Linear ADT : Linked List

Atul Gupta

Arrays vs. Linked List

+ Simple and easy to use	- Not so simple
+ Faster Access to elements (Constant time random access)	- Sequential access
- Fixed size	+ Dynamic size
- Inefficient insertions and deletions	+ Efficient insertions and deletions

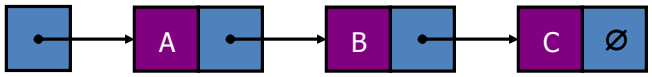
Array vs. Linked List

- Linked list
  - Insertion / deletion = O(1)
  - Indexing = O(n)
  - Easy to dynamically increase size of list, merge
- Array
  - Insertion / deletion = O(n)
  - Indexing = O(1)
  - Compact, uses less space
  - Better cache locality

Linked List Interface

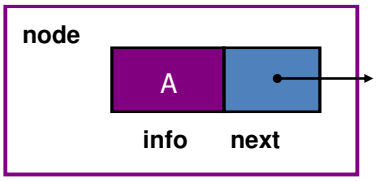
- Typical operations
  - void insert(x, L) : Insert element x into L in the last
  - void insertFirst(x, L) : Insert element x into L at the first place
  - void insertLast(x, L) : Insert element x into L at the last place
  - item deleteFirst(L) : delete first element from L
  - item deleteLast(L) : delete last element from L
  - void insertAt(x, L, index) : Insert element x into L at a given index
  - void insertBefore(x, L, y) : Insert element x into L before element y
  - void insertAfter(x, L, y) : Insert element x into L after element y
  - void insertAfter(x, L, ptr) : Insert element x into L after pointer ptr
  - …
  - item getElementAt(L, index) : Return the element at index
  - boolean isListEmpty(L): Return yes if L is empty

Singly Linked Lists

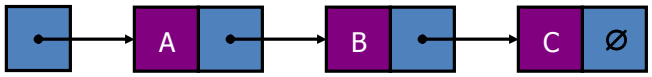


Head

- A *linked list* is a series of connected *nodes*
- Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list
- *Head*: pointer to the first node
- The last node points to `NULL`



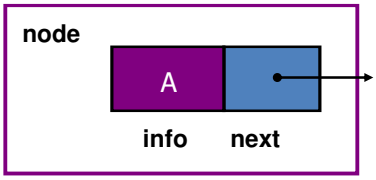
Singly Linked Lists



Head

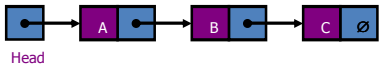
```
typedef struct node {
        item_type info ;
        struct node *next ;
} node;
```

```
void createList (struct node ** head ) {
    *head = NULL;
}
```



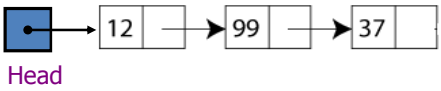
Singly Linked List: Insert()

```
void insert ( struct node **head, int item ) {
    struct node *newNode, *temp;
    newNode = (struct node*) malloc(sizeof (struct node ));
    if ( newNode == NULL )
            printf ( "Memory Error " );
    newNode -> info = item;
    newNode->next = NULL
    if (*head == NULL)
            *head = newNode;
    else  {
            temp = *head
            while (temp->next != NULL)
                    temp = temp->next;
            temp->next = newNode
          }
}
```



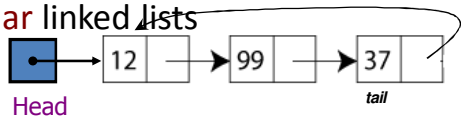
Linked List - variants

- Singly linked list



- Doubly linked lists

- Circular linked lists

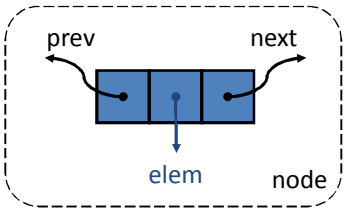


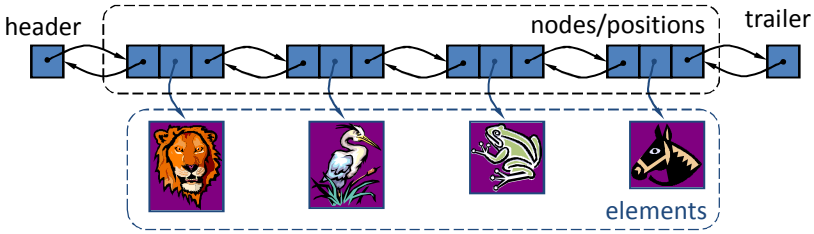
Linear ADT: Other Implementations

- Why different implementations?
  - More flexibility
  - Compare and contrast implementations
- Try by yourself

