CSE221

Linked Lists

2021 Fall

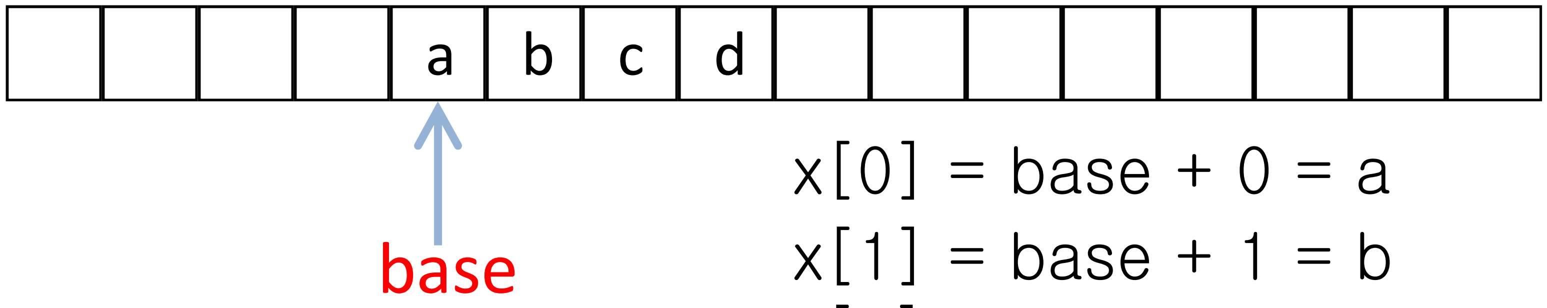
Young-ri Choi

Acknowledgment: The content of this file is based on the slides of the textbook as well as the slides provided in former lectures at UNIST.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Recap: Arrays

1-dimensional array



x[0] = base + 0 = a

x[1] = base + 1 = b

x[2] = base + 2 = c

x[3] = base + 3 = d

- Map into contiguous memory space
- Use index to locate a particular element
  - Location of ist element = base + i

Recap: Two Ways to Store a 2D-Matrix

1 integer

$$
\begin{array}{c|cccccc|}
 & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & 15 & 0 & 0 & 22 & 0 & -15 \\
1 & 0 & 11 & 3 & 0 & 0 & 0 \\
2 & 0 & 0 & 0 & -6 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 91 & 0 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 28 & 0 & 0 & 0
\end{array}
\longrightarrow
$$

3 integers
(3x more space)

	row	col	val
[0]	0	0	15
[1]	0	3	22
[2]	0	5	-15
[3]	1	1	11
[4]	1	2	3
[5]	2	3	-6
[6]	4	0	91
[7]	5	2	28

space saving = 12
(36/24 integers in dense/sparse matrix)

- Sparse representation is more effective when there are many empty elements

Recap: Sparse Matrix

- Array of triples <row, col, value> for nonzeros
  - Effective when there are many empty elements

$$\begin{array}{c|cccccc|} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 15 & 0 & 0 & 22 & 0 & -15 \\ 1 & 0 & 11 & 3 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & -6 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 91 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 28 & 0 & 0 & 0 \end{array} \longrightarrow$$

	row	col	val
[0]	0	0	15
[1]	0	3	22
[2]	0	5	-15
[3]	1	1	11
[4]	1	2	3
[5]	2	3	-6
[6]	4	0	91
[7]	5	2	28

In fact not a good use case: space saving = 1

Question: Use Arrays for Sparse Matrix?

- It depends
  - Sparse matrix is mainly for optimizing space
  - Recall Array Abstract Data Type (ADT)
    - Mapping between index and value
    - Operators like retrieve value or store value
    - Using array is just an implementation issue
  - Array vs Linked List can be decided by some factors
    - Does the program perform insert() and delete() heavily?
    - Does the program frequently scan the data?

Outline

- Singly linked list
- Doubly linked list
- Circular lists

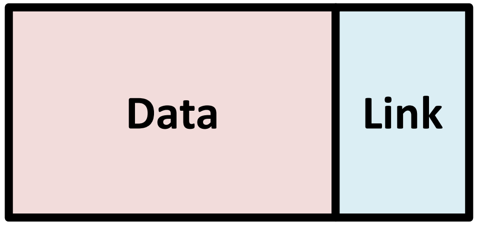
List

- Collection of elements (nodes)
  - that forms a linear ordering
- Examples
  - Shopping list, Laundry list, Black/white list
- Implement with an array
  - static, fixed

Linked List

- Elements are stored in an arbitrary order in memory
  - Each element can be put any physical location
  - Order is maintained by using link

