Chapter 2 – LIST

- Linear List Concepts
- List ADT
- Specifications for List ADT
- Implementations of List ADT
- Contiguous List
- Singly Linked List
- Other Linked Lists
- Comparison of Implementations of List
DEFINITION: Linear List is a data structure where each element of it has a unique successor.
Linear List Concepts (cont.)

- Linear lists
  - General
    - Unordered
    - Ordered
  - Restricted
    - FIFO (queue)
    - LIFO (stack)
General list:

- No restrictions on which operation can be used on the list
- No restrictions on where data can be inserted/deleted.

- Unordered list (random list): Data are not in particular order.
- Ordered list: data are arranged according to a key.
Restricted list:

- Only some operations can be used on the list.
- Data can be inserted/deleted only at the ends of the list.

- **Queue**: FIFO (First-In-First-Out).
- **Stack**: LIFO (Last-In-First-Out).
DEFINITION: A list of elements of type T is a finite sequence of elements of T together with the following operations:

Basic operations:
- **Construct** a list, leaving it empty.
- **Insert** an element.
- **Remove** an element.
- **Search** an element.
- **Retrieve** an element.
- **Traverse** the list, performing a given operation on each element.
List ADT (cont.)

Extended operations:

- Determine whether the list is *empty* or not.
- Determine whether the list is *full* or not.
- Find the *size* of the list.
- *Clear* the list to make it empty.
- *Replace* an element with another element.
- *Merge* two ordered list.
- *Append* an unordered list to another.
- …
Insertion

- Insert an element at a specified position p in the list
  - Only with *General Unordered List*.

- Insert an element with a given data
  - *With General Unordered List*: can be made at any position in the list (at the beginning, in the middle, at the end).
  - *With General Ordered List*: data must be inserted so that the ordering of the list is maintained (searching appropriate position is needed).
  - *With Restricted List*: depend on its own definition (FIFO or LIFO).
Insert an element at a specified position $p$ in the list.

Before:

$$\begin{array}{ccccccc}
1 & \text{20} & \cdots & \text{30} & \text{10} & \text{50} & \text{40} & n
\end{array}$$

After:

$$\begin{array}{ccccccc}
1 & \text{20} & \cdots & \text{30} & \text{10} & \text{60} & \text{50} & \text{40} & \cdots & n+1
\end{array}$$

Any element formerly at position $p$ and all later have their position numbers increased by 1.
Removal, Retrieval

- Remove/Retrieve an element at a specified position $p$ in the list
  
  - With General Unordered List and General Ordered List.

- Remove/Retrieve an element with a given data
  
  - With General Unordered List and General Ordered List: Searching is needed in order to locate the data being deleted/retrieved.
**Removal**

*Remove an element at a specified position* $p$ *in the list.*

**Before:**

<table>
<thead>
<tr>
<th></th>
<th>$p-2$</th>
<th>$p-1$</th>
<th>$p$</th>
<th>$p+1$</th>
<th>$n+1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
<td>30</td>
<td>10</td>
<td>60</td>
<td>50</td>
</tr>
</tbody>
</table>

**After:**

<table>
<thead>
<tr>
<th></th>
<th>$p-2$</th>
<th>$p-1$</th>
<th>$p$</th>
<th>$n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
<td>30</td>
<td>10</td>
<td>50</td>
</tr>
</tbody>
</table>

The element at position $p$ is removed from the list, and all subsequent elements have their position numbers decreased by 1.
Retrieve an element at a specified position $p$ in the list.

Before:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>p-2</th>
<th>p-1</th>
<th>p</th>
<th>p+1</th>
<th>n+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>30</td>
<td>10</td>
<td>60</td>
<td>50</td>
<td>40</td>
<td></td>
</tr>
</tbody>
</table>

After:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>p-2</th>
<th>p-1</th>
<th>p</th>
<th>p+1</th>
<th>n+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>30</td>
<td>10</td>
<td>60</td>
<td>50</td>
<td>40</td>
<td></td>
</tr>
</tbody>
</table>

retrieved data = 60

All elements remain unchanged.
Success of Basic Operations

- **Insertion** is successful when the list is not full.

- **Removal, Retrieval** are successful when the list is not empty.
Specification for List ADT

<void> Create()

<void> Traverse (ref <void> Operation ( ref Data <DataType>) )

<ErrorCode> Search (ref DataOut <DataType>) // DataOut contains values need to be found in key field, and will receive all other values in other fields.