Doubly Linked List

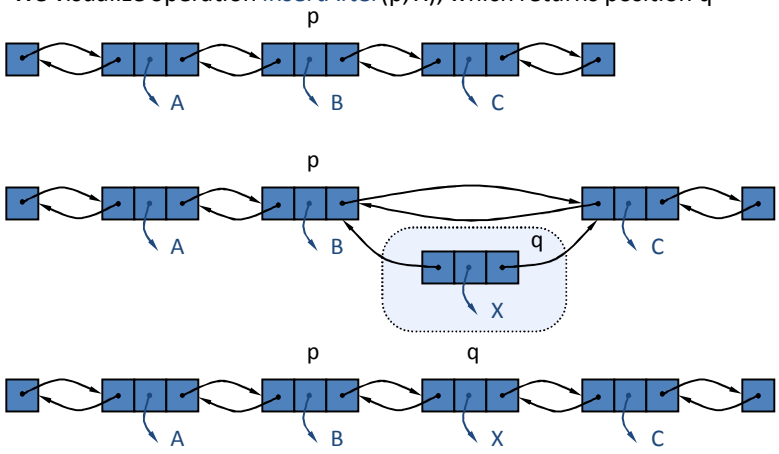
- A doubly linked list is often more convenient!
- Nodes store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





Insertion

- We visualize operation insertAfter(p, X), which returns position q

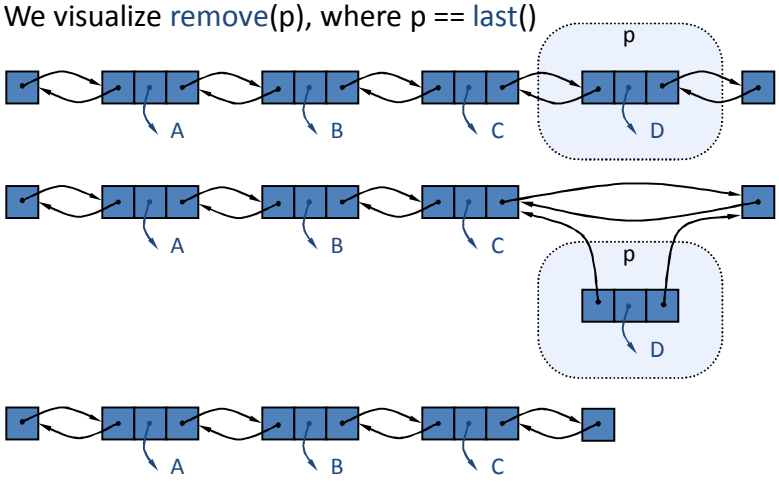


Insertion Algorithm

Algorithm insertAfter(*p,e*):
Create a new node *q*
q.info ← e
q.prev ← p {link *q* to its predecessor}
q. next ← p.next {link *q* to its successor}
p.next.prev ← *q* {link *p*'s old successor to *q*}
p.next ← *q* {link *p* to its new successor, *q*}
return *q* {the position for the element *e*}

Deletion

- We visualize remove(p), where p == last()



Deletion Algorithm

Algorithm remove(*p*):
t = *p*.element {a temporary variable to hold the return value}
p.prev.*n*ext ← *p*.next {linking out *p*}
p.next .*p*rev ← *p*.prev
p.prev ← null {invalidating the position *p*}
p.next ← null
return *t*

Worst-cast running time

- In a doubly linked list
  - + insertion at head or tail is in O(1)
  - + deletion at either end is on O(1)
  - -- element access is still in O(n)