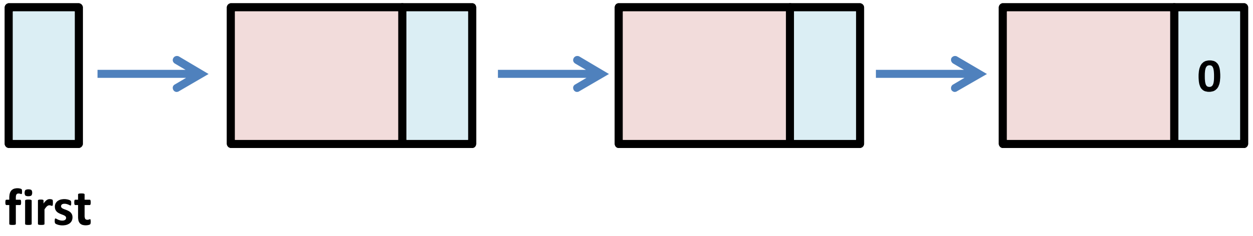
- Node of linked list
  - Data field
  - Link (pointer) fields



ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Singly Linked List Representation

- First (or head) pointer points to first node
- A sequential list of nodes through links
- Null terminated at the end

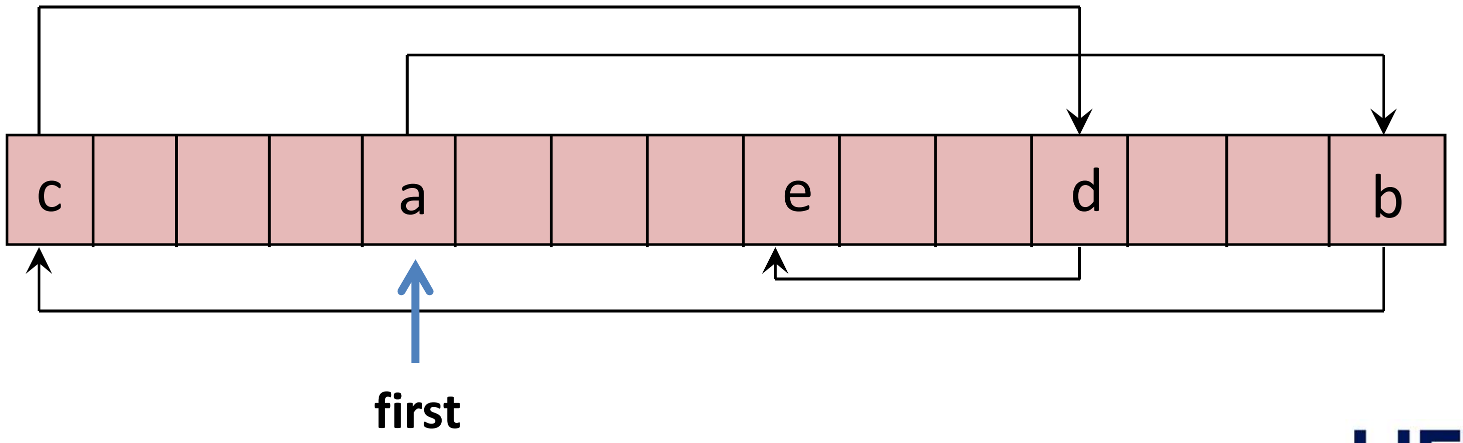


Memory Layout

- L=(a,b,c,d,e)
- Array representation