// For Unsorted List:

<ErrorCode> Insert (val DataIn <DataType>, val position <integer>)

<ErrorCode> Remove (ref DataOut <DataType>, val position <integer>)

<ErrorCode> Retrieve (ref DataOut <DataType>, val position <integer>)

<ErrorCode> Replace (val DataIn <DataType>,
    ref DataOut <DataType>, val position <integer>)

(Operations are successful when the required position exists).
Specification for List ADT (cont.)

// For Sorted List:

<ErrorCode> Insert (val DataIn <DataType>)

<ErrorCode> Remove (ref DataOut <DataType>) // DataOut contains values need to be found in key field, and will receive all other values in other fields.

<ErrorCode> Retrieve (ref DataOut <DataType>) // DataOut contains values need to be found in key field, and will receive all other values in other fields.

(Insertion is successful when the list is not full and the key needs to be inserted does not exist in the list.

Removal and Retrieval are successful when the list is not empty and the required key exists in the list.)
Specification of List ADT (cont.)

Samples of Extended methods:

<boolean> isFull()
<boolean> isEmpty()
<integer> Size()
<ErrorCode> Sort()

<ErrorCode> AppendList (ref ListIn <ListType>) // For Unordered Lists.
ListIn may be unchanged or become empty.

<ErrorCode> Merge (ref ListIn1 <ListType>, ref ListIn2 <ListType>)
// For Ordered Lists.

...
### Specification of List ADT (cont.)

#### Samples of variants of similar methods:

<code><void> Create()</void>
<code><void> Create (ref file <InOutType>) // made a list from content of a file</code>
<code><ErrorCode> Insert (val DataIn <DataType>, val position <integer>)</ ErrorCode>
<code><ErrorCode> InsertHead (val DataIn <DataType>)</ ErrorCode>
<code><ErrorCode> InsertTail (val DataIn <DataType>)</ ErrorCode>
<code><ErrorCode> Replace (val DataIn <DataType>, ref DataOut <DataType>, val position <integer>)</ ErrorCode>
<code><ErrorCode> Replace (val DataIn <DataType>, ref DataOut <DataType>)</ ErrorCode>

CuuDuongThanCong.com
https://fb.com/tailieudientucntt
Samples of variants of similar methods:

<ErrorCode> Remove (val position <integer>)
<ErrorCode> Remove (ref DataOut <DataType>,
    val position <integer>)
<ErrorCode> RemoveHead (val DataOut <DataType>)
<ErrorCode> RemoveTail (ref DataOut <DataType>)

<ErrorCode> Search (val DataIn <DataType>, ref ListOut <ListType>)
    // DataIn contains values need to be found in some fields, ListOut
    will contain all members having that values.

...
Implementations of List ADT

- **Contiguous implementation:**
  - Automatically Allocated Array with fixed size.
  - Dynamically Allocated Array with flexible size.

- **Linked implementations:**
  - Singly Linked List
  - Circularly Linked List
  - Doubly Linked List
  - Multilinked List
  - Skip List
  - . . .

- **Linked List in Array**
Array with *pre-defined maxsize* and has *n* elements.

**List** // Contiguous Implementation of List

- **count** <integer> // number of used elements *(mandatory)*.
- **data** <array of <DataType> > // *(Automatically Allocated Array)*

End List
Dynamically Allocated Array

List // Contiguous Implementation of List

- **count**: <integer> // number of used elements (mandatory).
- **data**: <dynamic array of <DataType>> // (Dynamically Allocated Array)
- **maxsize**: <integer>

End List
Sample of using List ADT

```cpp
#include <iostream>
#include <List> // uses Unordered List ADT.

int main()
{
    List<int> listObj;
    cout << "Enter 10 numbers: \n" << flush;
    int i, x;
    for (i=0; i<10; i++)
    {
        cin >> x;
        listObj.Insert( x, listObj.Size() ); // Insert at the end of the list.
    }
    cout << "Elements in the list: \n";
    for (i=0; i<10; i++)
    {
        listObj.Retrieve(x, i);
        cout << x << "t";
    }
    return 0;
}
```
Contiguous Implementation of List

In processing a contiguous list with n elements:

- **Insert** and **Remove** operate in time approximately proportional to n (require physical shifting).

- **Clear, Empty, Full, Size, Replace, and Retrieve** in constant time.
**Singly Linked List**

**List**  // Linked Implementation of List (for Singly Linked List)

head <pointer>

count <integer>  // number of elements (optional).

End List

An empty Singly Linked List having only head.

An empty Singly Linked List having head and count.
Singly Linked List (cont.)

