CS 2413: Data Structures (Fall’20)

Linked List

Chao Lan
Outline

Background: dynamic list of objects

Linked list: concepts, singly list, doubly list, circular list

Singly linked list: creation, traverse, insertion, deletion, other operations

Doubly linked list: creation, traverse, insertion, deletion, other operations
Background: a dynamic list of objects.

A dynamic list is a list of objects which can grow or shrink, often at target positions.

Original list.  

We can insert an object.  

We can delete an object.
Background: a dynamic list of objects.

We can do other things with dynamic lists, e.g., search, sort, attach, etc.

- Search for 6 in the list.

- Sort objects in the list.

- Attach a list to another.
Linked list, and its head and tail.

**Linked list** is a data structure used to implement dynamic list.

It is a list of linked objects, with a head and a tail at the two ends.

Let's treat this object as head of list.

Let's treat this object as tail of list.

* In the following, we will always treat the leftmost object as head and the rightmost as tail.
Object components

Each object has three components

- a key that holds data
- a pointer to previous object address
- a pointer to next object address

* Object addresses are made-up. They are not necessarily consecutive.
Three types of linked list: singly, doubly, circular

A **singly linked list** is one that only uses the next pointer but not the previous pointer.

![Diagram of a singly linked list](image)

Next pointer of tail is NULL.

This is the end and is pointing nowhere.
Singly linked list only allows one-way travel from head to tail.

001
prev key=8 next=17

017
prev key=3 next=5

005
prev key=6 next=NULL

This pointer leads us to the next object.

No pointer leads us back.
Three types of linked list: singly, doubly, circular

A **doubly linked list** is one that uses both the next pointer and the previous pointer.

 Prev pointer of head is NULL.

 Next pointer of tail is NULL.
Doubly linked list allows two-way travel between head and tail.

prev = NULL
key = 8
next = 017

prev = 001
key = 3
next = 005

prev = 017
key = 6
next = NULL

The next pointer leads us to the next object.

The previous pointer leads us back to the previous object.
A **circular linked list** is singly linked list with tail additionally pointing to the head.
Outline

Background: dynamic list of objects

Linked list: concepts, singly list, doubly list, circular list

Singly linked list: creation, traverse, insertion, deletion, other operations

Doubly linked list: creation, traverse, insertion, deletion, other operations
Class and its object representation.

We will represent an object using the following table.

<table>
<thead>
<tr>
<th>student name</th>
<th>address of the object</th>
<th>pointer to the previous object</th>
<th>pointer to the next object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tom</td>
<td></td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

We will call this object “Tom”.

```cpp
class Student {
    public:
    Student(string x); // address of the object
    string name; // student name
    Student* prev; // pointer to the previous object
    Student* next; // pointer to the next object
};
```

Initialize pointers to null.
A Running Example

A singly linked list of three objects: Tom, Jim and Sam.
Create a singly linked list.

```c
int main()
{
    Student Tom("Tom");
    Student Jim("Jim");
    Student Sam("Sam");
}
```

Note that all pointers are initialized to NULL.
Create a singly linked list.

```c
int main()
{
    Student Tom("Tom");
    Student Jim("Jim");
    Student Sam("Sam");

    Tom.next = &Jim;
}
```
Create a singly linked list.

```c
int main()
{
    Student Tom("Tom");
    Student Jim("Jim");
    Student Sam("Sam");
    Tom.next = &Jim;
    Jim.next = &Sam;
}
```

```c
Student* temp = &Tom;
```

Easier to traverse a list by pointer.

```c++
Student* temp = &Tom;

while (temp != NULL) {
    cout << temp->name;
    temp = temp->next;
}
```

```cpp
Student* temp = &Tom;

while (temp != NULL) {
    cout << temp->name;
    temp = temp->next;
}
```

```cpp
Student* temp = &Tom;

while (temp != NULL) {
    cout << temp->name;
    temp = temp->next;
}
```

Address of the current object.

