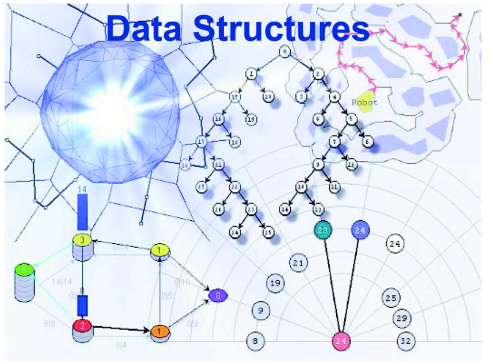
BBM 201
DATA STRUCTURES

Lecture 4:
Linked-Lists & General Lists



Ref Book: Data Structures Using C and C++. 2nd Edition. Y. Langsam, M. J. Augenstein, A. M. Tenenbaum, Prentice-Hall International Inc., 1996

Previous Lecture

Structures

Simple array implementation of Lists

```
// Array-based list implementation
class AList : public List {
  ListItemType* listArray;             // Array holding list elements
  static const int DEFAULT_SIZE = 10; // Default size
  int maxSize;                         // Maximum size of list
  int listSize;                        // Current # of list items
  int curr;                            // Position of current element
```

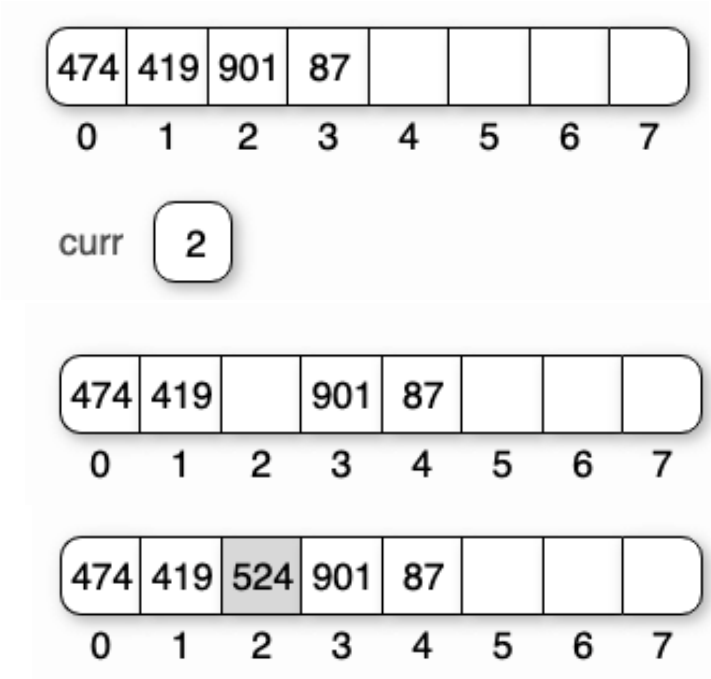
Previous Lecture

```
// Array-based list implementation
class AList : public List {
  ListItemType* listArray;            // Array holding list elements
  static const int DEFAULT_SIZE = 10; // Default size
  int maxSize;                        // Maximum size of list
  int listSize;                       // Current # of list items
  int curr;                           // Position of current element
```

Structures

Simple array implementation of Lists

insert 512 to current position



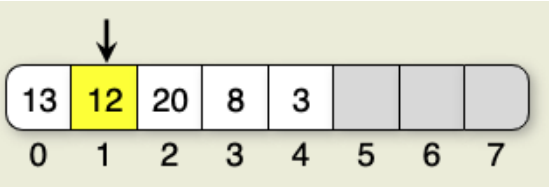
Previous Lecture

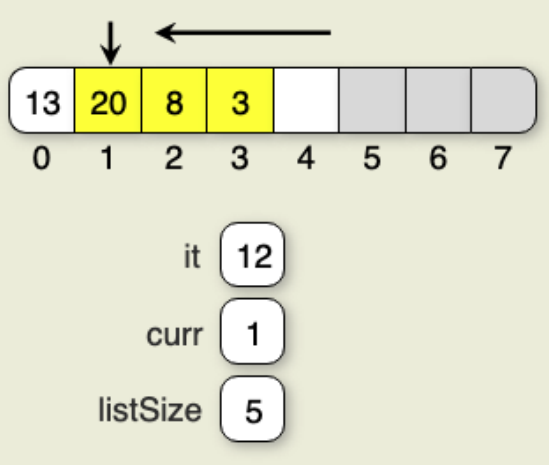
```
// Array-based list implementation
class AList : public List {
  ListItemType* listArray;            // Array holding list elements
  static const int DEFAULT_SIZE = 10; // Default size
  int maxSize;                        // Maximum size of list
  int listSize;                       // Current # of list items
  int curr;                           // Position of current element
```

Structures

Simple array implementation of Lists

Remove 12 in position 1





Previous Lecture

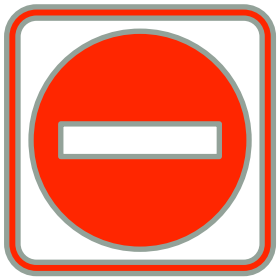
There are many situations where the list is built up by insertions at the high end,

and then only array accesses (i.e., getValue operations) occur.

In such a case, the array is a suitable implementation.

if insertions and deletions occur throughout the list and,

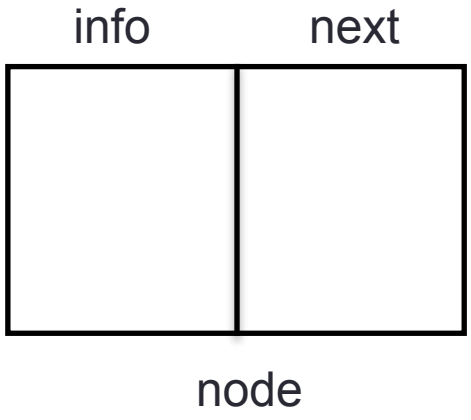
in particular, at the front of the list, then the array is not a good option.

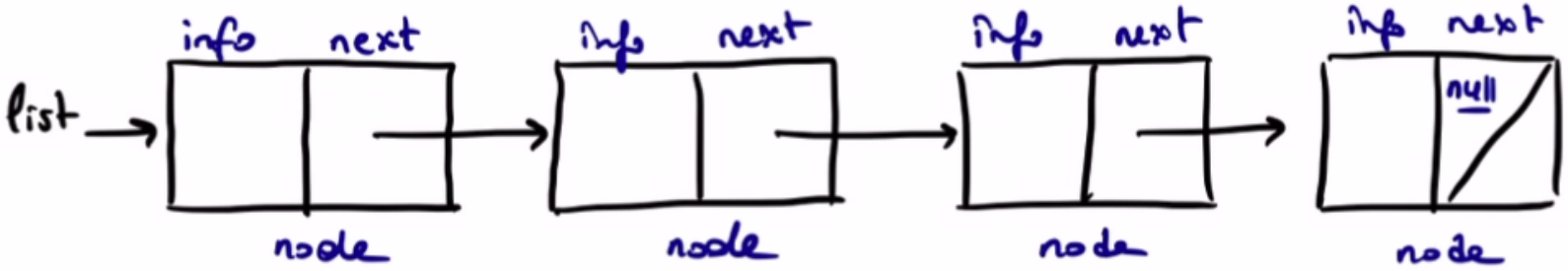


Outline

- Linked lists
  - Array implementation
  - Implementation using dynamic memory allocation
- General lists
  - List as ADT revisited
  - Abstract Operations

Linked List Structure

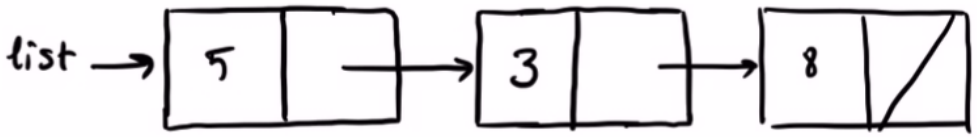




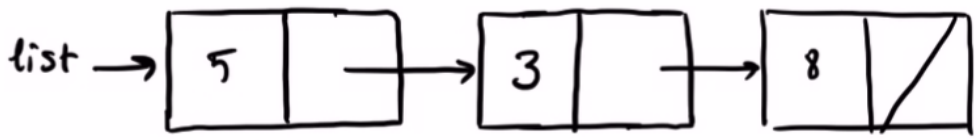
Basic functions

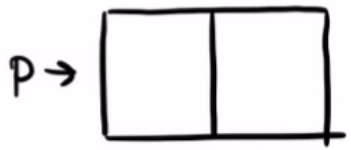
- list: external reference (i.e. pointer) to the first node
- Empty list: No nodes in the list
  - list is null
- Assume p is a *pointer* to a node:
  - info(p) : data section of the node p
  - next(p): the address of the next list item
  - *If next(p) is not null, what is* info(next(p)) *?*
- Assume that;
  - We have a getnode() function that creates an empty node for us
  - We have freenode() function that deallocates the memory pointed to by p

