Linked Lists

Διαφορές από πίνακες

- Εύκολη αυξομείωση στοιχείων
- Επακριβής χρήση μνήμης
- Δύσκολο random access
- Περισσότερες απαιτήσεις μνήμης
- Abstraction

Πίνακες: + και -

\+ Γρήγορη προσπέλαση.

-- Πολύπλοκη η αυξομείωση του μεγέθους.

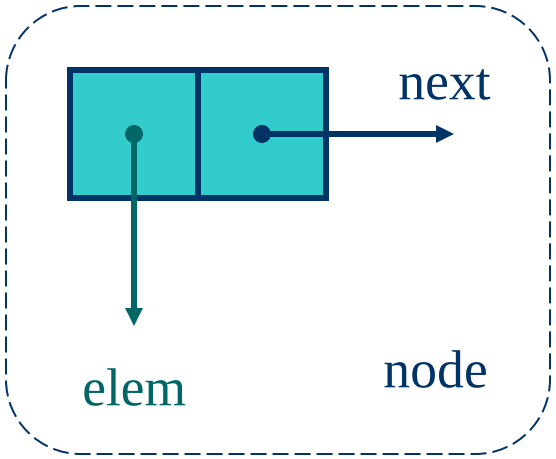
- Η αυξομείωση του μεγέθους των δομών δεδομένων απαιτείται σε πολλές εφαρμογές.
- Το μέγεθος μεταβάλλεται κατά την εκτέλεση (run time).

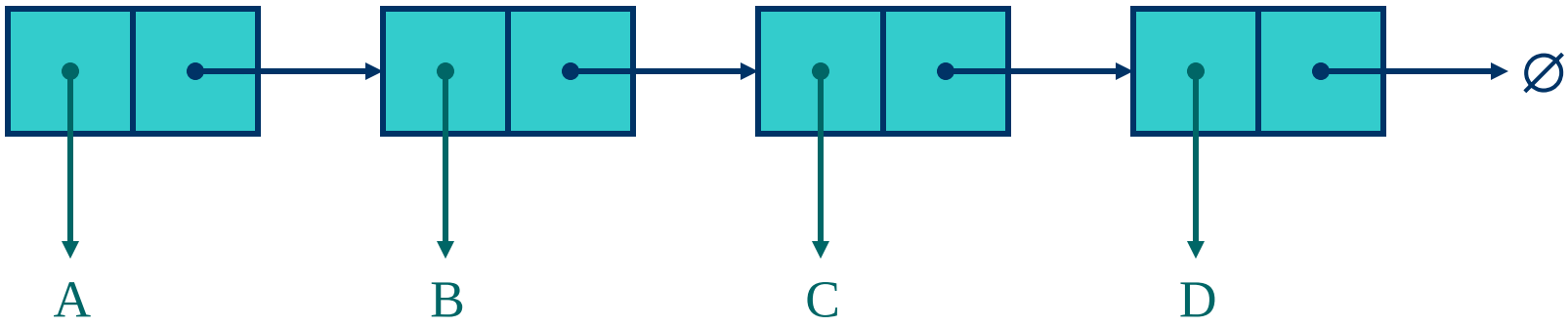
Συνδεδεμένες λίστες - Linked Lists

- Δυναμικές δομές δεδομένων
- Εφαρμογής
  - όταν απαιτείται το μέγεθος να μεταβάλλεται κατά την εκτέλεση (run time).

Απλά συνδεδεμένες λίστες

- Ακολουθία κόμβων
- Κάθε κόμβος περιέχει
  - πληροφορίες
  - σύνδεσμο στον επόμενο κόμβο





Ορισμοί και παραδείγματα

- Πεπερασμένη, διατεταγμένη ακολουθία από μηδέν η παραπάνω στοιχεία
  - $\{a_1, a_2, ..., a_n\}$
- E.g. the list of prime numbers less than 20
  - {2,3,5,7,11,13,17,19}
- A line of text of individual characters
  - {h,e,l,l,o, ,w,o,r,l,d}
- A list of lists!
  - {{1,2, , e,g,g,s},{1, , s,a,l,a,m,i},{1,k,g,r, ,o,f, ,to,m,a,t,o,e,s}}

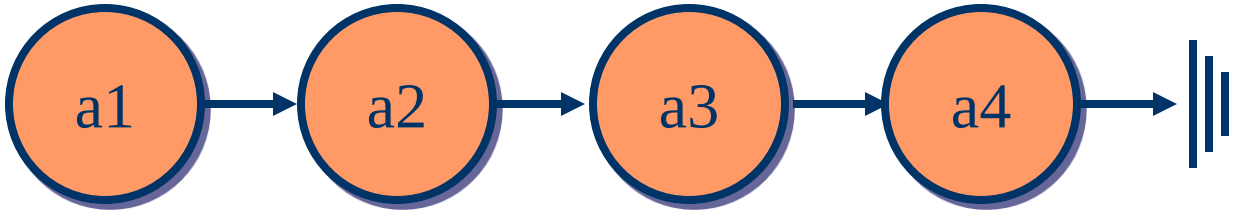
List Operations

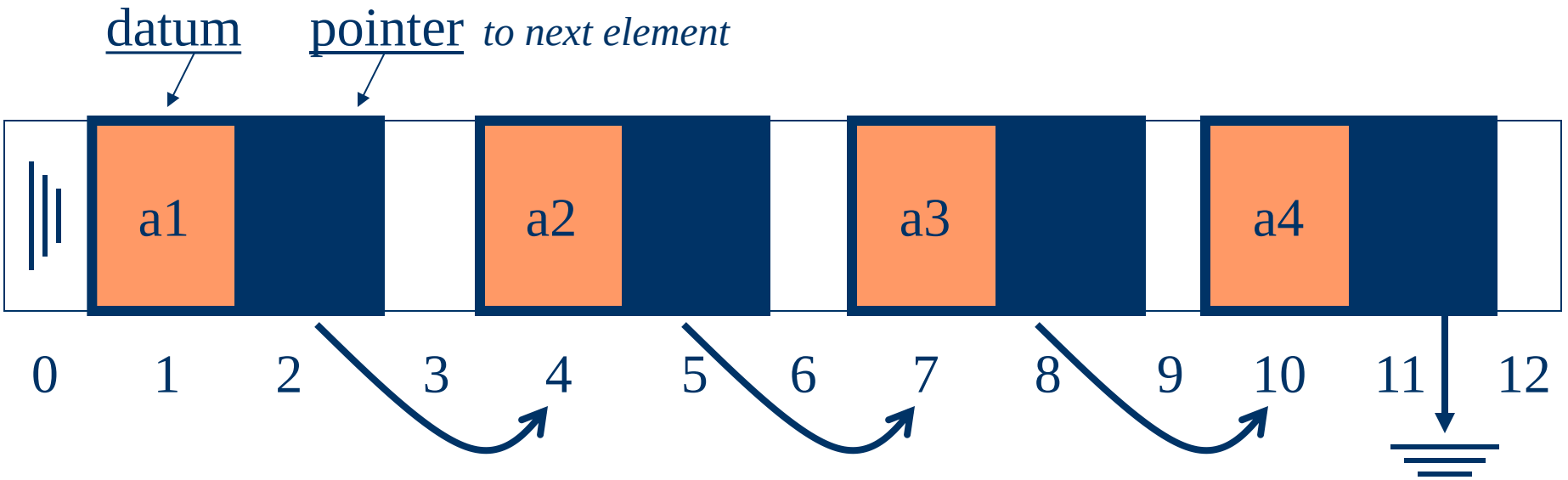
- Print
- Length
- Insert
- Remove
- Lookup

