Fakultas Informatika
School of Computing
Telkom University

CII2B4
STRUKTUR DATA

Variations of Linked List

Multi Linked List

Multi Linked List

- Linked list where each node may contain pointers to more than one nodes
  - Double linked list is also considered multi linked list
- Represents Parent-child relation
  - Tree
  - Graph

Multi Linked List – parent child relation

- Modification of single or double linked list so that they can perform better with the case at hand
- Aimed to illustrate the relation between data
  - 1 to N relation → tree
  - N to M relation → graph
- In form of :
  - List inside a list
  - Connection between 2 or more list

Example : Student - Class

- Let’s say we want to store student data and its relation with the default class
- What is the relation of the data?
  - 1 to N relation

Example : Student - Class

```
Type infotype_student <
    id      : string
    name    : string
>

Type adr_student : pointer to
elm_student

Type elm_student <
    info    : infotype_student
    next    : adr_student
>
```

```
Type infotype_class <
    class_name    : string
    supervisor    : string
>

Type adr_class : pointer to
elm_class

Type elm_class <
    info    : infotype_class
    next    : adr_class
>
```

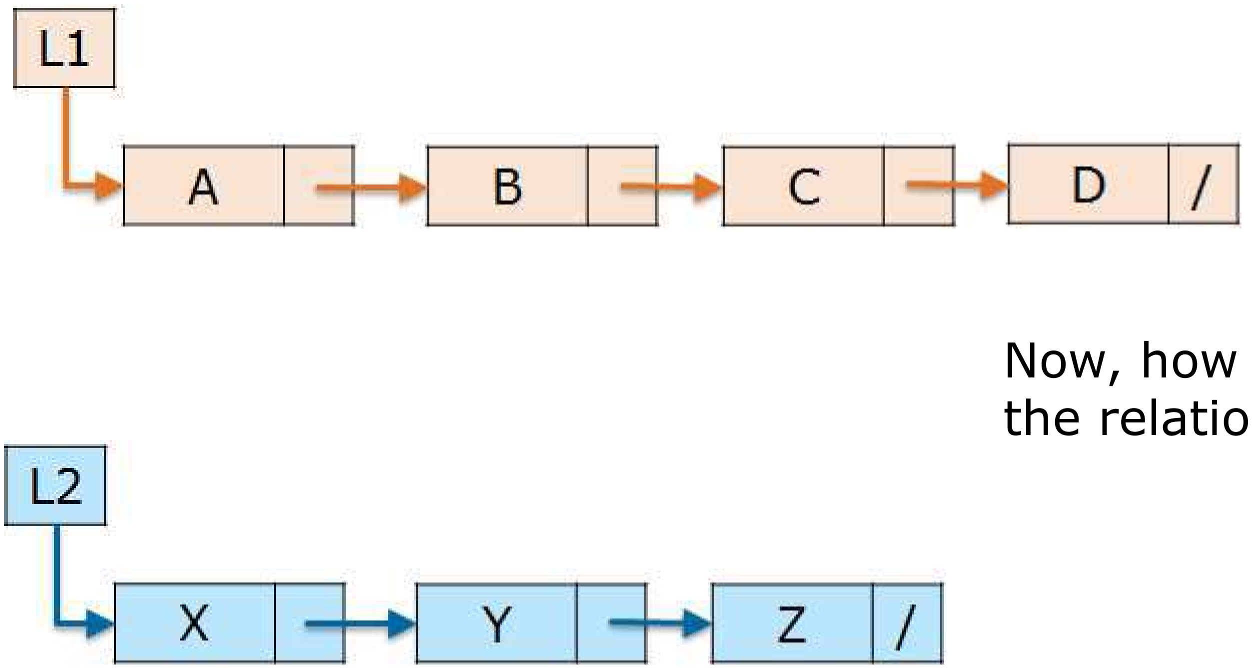
Example : Student - Class

```
Type list_student <
    first : adr_student
>
```

```
Type list_class <
    first : adr_class
>
```

```
L1 : list_student
L2 : list_class
```

