Department of Computer Science and Engineering
The Chinese University of Hong Kong

CSCI2100B CSCI2100S DATA STRUCTURES

Spring 2011

Linked Lists

Last updated: **22/02/2011**

Tang Wai Chung, Matthew

Contents

- Linked lists
  - Basic list operations
  - Circular/empty list conventions
  - Memory allocation & implementation issues
- The concept of abstract data type (ADT)
- List ADT
  - Array implementation
  - Linked list implementation
  - Application: polynomial ADT

Legend:

Important examples

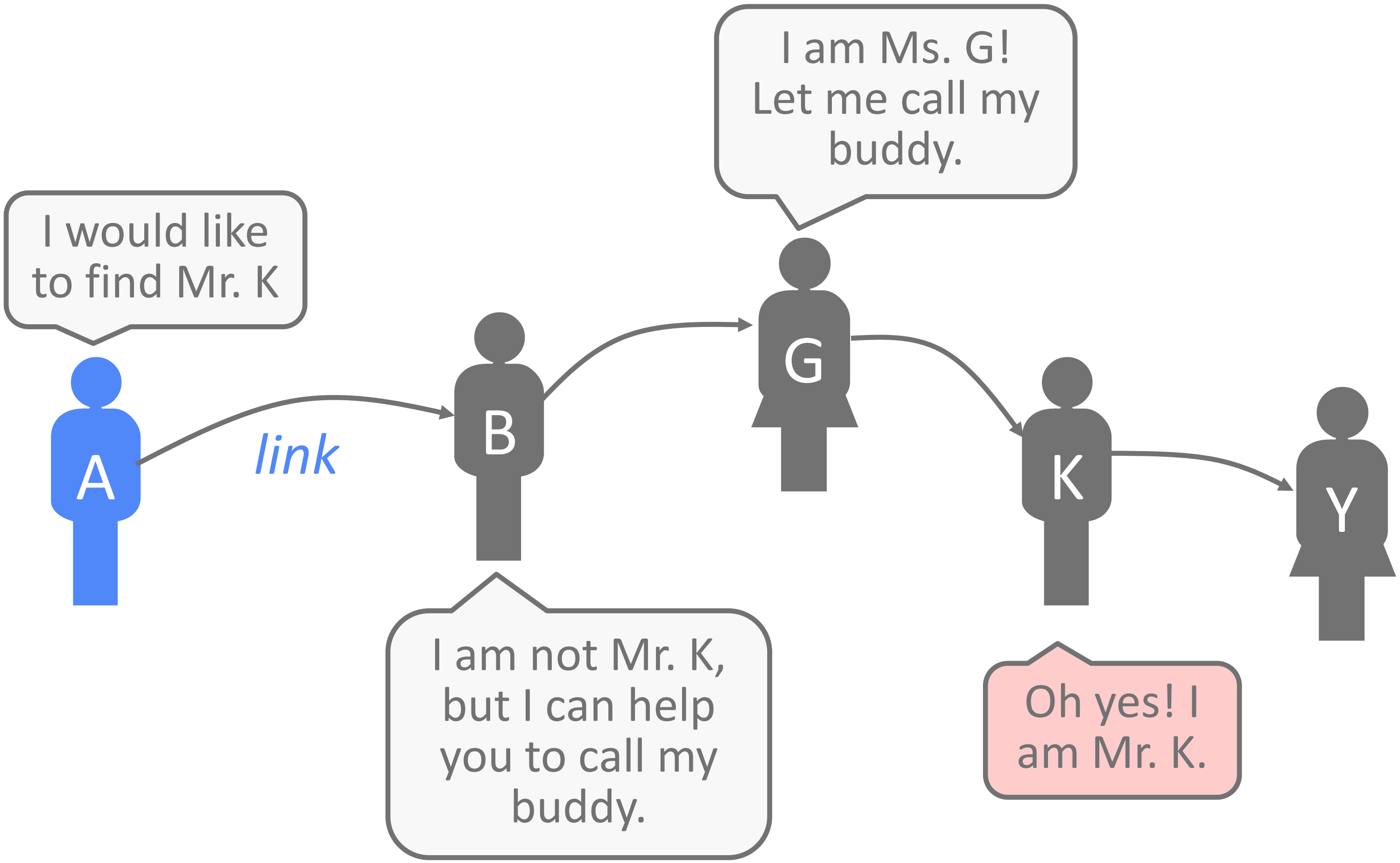
Advanced topics

Linked Lists

- When our primary interest is to go through a collection of items **sequentially**, we can organize the items as linked list.
- **Linked list**: a basic data structure where each item contains the information that we need to get to the next item (the link)
- **Advantage**: the capability to rearrange the items efficiently.

A **linked list** is a set of items where each item is part of a node that also contains a **link** a node.

Linked List: Visualization



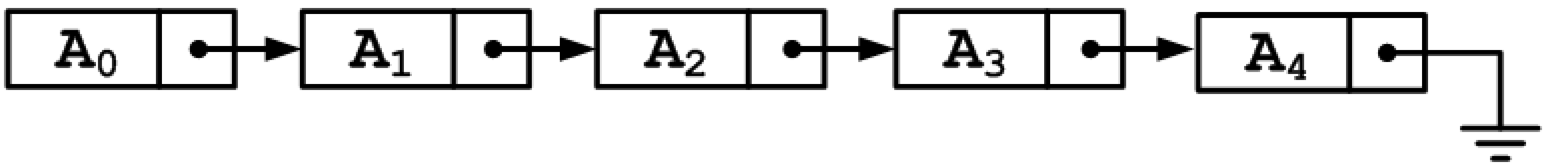
Starting at a given node (the first in the sequence), we follow its link, which gives the second item, and so forth.