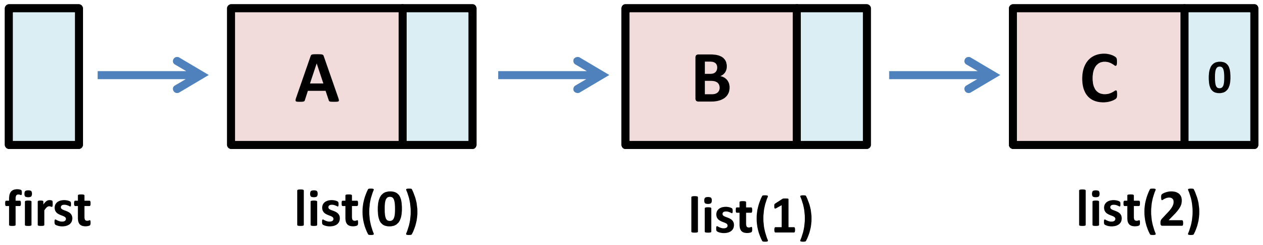
a	b	c	d	e										

- Linked list representation



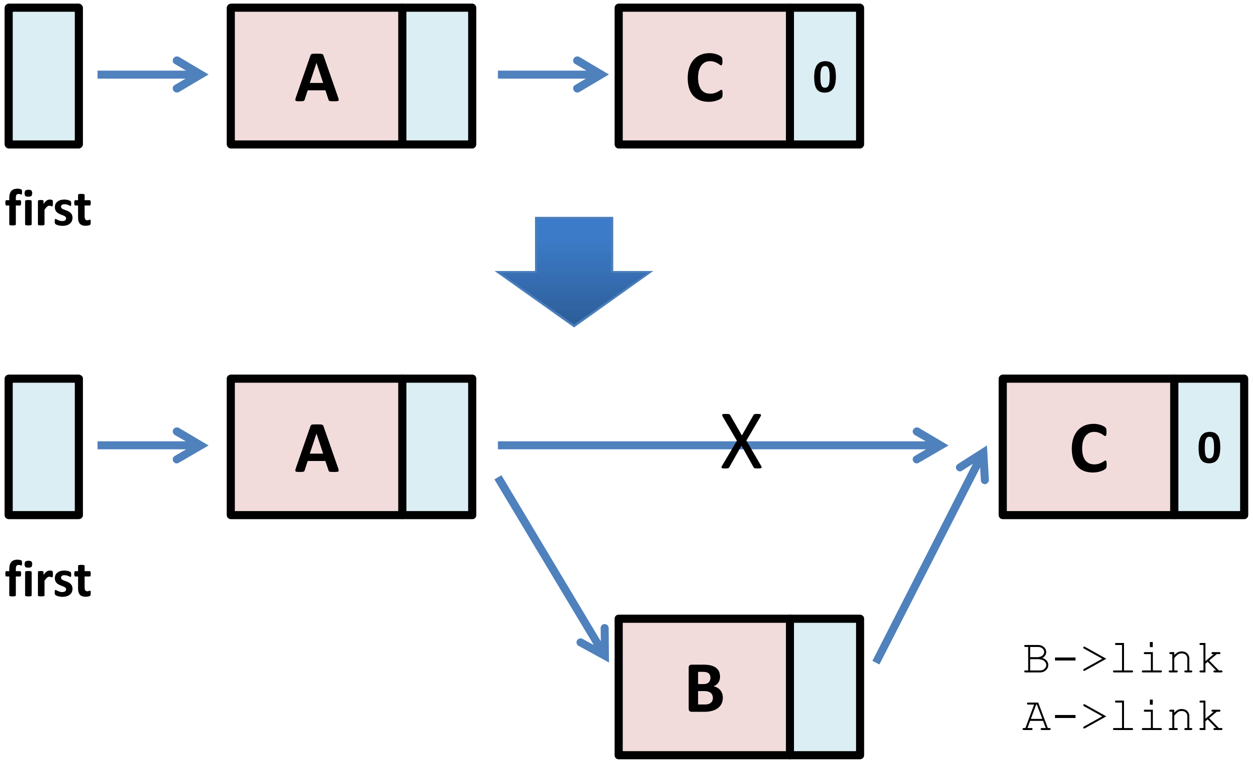
Basic Operations

- Access element
  - list(0) = first->data = A
  - list(1) = first->link->data = B
  - list(2) = first->link->link->data = C



Basic Operations

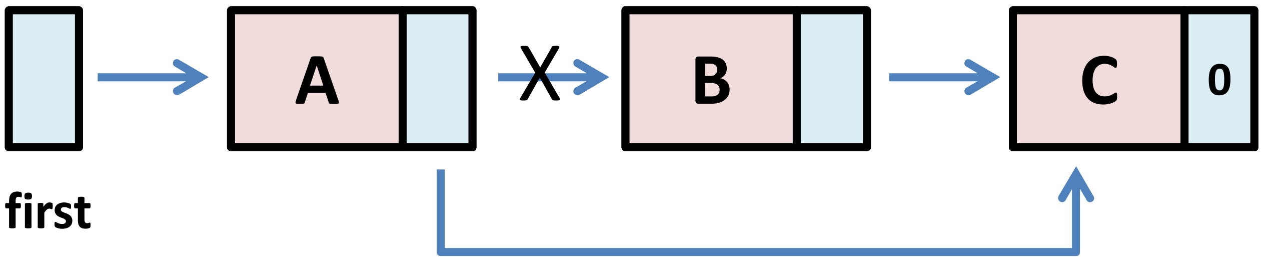
- Insert B after A



```
B->link = A->link
A->link = B
```