} *"Running" through the list.* ***Enumeration***

- Other, e.g. list cloning, sub-list.
- *Error Checking*

Representation

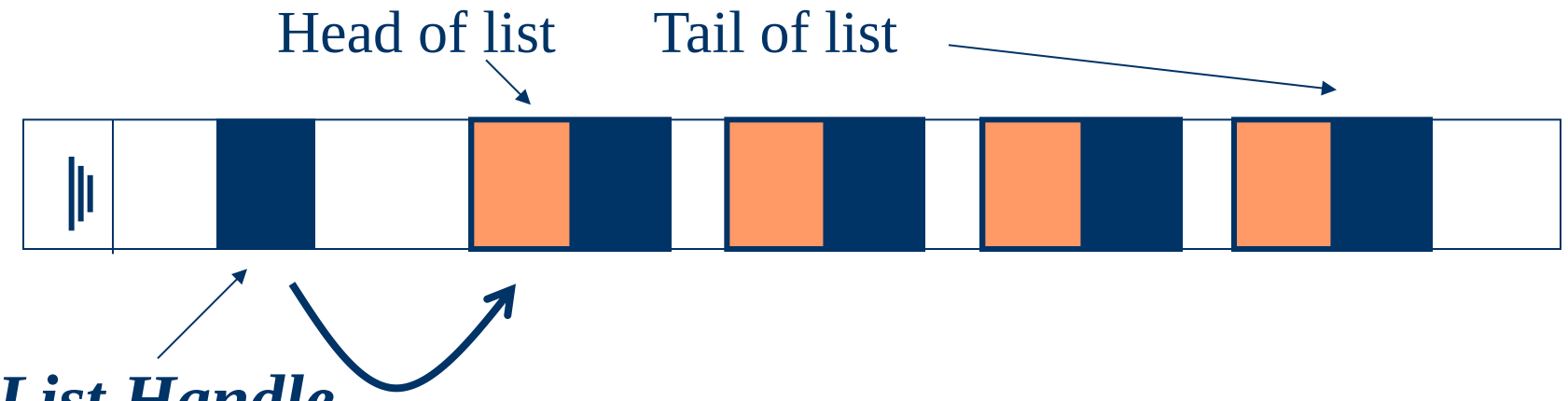




Implementation

Node { Handle datum, Handle pointerToNextElement}

e.g. struct Node {int myDatum, struct Node *next};
or class Node {int myDatum, Node *next};



List Handle

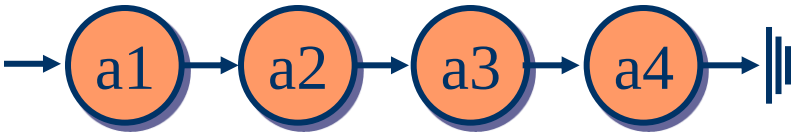
*where the pointer to the **head** of list is stored.*
Another such pointer is required if we retain a tail pointer to the list

Accessing the Data Structure

- List Handle
- setData(), getData(), setNext(), getNext()

E.g. void setData(struct Node *whichNode, int value);
Or Node.setData(int);

- Printing the list




```
Handle current = head;
while (notAtEndOfList(current))
{
        Print(current);
        Current = getNext(current);
}
```

Traversal

x = head

while current node (x) is not null

{

do something with **x->data**

x = x->next

}

Access the list

```
void
printTriangleList(ListEl *list)
{
   ListEl *x;
   for (x=list;
        x != NULL;
        x=x->next)
        cout<<x->datum<<endl;
        /* besides string or simple tyle (int, float etc),
           datum could be a structure,
           and printed accordingly */
}
```

Access the list

```
void
printTriangleList(ListEl *x)
{
   for (;
        x;
        x=x->next)
        cout<<x->datum<<endl;
        /* besides string or simple tyle (int, float etc),
           datum could be a structure,
           and printed accordingly */
}
```

Traverse

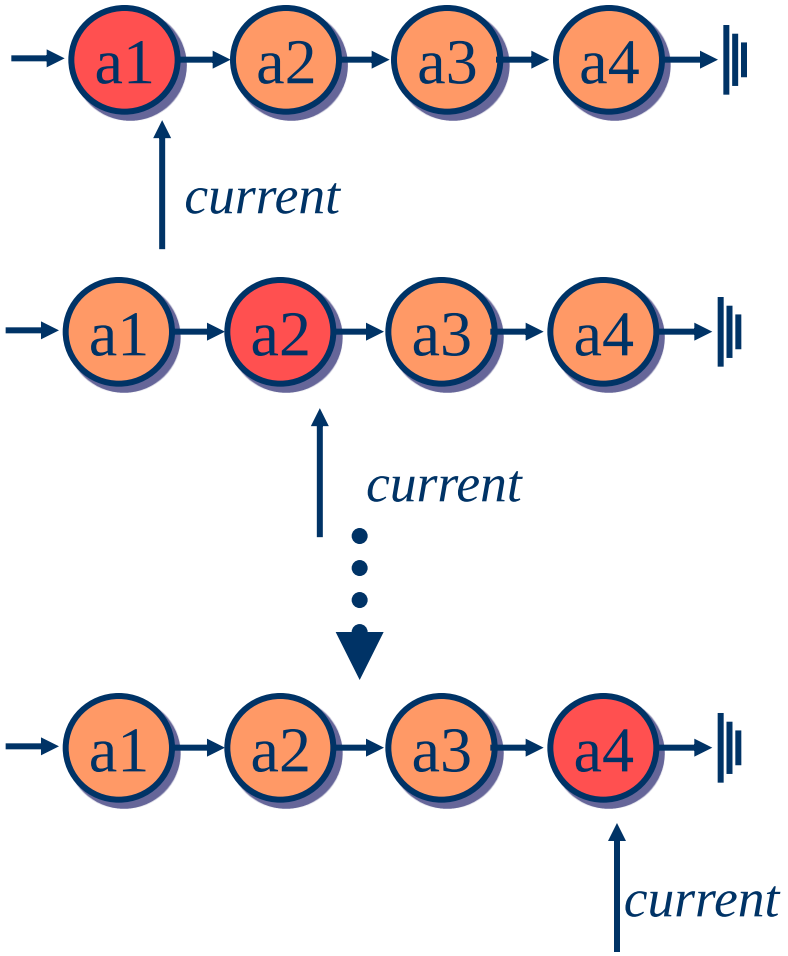
```
while( x!=NULL )
{
    cout >> x->datum;
    x= x->next;
}
```

Number of elements

```
int
triangleListLen(ListEl *list)
{
    ListEl *x;
    int result = 0;

    for (x=list;x;x=x->next)
        result++;

    return result;
}
```

Lookup

```
Enumerate through list
{
    if (current equals target)
        return current;
}
return NOT_FOUND;
```



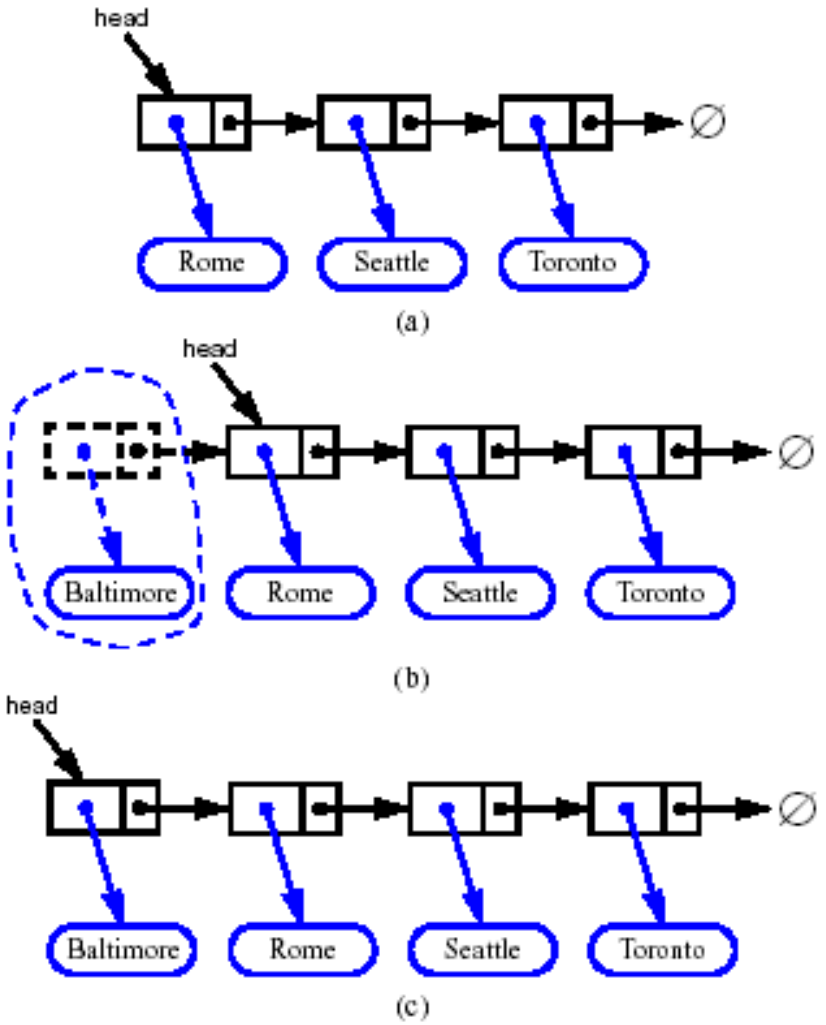
Return Value

index, handle, Boolean, numeric

```
list_el *
getByKey(list_el *list, char *mykey)
{
    list_el *x;

    for (x=list;x!=NULL;x=x->next)
    {
        if (strcmp(x->key,mykey)==0)
            return x;
    }

    return (list_el *) NULL;
}
```