Linked Lists: Conventions

- In principle, the list can be **cyclic**.
  - The sequence could seem **infinite**.
- We often work with a simple sequential arrangement of a **finite** set, then we have 3 options to denote the end of the list:
  - ***null link*** that points to no node
  - refers to a **dummy** node that contains no item.
  - refers back to the **first** node, making the list circular.
- Unless otherwise specified, we work with one-dimensional list.

Linked Lists: Declaration in C

- We will declare each node as a **structure** in C
- The list consists of a series of structures, which are not necessarily adjacent in memory, linked by **pointers**.
  - Each structure contains the element and a next pointer (**link**) to a structure of its successor
- **NULL** link convention is used to denote the end.



Structure declaration

```
typedef struct node_s node;
struct node_s {
    int e; /* the actual data */
    node *next; /* self-referent link */
};
```

Linked Lists: Memory Allocation

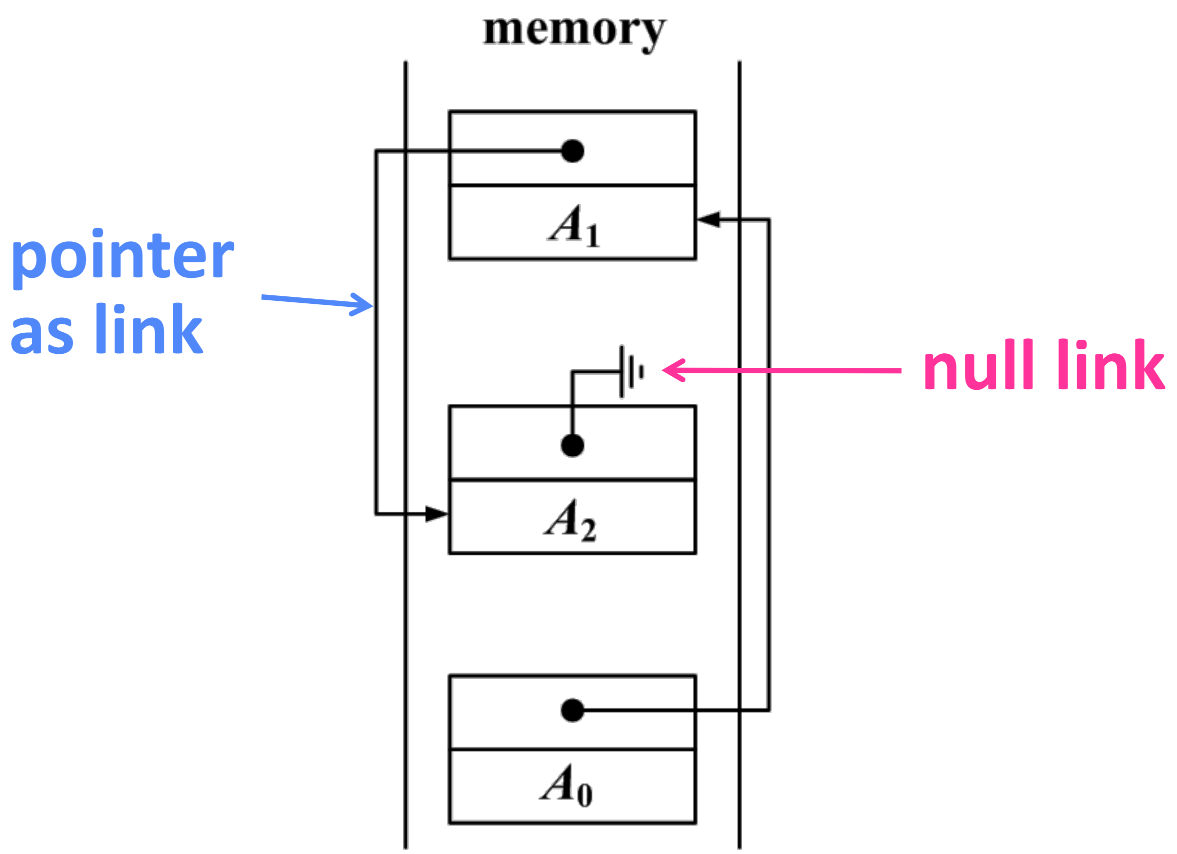
- We will create many **instances** of the same structure.
  - In general we **do not know** the number of nodes before our program executes.
- Whenever we want to use a new node, we need to create an instance and **reserve memory** for it.
  - use malloc() in C.
- When the node is no longer needed, we will **return** the allocated memory to the system.
  - use free() in C.

Structure allocation and freeing

```
node *x = malloc(sizeof(node));
node *y = malloc(sizeof *y);
...
free(x); free(y);
```

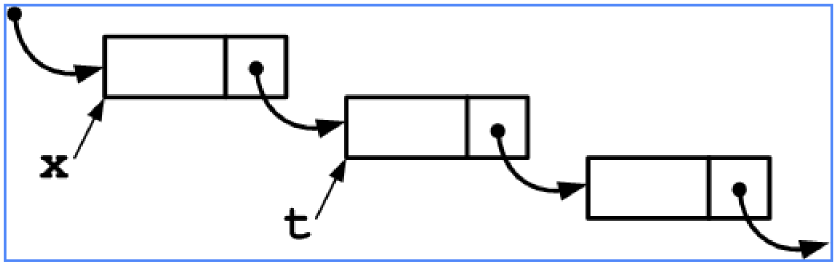
Linked List on Flat Memory

- If we try to fit the above implementation onto the memory model ...