Basic Operations

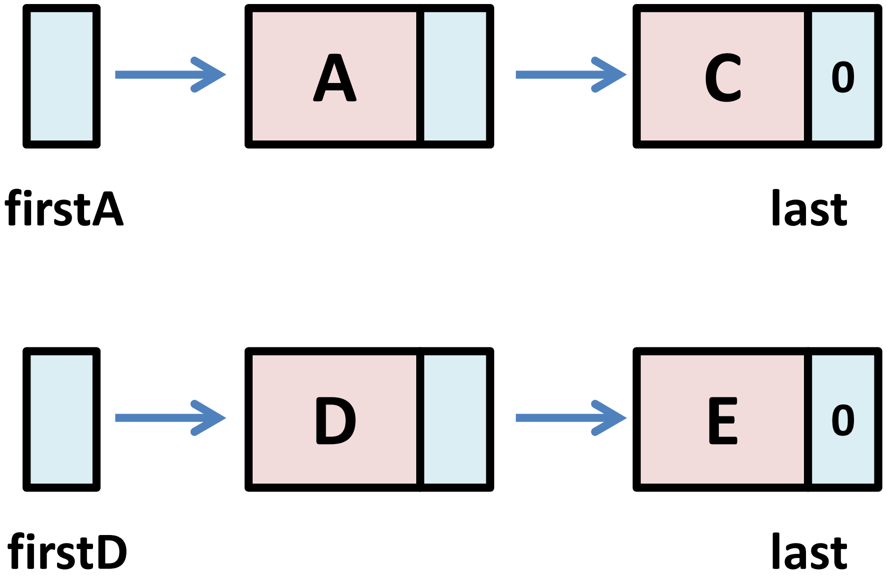
- Delete B
  - –Need A preceding B
    - A has to be either given or searched



```
A->link = B->link
delete B
```

