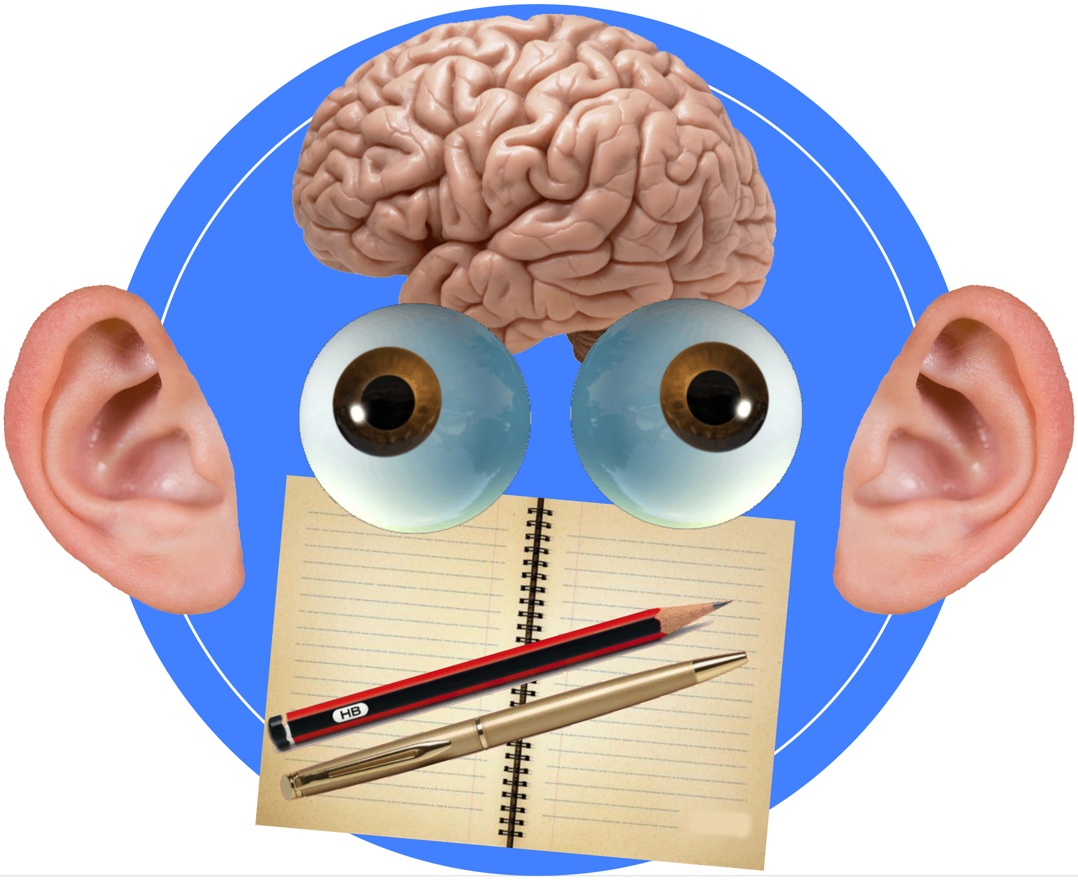


University of Guelph Prof. Pascal Matsakis

CIS2520

1. Lists

Required Textbook: “Data Structures...” by Standish (ISBN: 0-201-59118-9 © 1995)

CIS2520 Lists

LIST ADT

interface (in C)
sequential implementation
linked implementation
conclusion

LIST ADT: The Book Example 1.5

create	()
insert Web in 1st position	(Web)
insert C in 2nd position	(Web,C)
insert G in 1st position	(G,Web,C)
insert L in 1st	(L,G,Web,C)
insert Wes in 4th	(L,G,Web,Wes,C)
insert Wo in 2nd	(L,Wo,G,Web,Wes,C)
delete the item in 5th position	(L,Wo,G,Web,C)
insert B in 6th position	(L,Wo,G,Web,C,B)

LIST ADT: Main Operations 1.6

create	**create**: $\emptyset$ ➔ List[T]
insert an item in some position	**insert**: TxNxList[T] ➔ List[T]
delete the item in some position	**delete**: NxList[T] ➔ List[T]
determine whether **full**	**full**: List[T] ➔ Boolean
determine whether **empty**	**empty**: List[T] ➔ Boolean
find the **number** of items	**length**: List[T] ➔ N
find the **item** in some position	**peek**: NxList[T] ➔ T