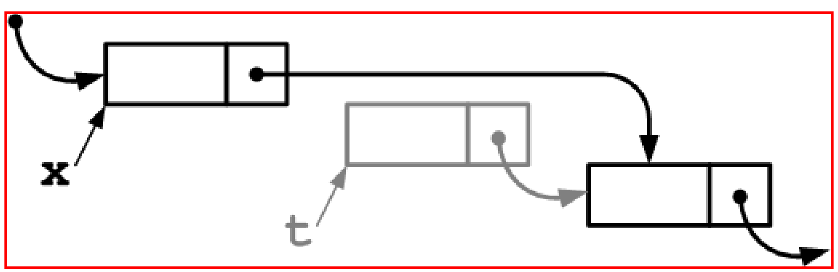


Linked List: Deletion

To delete the node following node x …

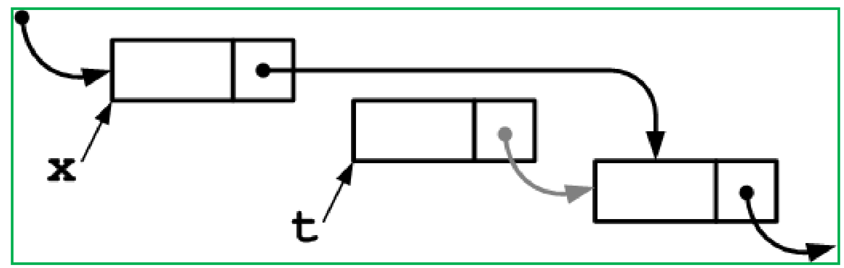



```
t = x->next;
x->next = t->next;
free(t);
```



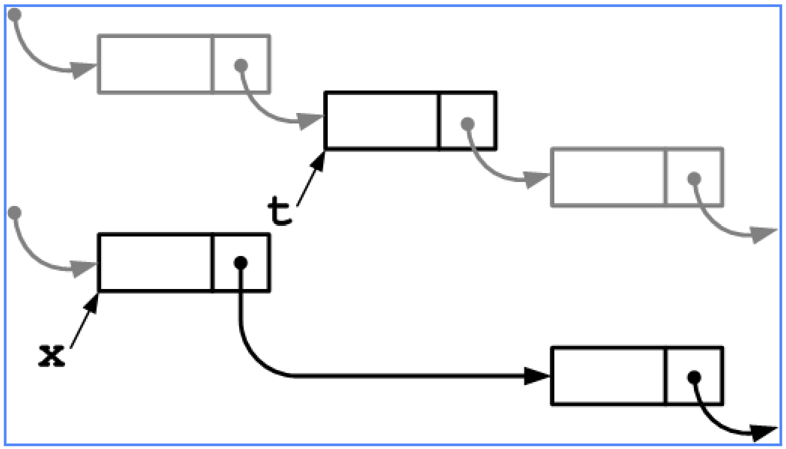
OR

```
x->next = x->next->next;
```

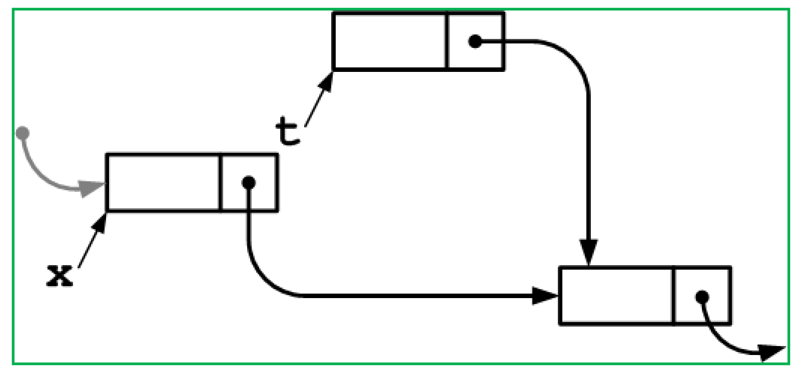


Linked List: Insertion

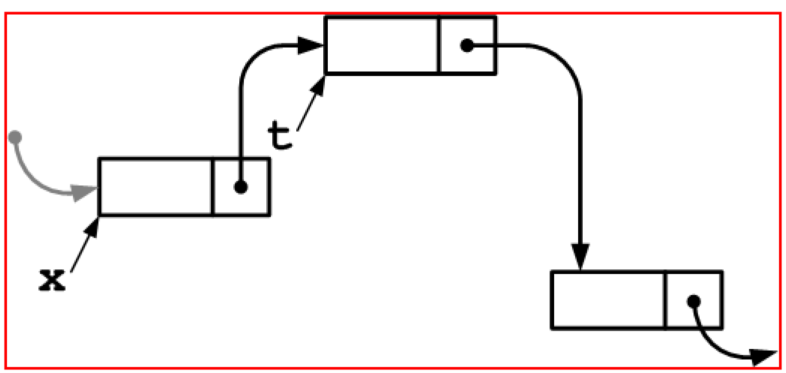
To insert node t into a list at a position following node x




```
t->next = x->next;
```




```
x->next = t;
```



Difficult Operations in Linked Lists

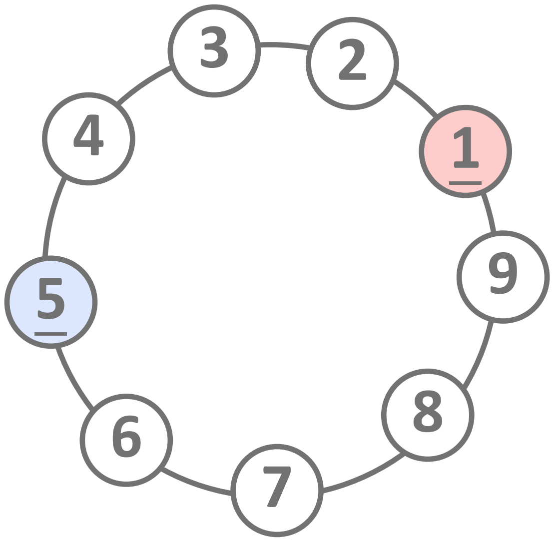
- Insertions and deletions are efficient in linked lists.
- By contrast, linked lists are not well suited for the "***find the k-th item***" that is efficient on arrays.
- Another unnatural operation on singly linked lists is "***find the item before a given item***".
  - We shall see some modifications to make this operation easier.