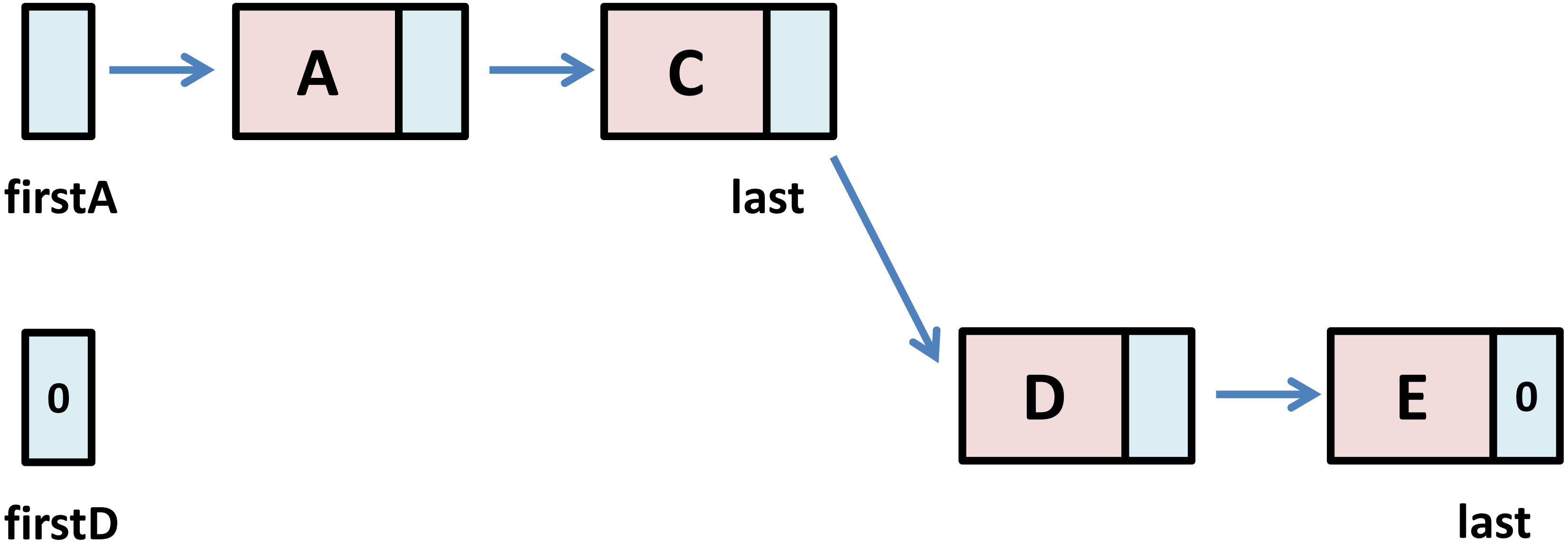
Basic Operations

- Concatenation



Basic Operations

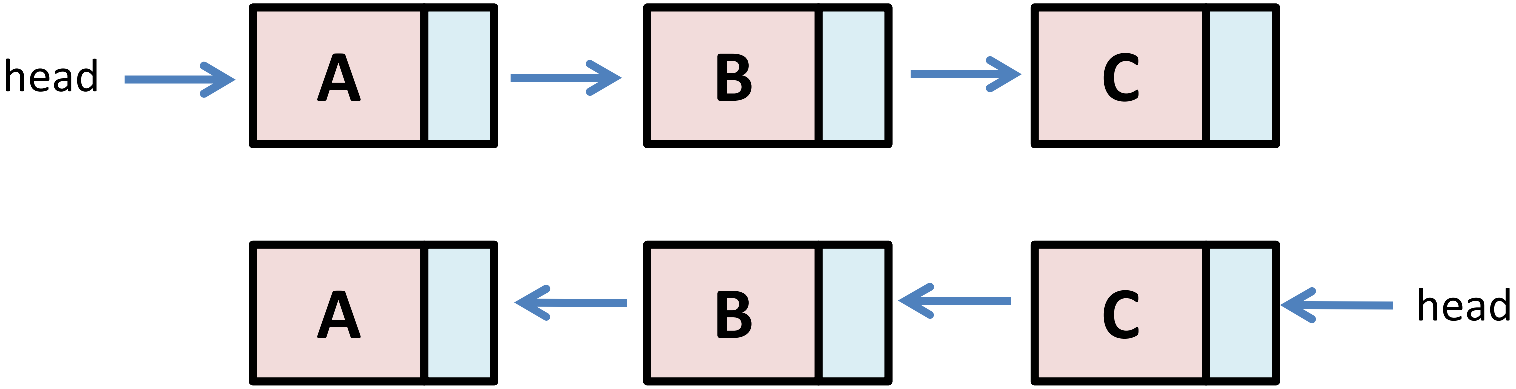
- Concatenation



```
C->link = firstD
firstD = NULL
```

Basic Operations

- Reverse



```
Initially, prev=NULL, current=head, next=NULL

while(current != NULL){
    next= current->next;
    current->next = prev;
    prev=current;
    current=next;
}
head=prev;
```

Recap: Array v.s. Linked List

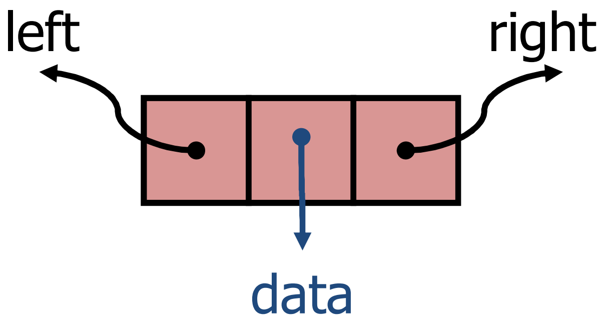
- Arrays
  - –Good for random access and sequential access
    - Indexing
    - Modern architecture reads data at a chunk
- Linked list
  - –Good for frequent inserting and deleting
    - Array must shift data for each inserting and deleting
  - –Resizing is easier
    - Array typically given with non-growable fixed space

