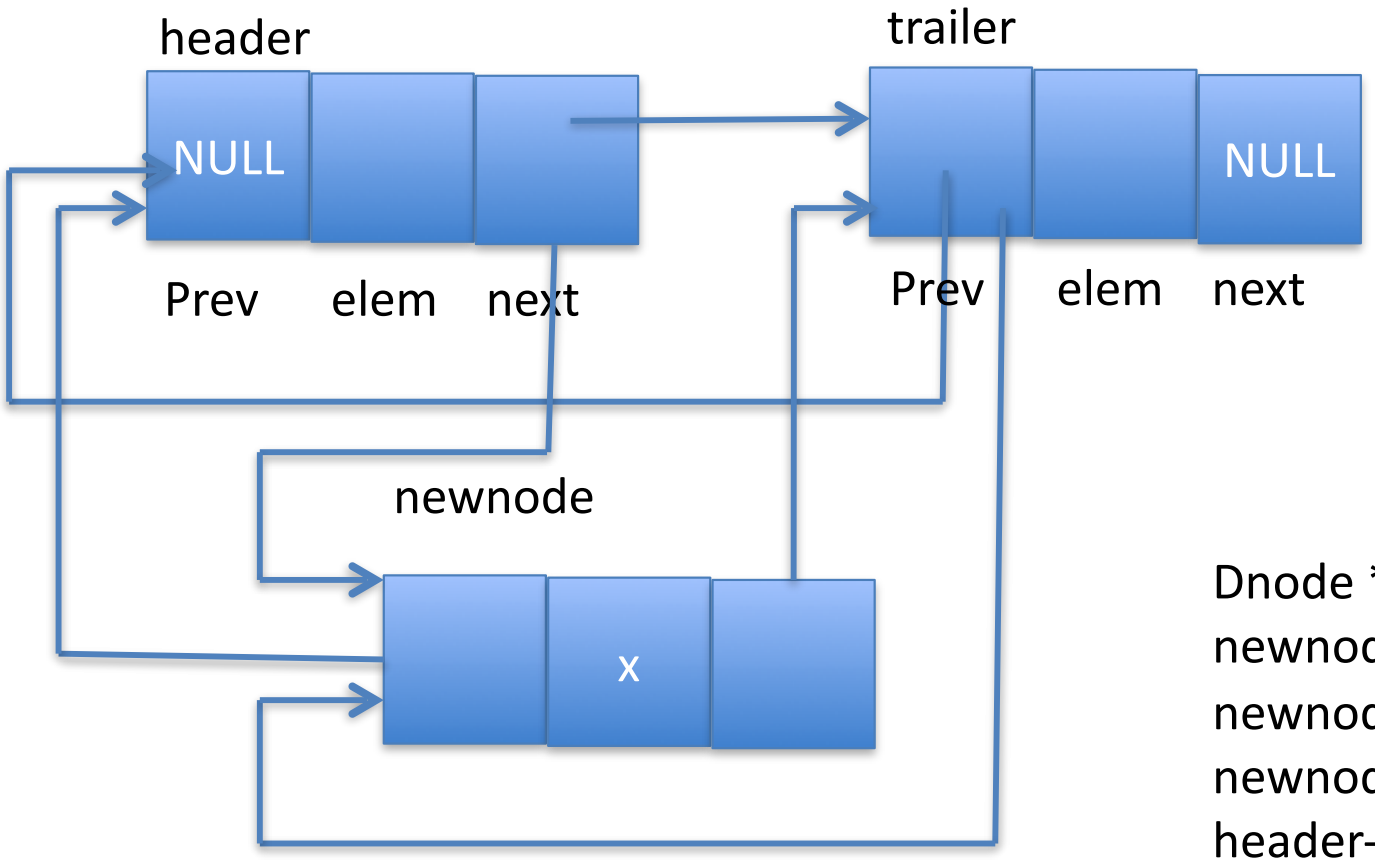
```
bool DLinkedList::empty() const  // is list empty
{ return (header->next == trailer);
 }

 const Elem& DLinkedList::front() const  // get the front element
  {      if (empty()) throw  “list is empty”;
         return (header->next ->elem) ;
 }

 const Elem& DLinkedList::back() const  // get the last element
  {      if (empty()) throw  “list is empty”;
         return (trailer->prev ->elem) ;
 }
```