LIST ADT: Preconditions and Postconditions 1.7

create: ∅ ➔ List[T]

constructor

insert: TxNxList[T] ➔ List[T]
delete: NxList[T] ➔ List[T]

mutators

full: List[T] ➔ Boolean
empty: List[T] ➔ Boolean
length: List[T] ➔ N
peek: NxList[T] ➔ T

accessors

{¬full(L) ∧ length(L)≥P} insert(I,P,L)
{length(L)>P} peek(P,L)

preconditions

insert(I,P,L) {¬ empty(L)}
delete(P,L) {length(L)=length(**old** L)−1}

postconditions

empty()=(length(L)=0)

invariant

LIST ADT: Axioms 1.8

create: ∅ ➔ List[T]

constructor

insert: TxNxList[T] ➔ List[T]
delete: NxList[T] ➔ List[T]

mutators

full: List[T] ➔ Boolean
empty: List[T] ➔ Boolean
length: List[T] ➔ N
peek: NxList[T] ➔ T

accessors

---

empty(create())
¬ empty(insert(I,P,L))
¬ full(delete(P,L))
peek(P,insert(I,P,L))=I
delete(P,insert(I,P,L))=L

axioms

LIST ADT: Other Operations 1.9

create: ∅ ➔ List[T] } **constructor**

insert: TxNxList[T] ➔ List[T]
delete: NxList[T] ➔ List[T] } **mutators**

full: List[T] ➔ Boolean
empty: List[T] ➔ Boolean
length: List[T] ➔ N
peek: NxList[T] ➔ T } **accessors**

Other typical terms and operations:

create | new, initialize,
print | show,
compare,
copy, clone,
free | destroy } **destructor ***

list ADT

INTERFACE (IN C)

sequential implementation
linked implementation
conclusion

INTERFACE (IN C): Why C? 1.11

- ✧ C is one of the most widely used programming languages of all time
- ✧ C allows efficient implementations of algorithms and data structures (e.g., GNU Scientific Library, Mathematica, MATLAB)
- ✧ C has directly or indirectly influenced many later languages (e.g., C++, Objective-C, C#, Java, JavaScript, Perl, PHP, Python)
- ✧ compilers, libraries, and interpreters of other programming languages are often implemented in C (e.g., Perl, PHP, Python)

INTERFACE (IN C): Example 1.12

```
#include "ListType.h"       /* imports the data type definitions of */
                           /* Item and List */

extern void Initialize (List *L);
extern void Insert (Item X, int position, List *L);
extern void Delete (int position, List *L);
extern int Full (List *L);
extern int Empty (List *L);
extern int Length (List *L);
extern void Peek (int position, List *L, Item *X);
extern void Destroy (List *L);
```

ListInterface.h

```
#include "ListInterface.h"
......
```

myProgram.c

list ADT

interface (in C)