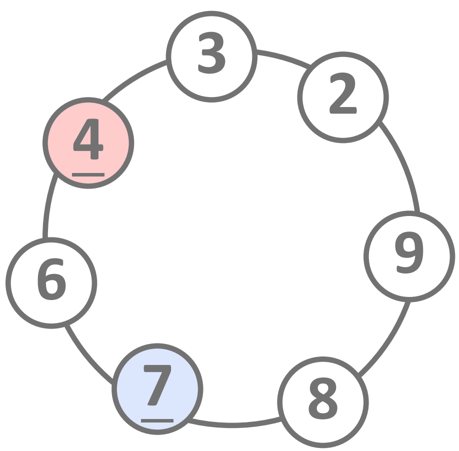
The Josephus Problem

- *N* people have to elect a leader by arranging themselves in a **circle**,
  - **eliminating** every *M*th person around the circle
  - **closing** ranks as each person drops out.
- In general, we may want to know the **order** in which the people are eliminated.
- For example, $N = 9$, $M = 5$, the order would be

5 1 7 4 3 6 9 2 <u>8</u>

The Josephus Problem (2)






```
int main(int argc, char *argv[]){
    int i, N = atoi(argv[1]), M = atoi(argv[2]);
    node *t = malloc(sizeof(node)), *x = t;

    t->e = 1;
    t->next = t; /* cyclic */

    for (i = 2; i <= N; i++){
        x = (x->next = malloc(sizeof *x));
        x->e = i;
        x->next = t;
    }
    while (x != x->next){
        for (i = 1; i < M; i++)
            x = x->next;
        printf("%d ", x->next->e);
        x->next = x->next->next;
        N--;
    }
    printf("%d\n", x->e);

    return 0;
}
```

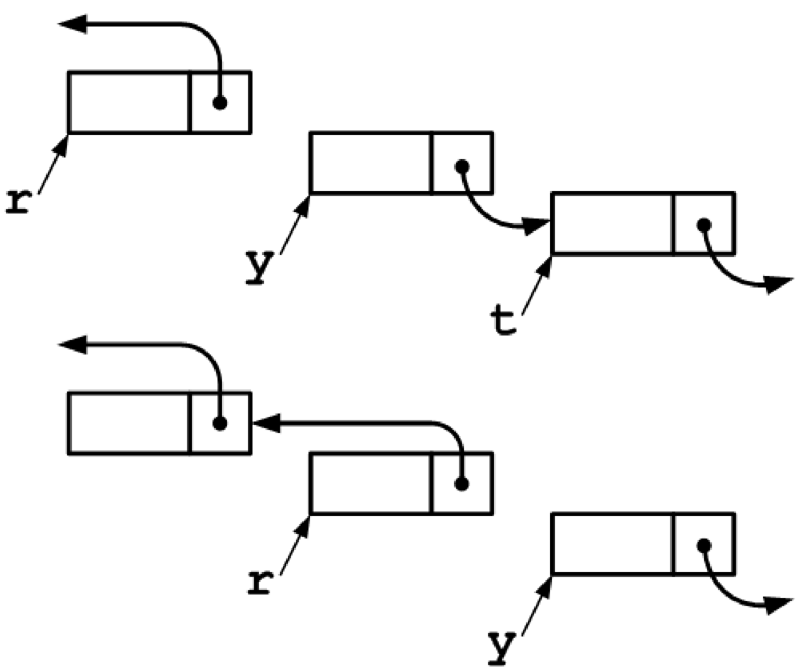
Elementary List Processing

Developing correct and efficient code for list-processing applications is an acquired programming skill that requires practice and patience to develop.

- Let us provide a mental model in coding the linked list:
  - ***A linked list is either a null link or a link to a node that contains an item and a link to a linked list.***
- For example, we might write the following for-loop to scan through every item on the list (**traversal**):

```
for (t = x; t != NULL; t = t->next)
    visit(t->e);
```

List Reversal




```
node *reverse(node *x){
    node *t, *y = x, *r = NULL;
    while (y != NULL){
        t = y->next;
        y->next = r;
        r = y;
        y = t;
    }
    return r;
}
```

List Insertion Sort

```
node heada, headb;
node *t, *u, *x, *a = &heada, *b;
for (i = 0, t = a; i < N; i++){
    t->next = malloc(sizeof *t);
    t = t->next;
    t->next = NULL;
    t->e = rand() % 1000;
}
b = &headb;
b->next = NULL;
for (t = a->next; t != NULL; t = u){
    u = t->next;
    for (x = b; x->next != NULL; x = x->next)
        if (x->next->e > t->e)
            break;

    t->next = x->next;
    x->next = t;
}
```