Node
- data <DataType>
- link <pointer>

End Node

***General DataType:***

**DataType**
- **key** <KeyType>
- field1 <…>
- field2 <…>
- …
- fieldn <…>

End DataType

---

**Element in the Singly Linked List**

- data
- link

A node with one data field

- number

A node with three data fields

- name
- id
- number

A node with one structured data field

- name
- id
- number

**DataType may be an atomic or a composite data**
Singly Linked List (cont.)

- Sample:
Singly Linked List (cont.)

- Sample: list representing polynomial

\[ 3x^5 - 2x^3 + x^2 + 4 \]
Create an Empty Linked List

**Before**

*List having head*

- head: ?
- count: ?

**After**

*List having head and count*

- head: NULL
- count: 0
Create an Empty Linked List (cont.)

<void> Create()
Creates an empty link list

Pre  none
Post An empty linked list has been created.

1. head = NULL
2. Return
end Create
1. Allocate memory for the new node and set up data.

2. Locate the pointer p in the list, which will point to the new node:
   - If the new node becomes the first element in the List: p is head.
   - Otherwise: p is pPre->link, where pPre points to the predecessor of the new node.
3. Update pointers:
   - Point the new node to its successor.
   - Point the pointer $p$ to the new node.

\[
\begin{align*}
\text{pNew} & \rightarrow \text{link} = \text{head} \quad (1) \\
\text{head} & = \text{pNew} \quad (2)
\end{align*}
\]
Insert Node to a Linked List (cont.)

- Insertion is successful when allocation memory for the new node is successful.

CuuDuongThanCong.com
There is **no difference** between

- insertion **in the middle** (a) and insertion **at the end** of the list (b)

(a)

(b)

\[
\begin{align*}
\text{pNew->link} &= \text{pPre->link} \quad (1) \\
\text{pPre->link} &= \text{pNew} \quad (2)
\end{align*}
\]
Insert Node to a Linked List (cont.)

- There is **no difference** between
  - insertion **at the beginning of the list** (a) and insertion to an empty list (b).

(a) \[\text{head} \rightarrow \cdots \rightarrow \text{pNew} \rightarrow X \]\n(b) \[\text{head} = \text{pNew} \]

\[\text{pNew->link} = \text{head} \]

CuuDuongThanCong.com | https://fb.com/tailieudientucntt
Insert Algorithm

<ErrorCode> **Insert** (val DataIn <DataType>)

// For ordered list.
Inserts a new node in a singly linked list.

**Pre** DataIn contains data to be inserted

**Post** If list is not full, DataIn has been inserted; otherwise, list remains unchanged.

**Return** success or overflow.
InsertNode Algorithm (cont.)

<ErrorCode> **Insert** (val DataIn <dataType>)

1. Allocate pNew
2. if (memory overflow)
   1. return *overflow*
3. else
   1. pNew->data = DataIn
   2. Locate pPre // pPre remains NULL if Insertion at the beginning or to an empty list
3. if (pPre = NULL) // Adding at the beginning or to an empty list
   1. pNew->link = head
   2. head = pNew
4. else // Adding in the middle or at the end of the list
   1. pNew->link = pPre->link
   2. pPre->link = pNew
5. return *success*
end Insert
Remove Node from a Linked List

1. Locate the **pointer** \( p \) in the list which points to the node to be deleted (\( p_{\text{Del}} \) will hold the node to be deleted).

   - If that node is the first element in the List: \( p \) is head.

   - Otherwise: \( p \) is \( p_{\text{Pre}} \rightarrow \text{link} \), where \( p_{\text{Pre}} \) points to the predecessor of the node to be deleted.
Remove Node from a Linked List (cont.)

2. Update pointers: $p$ points to the successor of the node to be deleted.

3. Recycle the memory of the deleted node.
Remove Node from a Linked List (cont.)

- Removal is successful when the node to be deleted is found.
There is no difference between

(a) Removal of a node from the middle and
(b) removal of a node from the end of the list.

```
(pPre->link = pDel->link)
Recycle pDel
```
There is **no difference** between

- removal the node **from the beginning** (a) of the list and
- removal the **only-remained node** in the list (b).

(a) ![Diagram for removing a node from the beginning of the list](https://fb.com/tailieudientucntt)

(b) ![Diagram for removing the only-remained node](https://fb.com/tailieudientucntt)
RemoveNode Algorithm

<ErrorCode> Remove (ref DataOut <DataType>)

Removes a node from a singly linked list.