Inserting at the Head

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



Insert

```
head = NULL;
while (1)
{
    p = new element();

    cin>>info;
    p->data = info;
    p->next = head ;
    head = p;
}
```

In a function

```
void functionAdd(struct node *head, int info)


struct node *head functionAdd(struct node *head, int info)
{
        p = new element();
        p->data = info;
        p->next = head;
        head = p;

        return head;
}
```

If we still wish for a void function

```
void
functionAdd(struct node **head, int info)
{
        p = new element();
        p->data = info;
        p->next = *head;
        *head = p;

        return *head;
}
```

Insert

function insertAfter(***Node*** ***node**, ***Node*** ***newNode**)

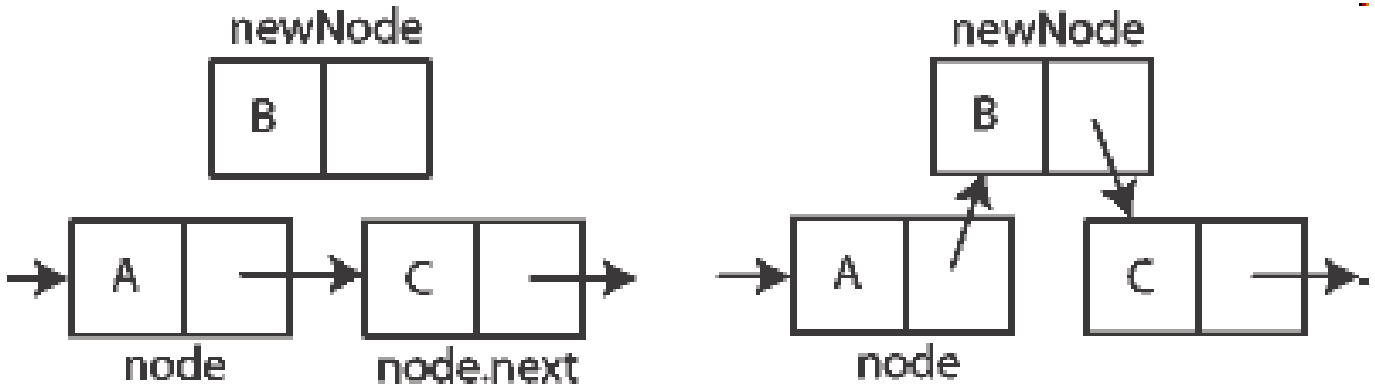
{

*// insert newNode **after** node*

newNode->next = node->next;

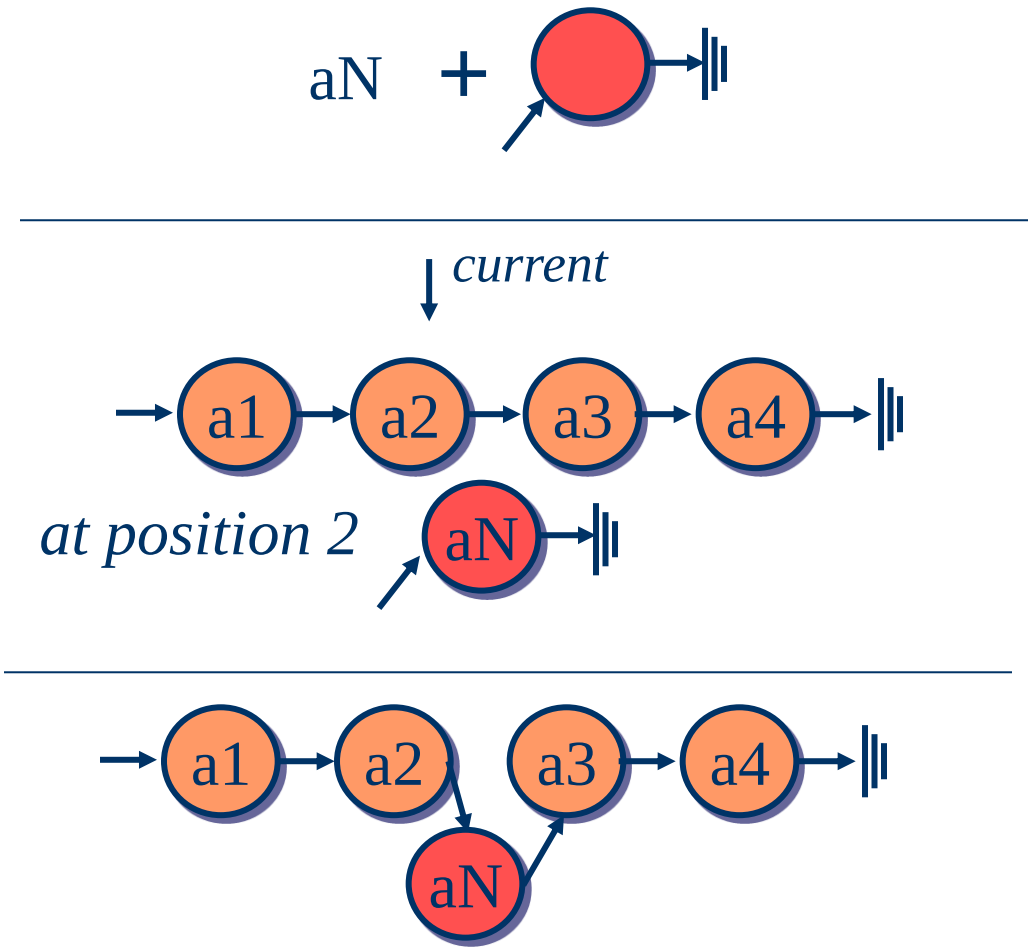
node->next = newNode;

}



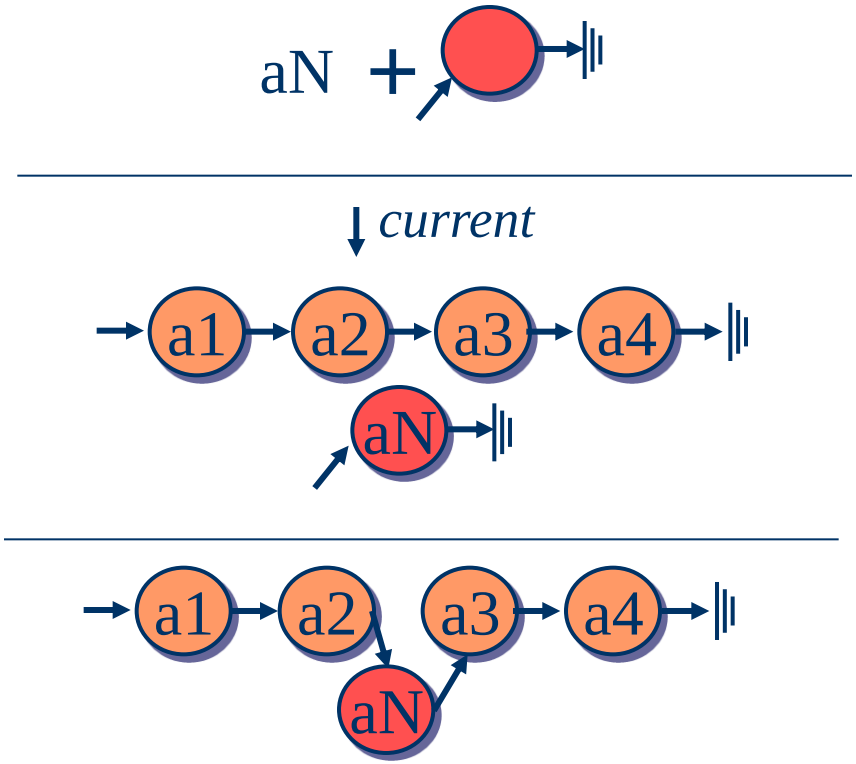
Insert

1. create new node
2. find handle to new position
3. update handles
4. return



Insert

```
// New node
aN = new Node(theNewValue);

// Handle to position
current = head;
loop
      until current indicates
      the target position

// Update handles
aN->next = current->next;
current->next = aN;
```