Adding an element to the front of a list



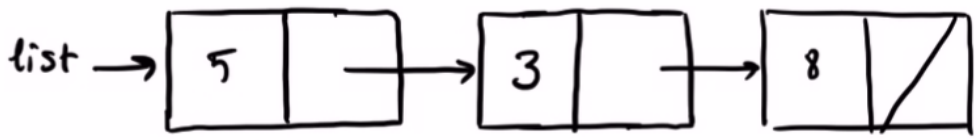
Adding an element to the front of a list

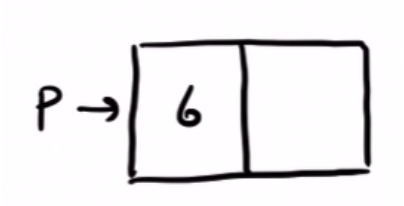




p=getnode();

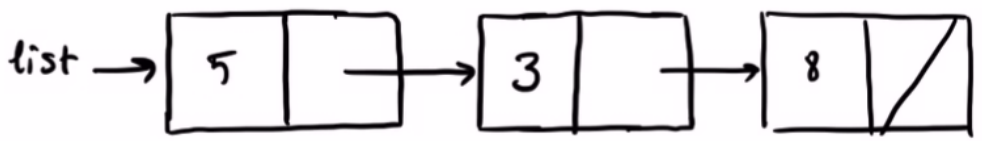
Adding an element to the front of a list

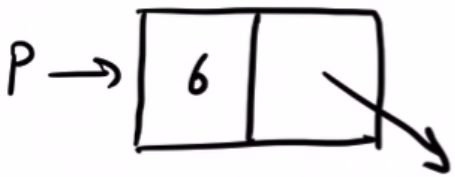




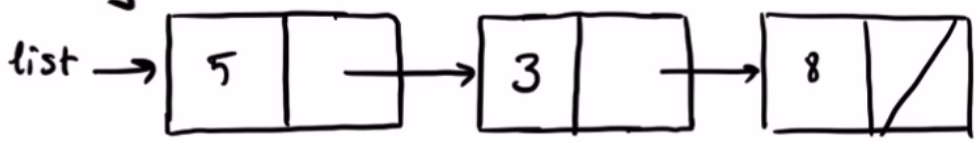
info(p) = 6;

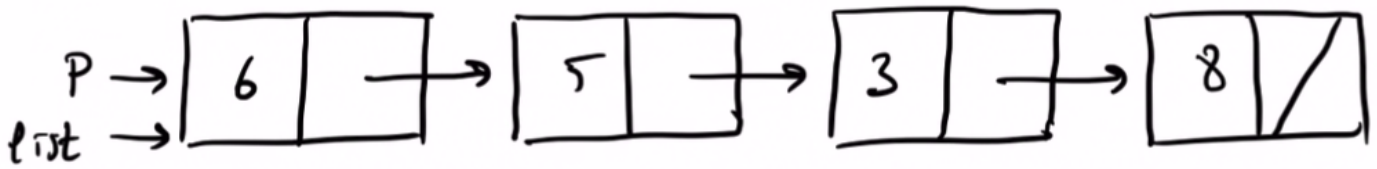
Adding an element to the front of a list





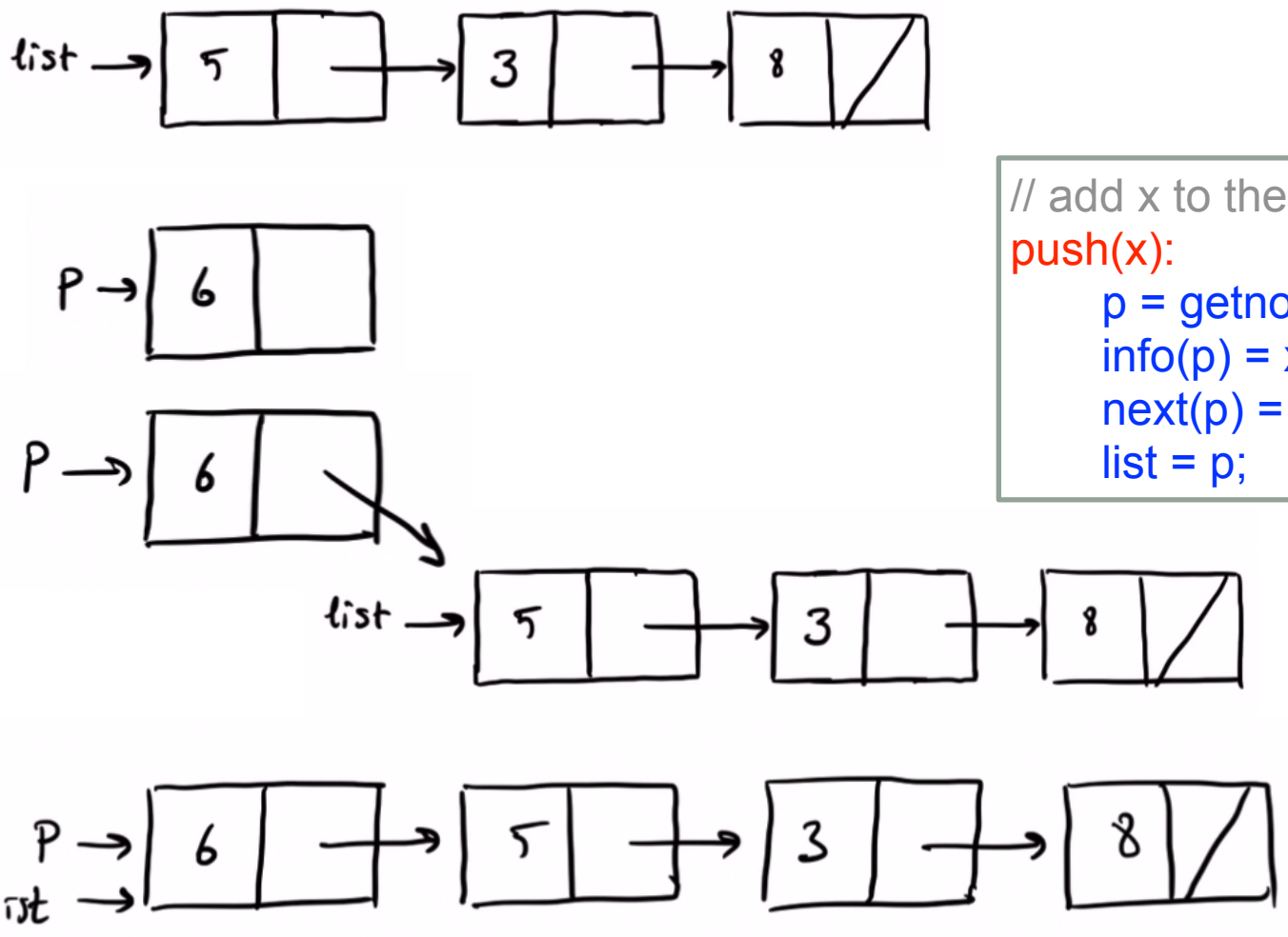
next(p) = list;





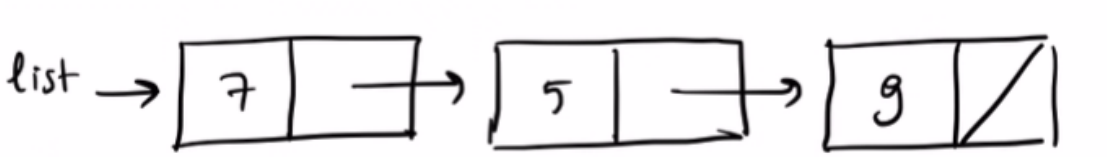
list = p;

Adding an element to the front of a list

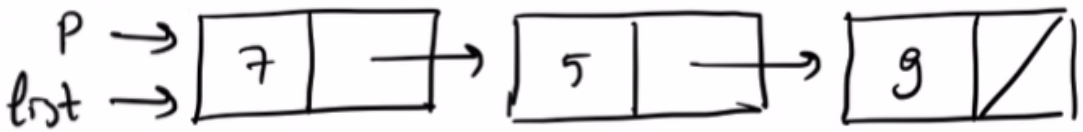



```
// add x to the front of the list:
push(x):
        p = getnode();
        info(p) = x;
        next(p) = list;
        list = p;
```