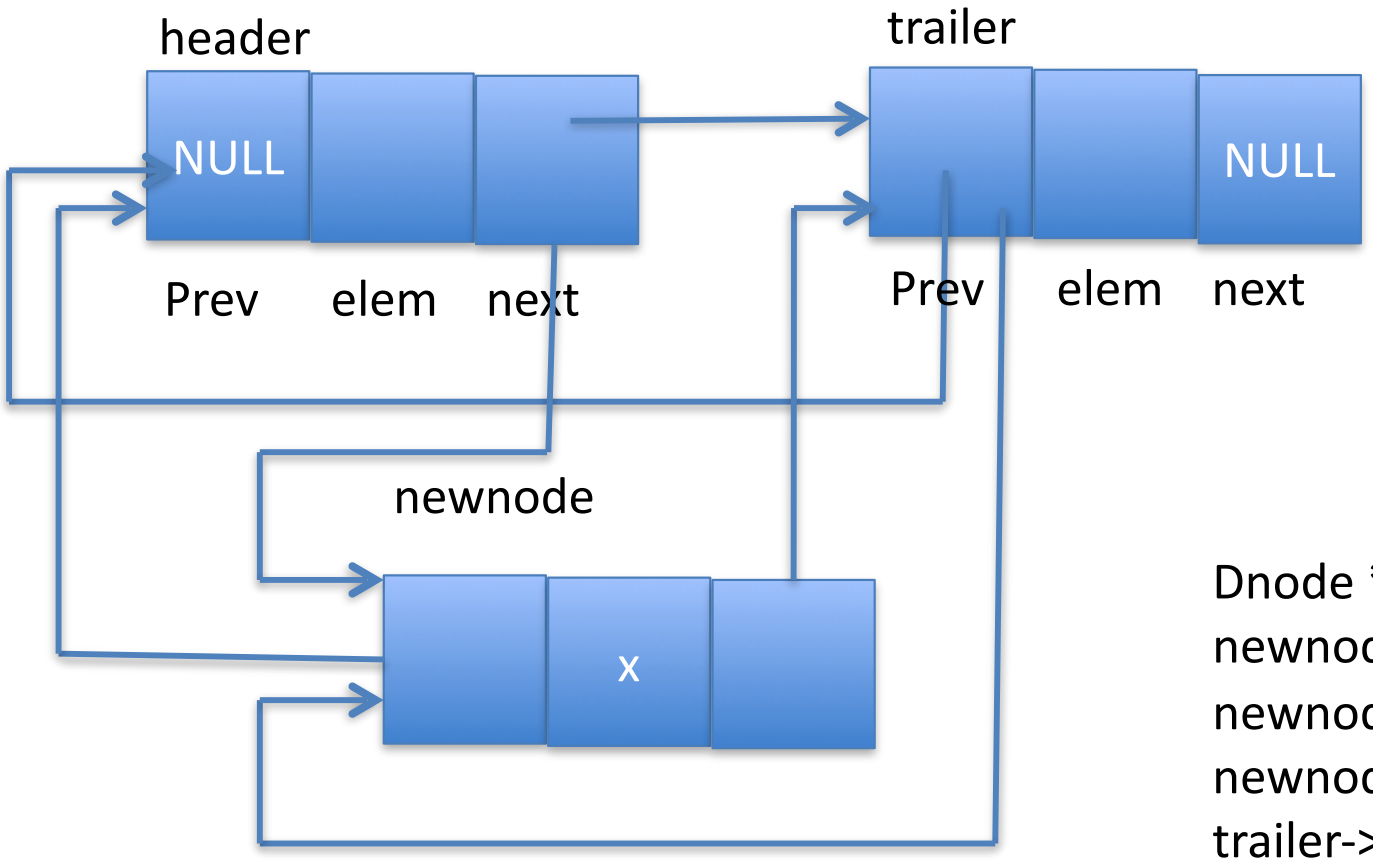


addFront



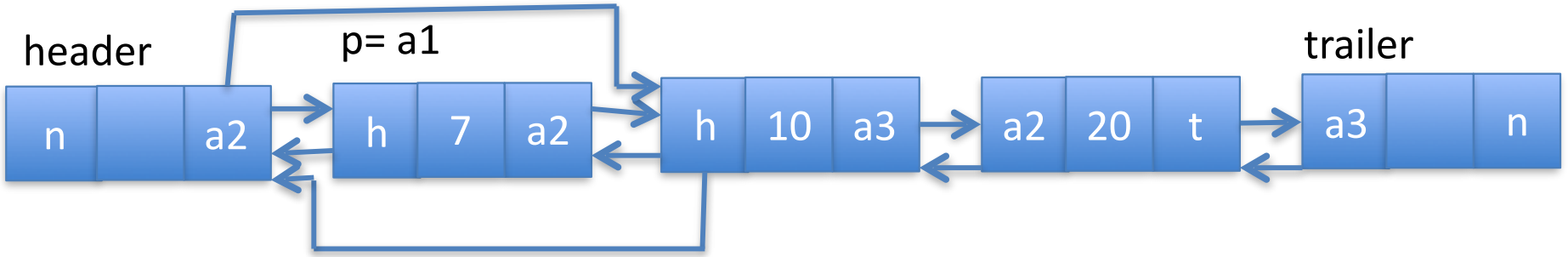

```
Dnode *newnode =new Dnode;
newnode->elem = x;
newnode->prev= header;
newnode-next = header->next;
header->next->prev = newnode;
header->next = newnode;
```

addBack



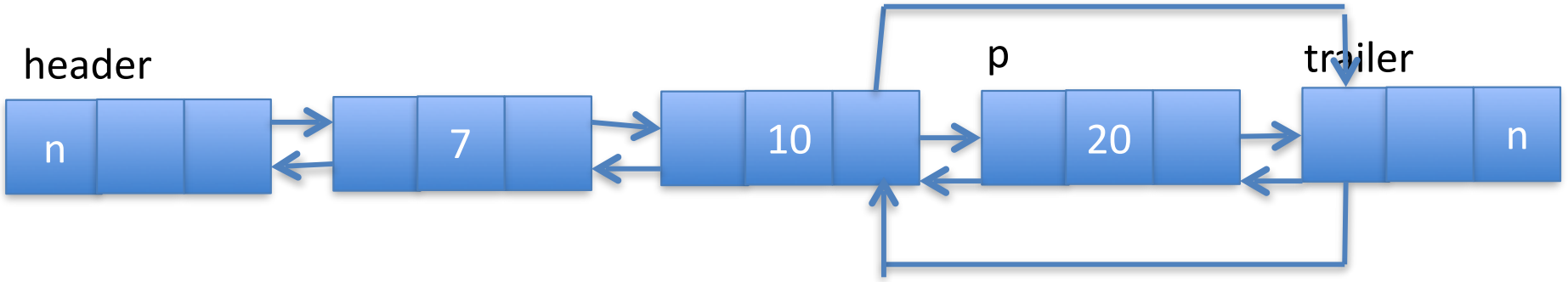

```
Dnode *newnode =new Dnode;
newnode->elem = x;
newnode->prev= trailer->prev;
newnode-next = trailer;
trailer->prev->next = newnode;
trailer->prev = newnode;
```

removeFront




```
if (empty()) throw “List is empty”;
Dnode *p = header->next;
header->next = p->next;
p->next->prev = p->prev;
delete p;
```

removeBack




```
if (empty()) throw “List is empty”;
Dnode *p = trailer->prev;
trailer->prev = p->prev;
p->prev->next = p->next;
delete p;
```