SEQUENTIAL IMPLEMENTATION

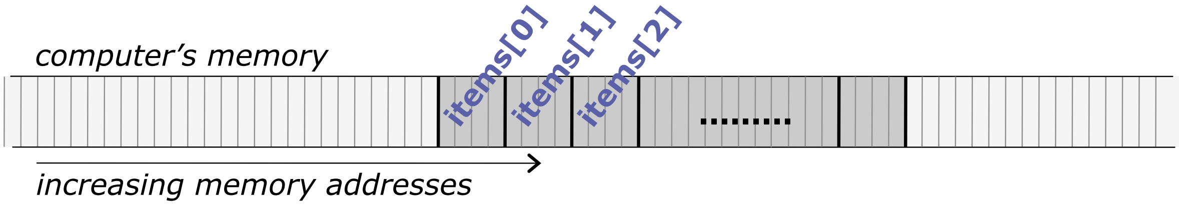
linked implementation

conclusion

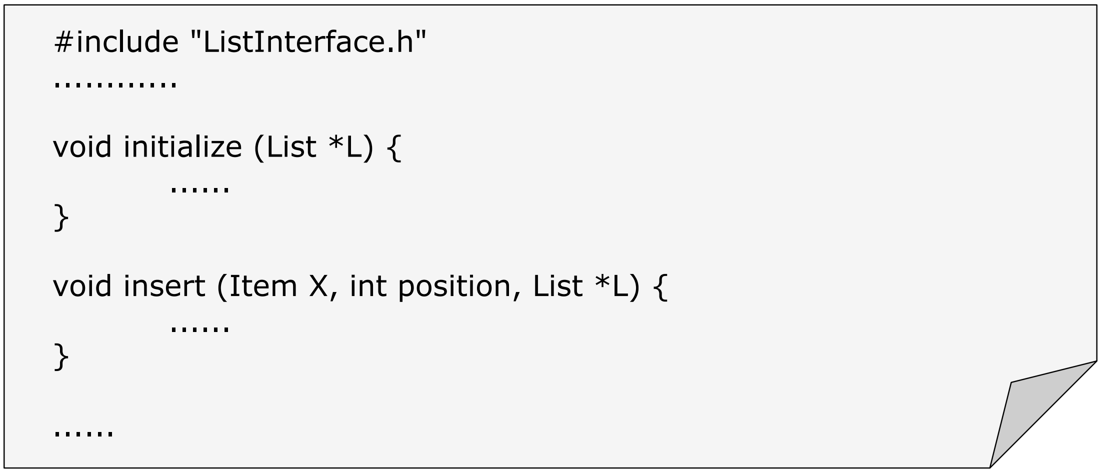
SEQUENTIAL IMPLEMENTATION: Example (1/2) 1.14

```
#include "ArbitraryInterface.h"        /* or could simply be, e.g., */
typedef Arbitrary Item;               /* typedef int Item; */

#define MAXLISTSIZE 100
typedef struct {
        Item items[MAXLISTSIZE];
        int size;
} List;
```

ListType.h



SEQUENTIAL IMPLEMENTATION: Example (2/2)

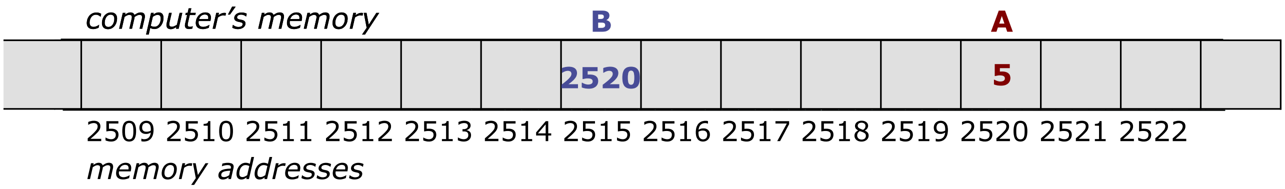


```
#include "ListInterface.h"
............

void initialize (List *L) {
          ......
}

void insert (Item X, int position, List *L) {
          ......
}

......
```

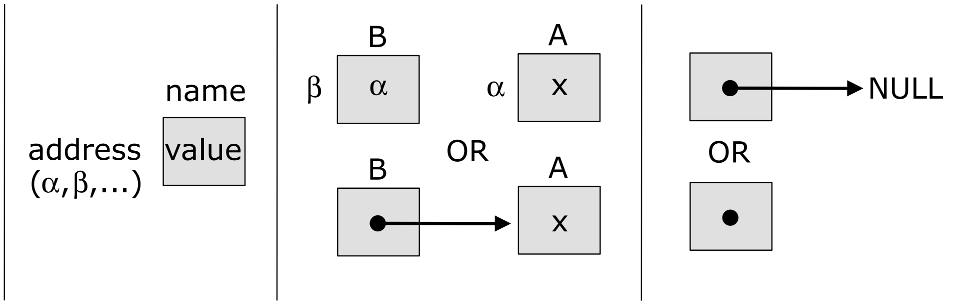
ListImplementation.c

list ADT
interface (in C)
sequential implementation
LINKED IMPLEMENTATION
conclusion