Removing a node from the front of a list

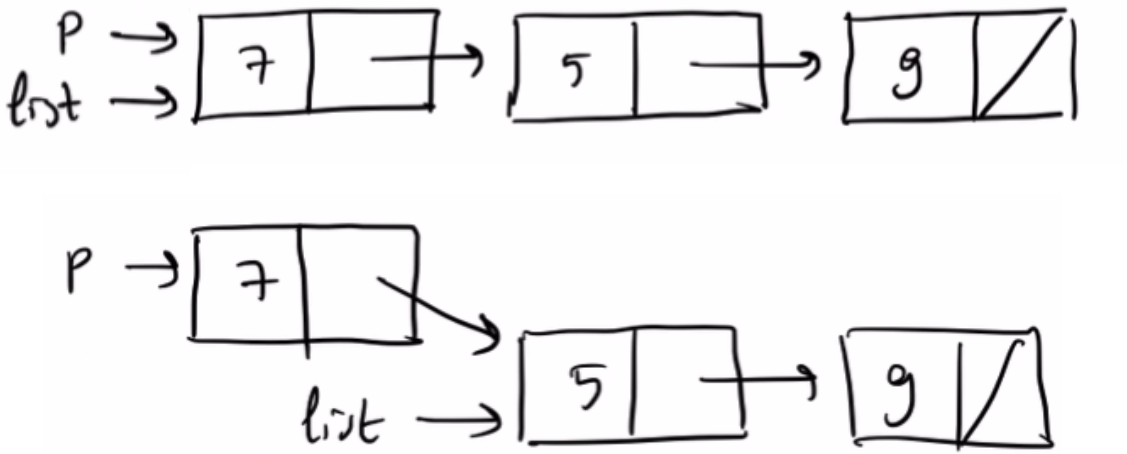


Removing a node from the front of a list



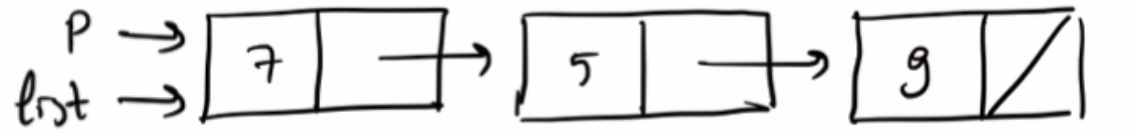
p=list;

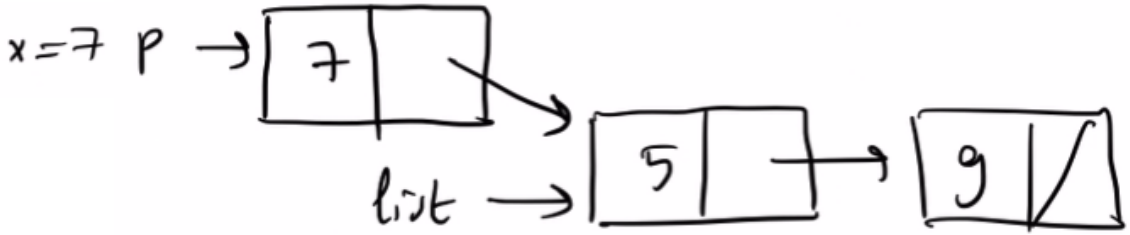
Removing a node from the front of a list



list=next(p);

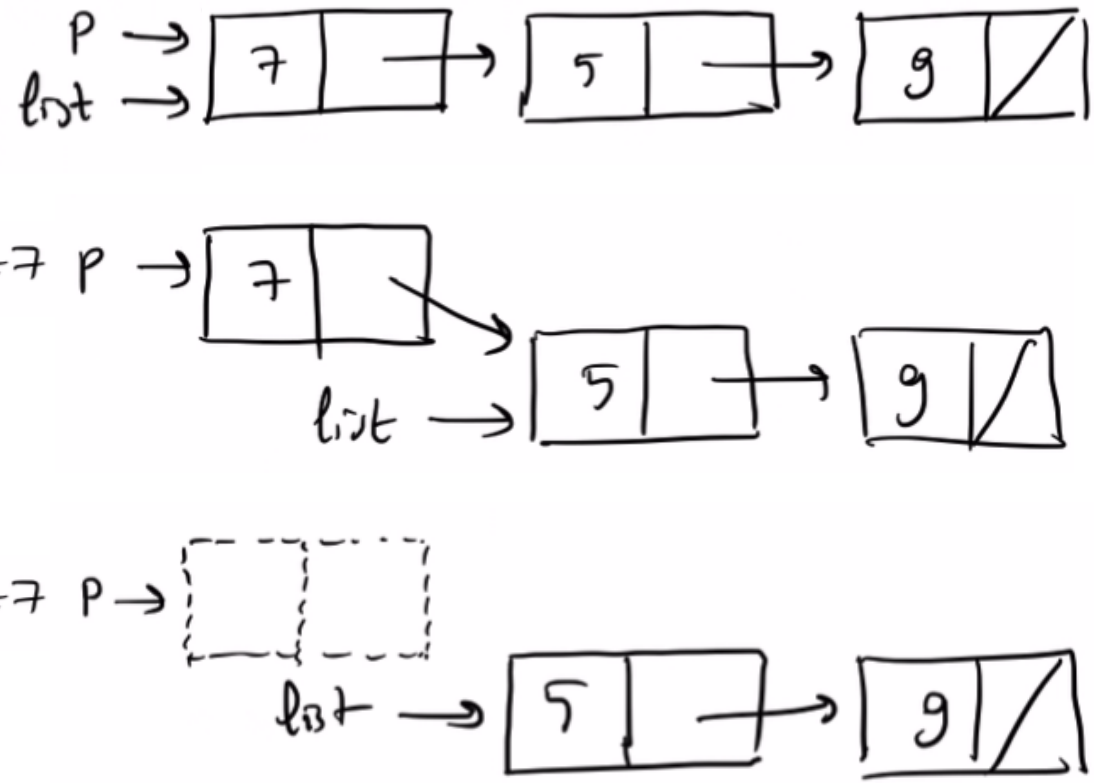
Removing a node from the front of a list





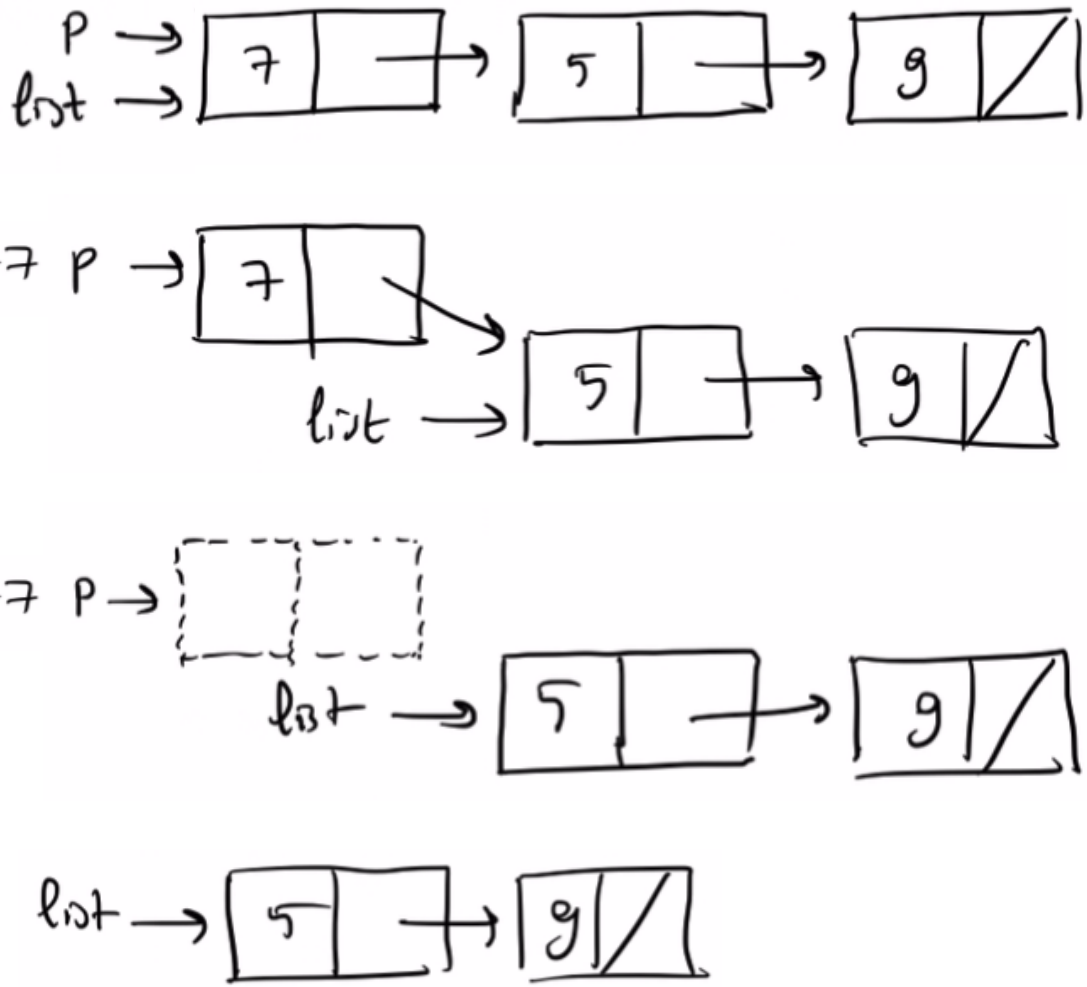
x=info(p);