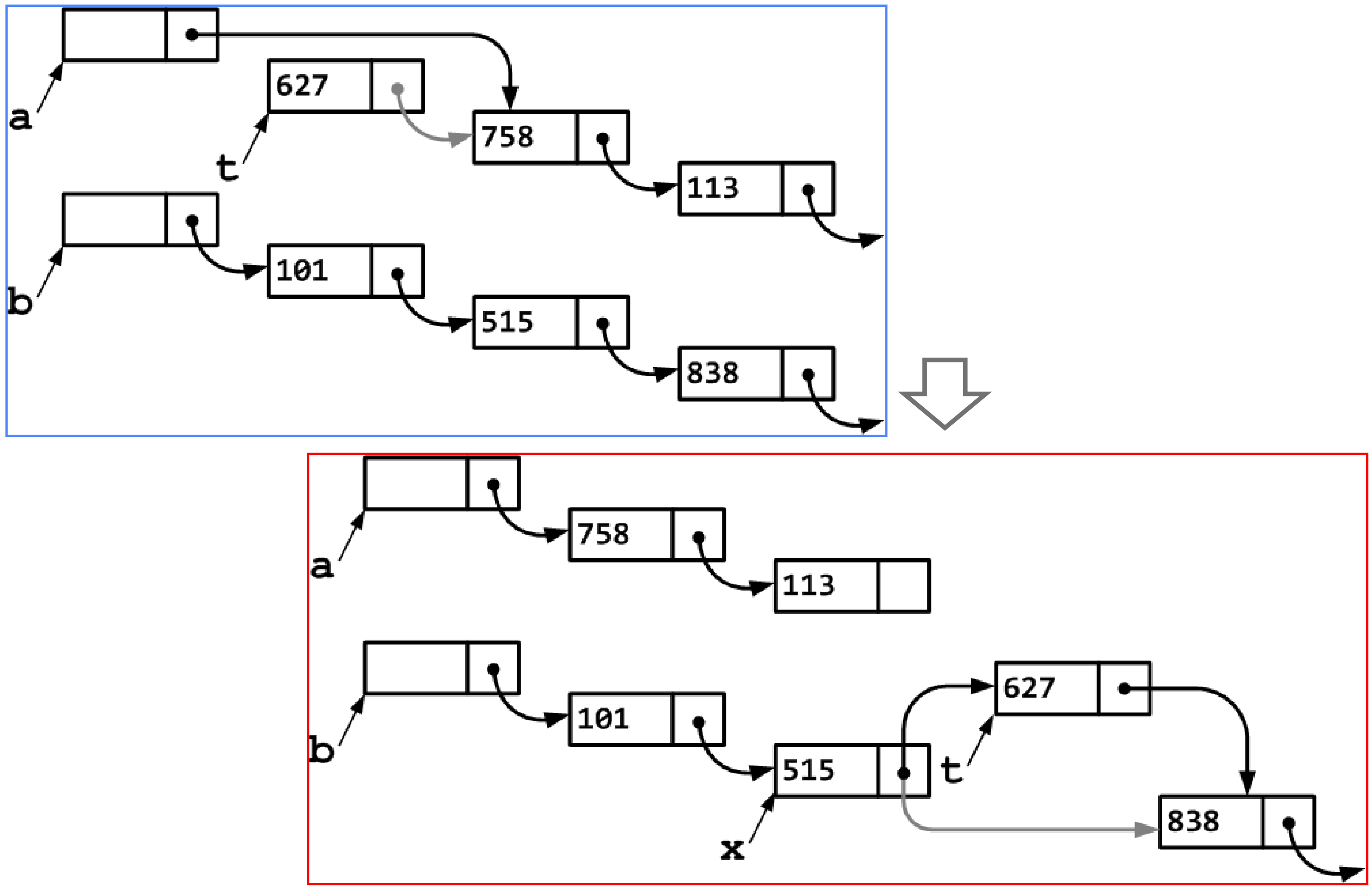
randomly generate a list of integers

initialize list *B*

locate point of insertion (list A traversal)

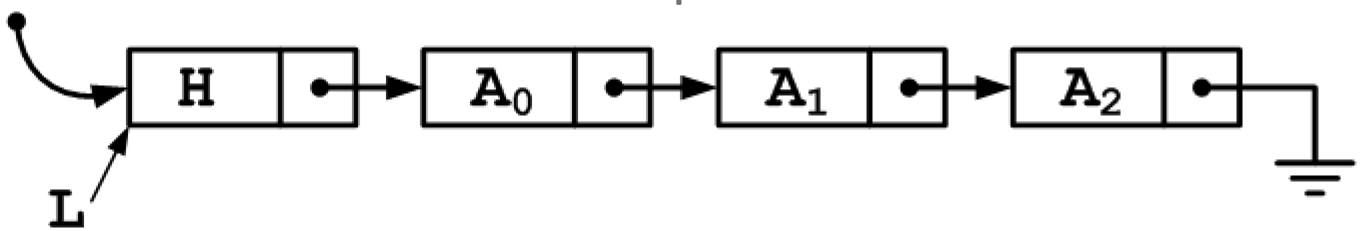
insertion

List Insertion Sort: Illustration



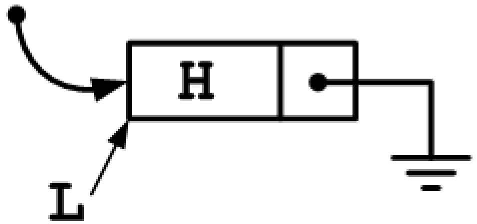
Header Node in Linked List

- The previous example illustrates an important convention: keep a **header**(**dummy**) node to specify the beginning of the list.
- This simplifies our coding since we do not have to **distinguish** between empty list & real list.
- It also allows the list to be passed to **functions** more easily.



Then an empty list is represented as:

No data is stored in the dummy node.