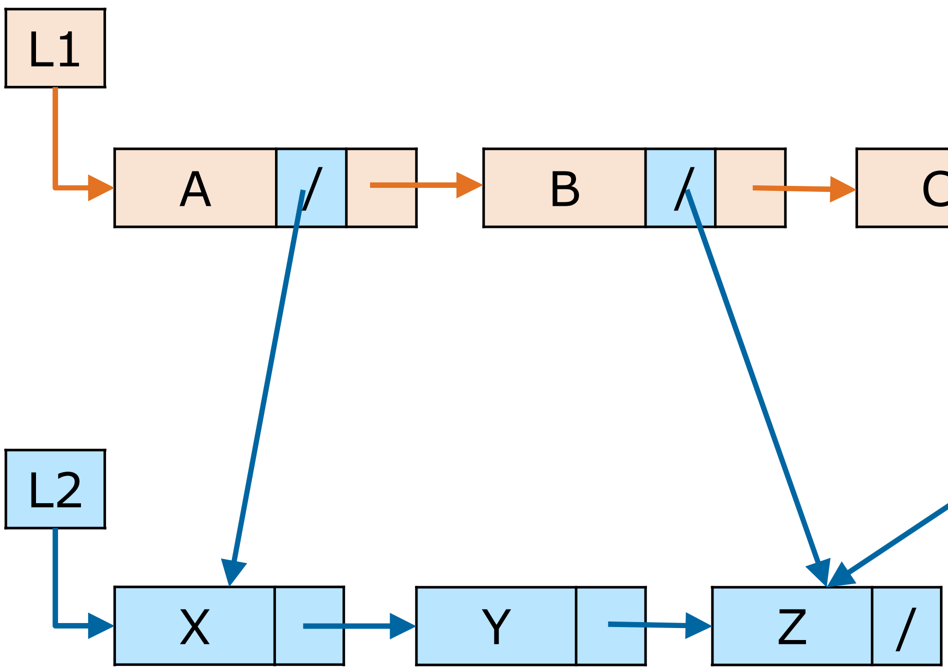
Example : Student - Class



Now, how to draw
the relation between data?

Example : Student - Class

1-to-n relation



The simplest way is
to add a pointer to elm_class
in elm_student to show which
Class the student is enrolled

Example : Student - Class

```
Type infotype_student <
    id      : string
    name    : string
>

Type adr_student : pointer to
elm_student

Type elm_student <
    info    : infotype_student
    next    : adr_student
    nextClass : adr_class
>
```

```
Type infotype_class <
    class_code    : string
    class_name    : string
    credit        : integer
>

Type adr_class : pointer to
elm_class

Type elm_class <
    info    : infotype_class
    next    : adr_class
>
```

Operations

- Insert Student and Insert Course
  - No change
- Search Student and Search Class
  - No change
- Set Class for a Student
- Search Class of a Student
- Search Student(s) of a Class

Operations

- Delete Student
  - No change
- Delete Relation
- Delete Class

Procedure Set Class

```
Procedure set_class(i/o: L1:list_student, i: L2 : list_class,
                                         id : string, class : string)
Dictionary
    define nextClass(P) (P)→nextClass
    P : adr_student
    Q : adr_class
Algorithm
    P ← search_student(L1, id)
    Q ← search_class(L2, class)
    if ( P ≠ Nil and Q ≠ Nil ) then
        nextClass(P) ← Q
```

Procedure Delete Class

```
Procedure del_class(i/o: L1:list_student, L2 : list_class, i: class : string)
Dictionary
    define nextClass(P) (P)→nextClass
    P : adr_student
    Q : adr_class
Algorithm
    Q ← search_class(L2, class)
    if ( Q ≠ Nil ) then
        P ← first(L1)
        while (P ≠ Nil) do
            if ( nextClass(P) = Q ) then
                nextClass(P) = Nil
            P ← next(P)
        deleteClass (L2, Q)
```

Example : Student - Course

- Let's say we want to store student data and its relation with the course taken
- What is the relation of the data?
  - N to M relation

Example : Student - Course

```
Type infotype_student <
    id      : string
    name    : string
>


Type adr_student : pointer to
elm_student

Type elm_student <
    info : infotype_student
    next : adr_student
>
```

```
Type infotype_course <
    course_id   : string
    course_name : string
    credit      : integer
>

Type adr_course : pointer to
elm_course

Type elm_course <
    info : infotype_course
    next : adr_course
>
```

