

Data Structures and Algorithms

(資料結構與演算法)

Lecture 4: Linked List

Hsuan-Tien Lin (林軒田)

`htlin@csie.ntu.edu.tw`

Department of Computer Science
& Information Engineering

National Taiwan University
(國立台灣大學資訊工程系)



motivation of linked list

Application: Polynomial Computation

$$x^4 + 4x^2 + 5$$

1	2	3
(1, 4)	(4, 2)	(5, 0)

solution 0: **ordered** array on (coefficient, **exponent**)

Issue of Ordered Array for Polynomial Computation

$$\begin{aligned} & x^4 + 4x^2 + 5 + 7x^3 \\ = & x^4 + 7x^3 + 4x^2 + 5 \end{aligned}$$

	1	2	3	4
before	(1, 4)	(4, 2)	(5, 0)	
after	(1, 4)	(7, 3)	(4, 2)	(5, 0)

- essentially **INSERT** in ordered array
- many **content moving** when cutting in

ordered array: less flexible for **insertion** (& removal)

Solution: Linked List for Flexible Insertion

$$x^4 + 4x^2 + 5 + 7x^3 = x^4 + 7x^3 + 4x^2 + 5$$

	1	2	3	4
	(1, 4)	(4, 2)	(5, 0)	
next (before)	2	3	NIL	
	(1, 4)	(4, 2)	(5, 0)	(7, 3)
next (after)	4	3	NIL	2

linked list $L \xrightarrow{L.head:1} (1, 4) \xrightarrow{4} (7, 3) \xrightarrow{2} (4, 2) \xrightarrow{3} (5, 0)$

no content moving, just changing clues (pointers)

overhead of *next* \implies flexible INSERT-AFTER

Algorithms (Operations) for Linked List

GET-DATA(*clue*)

1 **return** *data[clue]*

⇒

GET-DATA(*node*)

1 **return** *node.data*

GET-NEXT(*clue*)

1 **return** *next[clue]*

⇒

GET-NEXT(*node*)

1 **return** *node.next*

INSERT-AFTER(*node*, *data*)

```
1 newNode = NODE(data, node.next)  
2 node.next = newNode  
3
```

REMOVE-AFTER(*node*)

```
1 oldNode = node.next  
2 node.next = oldNode.next  
3 return oldNode
```

linked list: flexibility to allocate NODE **anywhere!**

Linked List: Ordered versus Unordered

Ordered Linked List

- output polynomial **orderly**
- INSERT: (sequential) **search** before insertion
- UPDATE: **remove then insert**

Unordered Linked List

- **general purposes**
- INSERT: usually just **head insertion**
- UPDATE: **simply change** the node

ordered versus **unordered**:
trade-off between **(orderly) output** & **maintenance** time

doubly linked list

REMOVE-HERE for Linked List

REMOVE-HERE(node@2)

	1	2	3	4
	(1, 4)	(4, 2)	(5, 0)	(7, 3)
next	4	3	NIL	2

$\xrightarrow{L.head:1} (1, 4) \xrightarrow{4} (7, 3) \xrightarrow{2} (4, 2) \xrightarrow{3} (5, 0)$

Need

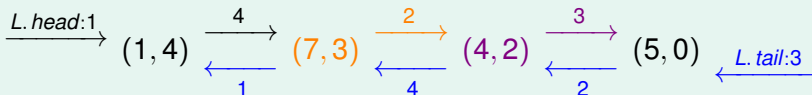
- $node@4.next = node@2.next$
- do not know where $node@4$ is, unless sequential search

REMOVE-HERE: hard for **singly** linked list

Doubly Linked List: More Flexible REMOVE-HERE

REMOVE-HERE(node@2)

	1	2	3	4
	(1, 4)	(4, 2)	(5, 0)	(7, 3)
next	4	3	NIL	2
prev	NIL	4	2	1



REMOVE-HERE(*node*)