Interface for Linked List Operations

- When we do not want to repeat the basic list operations inline, we can choose to make a set of **black-box functions**.
  - This usually works better with the **header** node convention so that the functions can easily return an empty or non-empty list through a single interface.

```
node *new_node(int);
void free_node(node *);
void insert_next(node *, node *);
node *delete_next(node *);
node *next(node *);
int item(node *);
```

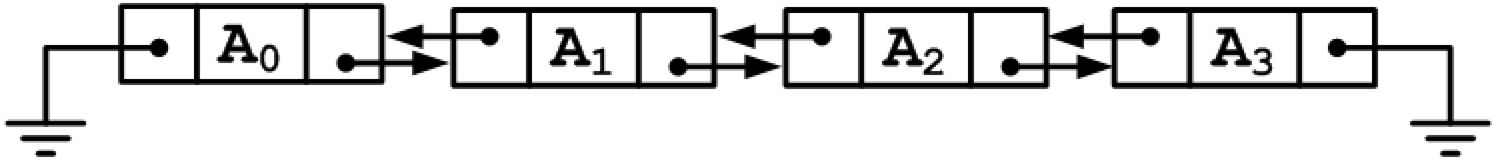
Revisiting Josephus Problem

- We may rewrite our solution to Josephus problem using the newly created interface.
  - Contrast with the previous inline version of the solution.

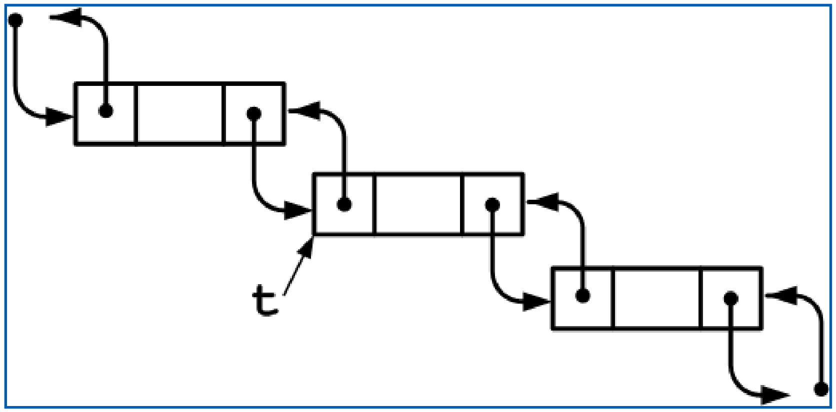
```
int main(int argc, char *argv[]){
    int i, N = atoi(argv[1]), M = atoi(argv[2]);
    node *t, *x;
    for (i = 2, x = new_node(1); i <= N; i++){
        t = new_node(i);
        insert_next(x, t);
        x = t;
    }
    while (x != next(x)){
        for (i = 1; i < M; i++) x = next(x);
        t = delete_next(x);
        printf("%d ", item(t));
        free_node(t);
    }
    printf("%d\n", item(x));
```

Doubly Linked Lists

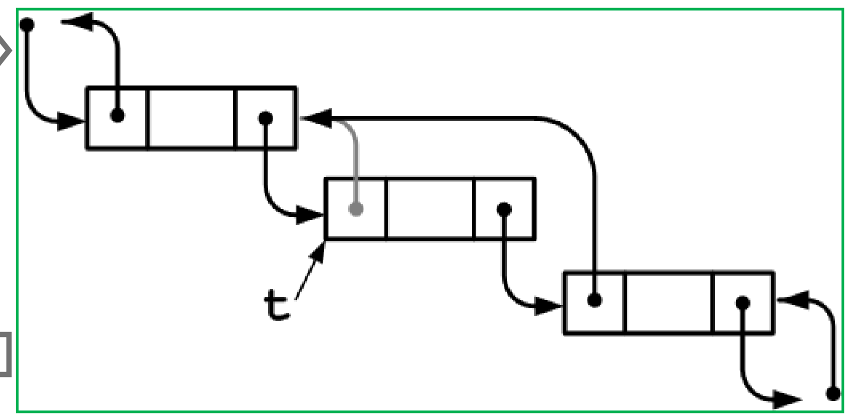
- Traverse linked lists in a backward direction is **not** convenient.
- To facilitate **to** and **fro** movement on a linked list, we add an extra pointer to the predecessor.
- But we have to take care of **more** pointers when you insert or delete.
- Deletion is simplified and is $O(1)$ in doubly linked lists.



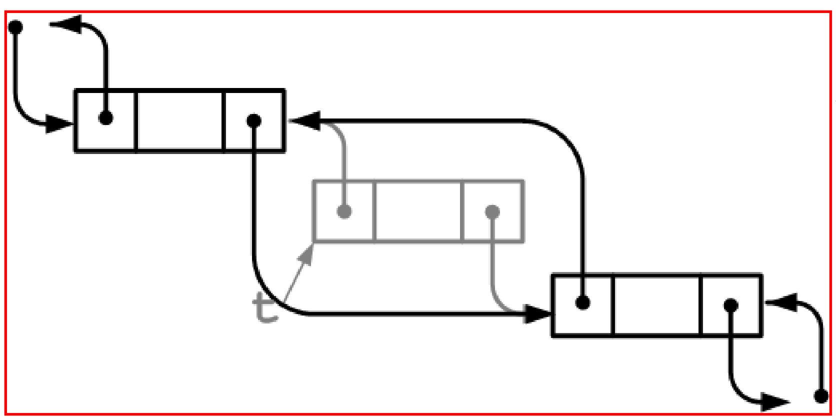
Doubly Linked Lists: Deletion



`t->next->prev = t->prev;`

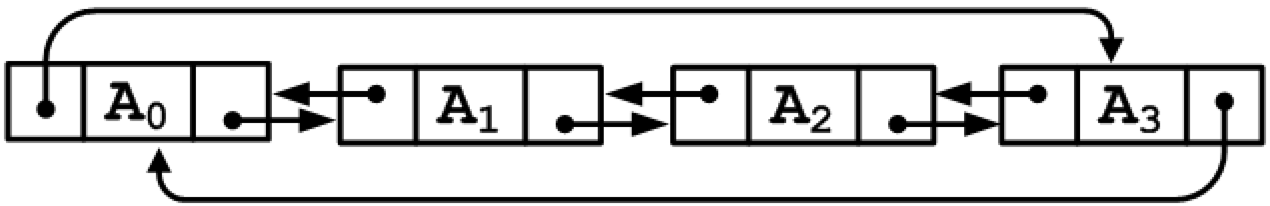


`t->prev->next = t->next;`



Doubly Circular List

- Another popular convention is to have the last cell keep a pointer back to the first (with or without header).