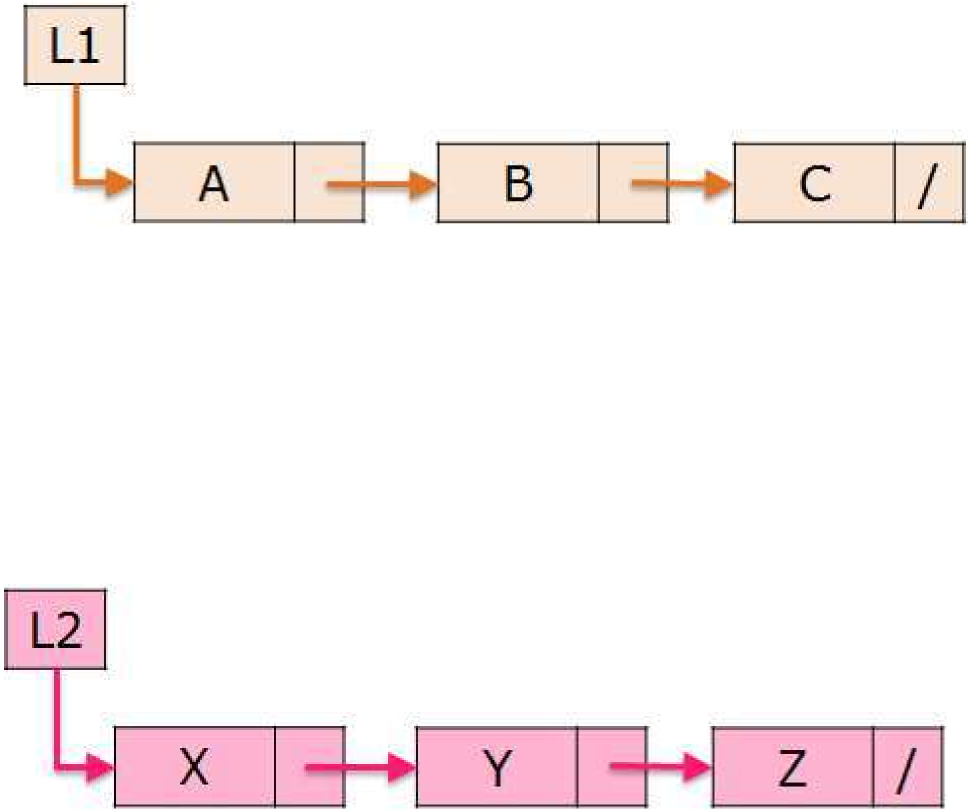
Example : Student - Course

```
Type list_student <
    first : adr_student
>
```

```
Type list_course <
    first : adr_course
>
```

```
L1 : list_student
L2 : list_course
```

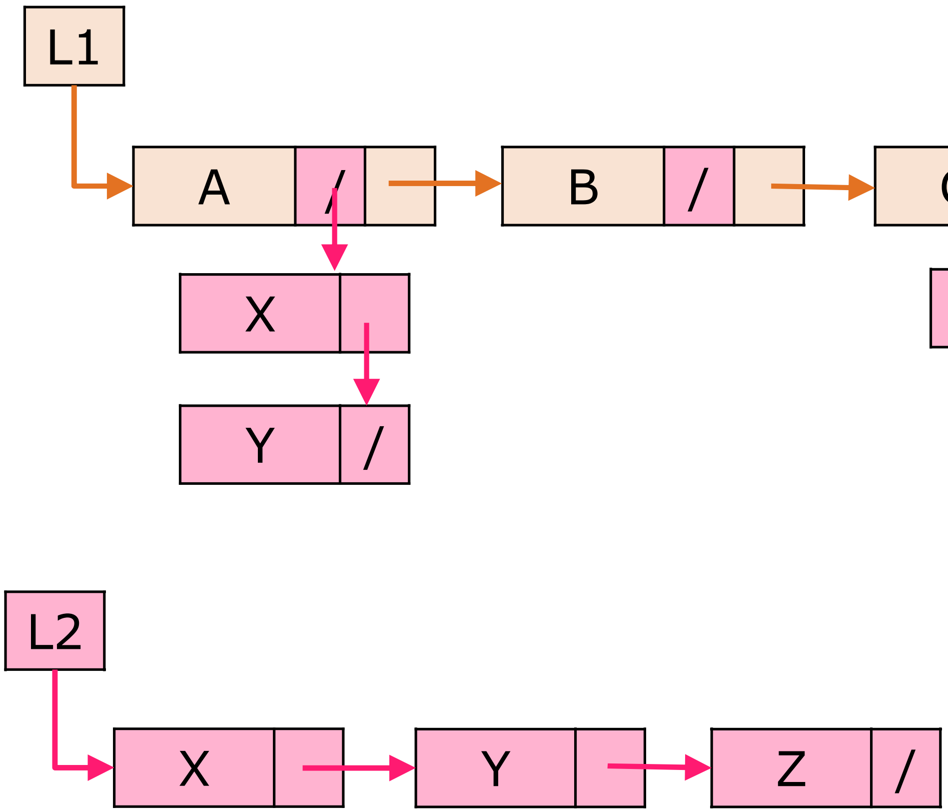
Example : Student - Course



Now, how to draw
the relation between data?