Address of the next object.

```cpp
Student* temp = &Tom;

while (temp != NULL) {
    cout << temp->name;
    temp = temp->next;
}
```

```c++
Student* temp = &Tom;

while (temp != NULL) {
    cout << temp->name;
    temp = temp->next;
}
```
Traverse a singly linked list.

```cpp
Student* temp = &Tom;

while (temp != NULL) {
    cout << temp->name;
    temp = temp->next;
}
```

```cpp
Student* temp = &Tom;
while (temp != NULL) {
    cout << temp->name;
    temp = temp->next;
}
```

```cpp
Student* temp = &Tom;
while (temp != NULL) {
    cout << temp->name;
    temp = temp->next;
}
```
Traverse a singly linked list.

```c++
Student* temp = &Tom;
while (temp != NULL) {
    cout << temp->name;
    temp = temp->next;
}
```
Traverse a singly linked list.

```c++
Student* temp = &Tom;
while (temp != NULL) {
    cout << temp->name;
    temp = temp->next;
}
```

Terminate while loop.
Terminate traverse.
Can only travel forward, not backward, on singly linked list.

```c++
Student* temp = &Tom;

while (temp != NULL) {
    cout << temp->name;
    temp = temp->next;
}
```
[2++] Store head (or, its address) == Store the entire singly linked list.

```c
Student* head = &Tom;
```

To get data from any object, we can travel to it from head.
[2++] Store head (or, its address) == Store the entire singly linked list.

```
Student* head = &Tom;
cout << head->name;
cout << head->next->name;
cout << head->next->next->name;
```

To get data from any object, we can travel to it from head.

We will assume only head is stored for a singly linked list in the following.

In this case, if any link is broken, we will lose track of its pointing object and all the following objects in the list.

If we lose this link, e.g., reset Jim.next = null, we lose Sam.

We will assume only head is stored for a singly linked list in the following.

In this case, if any link is broken, we will lose track of its pointing object and all the following objects in the list.

If we lose this link, e.g., reset Tom.next = 023, we lose both Jim and Sam.
[3] Insert an object to the list.

Suppose we want to insert “Lin”, between Jim and Sam.

Because Lin is a newly declared object, we have its address.
Theoretically, we are supposed to…

We need to (i) link Jim to Lin and (ii) link Lin to Sam.
Q: Does the order of (i) and (ii) matter?

We need to (i) link Jim to Lin and (ii) link Lin to Sam.
Order 1: first (Lim, Sam), then (Jim, Lin).
Order 1: first (Lim, Sam), then (Jim, Lin).

Lin→next = head→next→next.
Order 1: first (Lim, Sam), then (Jim, Lin).

head→next→next = &Lin.
Order 2: first (Jim, Lin), then (Lin, Sam).
Order 2: first (Jim, Lin), then (Lin, Sam).

Previously, Sam’s address was stored here, but it’s overwritten by Lin’s address!

We lose Sam’s address!

head→next→next = &Lin.
Order 2: first (Jim, Lin), then (Lin, Sam).

Can't build the link. Don't know the address of Sam.
Suppose we want to add Lin to the head of the list, replacing Tom.
Theoretically, we are supposed to …

We need to (i) point head to Lin and (ii) link Lin to Tom.
Q: Does the order of (i) and (ii) matter?

We need to (i) point head to Lin and (ii) link Lin to Tom.
Order 1: first (Lin, Tom), then (head, Lin).
Order 1: first (Lin, Tom), then (head, Lin).

Lin
null
001

Tom
null
007

Jim
null
005

Sam
null
null

Lin.next = head.
Order 1: first (Lin, Tom), then (head, Lin).

head = &Lin.
Order 2: first (head, Lin), then (Lin, Tom).
Order 2: first (\texttt{head, Lin}), then (Lin, Tom).

head = &Lin.

Lose address of Tom.

head
\downarrow
023

Lin
null
null

007

Tom
null

007

Jim
null

005

Sam
null
null
Order 2: first (head, Lin), then (Lin, Tom).
[3++] Insert an object to the tail.