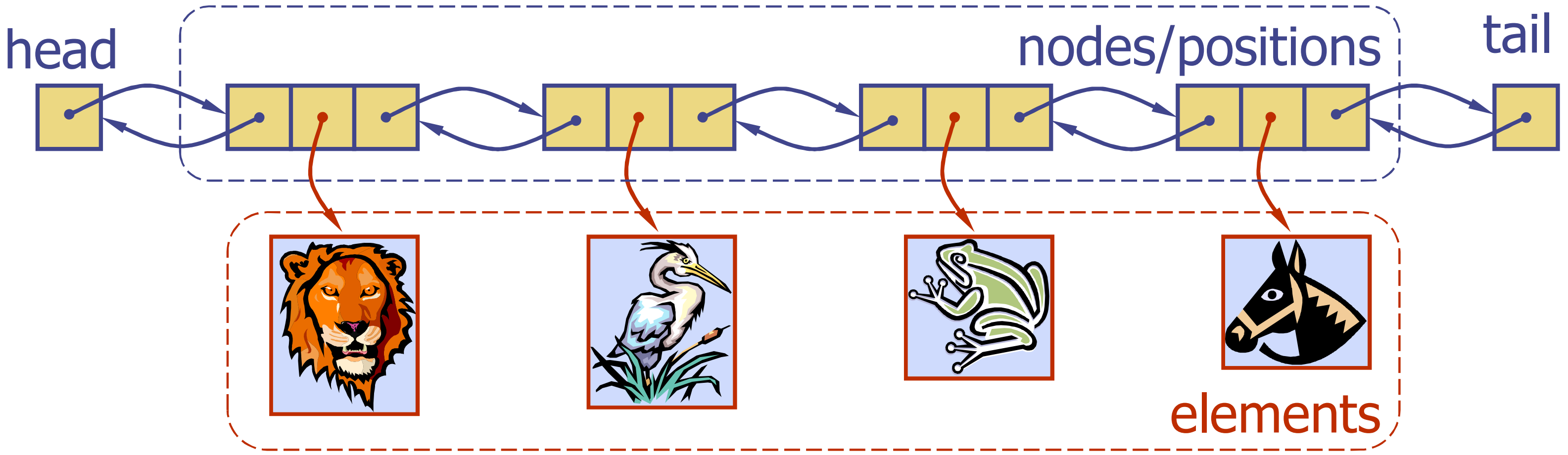
Outline

- Singly linked list
- Doubly linked list
- Circular lists

Doubly Linked List

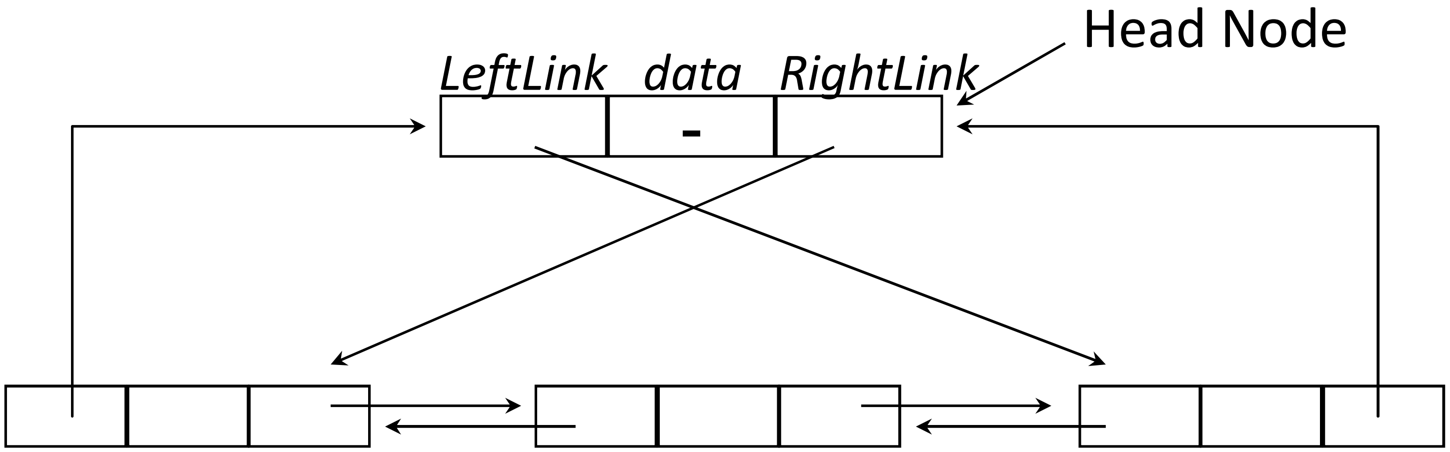
- Two link pointers
  - –Left, right
- Can traverse both directions





Doubly Linked List

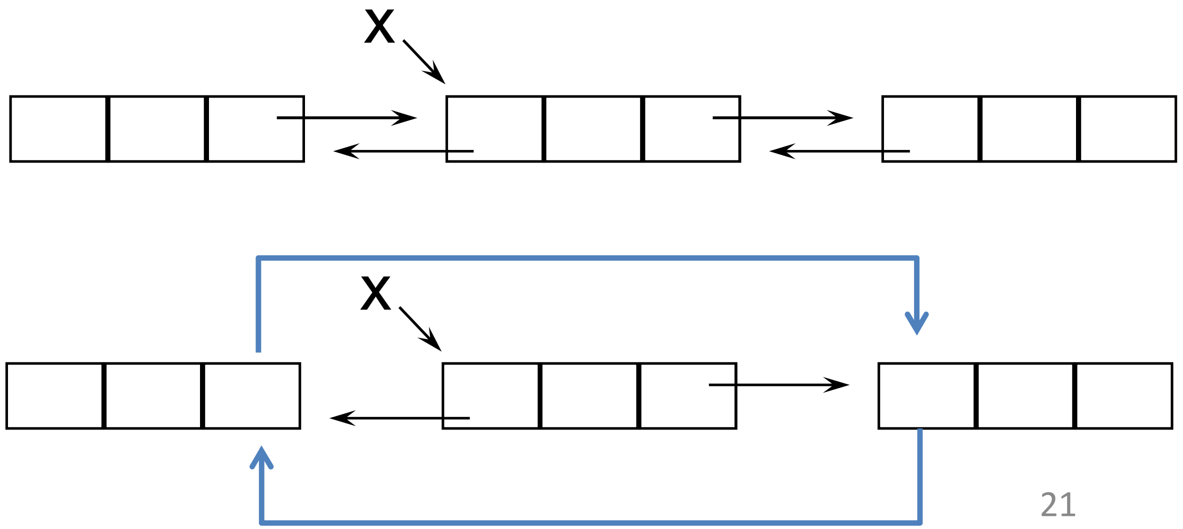
- Using a head node (version 2)
  - Can point to head and tail using a single node