Removing a node from the front of a list

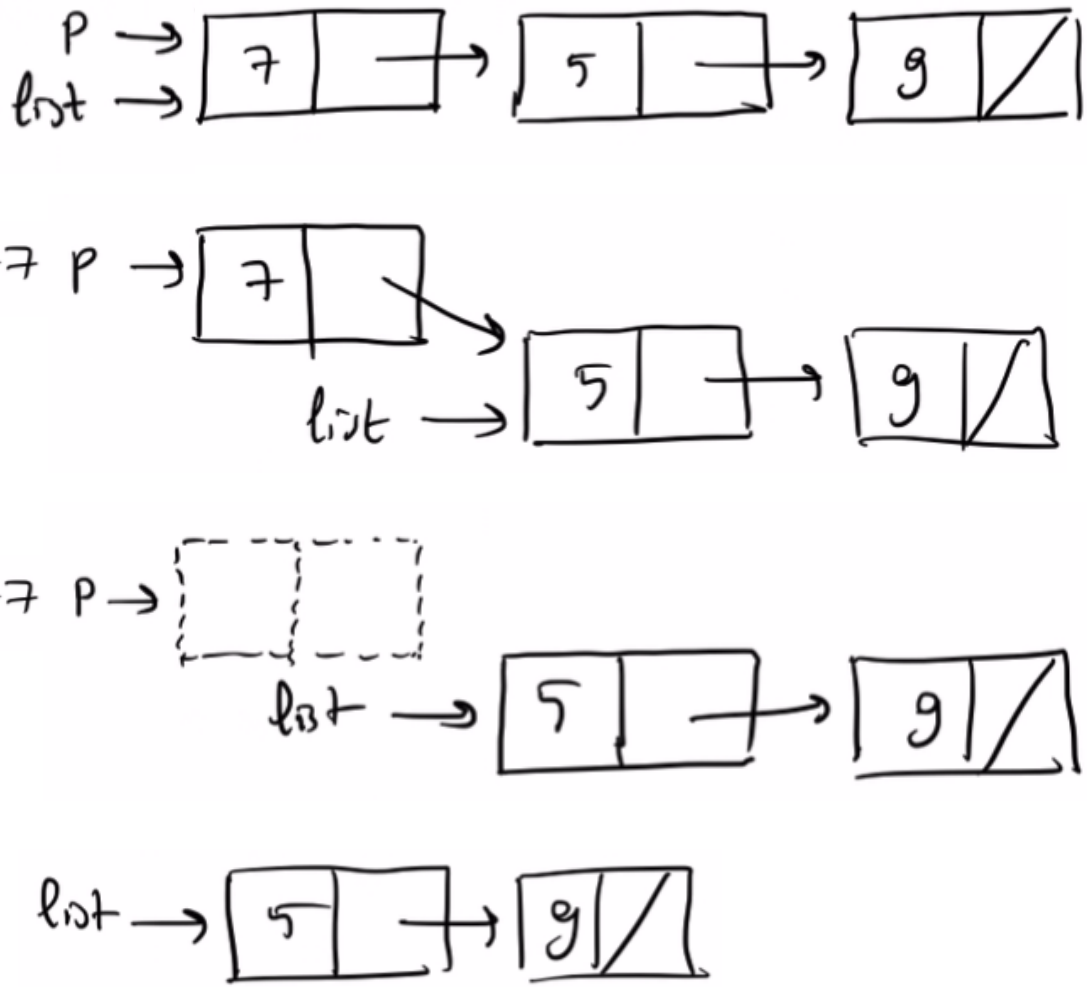


freenode(p);

Removing a node from the front of a list

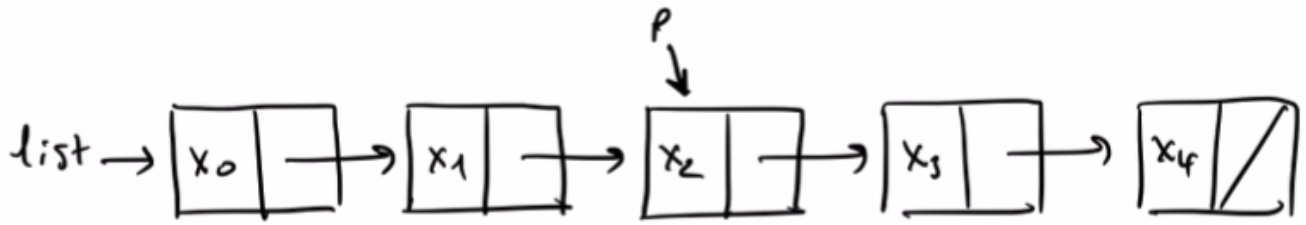


Removing a node from the front of a list

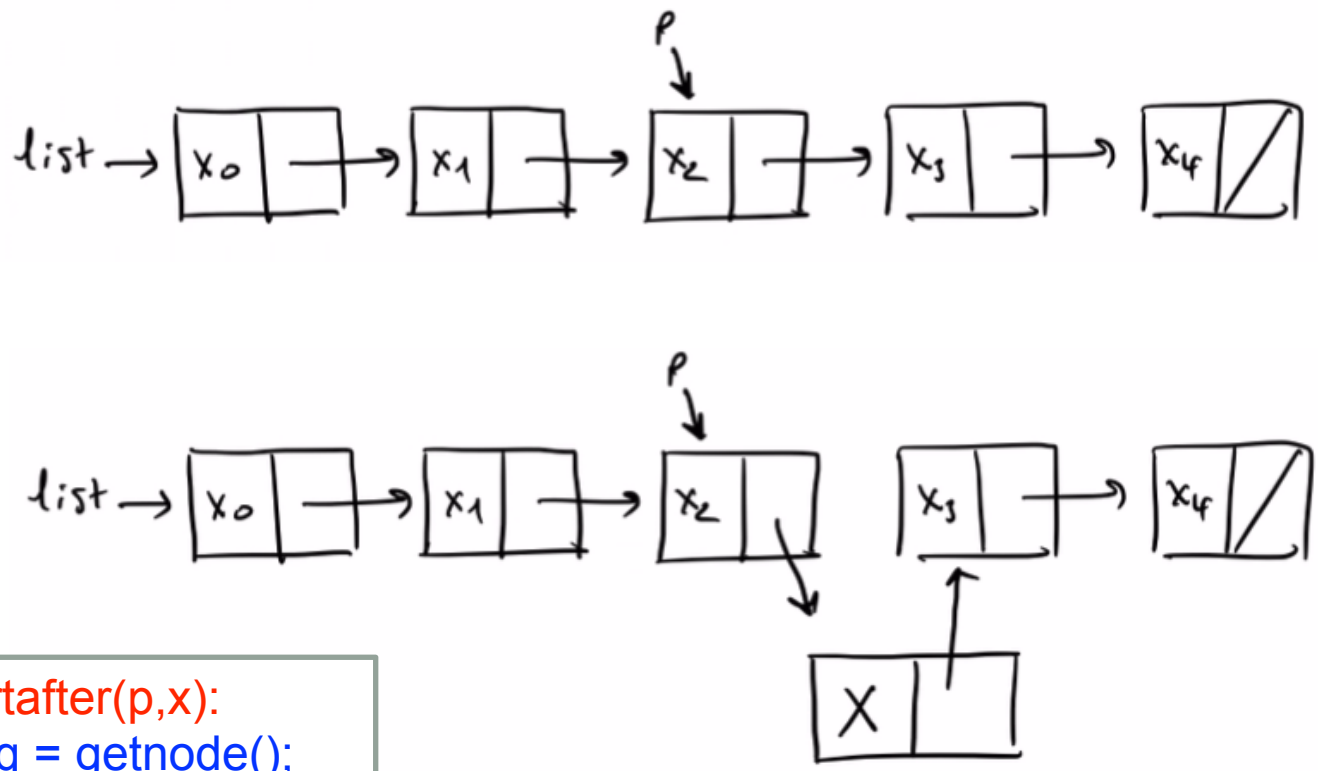



```
// remove a node from the front of the list:
pop():
        p = list;
        list = next(p);
        x = info(p);
        freenode(p);
```