Pre   DataOut contains the key need to be removed.

Post   If the key is found, DataOut will contain the data corresponding to it, and that node has been removed from the list; otherwise, list remains unchanged.

Return success or failed.
<errorCode> Remove (ref DataOut <DataType>)

1. Allocate pPre, pDel // pPre remains NULL if the node to be deleted is at the beginning of the list or is the only node.

2. if (pDel is not found)
   1. return failed

3. else
   1. DataOut = pDel->data
   2. if (pPre = NULL) // Remove the first node or the only node
      1. head = pDel->link
   3. else // Remove the node in the middle or at the end of the list
      1. pPre->link = pDel->link
   4. recycle pDel
   5. return success

end Remove
Search Algorithm for Auxiliary Function in Class

- This search algorithm **is not a public method** of List ADT.

- Sequence Search has to be used for the linked list.

- This studying for the case: List is ordered accordingly to the key field.
Search Algorithm for Auxiliary Function in Class

- **Public method Search of List ADT:**
  
  `<ErrorCode> Search (ref DataOut <DataType>)`

  *Can not return a pointer to a node if found.*

- **Auxiliary function Search of List ADT:**

  `<ErrorCode> Search (val target <KeyType>,
    ref pPre <pointer>,
    ref pLoc <pointer>)`

  *Searches a node and returns a pointer to it if found.*
Successful Searches

Located first

5 \rightarrow 10 \rightarrow \ldots

pPre
pLoc

target

Located middle

15 \rightarrow 20 \rightarrow \ldots

pPre
pLoc

target

Located last

95 \rightarrow 100

pPre
pLoc

target

Unsuccessful Searches

Less than first

target < 5

5 \rightarrow 10 \rightarrow \ldots

pPre
pLoc

target < 5

Greater than last

target > 100

95 \rightarrow 100

pPre
pLoc

target > 100

target > 15

15 \rightarrow 20 \rightarrow \ldots

pPre
pLoc

target > 15

target < 20
<ErrorCode> Search (val target <KeyType>,
    ref pPre <pointer>,
    ref pLoc <pointer>)
Searches a node in a singly linked list and return a pointer to it if found.

// For Ordered List
Pre   target is the key need to be found
Post  pLoc points to the first node which is equal or greater than key,
       or is NULL if target is greater than key of the last node in the list.
       pPre points to the largest node smaller than key, or is NULL if
       target is smaller than key of the first node.
Return found or notFound
<ErrorCode> Search (val target <KeyType>,
    ref pPre <pointer>,
    ref pLoc <pointer>)

// For Ordered List
1. pPre = NULL
2. pLoc = head
3. loop ( (pLoc is not NULL) AND (target > pLoc ->data.key) )
   1. pPre = pLoc
   2. pLoc = pLoc ->link
4. if (pLoc is NULL)
   1. return notFound
5. else
   1. if (target = pLoc ->data.key)
      1. return found
   2. else
      1. return notFound
end Search
Retrieve Algorithm

- Using Search Algorithm to locate the node
- Retrieving data from that node
Retrieve Algorithm (cont.)

<ErrorCode> Retrieve (val target <KeyType>,
                    ref DataOut <DataType>)

Retrieves data from a singly linked list

Pre   target is the key its data need to be retrieved
Post  if target is found, DataOut will receive data
Return success or failed
Uses  Auxiliary function Search of class List ADT.
RetrieveNode Algorithm (cont.)

<ErrorCode> Retrieve (val target <KeyType>,
                    ref DataOut <DataType>)

1.  errorCode = Search (target, pPre, pLoc)
2.  if (errorCode = notFound)
    1.  return failed
3.  else
    1.  DataOut = pLoc->data
    2.  return success
end Retrieve
Traverse List

Traverse Module controls the loop:

Calling a user-supplied operation to process data

```c
<void> Traverse (ref <void> Operation ( ref Data <DataType>) )
```

Traverses the list, performing the given operation on each element.

**Pre**  Operation is user-supplied.

**Post** The action specified by Operation has been performed on every element in the list, beginning at the first element and doing each in turn.

1.  pWalker = head

2.  **loop** (pWalker is not NULL)
   1.  Operation(pWalker->data)
   2.  pWalker = pWalker->link

end Traverse
Traverse List (cont.)

User controls the loop:

Calling GetNext Algorithm to get the next element in the list.

<ErrorCode> **GetNext** (val fromWhere <boolean>,
ref DataOut <DataType>)

Traverses the list, each call returns data of an element in the list.