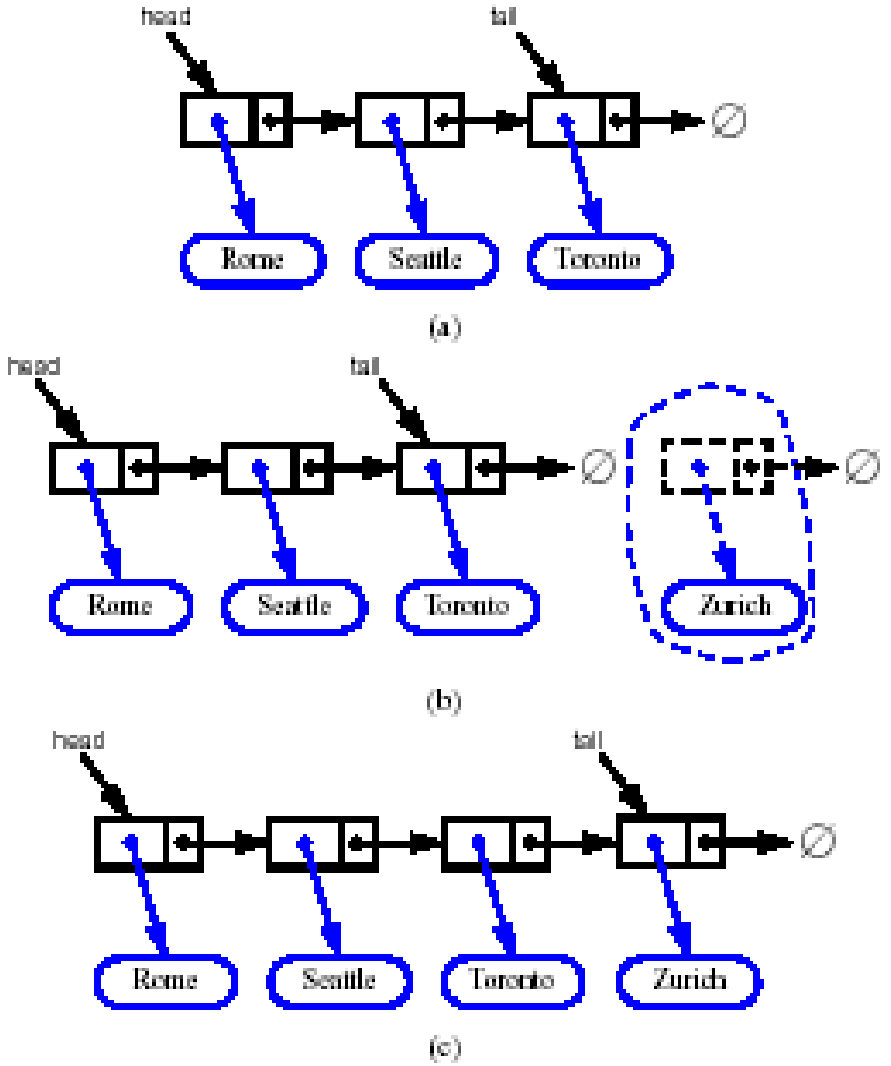


Insert at n-th position

```
for( int i = 1 ; i < node_number ; i++ )
    current = current ->next;
node *temp;
temp = new node();
temp->data = info;
temp->next = temp1->next;
temp1->next = temp;
```

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



Insert at the end

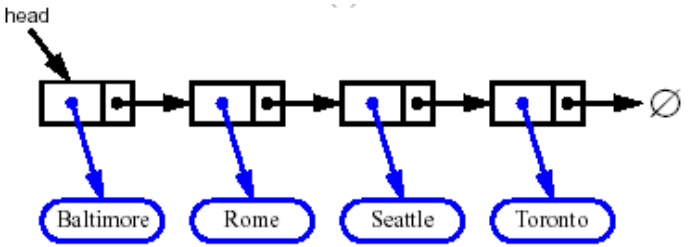
```
node *temp1, *temp;

temp1 = head;
while(temp1->next!=NULL)
    node temp1 = temp1->next;

temp = new node();
temp->data = info;
temp->next = NULL;
temp1->next = temp;
```

```
ListEl *addToTriangleList(ListEl *list, int datum)
{
    ListEl *result = NULL, *newEl = NULL;

    newEl = new ListEl();
    newEl->datum = datum;
    newEl->next = (ListEl *) NULL;

    if (list == NULL)  // first time
    {
        result = newEl;
        result->next = NULL;
    }
    else
    {
        ListEl *x,

        for (x=list;x;x=x->next)
            if (x->next == NULL)
                break;

        x->next = newEl;
        x->next->next = NULL;
        result = list;
    }
    return result;
}
```

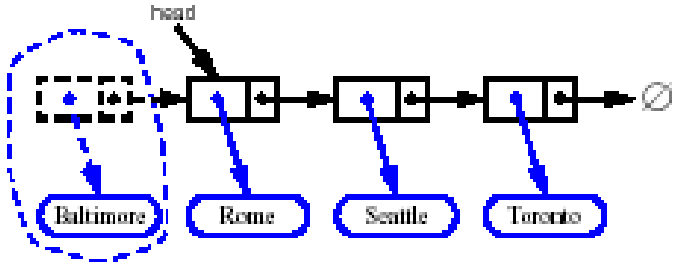
```
ListEl *x,

for (x=list;x;x=x->next)
      if (x->next == NULL)
         break;

x->next = newEl;
x->next->next = NULL;
result = list;
```

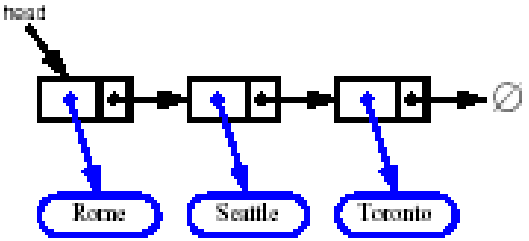
Removing at the Head

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




```
tmp = head;
head = head->next;
delete tmp->cityName;
delete tmp;
```





Remove function

```
// assumes head is not NULL
ListEl *remove_first(List *head)
{
    tmp = head;
    head = head->next;
    delete tmp->cityName;
    delete tmp;
    return head;
}
```

Remove all !

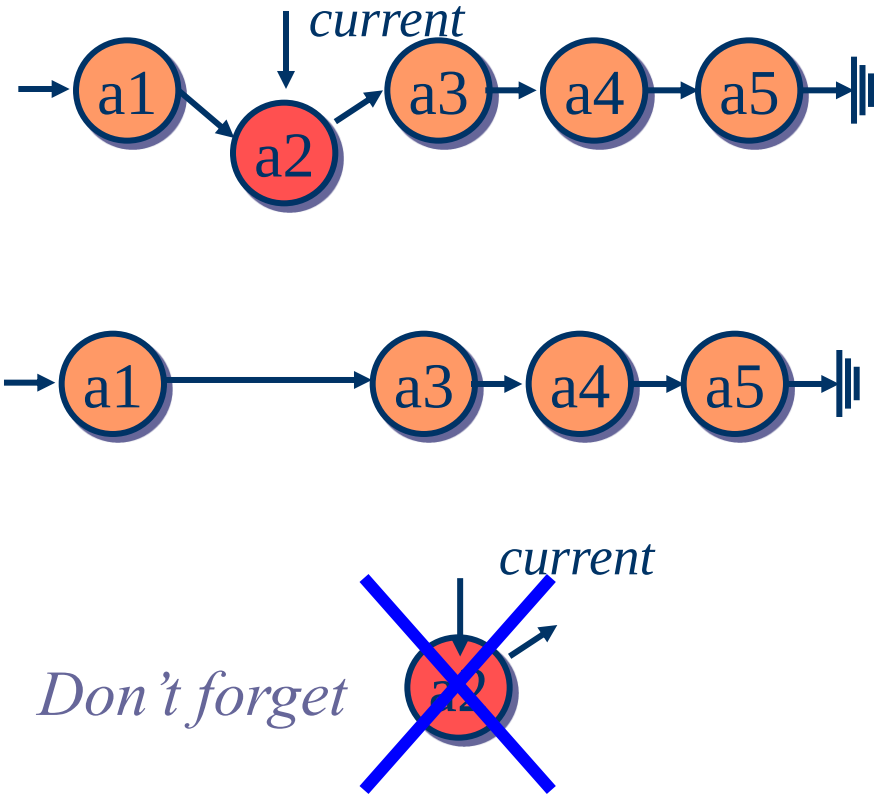
```
void freeList(ListEl *list)
{
        ListEl *clist = list;
        while (1)
        {
                if (clist == NULL)
                        break;
                clist = remove_first(clist);
        }
}
```