Example : Student - Course

n-to-n relation



For this case,
insert a list of course
inside each elm_student
to indicate every course
taken by the student

Example : Student - Course

```
Type infotype_student <
    id          : string
    name        : string
>

Type adr_student : pointer to
elm_student

Type elm_student <
    info : infotype_student
    next : adr_student
    course : list_course
>
```

```
Type infotype_course <
    course_id   : string
    course_name : string
    credit      : integer
>

Type adr_course : pointer to
elm_course

Type elm_course <
    info : infotype_course
    next : adr_course
>
```

Operations

- Insert Student and Insert Course
  - No change
- Search Student and Search Class
  - No change
- Add course to a student
- Delete course in a student
- Delete a student
- Delete a course

Procedure add Course

<u>Procedure</u> add_course(i/o: L1:list_student,
i: L2 : list_course, id : string, course_id : string)

```
Dictionary
    define info(R) (R)→info
    define info(Q) (Q)→info
    define course(P) (P)→course
    P : adr_student
    Q, R : adr_course
    L3 : list_course
Algorithm
    P ← search_student(L1, id)
    Q ← search_course(L2, id_course)
    …
```

```
…
if ( P ≠ Nil and Q ≠ Nil ) then
    Allocate(R)
    info(R) ← info(Q) //duplicate Q
    L3 ← course(P)
    insertLastCourse(L3, R)
```

Procedure Delete Course Student

```
Procedure del_course_student(i/o: L1:list_student,
                             i:  id : string, course_id : string)
Dictionary
    define course(P) (P)→course
    P : adr_student
    L3 : list_course
Algorithm
    P ← search_student(L1, id)
    if ( P ≠ Nil ) then
        L3 ← course(P)
        deleteCourse(L3, course_id)
```

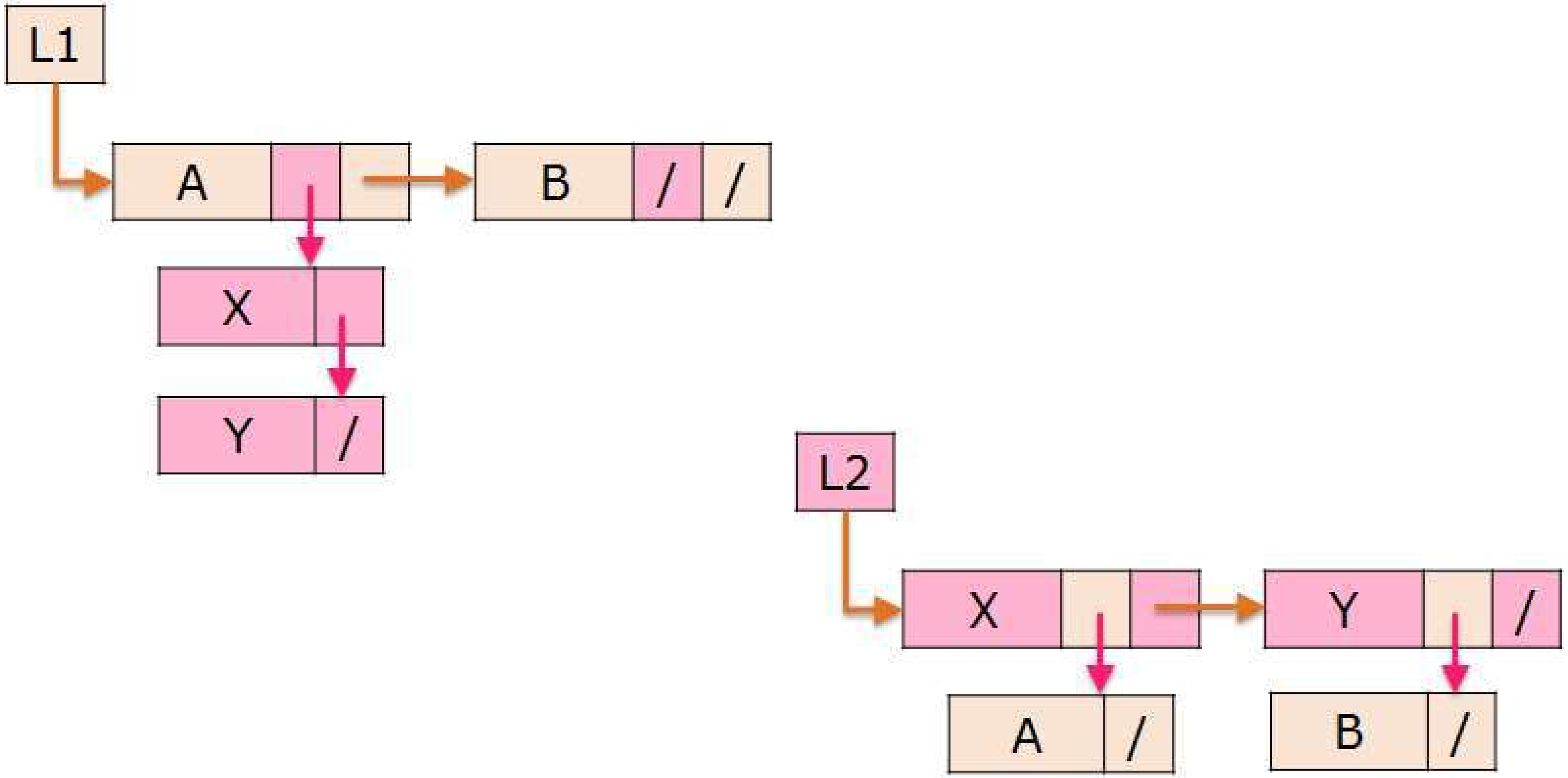
Procedure Delete Student

```
Procedure del_student(i/o: L1:list_student, i: id : string)
Dictionary
     define course(P) (P)→course
     define info(P) (P)→info
     P : adr_student
     L3 : list_course
Algorithm
     P ← search_student(L1, id)
     if ( P ≠ Nil ) then
          L3 ← course(P)
          emptyList(L3)
          deleteStudent(L1, info(P).id )
```