Inserting data after a node

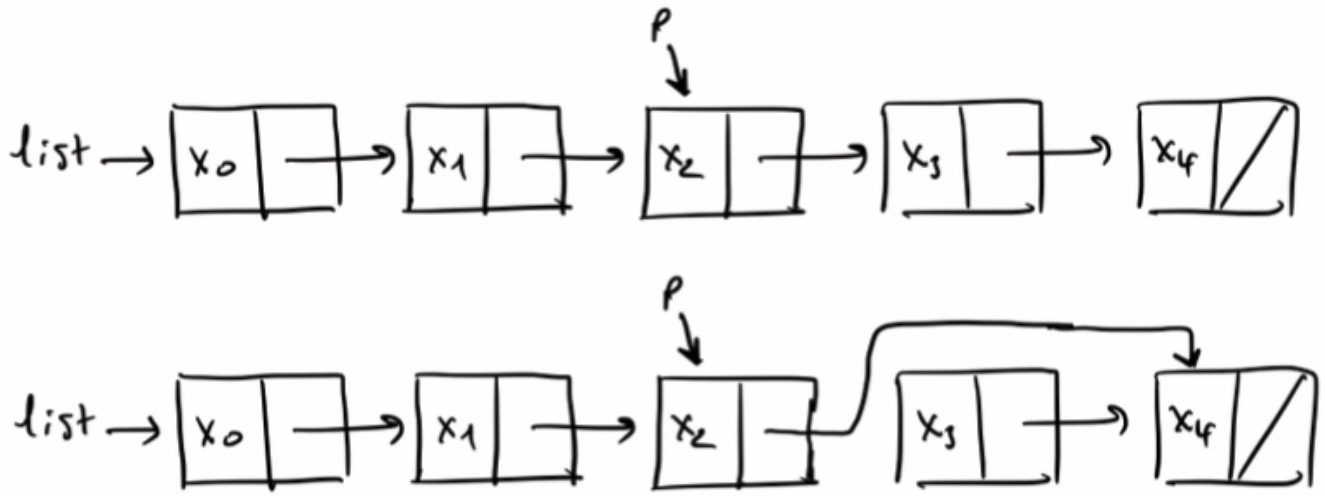


Inserting data after a node

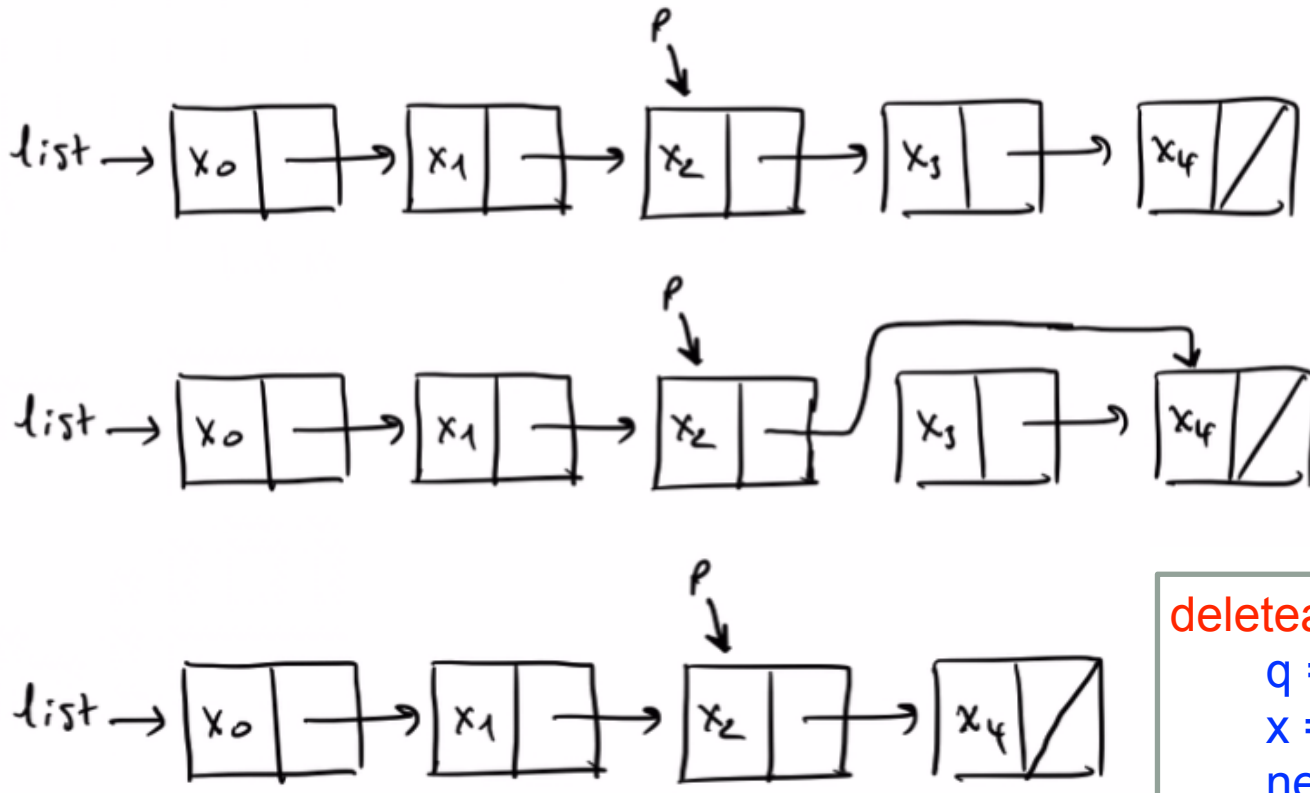



```
insertafter(p,x):
        q = getnode();
        info(q) = x;
        next(q) = next(p);
        next(p) = q;
```

Deleting data after a node



Deleting data after a node




```
deleteafter(p):
        q = next(p);
        x = info(q);
        next(p) = next(q);
        freenode(q);
```

Examples of List Operations

Delete all occurrences of 4 from a list *list*

Algorithm:

- -Traverse the list in a search for nodes that contain 4 in their info fields
- -Delete each such node (p):
  - -keep a pointer to the predecessor of that node (q)

Examples of List Operations

Delete all occurrences of 4 from a list *list*

Algorithm:

-Traverse the list in a search for nodes that contain 4 in their info fields

-Delete each such node (p):

-keep a pointer to the predecessor of that node (q)

Use ***pop*** operation to remove the nodes from the beginning of the list

Use ***deleteafter*** operation to remove nodes from the middle of the list

Examples of List Operations

Delete all occurrences of 4 from a list *list*

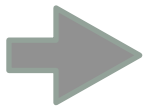
Algorithm:

-Traverse the list in a search for nodes that contain 4 in their info fields

-Delete each such node (p):

-keep a pointer to the predecessor of that node (q)

```
q = null;
p = list;
while (p!=null){
    if (info(p) == 4)
        if(q==null){
            //remove the first node of the list
            x = pop(list);
            p = list;
        }
```



Cntd. from the next slide

Examples of List Operations

Delete all occurrences of 4 from a list *list*

```
q = null;
p = list;
while (p!=null){
    if (info(p) == 4)
        if(q==null){
            //remove the first node of the list
            x = pop(list);
            p = list;
        }
        else {
            // delete the node after q and move up p
            p = next(p);
            x = deleteafter(q);
        }
    else {
        //continue traversing the list
        q = p;
        p = next(p);
    }
}
```

Examples of List Operations

Delete all occurrences of 4 from a list *list*

```
q = null;
p = list;
while (p!=null){
    if (info(p) == 4)
        if(q==null){
            //remove the first node of the list
            x = pop(list);
            p = list;
        }
        else {
            // delete the node after q and move up p
            p = next(p);
            x = deleteafter(q);
        }
    else {
        //continue traversing the list
        q = p;
        p = next(p);
    }
}
```

The practice of using two pointers, one following the other, is very common in working with lists

Examples of List Operations

Insert an item x to an ordered list in its proper place.

-Use ***push*** operation to add a node to the front of the list
-Use ***insertafter*** operation to add a node in the middle of the list

Examples of List Operations

Insert an item x to an ordered list in its proper place.