Code snippets

```
struct node_s {
    int e;
    node *next;
    node *prev;
};
```

```
void delete(node *t){
    t->prev->next = t->next;
    t->next->prev = t->prev;
    free(t);
}
```

LIST ABSTRACT DATA TYPE (ADT)

ADT: Definition

Abstract Data Type

An abstract data type (ADT) a data type (a set of **values** and a collection of **operations** on those values) that is accessed only through an **interface**.

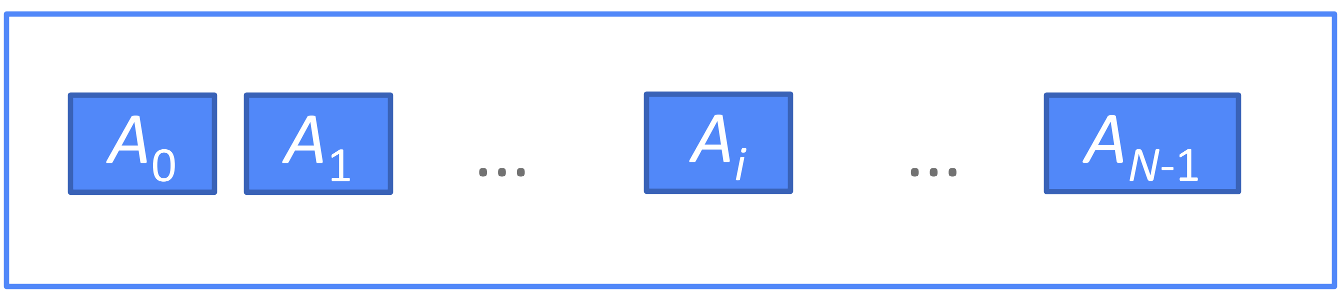
We refer to a program that uses an ADT as a **client**, a program that specifies the data type as an **implementation**.

Abstract Data Type (ADT)

- Integers, floating point numbers, characters are data type. They have associated operations (addition, multiplications, etc.)
- Abstract data type consists of a sets of operations, yet how the operations are implemented is **hidden**.
- For example, given a **set** ADT, we want operations like union, intersection, size and complement.
- ADTs may be implemented in different ways, but the programs that use them can **safely ignore** which implementation was used.

The List ADT

- A general list of the form A_0, A_1, A_2, ..., $A_{N\text{-}1}$.
- The size of the list is N.
- The special list with size 0 is called empty list.
  - A_{i+1} **follows** A_i and $A_{i\text{-}1}$ **precedes** A_i.
  - The **position** of A_i is i.
  - The elements in the list may be simplified to **integers** for **simplicity**. Although complex elements can be used.



We use 0-based counting scheme.

List Operations

- Utility: ***print_list*** and ***make_empty***
- Searching: ***find*** returns the position of the first occurrence of a key
- ***insert*** and ***delete***: insert a new element in some position and delete a key from the list.
- ***find_kth***: return the element in some position *k*.

Examples: Given *L*: 34, 12, 52, 16, 12

[find(52)]	returns 2
[insert(X, 3)]	*L*: 34, 12, 52, X ,16, 12
[delete(52)]	*L*: 34, 12, X, 16, 12

- ***next*** and ***previous*** are other possible operations

The List ADT (Coding)

- The client programs do not have to understand the **actual implementation** of the ADT.
  - Instead, an interface is **well-defined** to allow easy manipulations of the lists

List ADT interface declaration

```
int list_is_empty(list_t list);
list_t list_create(void); /* create a new empty list */
void list_free(list_t list); /* free(destroy) the list */
void list_insert(list_t list, pos_t p, int x); /* normal insert */
void list_insert_end(list_t list, int x); /* insert at the end */
void list_insert_begin(list_t list, int x); /* insert at the front */
void list_delete(list_t list, int x); /* delete a specific item */
pos_t list_find(list_t list, int x); /* searching */
pos_t list_find_kth(list_t list, int k); /* searching by index */
pos_t list_begin(list_t list); /* iterator */
pos_t list_next(list_t list, pos_t p); /* iterator */
int list_is_end(list_t list, pos_t p); /* iterator */
int list_get(list_t list, pos_t p); /* accessor */
void list_set(list_t list, pos_t p, int x); /* accessor */
void list_print(list_t list); /* utility */
```

Simple Array Implementation of List ADT

- An **estimate** of the maximum size of the list is required.
- A dynamically-growing array is accepted but the insertion would be slow if the initial size is not well estimated.
- An array implementation allows:
  - ***print_list*** and ***find*** in $O(N)$
  - ***find_kth*** in $O(1)$
  - ***insert***: inserting at pos. 0 requires pushing the entire array down. $O(N)$
  - ***delete***: deleting the first element requires shifting all elements 1 position up. $O(N)$
- Array implementation is **slow** when ***insert*** and ***delete*** are frequent.

Array Implementation of List ADT

```
typedef struct list_s *list_t;
typedef int pos_t;
#define NPOS -1
#define MAX_SIZE 10000
struct list_s {
    int *e;
    int n;
};
```

The list is actually an array with its size accounted.

```
pos_t list_find(list_t list, int x){
    int i;
    for (i = 0; i < list->n; i++)
        if (list->e[i] == x)
            return i;
    return NPOS;
}
```

The classic sequential search on the items stored.