Procedure Delete Course

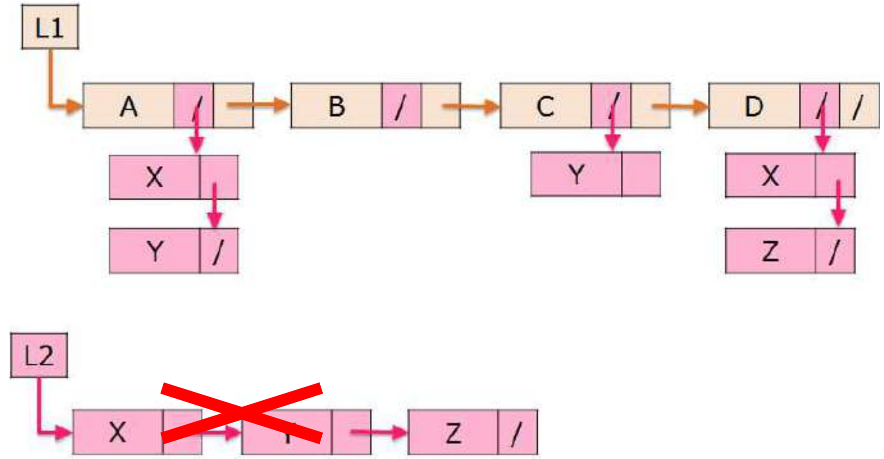
```
Procedure del_course(i/o: L1:list_student,
                          i: L2:list_course, id_course : string)
Dictionary
    define first(L1) ((L1).first)
    define course(P) (P)→course
    P : adr_student
    Q : adr_course
    L3 : list_course
Algorithm
    P ← first(L1)
    while ( P ≠ Nil ) do
        L3 ← course(P)
        deleteCourse(L3, course_id)
    deleteCourse(L2, course_id)
```

Which one is the parent?



Variation of Multi List

- Depend on the case, List L2 might not be needed
  - When child list is not too important to be listed independently
  - When the child list is too vary
  - Example : List of Employee and list of their children



Procedure add Course

Procedure add_course(i/o: L1:list_student, i: L2 : list_course, id : string)

```
Dictionary
    define course(P) (P)→course
    P : adr_student
    R : adr_course
    L3 : list_course
Algorithm
    P ← search_student(L1, id)
    …
```

```
…
if ( P ≠ Nil ) then
    Allocate(R)
    input(info(R)) //input new data for child
    L3 ← course(P)
    insertLastCourse(L3, R)
```

Question?