Doubly Linked List

- Delete: no preceding node needed

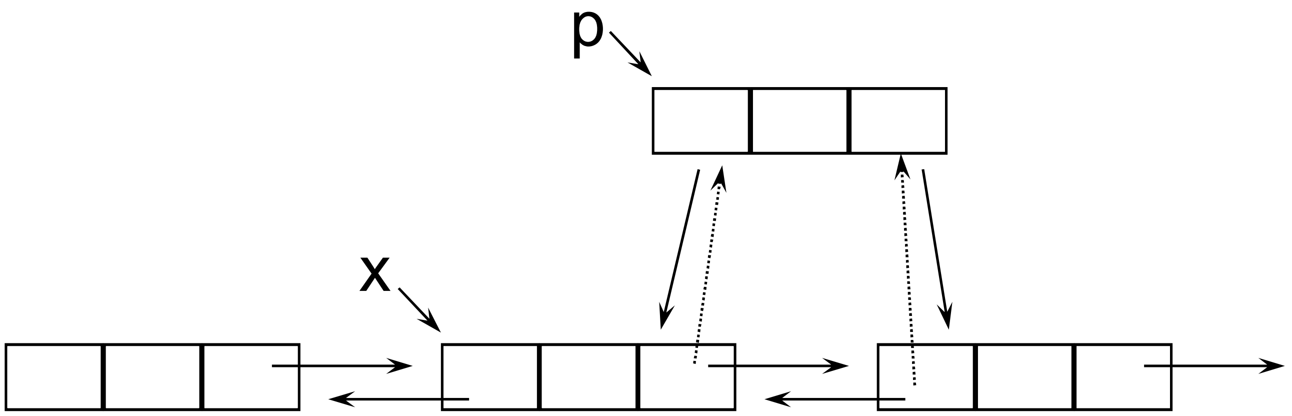
```
void DblList::Delete(DblListNode *x)
{
   x->left->right = x->right;
   x->right->left = x->left;
   delete x;
}
```



Doubly Linked List

- Insert

```
void DblList::Insert(DblListNode *p, DblListNode *x)
{
    p->llink = x;
    p->rlink = x->rlink;
    x->rlink->llink = p;
    x->rlink = p;
}
```

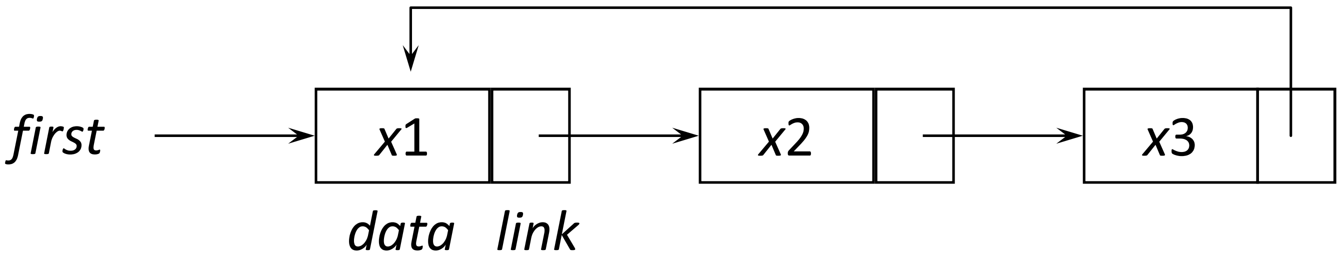


Outline

- Singly linked list
- Doubly linked list
- Circular lists

Circular Linked List

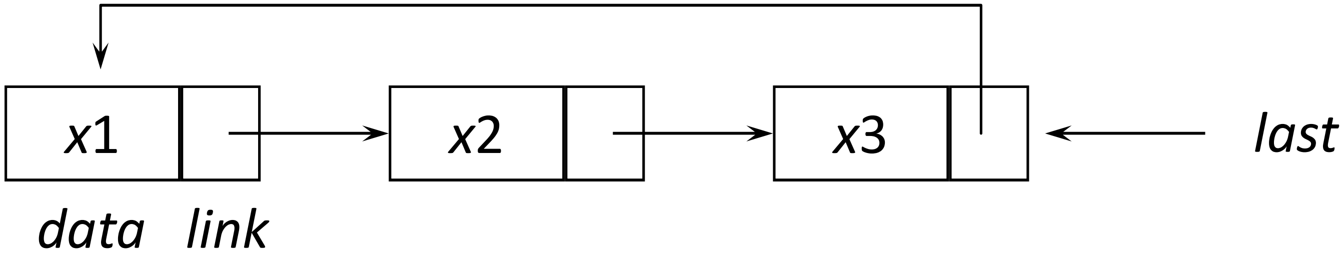
- Link of the last node points to the first node
  - last->link = first
  - No null pointer
- Efficient for circular accessing problems
  - Round-robin scheduling