-Use ***push*** operation to add a node to the front of the list
-Use ***insertafter*** operation to add a node in the middle of the list

```
q = null;
for (p = list; p != null && x > info(p); p = next(p))
    q = p;
//insert the node containing x at this point
if(q == null) //insert x at the head of the list
    push(list, x);
else
    insertafter(q, x);
```

How many nodes are accessed on the average to insert a new element?

Examples of List Operations

Insert an item x to an ordered list in its proper place.

How many nodes are accessed on the average to insert a new element?

```
q = null;
for (p = list; p != null && x > info(p); p =
next(p))
        q = p;
//insert the node containing x at this point
if(q == null) //insert x at the head of the list
        push(list, x);
else
        insertafter(q, x);
```

Assume that the list contains n nodes;

Then, x can be placed in one of the n+1 positions:

$$A = (1/(n+1))*1+(1/(n+1))*2+\ldots+(1/(n+1))*(n-1)+(1/(n+1))*n+(1/(n+1))*n$$

if x is less than the first node; only one node is accessed

Examples of List Operations

Insert an item x to an ordered list in its proper place.

How many nodes are accessed on the average to insert a new element?

```
q = null;
for (p = list; p != null && x > info(p); p =
next(p))
        q = p;
//insert the node containing x at this point
if(q == null) //insert x at the head of the list
        push(list, x);
else
        insertafter(q, x);
```

Assume that the list contains n nodes;

Then, x can be placed in one of the n+1 positions:

$$A = (1/(n+1))*1+(1/(n+1))*2+\ldots+(1/(n+1))*(n-1)+(1/(n+1))*n+(1/(n+1))*n$$

if x is less than the second node; two nodes are accessed

Examples of List Operations

Insert an item x to an ordered list in its proper place.

How many nodes are accessed on the average to insert a new element?

```
q = null;
for (p = list; p != null && x > info(p); p =
next(p))
        q = p;
//insert the node containing x at this point
if(q == null) //insert x at the head of the list
        push(list, x);
else
        insertafter(q, x);
```

Assume that the list contains n nodes;

Then, x can be placed in one of the n+1 positions:

$$A = (1/(n+1))*1+(1/(n+1))*2+\ldots+(1/(n+1))*(n-1)+(1/(n+1))*n+(1/(n+1))*n$$

The probability of inserting any position (all positions are equally likely)

Examples of List Operations

Insert an item x to an ordered list in its proper place.

How many nodes are accessed on the average to insert a new element?

```
q = null;
for (p = list; p != null && x > info(p); p =
next(p))
        q = p;
//insert the node containing x at this point
if(q == null) //insert x at the head of the list
        push(list, x);
else
        insertafter(q, x);
```

Assume that the list contains n nodes;

Then, x can be placed in one of the n+1 positions:

$$A = (1/(n+1))*1+(1/(n+1))*2+\ldots+(1/(n+1))*(n-1)+(1/(n+1))*n+(1/(n+1))*n$$

$$A = (1/(n+1))*(1+2+\ldots+n)+(n/(n+1))$$

$$A = (1/(n+1))*(n*(n+1)/2) + (n/(n+1))$$

$$A = n/2 + n/(n+1)$$

$$\sim n/2$$

Array Implementation of Linked Lists

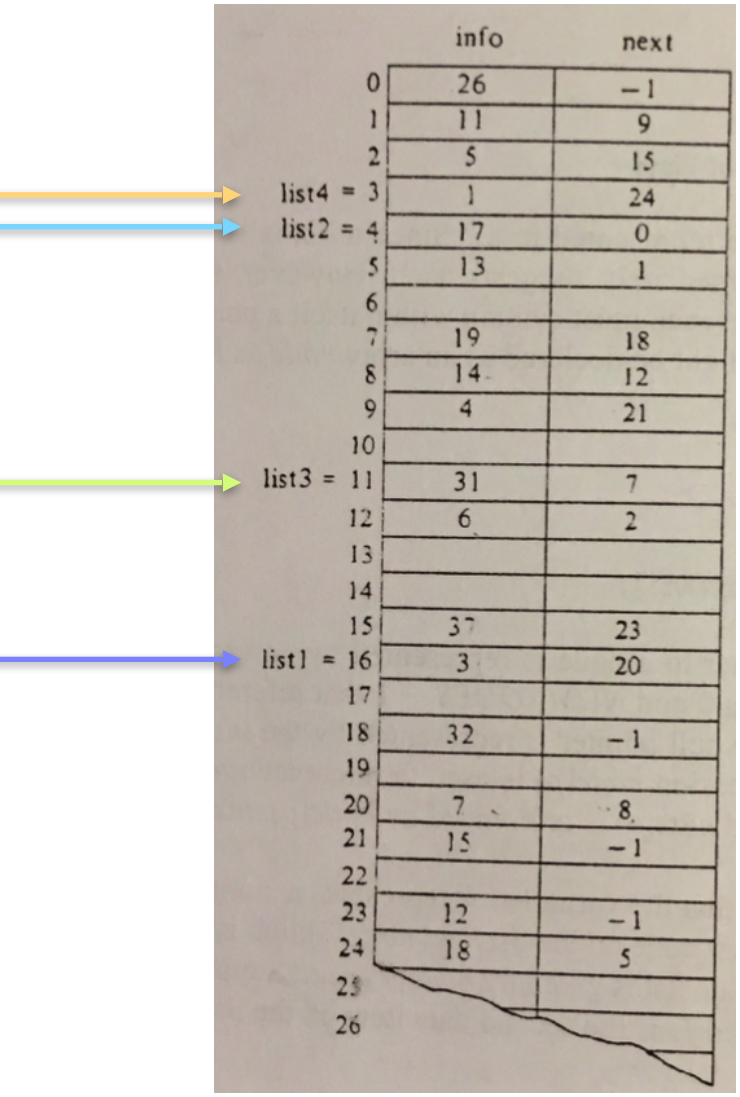
```
struct nodetype {
    int info, next;
};
struct nodetype node[NUMNODES];
```

A pointer to a node is represented by an array index.
The NULL pointer is represented by the integer -1

node(p) —-> node[p]
info(p) ——> node[p].info
next(p)——> node[p].next

Note that NULL is used in the implementations of abstract operations to represent null

Array Implementation of Linked Lists



		info	next
	0	26	−1
	1	11	9
	2	5	15
list4 =	3	1	24
list2 =	4	17	0
	5	13	1
	6		
	7	19	18
	8	14	12
	9	4	21
	10		
list3 =	11	31	7
	12	6	2
	13		
	14		
	15	37	23
list1 =	16	3	20
	17		
	18	32	−1
	19		
	20	7	8
	21	15	−1
	22		
	23	12	−1
	24	18	5
	25		
	26		

Array nodes can be scattered throughout the array node in any arbitrary order.

Each node carries within itself the location of its successor until the last node of the list.

More than one list can be stored in the allocated array:
list1->{3,7,14,6,5,37,12}
list2->{17,26}
list3->{31,19,32}
list4->{1,18,13,11,4,15}

Figure ref: Data Structures Using C and C++, Langsam et al. Prentice Hall (Second Edn.) Page 204.

Array Implementation of Linked Lists

Create an available list for node allocations
Initially all nodes are unused; available

```
avail = 0;
for(int i=0;i<NUMNODES-1;i++)
    node[i].next = i+1;
node[NUMNODES-1].next = -1;
```

Array Implementation of Linked Lists

Create an available list for node allocations
Initially all nodes are unused; available

```
avail = 0;
for(int i=0;i<NUMNODES-1;i++)
    node[i].next = i+1;
node[NUMNODES-1].next = -1;
```

Nodes are linked in their natural order.
There is no reason other than convenience to link in this order

Array Implementation of Linked Lists

Assume node and avail are accessible from all the functions

```
int getnode()
{
    int p;
    if(avail == -1){
        cout<<"ERROR: overflow!"<<endl;
        exit(1);
    }
    p = avail;
    avail = node[avail].next;
    return p;
}
```

When a node is needed, it is obtained from the available list!
if avail equals -1 —> no node available!