Alternative Solution

- With the duplication of the child node, the size might expand became too big
- Difficult modification at child node
- Solution :
  - Make the child list is a pointer element to point the second list as a relation list

Alternative : Student - Course

```
Type infotype_student <
      id          : string
      name        : string
>
Type adr_student : pointer to elm_student

Type elm_student <
      info : infotype_student
      next : adr_student
      course : list_relation
>
```

```
Type infotype_course <
      course_id : string
      course_name   : string
      credit              : integer
>
Type adr_course : pointer to elm_course

Type elm_course <
      info : infotype_course
      next : adr_course
>
```

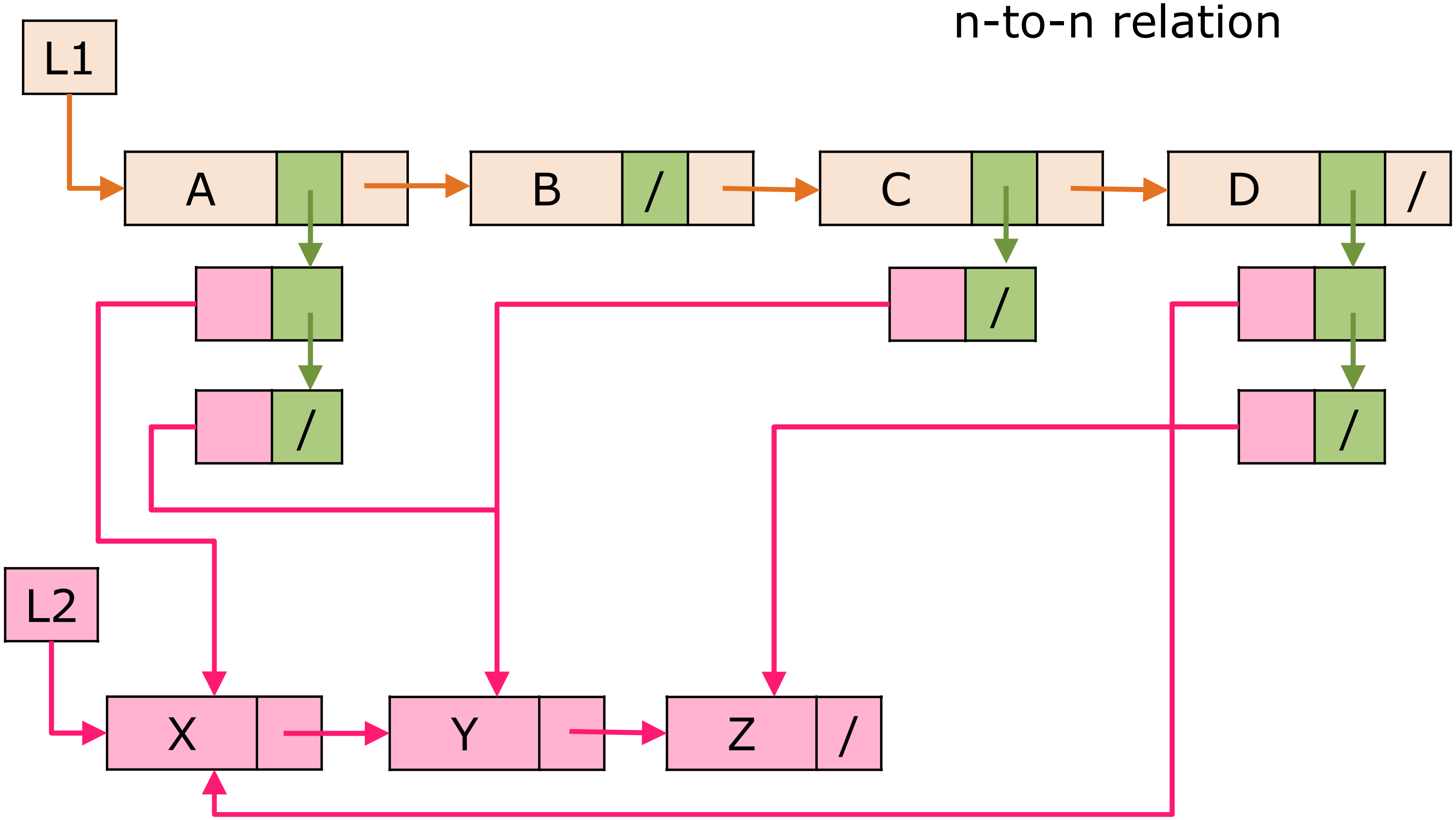
```
Type adr_relation :
      pointer to elm_relation

Type elm_relation <
      next_course : adr_course
      next : adr_relation
>
```

```
Type list_relation <
      first : adr_relation
>
```