Remove

1. find handle to removal position
2. update handles
3. free memory

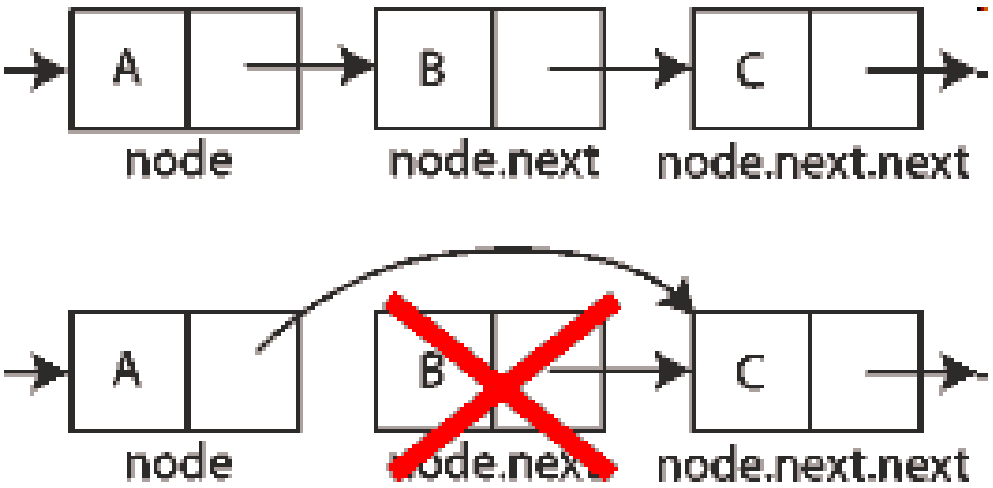
   free(current);

4. return



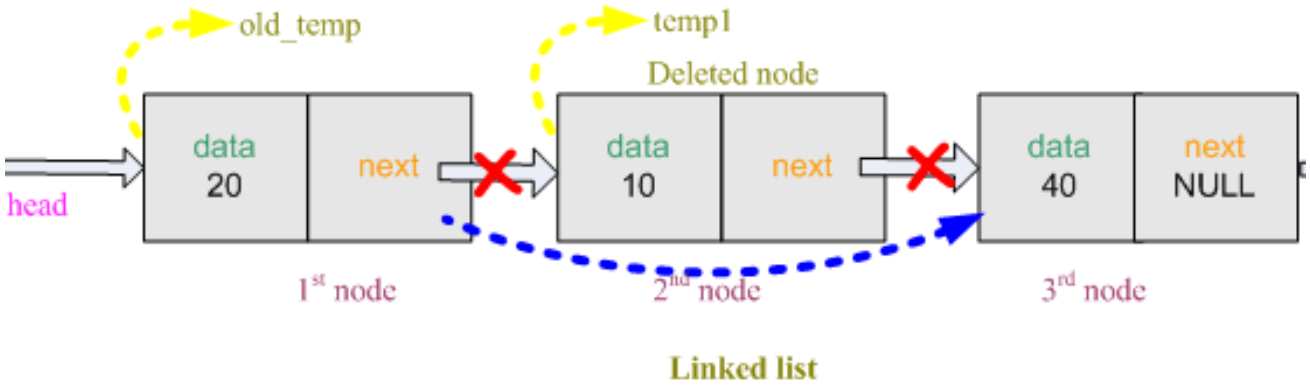
Remove

```
void removeAfter(node *node)
{
    // remove node past this one obsoleteNode = node->next
    node->next = node->next->next
    delete obsoleteNode
}
```



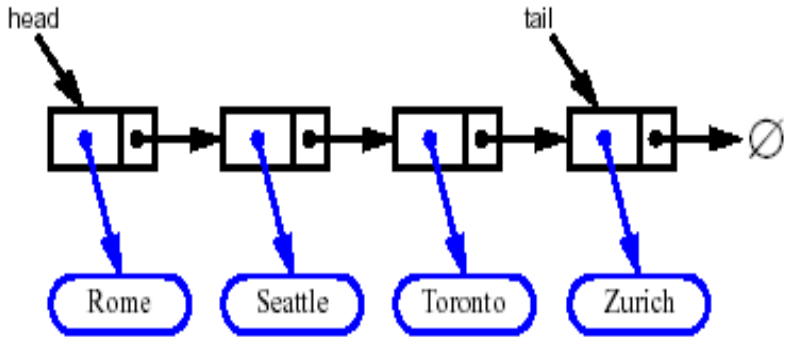
Delete n-th

```
old_temp->next = temp1->next;
free(temp1);
```



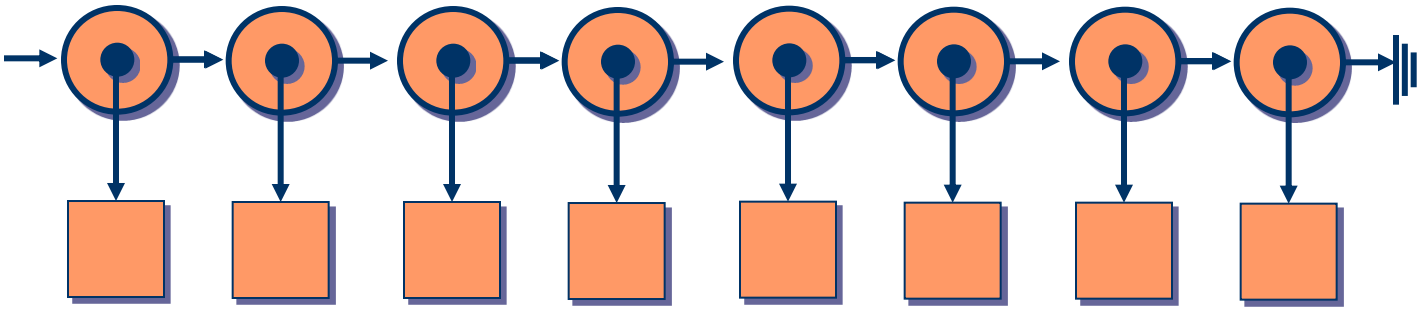
Removing at the Tail

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

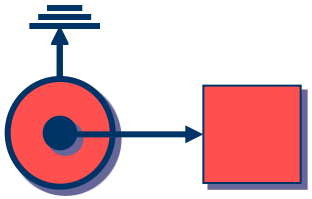


*Unless we also maintain a pointer to the **penultimate** node (a bit more complicated, may need to keep track of cases for null list, one node list, and the rest)*