Array Implm. of List ADT (2)

```
void list_insert(list_t list, pos_t p, int x){
    int i;
    assert(list->n + 1 <= MAX_SIZE);
    for (i = list->n; i > p; i--)
        list->e[i] = list->e[i - 1];
    list->e[p] = x;
    list->n++;
}
```

Be careful with the size limitation

Then shift the stored data and make room

```
void list_delete(list_t list, int x){
    int i, p;
    if ((p = list_find(list, x)) == -1)
        return;
    for (i = p; i < list->n - 1; i++)
        list->e[i] = list->e[i + 1];
    list->n--;
}
```

Safe: Just report error if the key is not found

Then shift down the stored data

List ADT: Linked List Implm.

- Instances of the node structure are created on **demand** (with malloc()).
- The beginning of the list is marked by the **dummy** header node.
- Insertion at a specific position is **fast**.

```
struct node_s;
typedef struct node_s *list_t;
typedef struct node_s *pos_t;
#define NPOS NULL

typedef struct node_s node;
struct node_s {
    int e;
    node *next;
};
```

List ADT w/ LL: Basic Operations

is_empty: check whether the given list is an empty list

```
int list_is_empty(list_t list){
    return (list->next == NULL);
}
```

is_last: check whether the given position in the list is the last element.

```
int list_is_end(list_t p, pos_t p){
    return (p == NULL);
}
```

create: generate an empty list.

Remember to free the list when it is not used anymore.

```
list_t list_create(void){
    node *t = malloc(sizeof *t);
    t->e = INT_MIN;
    t->next = NULL;
    return t;
}
```

List ADT w/ LL: Insertion & Deletion

```
void list_insert(list_t list, pos_t p, int x){
    node *t = malloc(sizeof *t);
    t->e = x;
    t->next = p->next;
    p->next = t;
}
```

Insert in a given position. $O(1)$

Deletion based on key requires a searching. $O(N)$

```
void list_delete(list_t list, int x){
    pos_t t, u;

    for (t = list; t->next != NULL && t->next->e != x; t = t->next);

    if (t->next == NULL) return; /* 'x' does not exist */

    u = t->next;
    t->next = u->next;
    free(u);
}
```

When we are given an element to be deleted in the list, we first have to check whether the element exists in the list.

If it exists, we need to find it predecessor so that we can remove the node containing the element.

List ADT w/ LL: *find & free*

Similar to ***delete***, we use a for-loop to **traverse** the list.

```
pos_t list_find(list_t list, int x){
    pos_t t;
    for (t = list->next; t != NULL && t->e != x; t = t->next);

    return (t == NULL ? NPOS : t);
}
```

CAREFUL: Always copy the next pointer before you free the node, or you lose the link.
Traverse each node. **Copy** the link then free the node.

```
void list_free(list_t list){
    node *t, *u;
    for (t = list->next; t != NULL; t = u){
        u = t->next;
        free(t);
    }
    free(t); /* the header */
}
```

List ADT: Comparison

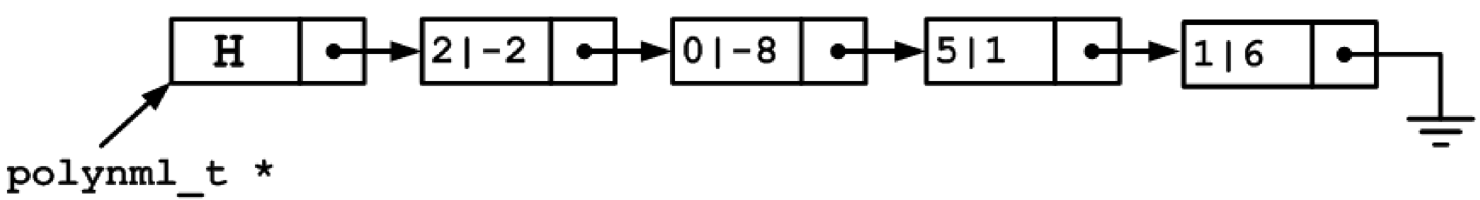
Operation	Array Implm.	Linked List Implm.
create	$O(1)$	$O(1)$
insert	$O(N)$	$\mathbf{O(1)}$
delete	$O(N)$	$O(N)$
find	$O(N)$	$O(N)$
find_kth	$\mathbf{O(1)}$	$O(N)$
free	$\mathbf{O(1)}$	$O(N)$

Example: The Polynomial ADT

- Array implementation: **inefficient** for sparse polynomials; **waste time** in adding and multiplying zeros.
- Singly linked list implementation: good for both **sparse** and dense polynomials.
  - A very good application of linked lists.

```
typedef struct node_s node;
typedef node polynml_t;
struct node_s {
    int coeff;
    int degree;
    node *next;
};
```

Example: $x^5 - 2x^2 + 6x - 8$



The Polynomial ADT (Cont')

- **Addition**: Find like terms and add up coefficients. Add all unlike terms.
- **Multiplication**: nested list traversal (nested for-loops)

```
polynml_t *add(polynml_t *p1, polynml_t *p2){
    polynml_t *psum = list_copy(p1);
    node *p, *t;

    for (p = p2->next; p != NULL; p = p->next)
        if ((t = list_find(psum, p->degree)) != NULL)
            t->coeff += p->coeff; /* add up like terms */
        else  /* unlike terms just add to list */
            list_insert(psum, psum, *p);
    return psum;
}
```

Complexity: $O(|p_1||p_2|)$

$|p|$ denotes the no. of terms in the polynomial p.

Summary

- Concept and definition of linked lists
- Basic operation of linked lists: insertion, deletion, traversal
- Linked list implementation details in C
- Different conventions in realizing linked lists and extension to doubly linked lists.
- Introduce the concept of abstract data type.
- The list ADT with
  - Array implementation
  - Linked list implementation
- Application of linked list in abstracting polynomials.