Example : Student - Course

n-to-n relation



Procedure add Course

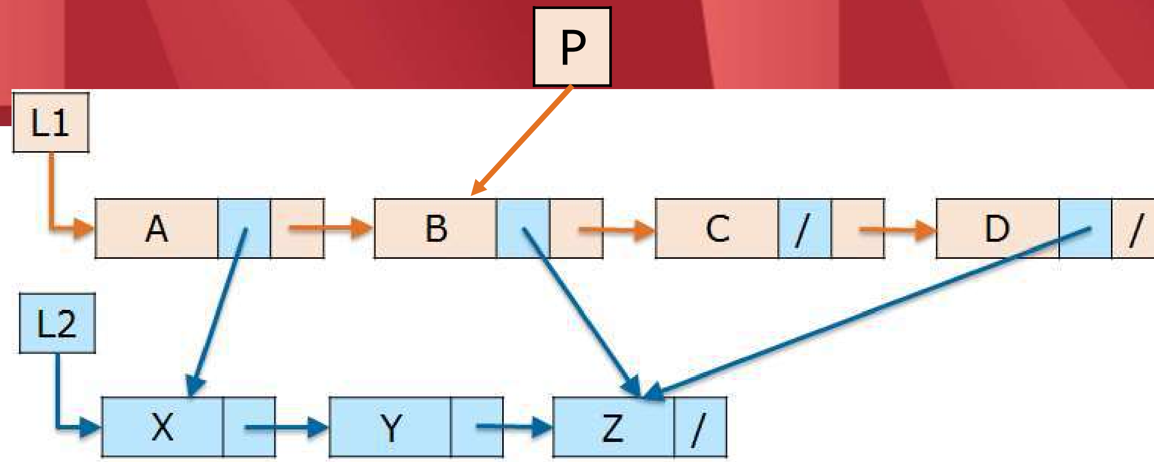
<u>Procedure</u> add_course(i/o: L1:list_student, i: L2 : list_course,
id : string, course_id : string)

```
Dictionary
    define next_course(R) (R)→next_course
    define course(P) (P)→course
    P : adr_student
    Q : adr_course
    R : adr_relation
    L3 : list_course
Algorithm
    P ← search_student(L1, id)
    Q ← search_course(L2, id_course)
    …
```

```
…
if ( P ≠ Nil and Q ≠ Nil ) then
    Allocate(R)
    next_course(R) ← Q
          //create pointer to Q
    L3 ← course(P)
    insertLastCourse(L3, R)
```

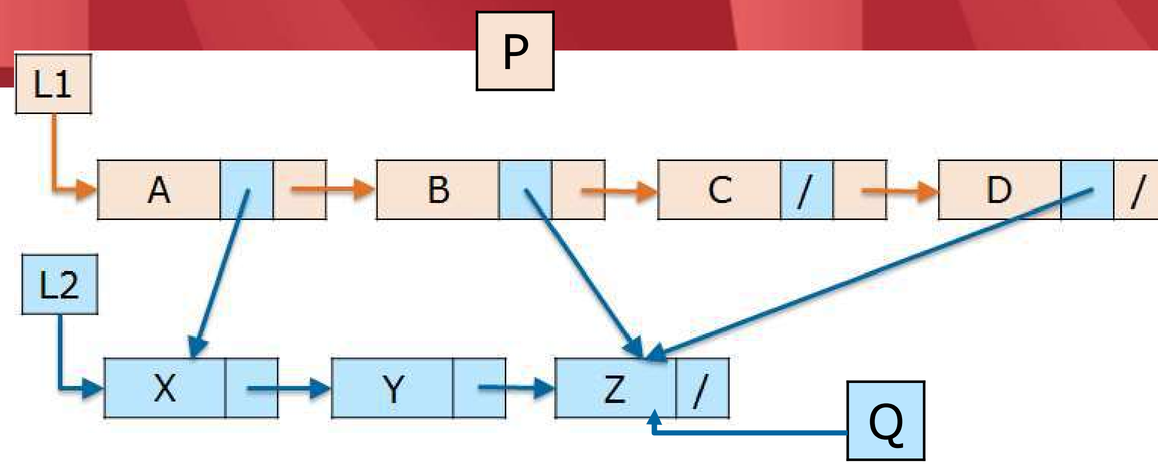
Question?