Suppose we want to add Lin to the tail of the list, replacing Sam.
Insert an object to the tail.

Just link Sam to Lin. (But first travel from head to Sam.)

head→next→next→next = &Lin.
Suppose we want to delete Jim from the list.
Delete an object from the singly linked list.

Just link Tom to Sam (bypass Jim).

Jim’s record will remain the memory. We just lose track of it. And its link to Sam is useless now.

\[\text{head} \rightarrow \text{next} = \text{head} \rightarrow \text{next} \rightarrow \text{next}.\]
[4+] Delete head from the singly linked list.

Suppose we want to delete Tom, and make Jim the head.
Delete head from the singly linked list.

Just update head to Jim.

head = head→next.
[4++] Delete tail from the singly linked list.

Suppose we want to delete Sam, and make Jim the tail.
[4] Delete tail from the singly linked list.

head

001

Tom

null

007

Just set Jim's next to null.

007

Jim

null

head→next→next = NULL.

005

Sam

null

null
Other tasks on singly linked list.

Search: look for Jim in the list and return its position or failure.

Size: get the size of the list.

Attach: attach a list to end of another.
Outline

Background: dynamic list of objects

Linked list: concepts, singly list, doubly list, circular list

Singly linked list: creation, traverse, insertion, deletion, other operations

Doubly linked list: creation, traverse, insertion, deletion, other operations
Class and its object representation.

We will represent an object using the following table.

We also assume all members are public.
A Running Example

A doubly linked list of three objects: Tom, Jim and Sam.

* Note there are links backwards.
[1] Create a doubly linked list.

Student Tom("Tom");
Student Jim("Jim");
Student Sam("Sam");
Create a doubly linked list.

```c
Student Tom("Tom");
Student Jim("Jim");
Student Sam("Sam");

Tom.next = &Jim;
Jim.next = &Sam;
```
Create a doubly linked list.

```java
Student Tom("Tom");
Student Jim("Jim");
Student Sam("Sam");

Tom.next = &Jim;
Jim.next = &Sam;
Jim.prev = &Tom;
Sam.prev = &Jim;
```
[2] Traverse a doubly linked list: forward!

Student* temp = &Tom;

```
temp

temp->next
```
Traverse a doubly linked list: forward!

```cpp
Student* temp = &Tom;

while (temp != NULL) {
    cout << temp->name;
    temp = temp->next;
}
```
Traverse a doubly linked list: forward!

```cpp
Student* temp = &Tom;

while (temp != NULL) {
    cout << temp->name;
    temp = temp->next;
}
```
Traverse a doubly linked list: forward!

```cpp
Student* temp = &Tom;
while (temp != NULL) {
    cout << temp->name;
    temp = temp->next;
}
```
Traverse a doubly linked list: forward!

```cpp
Student* temp = &Tom;
while (temp != NULL) {
    cout << temp->name;
    temp = temp->next;
}
```
Traverse a doubly linked list: forward!

Student* temp = &Tom;

while (temp != NULL) {
    cout << temp->name;
    temp = temp->next;
}
Traverse a doubly linked list: forward!

```cpp
Student* temp = &Tom;

while (temp != NULL) {
    cout << temp->name;
    temp = temp->next;
}
```
Traverse a doubly linked list: forward!

```cpp
Student* temp = &Tom;

while (temp != NULL) {
    cout << temp->name;
    temp = temp->next;
}
```
Traverse a doubly linked list: forward!

```cpp
Student* temp = &Tom;

while (temp != NULL) {
    cout << temp->name;
    temp = temp->next;
}
```
Traverse a doubly linked list: forward!

```cpp
Student* temp = &Tom;

while (temp != NULL) {
    cout << temp->name;
    temp = temp->next;
}
```
Traverse a doubly linked list: forward!