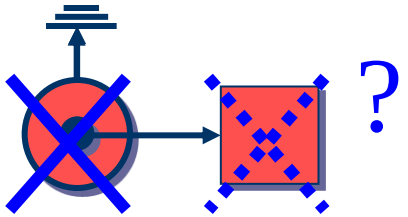
List of Data Structures



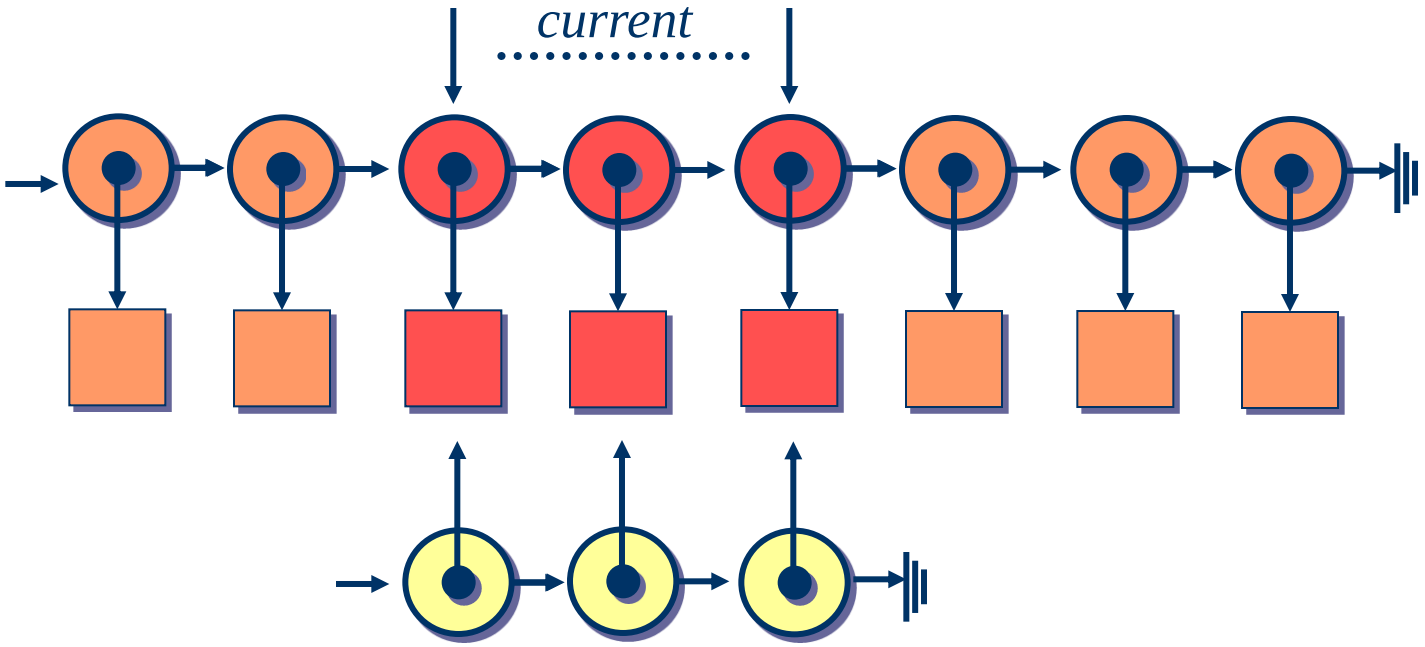
- Insertion



- Removal



Sub - List



May be composed of simpler operations

Abstraction

- Abstraction of Representation w.r.t. Stored Data Type
- Code Reusability
- Implementation
  - Pointer Abstraction *C*
  - Object abstraction *Java*
  - Templates *C++*

A list of Triangles

```
typedef struct Triangle
{
   Point p1,p2,p3;
};
```

```
typedef struct ListEl
{
    Triangle datum;
    ListEl *next;
};
```

List of lists

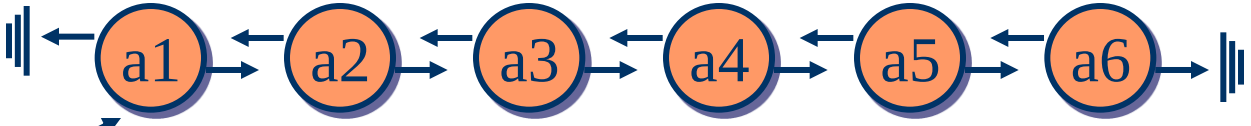
```
typedef struct ListList
{
  ListEl *head;      // of triangle list
  ListList *next;
};
```

```
void
printCardinalities(ListList *list)
{
    ListList *x;
    int cnt = 1;

    for (x=list;x;x=x->next)
    {
        cout <<  triangleListLength(x->head);
        cnt++;
    }

    cout << “All elems are ” << cnt << endl;
}
```

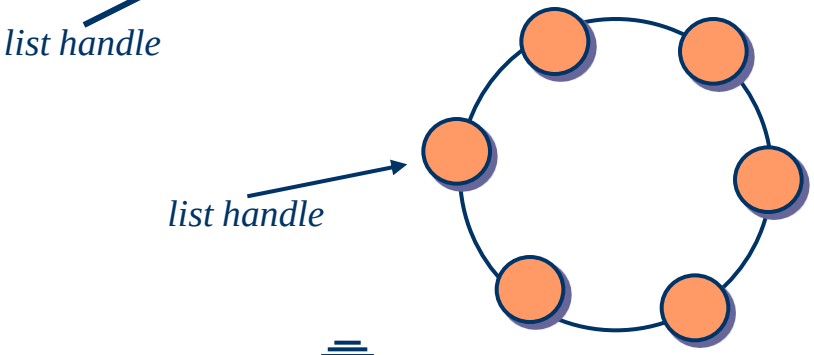
```
void
freeListList(ListList *list)
{
    ListList *x;

    for (x=list;x;x=x->next)
        freeTriangleList(x->datum);

    freeListList();
}
```

Structural Variants

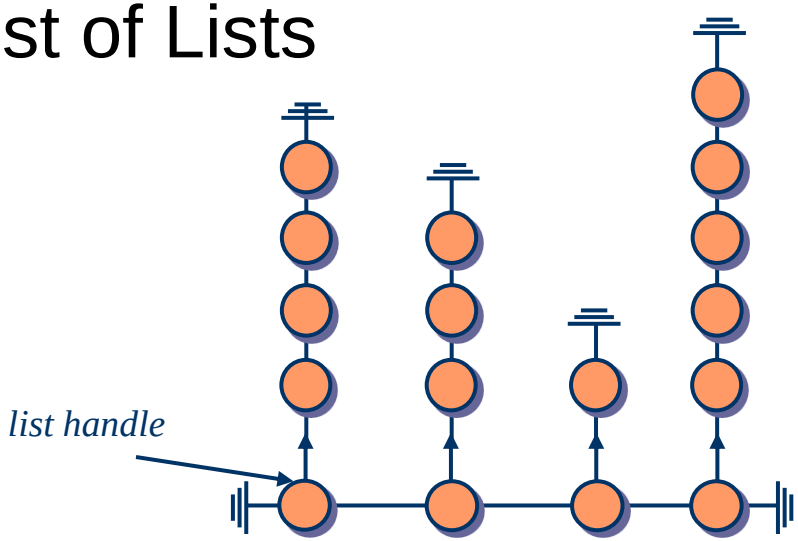
- Doubly Linked List



- Cyclic List



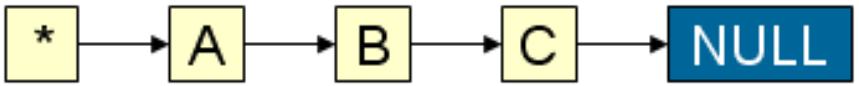
- List of Lists



A few basic types of Linked Lists

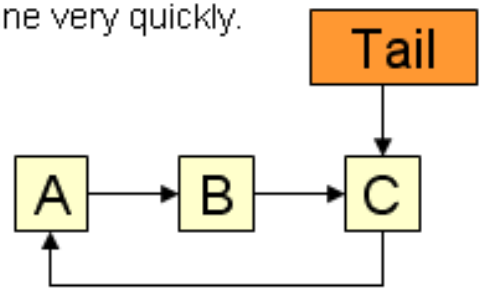
Singly Linked List

Root node links one way through all the nodes. Last node links to null.



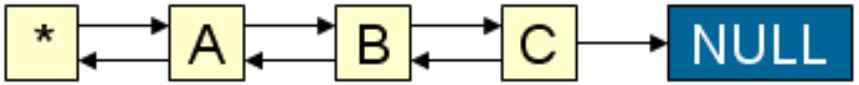
Circular Linked List

Circular linked lists have a reference to one node which is the tail node and all the nodes are linked together in one direction forming a circle. The benefit of using circular lists is that appending to the end can be done very quickly.