**Pre** fromWhere is 0 to start at the first element, otherwise, the next element of the current needs to be retrieved.

**Post** According to fromWhere, DataOut contains data of the first element or the next element of the current (if exists) in the list. That element becomes the current.

**Return** *success* or *failed*.
Traverse List (cont.)

User controls the loop:
Calling GetNext Algorithm to get the next element in the list.

<void> Operation ( ref Data <DataType>)// the function needs to apply to each element in the list.

User controls the loop:
1. errorCode = GetNext(0, DataOut)
2. loop (errorCode = success)
   1. Operation(DataOut)
   2. errorCode = GetNext(1, DataOut)
GetNext Algorithm

- Singly Linked List has additional attribute pCurr pointing to the current element (the last element has just been processed).
GetNext Algorithm (cont.)

<ErrorCode> GetNext (val fromWhere <boolean>,
   ref DataOut <DataType>)

1. if (fromWhere is 0)
   1. if (count = 0)
      1. return failed
   2. else
      1. pCurr = head
      2. DataOut = pCurr->data
      3. return success
2. else
   1. if (pCurr->link is NULL)
      1. return failed
   2. else
      1. pCurr = pCurr->link
      2. DataOut = pCurr->data
      3. return success
end GetNext
Clear List Algorithm

<void> Clear()
Removes all elements from a list.

Pre none.
Post The list is empty.

1. loop (head is not NULL)
   1. pDel = head
   2. head = head->link
   3. recycle pDel
2. return
end Clear
Comparison of Implementations of List

- Contiguous storage is generally preferable
  - When the entries are individually very small;
  - When the size of the list is known when the program is written;
  - When few insertions or deletions need to be made except at the end of the list; and
  - When random access is important.

- Linked storage proves superior
  - When the entries are large;
  - When the size of the list is not known in advance; and
  - When flexibility is needed in inserting, deleting, and rearranging the entries.
Doubly Linked List allows going forward and backward.

Insert an element in Doubly Linked List
Circularly Linked List

current

CuuDuongThanCong.com

https://fb.com/tailieudientucntt
Double Circularly Linked List
Multilinked List

- Multilinked List allows traversing in different order.
Skip List

Skip List improves sequential searching.
Choice of variants of Linked List

To choose among linked Implementations of List, consider:

- Which of the operations will actually be performed on the list and which of these are the most important?
- Is there locality of reference? That is, if one entry is accessed, is it likely that it will next be accessed again?
- Are the entries processed in sequential order? If so, then it may be worthwhile to maintain the last-used position as part of list.
- Is it necessary to move both directions through the list? If so, then doubly linked lists may prove advantageous.
There are two linked lists in array:

- One (head) manages used entries.
- Another (available) manages empty entries (have been used or not yet)
# Multilinked List In Array

<table>
<thead>
<tr>
<th>name</th>
<th>next_name</th>
<th>math</th>
<th>next_math</th>
<th>CS</th>
<th>next_CS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clark, F.</td>
<td>5</td>
<td>70</td>
<td>4</td>
<td>50</td>
<td>-1</td>
</tr>
<tr>
<td>Smith, A.</td>
<td>-1</td>
<td>75</td>
<td>0</td>
<td>92</td>
<td>3</td>
</tr>
<tr>
<td>Garcia, T.</td>
<td></td>
<td>83</td>
<td>1</td>
<td>90</td>
<td>3</td>
</tr>
<tr>
<td>Hall, W.</td>
<td>4</td>
<td></td>
<td></td>
<td>55</td>
<td>5</td>
</tr>
<tr>
<td>Evans, B.</td>
<td>1</td>
<td>82</td>
<td></td>
<td>85</td>
<td>5</td>
</tr>
<tr>
<td>Arthur, E.</td>
<td>3</td>
<td></td>
<td></td>
<td>60</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
### Application: Sparse Matrices

<table>
<thead>
<tr>
<th>students</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>8,000</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>courses</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>300</td>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>

The image shows a sparse matrix with students and courses. The matrix is filled with elements A, B, and C.
Two one-dimensional arrays of Linked List are used.
Sparse Matrices

student array of linked list

- **Student no.**
- **Course no.**
- **Grade**

Row link

Column link

- A
- B
- C

CuuDuongThanCong.com

https://fb.com/tailieudientucntt
Sparce Matrices

- Why *two* arrays of linked lists?
- How about *two linked lists* of linked lists?
- How about *3-D* sparse matrices?
Variants of List are used for Graph and Hash Table, we will see later.