Array Implementation of Linked Lists

Accepts a pointer to a node and returns that node to the available list

```
void freenode(int p)
{
    node[p].next = avail;
    avail = p;
    return;
}
```

Array Implementation of Linked Lists

Primitive operations are straightforward:

```
//inserts x into a node following the node
//pointed to by p
void insertafter(int p, int x)
{
    int q;
    if(p == -1){ // ensure that p is not NULL
        cout<<"ERROR: void insertion!"<<endl;
        return;
    }
    q = getnode();
    node[q].info = x;
    node[q].next = node[p].next;
    node[p].next = q;
    return;
}
```

Array Implementation of Linked Lists

Primitive operations are straightforward:

```
// deletes the node following node(p)
// returns the content of the deleted node
int deleteafter(int p)
{
    int x, q;
    if((p == -1) || (node[p].next == -1)){
        cout<<"ERROR: void deletion!"<<endl;
        return ERRORCODE; //INT_MIN as error code
    }
    q = node[p].next;
    x = node[q].info;
    node[p].next = node[q].next;
    freenode(q);
    return x;
}
```

Limitations of Array Implementation

- A fixed set of nodes are represented in an array at the start of execution
- A pointer to a node is represented by the array indexes

There are two disadvantages:

- The number of nodes that are needed can not be predicted beforehand
  - -The allocated space may be insufficient
- All the declared memory must remain allocated
  - -Even when it is not needed.

Limitations of Array Implementation

- A fixed set of nodes are represented in an array at the start of execution
- A pointer to a node is represented by the array indexes

There are two disadvantages:

- The number of nodes that are needed can not be predicted beforehand
  - -The allocated space may be insufficient
- All the declared memory must remain allocated
  - -Even when it is not needed.

SOLUTION: USE DYNAMIC ALLOCATION OF NODES

When a node is needed -> allocate it

When it is no longer needed -> release it

Linked List Implementation with Dynamic Allocation

```
struct node {
    int info;
    struct node *next;
};
typedef struct node* NODEPTR;
```

Linked List Implementation with Dynamic Allocation

```
struct node {
    int info;
    struct node *next;
};
typedef struct node* NODEPTR;
```

```
NODEPTR getnode()
{
    NODEPTR p;
    p = new node;
    return p;
}
```

Linked List Implementation with Dynamic Allocation

```
struct node {
    int info;
    struct node *next;
};
typedef struct node* NODEPTR;
```

```
void freenode(NODEPTR p)
{
    delete p;
}
```

Linked List Implementation with Dynamic Allocation

```
void insertafter(NODEPTR p, int x)
{
    NODEPTR q;
    if(p == NULL){
        cout<<"ERROR: void insertion!"<<endl;
        exit(1);
    }
    q = getnode();
    q->info = x;
    q->next = p->next;
    p->next = q;
}
```

Linked List Implementation with Dynamic Allocation

```
int deleteafter(NODEPTR p)
{
    int x;
    NODEPTR q;
    if ((p == NULL) || (p->next == NULL)){
        cout<<"ERROR: void deletion!"<<endl;
        exit(1);
    }
    q = p->next;
    x = q->info;
    p->next = q->next;
    freenode(q);
    return x;
}
```

Similarity of both implementations

```
//inserts x into a node following the node
//pointed to by p
void insertafter(int p, int x)
{
    int q;
    if(p == -1){ // ensure that p is not NULL
        cout<<"ERROR: void insertion!"<<endl;
        return;
    }
    q = getnode();
    node[q].info = x;
    node[q].next = node[p].next;
    node[p].next = q;
}
```

Array

```
void insertafter(NODEPTR p, int x)
{
    NODEPTR q;
    if(p == NULL){
        cout<<"ERROR: void insertion!"<<endl;
        exit(1);
    }
    q = getnode();
    q->info = x;
    q->next = p->next;
    p->next = q;
}
```

Dynamic allocation

Similarity of both implementations

```
// deletes the node following node(p)
// returns the content of the deleted node
int deleteafter(int p)
{
    int x, q;
    if((p == -1) || (node[p].next == -1)){
        cout<<"ERROR: void deletion!"<<endl;
        return ERRORCODE; //INT_MIN as error
code
    }
    q = node[p].next;
    x = node[q].info;
    node[p].next = node[q].next;
    freenode(q);
    return x;
}
```

Array

```
int deleteafter(NODEPTR p)
{
    int x;
    NODEPTR q;
    if ((p == NULL) || (p->next == NULL)){
        cout<<"ERROR: void deletion!"<<endl;
        exit(1);
    }
    q = p->next;
    x = q->info;
    p->next = q->next;
    freenode(q);
    return x;
}
```

Dynamic allocation

Array vs. Linked List:

Memory requirements

Array	Linked List
Has fixed size	No unused memory
	Extra memory for pointer variables
Memory may not be available as one large block	Memory may be available as multiple small blocks

Array vs. Linked List:

Cost of accessing an element

- If we know the starting address of the array, we can calculate the address of the ith element in the array. This takes constant (O(1)) time for any element in the array!
- To find the address of the i'th element in the linked list, we need to traverse all elements until that element (O(n) time).

Array vs. Linked List:

Cost of inserting an element

Cost of inserting a new element (worst case):	Array	Linked List
a) At the beginning	O(n)	O(1)
b) At the end	O(1) (if array is not full)	O(n)
c) At *i*'th position	O(n)	O(n)

- To insert an element to the beginning of an array all elements need to be shifted by one to the next address.

- To add an element to the end of a linked list all elements need to be traversed.

General Lists: List as ADT

- A list is a sequence of objects called elements
- Associated with each list element is a value

  -the number 5 may appear on a list twice. Each appearance is a distinct element of the list; but the values of the two elements are the same.

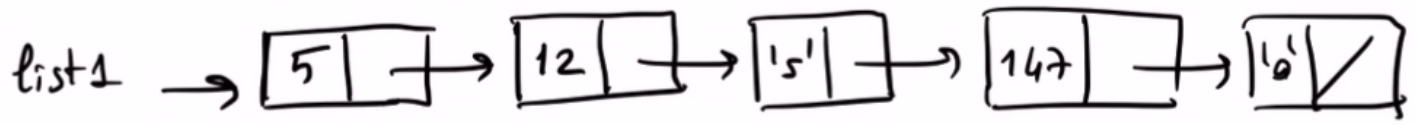
- There is no reason to assume that the elements of a list must be of the same type.

General Lists: List as ADT

- A list is a sequence of objects called elements
- Associated with each list element is a value

  -the number 5 may appear on a list twice. Each appearance is a distinct element of the list; but the values of the two elements are the same.

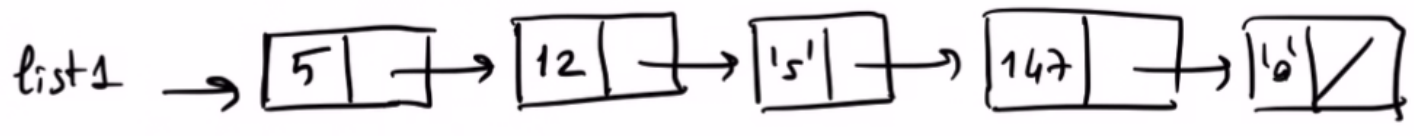
- There is no reason to assume that the elements of a list must be of the same type.



List of integers and characters

Notation for abstract general lists

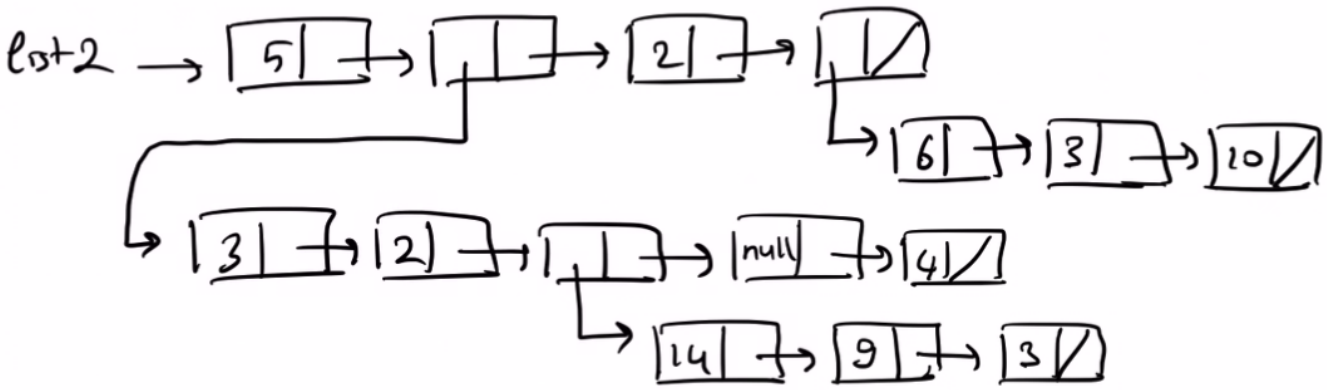
A list may be denoted by a parenthesized enumeration of its elements separated by commas



list1=(5,12,’s’,147,’a’)

Notation for abstract general lists

The null list is denoted by an empty paranthesis pair, such as ().