Exercise 1



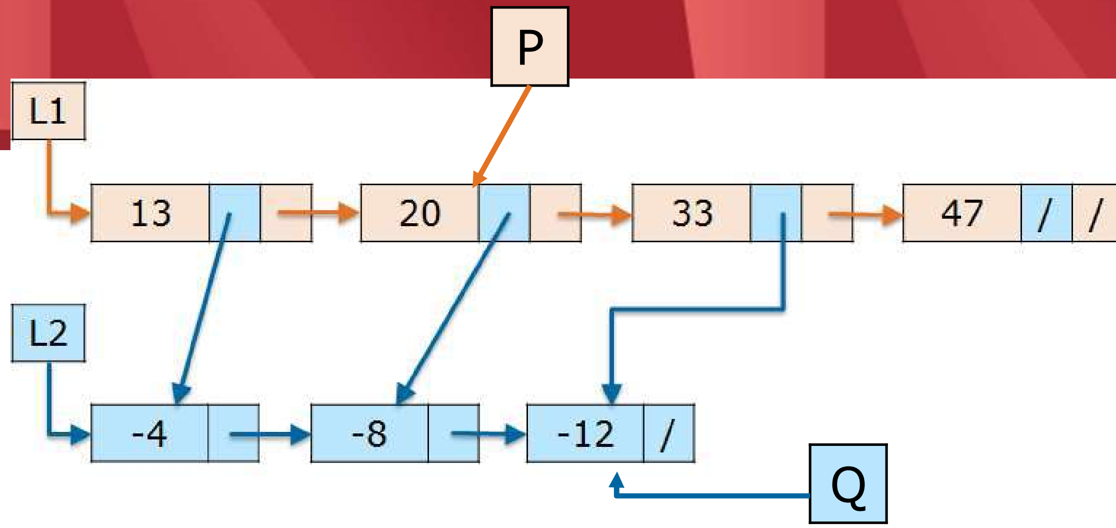
1	Info(P) Info(child(P))	
2	Info(child(next(P)))	
3	Info(next(child(first(L1))))	
4	Info(next(P))	
5	P ← next(P) Child(P) ← next(first(L2)) Info(child(P))	
6	create algorithm to count member of L1	

Exercise 2



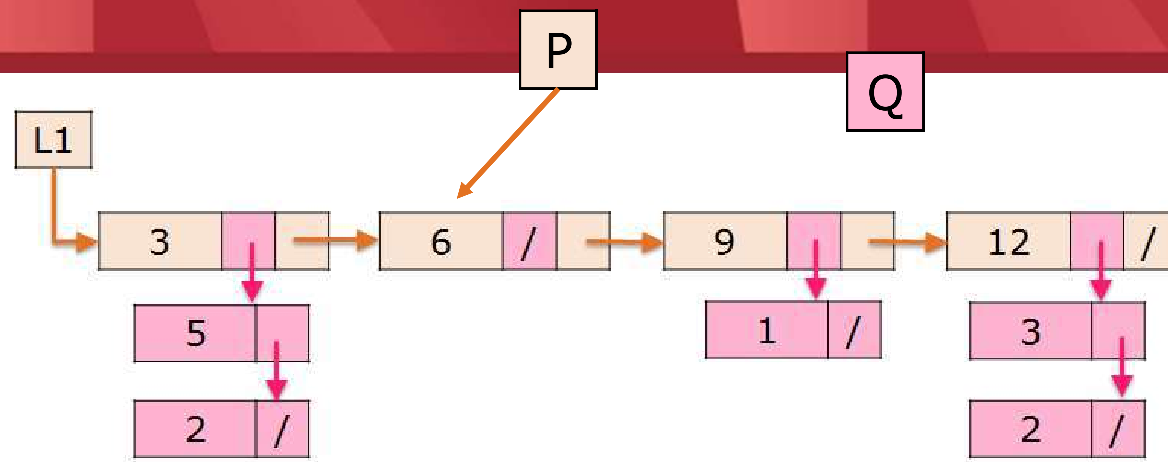
1	Create algorithm to count the parent of Q	
2	P ← first(L1) Q ← child(P) info(Q) ← info(next(Q) Child(next(P)) ← next(Q) Output(Info(child(P)))	
	Output(Info(child(next(P))))	
3	Make child A = Y	

Exercise 3



1	info(first(L1)) + info(child(P))	
2	Q ← next(first(L2)) Info(Q) + info(child(next(P)))	
3	P ← next(next(P)) Info(next(child(first(L1)))) – info(P)	
4	Create algorithm to show info parent with no child	

Exercise 4



1	Info(next(child(first(L1))))	
2	P ← next(P) Q ← child(P) Info(Q) + info(P)	
	Q ← next(child(next(P)))) P ← next(first(L1)) Info(P) - info(Q)	
3	Create algorithm to count child of P	


Fakultas Informatika
School of Computing
Telkom University

THANK YOU