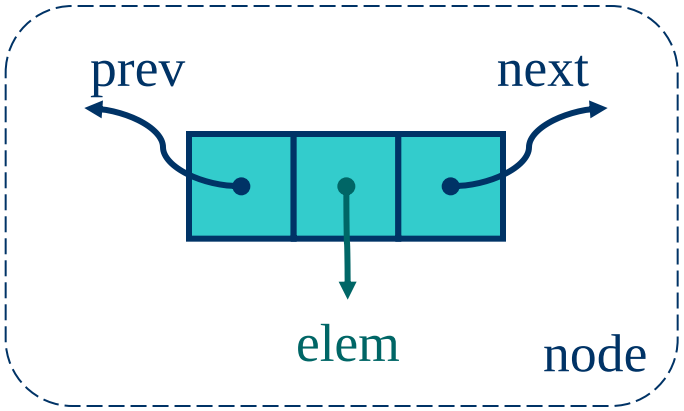
Doubly Linked List

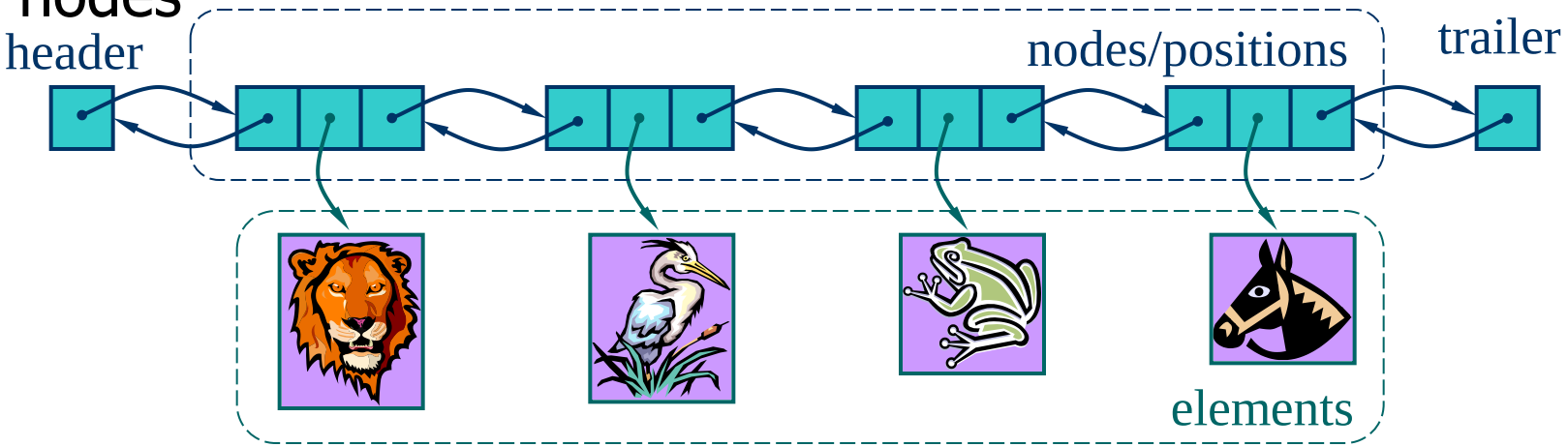
Every node stores a reference to its previous node as well as its next. This is good if you need to move back by a few nodes and don't want to run from the beginning of the list.



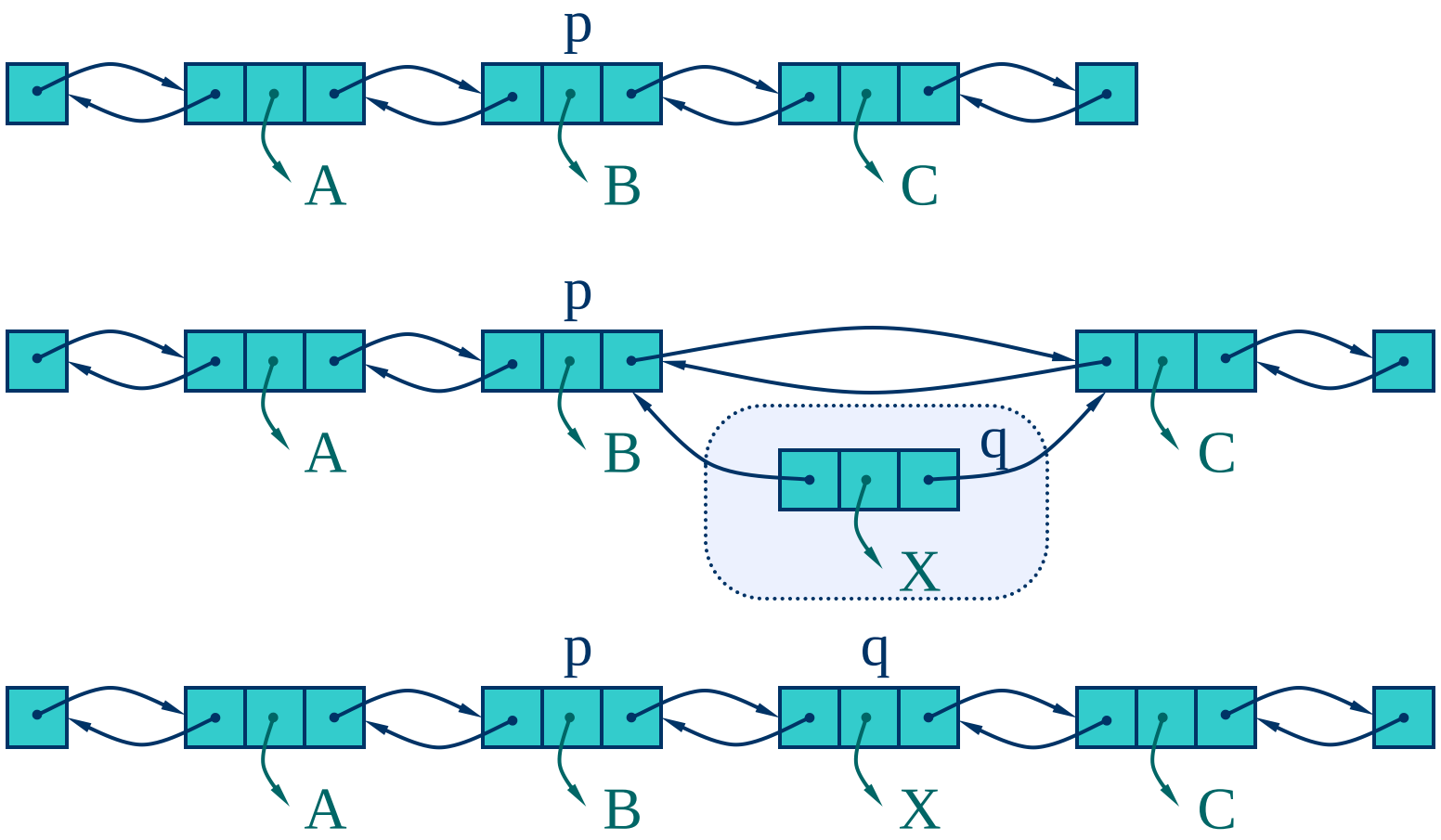
Διπλά συνδεδεμένη λίστα

- Δυνατότητα γρήγορης προσπέλασης και προς τις δύο φορές της ακολουθίας
- Κάθε κόμβος περιέχει
  - πληροφορίες
  - σύνδεσμο στον επόμενο κόμβο
  - σύνδεσμο στον προηγούμενο κόμβο
- Μπορεί να έχει επιπλέον "handle nodes"