Circular Linked List

- Insert at the end is inefficient
  - Need to search *last* from *first*

- Keep *last* instead of *first*
  - Insert at the end and front can be O(1)
  - first = last-> link



Tradeoff among Linked Lists

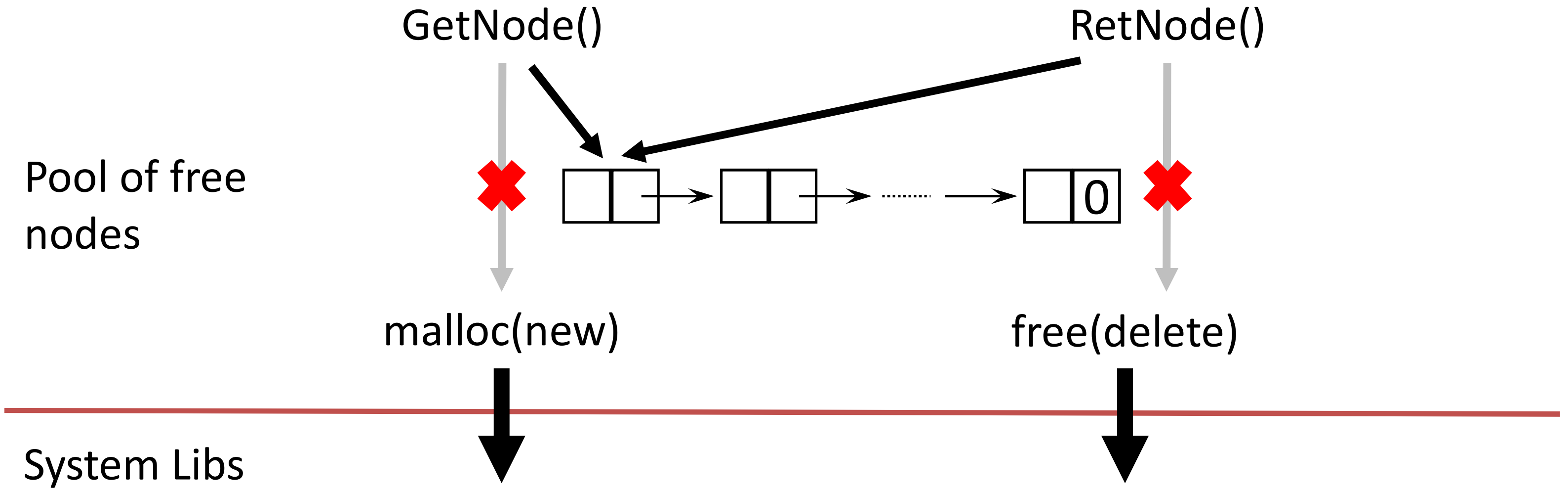
- Singly Linked List ↔ Circular Linked List
  - No difference in space consumption
  - When heavily accessing the first and last element
    - Circular can keep only one pointer (last)
    - Singly needs to keep two pointers (first and last)
- Circular Linked List ↔ Doubly Linked List
  - Doubly consumes roughly 50% more space for pointers
  - More efficient in several use cases, though
    - E.g., 1: accessing the second last elem
    - E.g., 2: delete(index) incurs half memory accesses (no prev pt)

Available Space Lists

- New (malloc) and delete (free) are expensive
  - Need O(n) time to delete all nodes in list of size n
- We can manage a pool (list) of free nodes
  - When allocating a new node, we instead get a free node from the pool
  - When a node is deleted, we return it to the pool
  - Can delete all nodes at O(1)

Available Space Lists

- Deferring and reducing memory function calls
  - –Call malloc() when the list is empty
  - –Call free() when the list is too full



Expensive!

Available Space Lists

- *avail*: first pointer of available space list
- GetNode()

```
template <class Type>
ListNode<Type>* CircList::GetNode()
// Getting a node from the pool
{
    ListNode<Type>* x;
    if(!avail) x = new ListNode<Type>;
    else { x = avail; avail = avail->link; }
    return x;
}
```

System lib call:
expensive

Available Space Lists

- RetNode()

```
template <class Type>
void CircList<Type>::RetNode(ListNode<Type>* x)
// Return x to the free node pool
{
   x->link = avail;
   avail = x;
   x = 0;
}
```

Zeroing. Why?

Available Space Lists

- Delete entire circular list in O(1)

```
template <class KeyType>
void CircList<Type>::~CircList()
// Delete the circular linked list
{
   if (last) {
      ListNode<Type>* first = last->link; // assume we store last
      last->link = avail; // last node linked to avail
      avail = first;  // first node of list becomes front of avail
                         list
      last = 0;
   }
}
```

Questions?