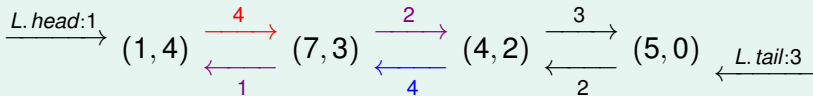
- 1 *node.prev.next* = *node.next*
- 2 *node.next.prev* = *node.prev*
- 3 **return** *node*

overhead of *prev* in doubly linked list
 \Rightarrow flexible REMOVE-HERE

INSERT for Doubly Linked List

$$x^4 + 4x^2 + 5 + 7x^3 = x^4 + 7x^3 + 4x^2 + 5$$

	1	2	3	4
	(1, 4)	(4, 2)	(5, 0)	(7, 3)
next	4	3	NIL	2
prev	NIL	4	2	1



INSERT-AFTER(*node*, *data*)

```

1  newNode =
    NODE(data, node.next, node)
2  node.next = newNode
3  newNode.next.prev = newNode
  
```

INSERT-BEFORE(*node*, *data*)

```

1  newNode =
    NODE(data, node, node.prev)
2  node.prev = newNode
3  newNode.prev.next = newNode
  
```

not just space overhead,
but also **time overhead** for **doubly linked list**

linked list for sparse vector

Application: Sparse Vector in Scientific Computing

(Coordinate) Vector

e.g.

$$[1, 1, 2, 6, 1, 3, 1, 4] \in \mathbb{R}^8$$

—essentially a fixed-dimensional array

Sparse Vector

vector with **many 0's** and **few non-zero**, e.g.

$$[1, 0, 0, 0, 0, 3, 0, 4] \in \mathbb{R}^8$$

—frequently used in signal processing, scientific computing, etc.

polynomial: a special case of **sparse** vector
(e.g. $x^0 + 3x^5 + 4x^7$)

Sparse Vector: Dense Array versus Linked List

$$[1, 0, 0, 0, 0, 3, 0, 4] \in \mathbb{R}^8$$

Dense Array Representation

store every value (8 integers)

index	1	2	3	4	5	6	7	8
value	1	0	0	0	0	3	0	4

Linked List Representation

store **non-zero** values with ordered linked list of (index, value)

$$(1, 1) \longrightarrow (6, 3) \longrightarrow (8, 4)$$

storing only **non-zeros**: time/space efficient if **many zeros**

Merging Sparse Vectors

$$\begin{aligned} & [1, 0, 0, 0, 0, 3, 0, 4] + [0, 0, 0, 0, 0, 6, 9, 0] \\ & (1, 1) \longrightarrow (6, 3) \longrightarrow (8, 4) + (6, 6) \longrightarrow (7, 9) \end{aligned}$$

SUM(A, B)

```
1  L = empty list, ca = A.head, cb = B.head
2  while ca ≠ NIL and cb ≠ NIL
3      if ca.index equals cb.index
4          if (ca.value + cb.value) ≠ 0
5              L.INSERT-TAIL(NODE(ca.value + cb.value, NIL))
6              ca = ca.next, cb = cb.next
7      elseif ca.index > cb.index
8          L.INSERT-TAIL(NODE(cb.value, NIL))
9          cb = cb.next
10     else
11         L.INSERT-TAIL(NODE(ca.value, NIL))
12         ca = ca.next
13 [L.INSERT-TAIL on the non-NIL cursor]
14 return L
```

running **cursors** algorithm for merging two (linked) lists

Real-World Use of Sparse Vector: LIBSVM

svm.cpp

```
double Kernel::dot(const svm_node *px, const svm_node *py){
    double sum = 0;
    while(px->index != -1 && py->index != -1){
        if(px->index == py->index){
            sum += px->value * py->value;
            ++px;
            ++py;
        }
        else{
            if(px->index > py->index)
                ++py;
            else
                ++px;
        }
    }
    return sum;
}
```

similar to **sparse vector merging**;
good data structure needed everywhere

linked list in job interviews

Linked List Reversal

nothing special, but important to “code on board”

“Cycle” in Linked List?

tortoise-hare (turtle-rabbit) algorithm

Middle of Linked List

two pass, or tortoise-hare algorithm

Summary

Lecture 4: Linked List

- motivation of linked list
overhead of *next* for flexibility
- doubly linked list
overhead of *prev* for more flexibility
- linked list for sparse vector
storing non-zeros for time/space efficiency
- linked list in job interviews
often used to test basic computational thinking