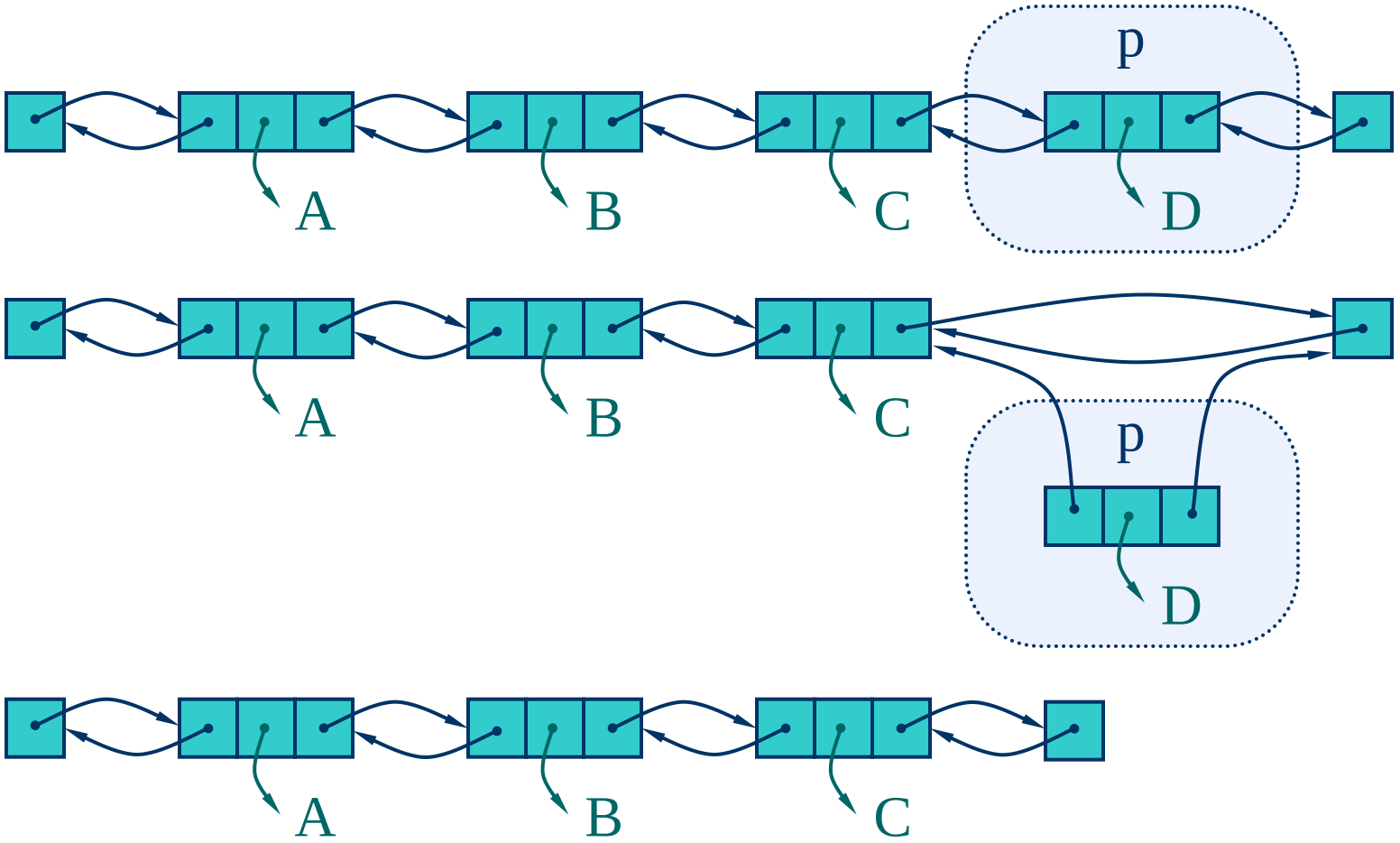




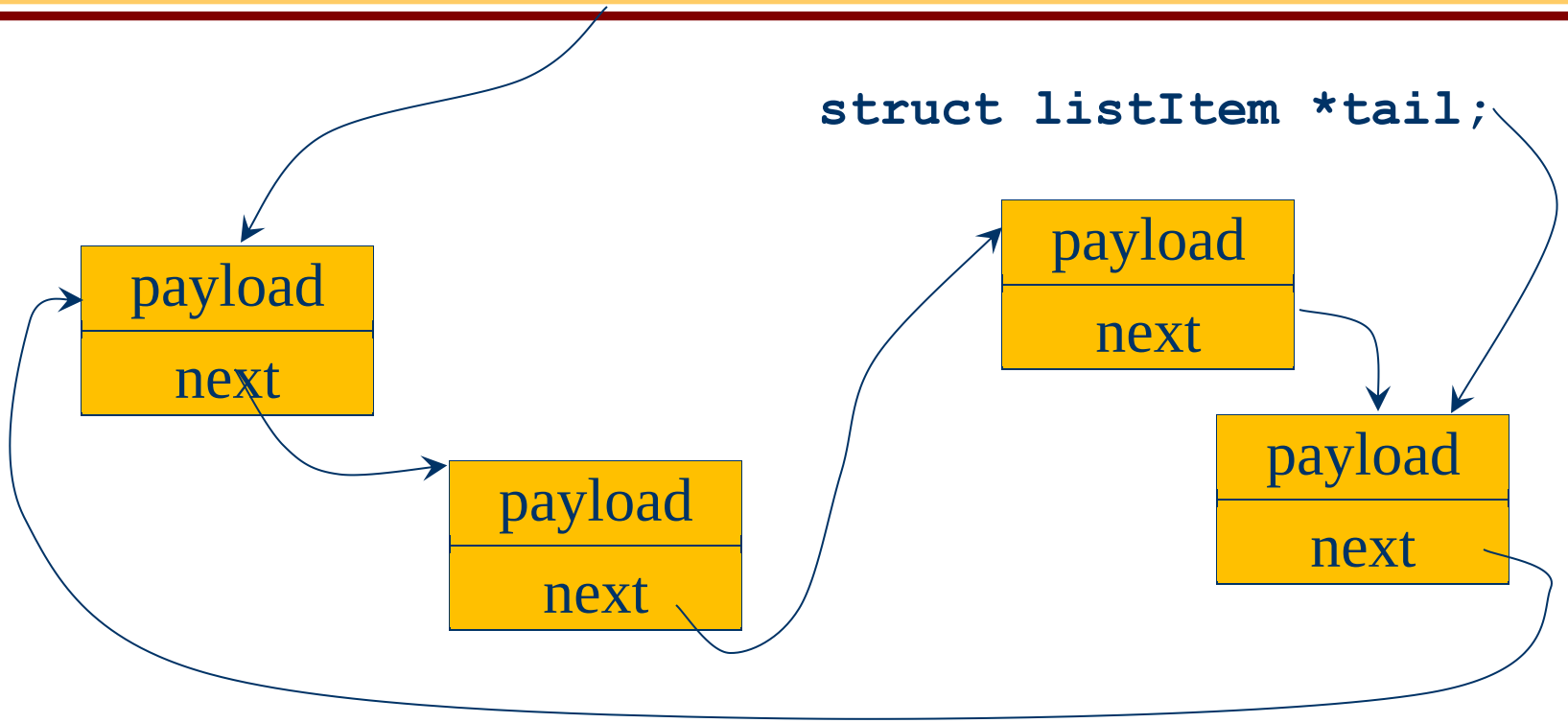
Insertion



Deletion



Circular List

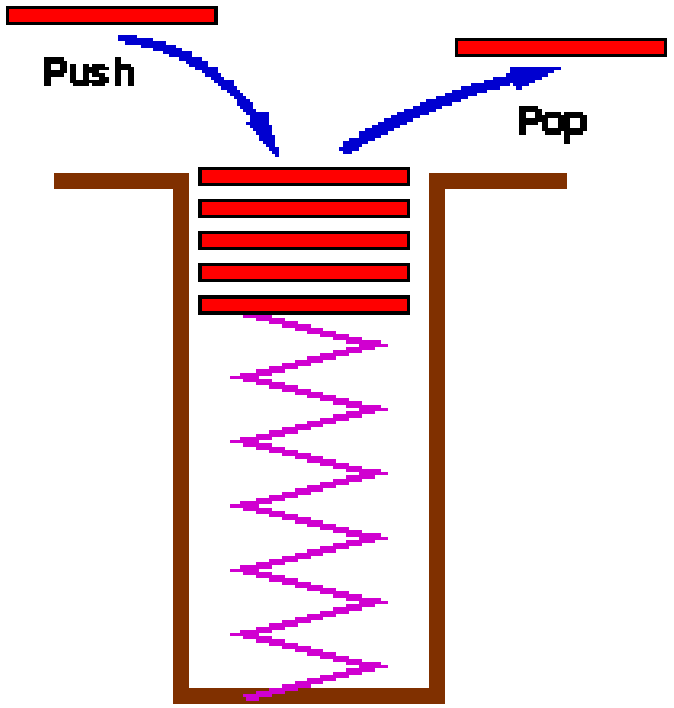



```
listItem *addAfter (listItem *p, listItem *tail){
   if (p && tail) {
        p -> next = tail -> next;
        tail = p;
   } else if (p) {
        tail p -> next = p;
   }
   return tail;
}
```

Optional:–
`struct listItem *head;`

Algorithmic Variants

- Push / Pop *Stack*



- Enqueue, Dequeue *Queues*
  - *FILO (stack)*
  - *FIFO (priority queue)*

How to remember the length?

```
struct ListHolder {
    int numberOfElements; /* private, init=-1*/
    ListElement *head;
}

// object orientation (C++) treats this in a more
    elegant fashion for the programmer, still using the
    same data structures in memory
```