LINKED IMPLEMENTATION: Notation 1.17



```
int *B;
int A;
A=5;
B=&A;
```

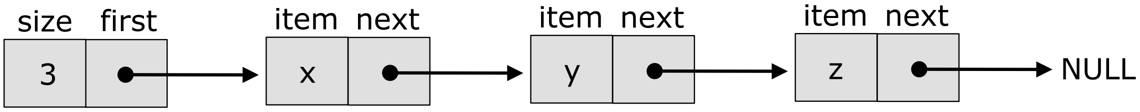


LINKED IMPLEMENTATION: One-Way (1/3) 1.18

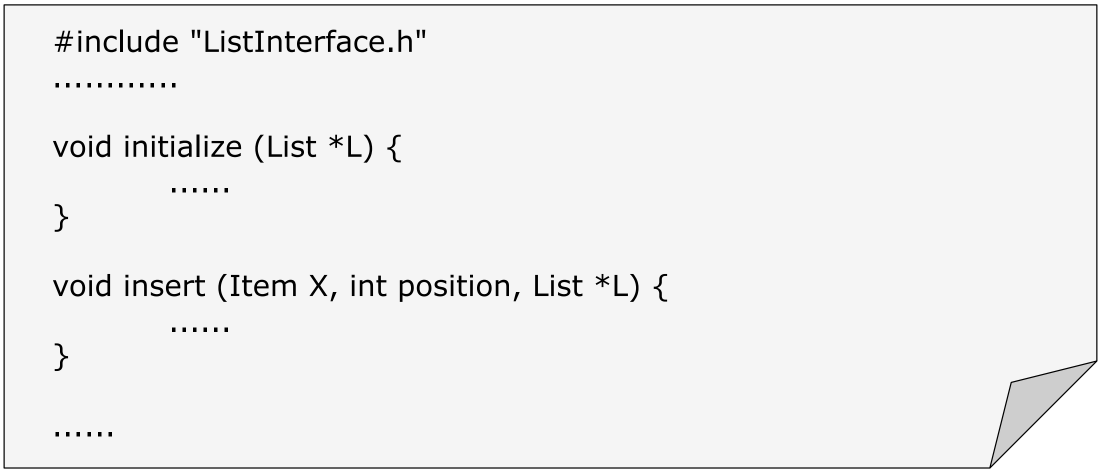
```
#include "ArbitraryInterface.h"           /* or could simply be, e.g., */
typedef Arbitrary Item;                  /* typedef int Item; */

typedef struct ListNodeTag {
                    Item item;
                    struct ListNodeTag *next;
} ListNode;

typedef struct {
                    int size;
                    ListNode *first;
} List;
```

ListType.h

LINKED IMPLEMENTATION: One-Way (2/3)



LINKED IMPLEMENTATION: One-Way (3/3)



```
#include "ListInterface.h"
...........

void initialize (List *L) {
          ......
}

void insert (Item X, int position, List *L) {
          ......
}

......
```

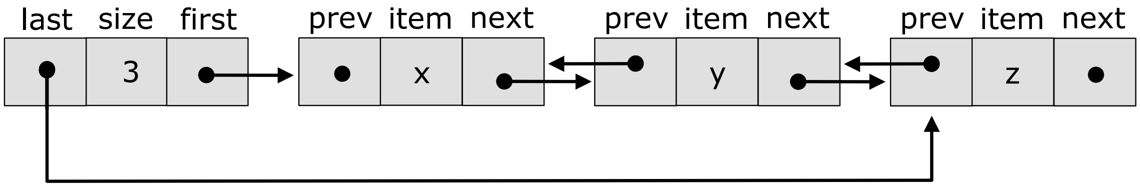
ListImplementation.c

LINKED IMPLEMENTATION: Two-Way (1/3) 1.21

```
#include "ArbitraryInterface.h”
typedef Arbitrary Item;

typedef struct ListNodeTag {
                Item item;
                struct ListNodeTag *next;
                struct ListNodeTag *prev;
} ListNode;

typedef struct {
                int size;
                ListNode *first;
                ListNode *last;
} List;
```

ListType.h

LINKED IMPLEMENTATION: Two-Way (2/3) 1.22



LINKED IMPLEMENTATION: Two-Way (3/3) 1.23

```
#include "ListInterface.h"
...........

void initialize (List *L) {
          ......
}

void insert (Item X, int position, List *L) {
          ......
}

......
```

ListImplementation.c

list ADT
interface (in C)
sequential implementation
linked implementation

CONCLUSION

CONCLUSION: Advantages of ADT 1.25

- ✧ ADT user just needs to know the ADT properties and abilities; no need to know how the ADT implementation works
- ✧ ADT implementation may change; but no need to change the code that uses the ADT
- ✧ ADT user can use most efficient implementation for given situation

Required Textbook: “Data Structures...” by Standish (ISBN: 0-201-59118-9 © 1995)

CONCLUSION: Sequential vs. Linked 1.26

Sequential implementation:

- ✧ static storage allocation
- ✧ storage is contiguous
- ✧ random access (index)
- ✧ *insert* and *delete* must shift existing data

Linked implementation:

- ✧ dynamic storage allocation
- ✧ storage is not contiguous
- ✧ sequential access only
- ✧ *insert* and *delete* do not change existing data

Required Textbook: “Data Structures...” by Standish (ISBN: 0-201-59118-9 © 1995)