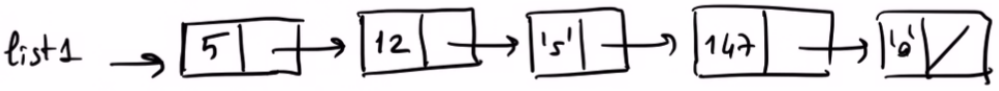
list2=(5,(3,2,(14,9,3),(),4),2,(6,3,10))

Abstract operations on general lists

- If a list is a nonempty list, *head(list)* is defined as the value of the first element of the list.
  - head(list) may be either a list or a simple data item.
- If a list is a nonempty list, *tail(list)* is defined as the list obtained by removing the first element of list.
  - tail(list) must be a (possibly null) list.
- If a list is the empty list, head(list) and tail(list) are not defined.

Abstract operations on general lists

- If a list is a nonempty list, *head(list)* is defined as the value of the first element of the list.
  - head(list) may be either a list or a simple data item.
- If a list is a nonempty list, *tail(list)* is defined as the list obtained by removing the first element of list.
  - tail(list) must be a (possibly null) list.
- If a list is the empty list, head(list) and tail(list) are not defined.

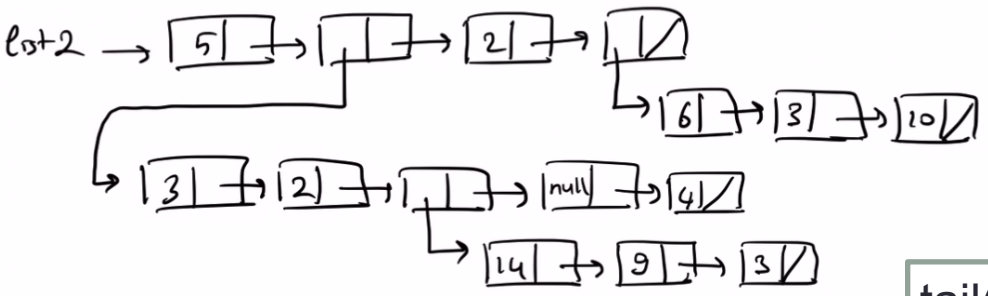


list1=(5,12,'s',147,'a')

head(list1) = 5
tail(list1) = (12,'s',147,'a')
head(tail(list1)) = 12
tail(tail(list1)) = ('s',147,'a')

Abstract operations on general lists

- If a list is a nonempty list, *head(list)* is defined as the value of the first element of the list.
  - head(list) may be either a list or a simple data item.
- If a list is a nonempty list, *tail(list)* is defined as the list obtained by removing the first element of list.
  - tail(list) must be a (possibly null) list.
- If a list is the empty list, head(list) and tail(list) are not defined.



list2=(5,(3,2,(14,9,3),(),4),2,(6,3,10))

tail(list2) = ((3,2,(14,9,3),(),4),2,(6,3,10))
head(tail(list2)) = (3,2,(14,9,3),(),4)
head(head(tail(list2))) = 3

Abstract operations on general lists

Operations that extract information from a list

- *first(list)* returns the first element of list *list*.
  - If list is empty, *first(list)* is a special null element, i.e. *nullelt*.
  - Note that head(list) produces a *value*, whereas the first operation produces an *element*.
- *info(elt)* returns the value of the list element ***elt***.
  - *head(list)* equals *info(first(list))*
- *next(elt)* returns the element that follows the element ***elt*** on the list.
  - Assumption: An element can have only one follower.
- *nodetype(elt)* returns the type of the element ***elt***.
  - If the enumerated constants *CH, INTGR, LST* represents the types *character, integer, list*, respectively, nodetype(first(list1)) equals *INTGR*.

Abstract operations on general lists

Operations that modify a list

- *addon(list, x)* returns a new list that has x as its head and list as its tail.
  - Ex: L1 = (3,4,7), L2 = addon(L1, 5); L2 is a new list = (5,3,4,7)
- *setinfo(elt, x)* sets the value of the list element ***elt*** to x.
  - *setinfo(first(list), x) :* sets the value of the first element of the list to x
  - This operation can be abbreviated as *sethead(list, x).*
    - *Ex: L1 = (5,10,8), sethead(L1, 18)* changes L1 to *(18,10,8)*
  - *sethead(list, x) is equivalent to list = addon(tail(list),x)*
- *setnext(elt1,elt2)* modifies the list containing elt1 such that *next(elt1) = elt2*.
  - elt1 can not be the null element.
  - next(elt2) is unchanged.
- *settail(list1, list2)* sets the tail of list1 to list2.
  - is defined as *setnext(first(list1),first(list2))*
  - *Ex: list = (5,9,3,7,8,6), settail(list,(8))* changes the value of *list* to *(5,8).* *settail(list,(4,2,7))* changes its value to *(5,4,2,7).*
  - *Note that settail(list, l) is* equivalent to *list = addon(l, head(list))*

Abstract operations on general lists

Sample algorithms that use these operations

Example 1: Add 1 to every integer that is an element of list *list*.

```
p = first(list);
while (p != nullelt) {
  if (nodetype(p) == INTGR)
    setinfo(p, info(p) + 1);
  p = next(p);
}
```

Abstract operations on general lists

Sample algorithms that use these operations

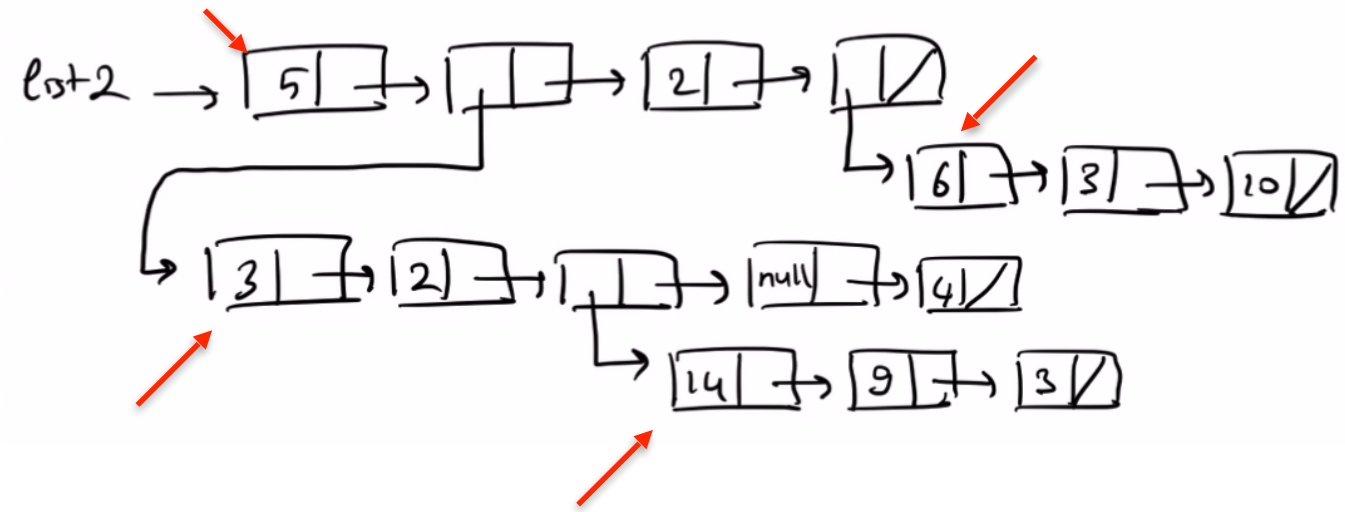
Example 2: Delete any character element in the list *list*, whose value is ‘w’.

```
q = nullelt;
p = first(list);
while (p != nullelt)
  if (info(p) == ‘w’){
    // remove node from the list
    p = next(p);
    if (q == nullelt)
      list = tail(list);
    else
      setnext(q,p);
    }
  else {
    q = p;
    p = next(p);
  }
```

Abstract operations on general lists

Sample algorithms that use these operations

Example 3: Extend add 1 to every integer elements including the ones in the list elements of the list *list*.



Define a recursive solution for simplicity

Abstract operations on general lists

Sample algorithms that use these operations

Example 3: Extend add 1 to every integer elements including the ones in the list elements of the list *list*.

```
addoneRec(list):
  p = first(list);
  while (p != nullelt) {
    if (nodetype(p) == INTGR)
      setinfo(p, info(p) + 1);
    else
      if (nodetype(p) == LST)
        addoneRec(info(p));
    p = next(p);
  }
```

Representation of General Lists

How these operations change in the linked list representation of the general lists?

Reading Assignment: Read the Linked List Representation of General Lists from the reference book Chapter 9. Supplementary file will be provided through Piazza for the related part.