Student* temp = &Tom;

while (temp != NULL) {
    cout << temp->name;
    temp = temp->next;
}

Terminate while loop.
Terminate traverse.
[2+] Traverse a doubly linked list: backward!

```c
temp = &Sam;
```

Diagram:
```
    temp
     |
    ▼
    prev

Tom
null
007

Jim
001
005

Sam
007
null
```
Traverse a doubly linked list: backward!

temp = &Sam;

while (temp != NULL) {
    cout << temp->name;
    temp = temp->prev;
}
Traverse a doubly linked list: backward!

temp = &Sam;

while (temp != NULL) {
    cout << temp->name;
    temp = temp->prev;
}
Traverse a doubly linked list: backward!

```cpp
temp = &Sam;
while (temp != NULL) {
    cout << temp->name;
    temp = temp->prev;
}
```

Address of the current object.

Address of the previous object.
Traverse a doubly linked list: backward!

```c++
temp = &Sam;
while (temp != NULL) {
    cout << temp->name;
    temp = temp->prev;
}
```
Traverse a doubly linked list: backward!

temp = &Sam;

while (temp != NULL) {
    cout << temp->name;
    temp = temp->prev;
}
Traverse a doubly linked list: backward!

temp = &Sam;

while (temp != NULL) {
    cout << temp->name;
    temp = temp->prev;
}
Traverse a doubly linked list: backward!

temp = &Sam;
while (temp != NULL) {
    cout << temp->name;
    temp = temp->prev;
}
Traverse a doubly linked list: backward!

```
temp = &Sam;
while (temp != NULL) {
    cout << temp->name;
    temp = temp->prev;
}
```
Traverse a doubly linked list: backward!

temp = &Sam;

while (temp != NULL) {
    cout << temp->name;
    temp = temp->prev;
}
Traverse a doubly linked list: backward!

```cpp
temp = &Sam;
while (temp != NULL) {
    cout << temp->name;
    temp = temp->prev;
}
```

Terminate while loop. Terminate traverse.
To get any object, we can travel to it from any other object.
[3+] Assume a head pointer and a tail pointer in the following examples.

- Head pointer:
  - 001
  - Tom
  - null
  - 007

- Tail pointer:
  - 005
  - Sam
  - null

* You don’t have to practice this.
Q: why don’t we assume the two pointers on singly linked list?
[4] Insert an object to the list.

Suppose we want to insert Lin between Jim and Sam.
Theoretically, we are supposed to...

We need to (i) link Jim to Lin, (ii) link Lin to Sam, (iii) link Sam to Lin, (iv) link Lin to Jim.
Q: Does the order of (i), (ii), (iii) and (iv) matter?

We need to (i) link Jim to Lin, (ii) link Lin to Sam, (iii) link Sam to Lin, (iv) link Lin to Jim.
Order: (Jim, Lin), (Lin, Sam) [fails on singly], (Sam, Lin), (Lin, Jim)
Order: (Jim, Lin), (Lin, Sam) [fails on singly], (Sam, Lin), (Lin, Jim)
Order: (Jim, Lin), (Lin, Sam) [fails on singly], (Sam, Lin), (Lin, Jim)

Undoable on singly list!
Order: (Jim, Lin), (Lin, Sam) [fails on singly], (Sam, Lin), (Lin, Jim)

tail→prev = head→next→next
Order: (Jim, Lin), (Lin, Sam) [fails on singly], (Sam, Lin), (Lin, Jim)

tail→prev→prev = head→next

or, head→next→next→prev = head→next
Suppose we want to add Lin to the head of the list, replacing Tom.
We can (i) link Lin to Tom, (ii) link Tom to Lin, (iii) assign head to Lin.

* Order does not matter!
Example: (Lin, Tom), (Tom, Lin), (Lin, Head)
Example: (Lin, Tom), (Tom, Lin), (Lin, Head)

Lin.next = head
Example: (Lin, Tom), (Tom, Lin), (Lin, Head)

head→prev = &Lin
Example: (Lin, Tom), (Tom, Lin), (Lin, Head)

head = &Lin
Suppose we want to add Lin to the tail, replacing Sam.
Insert an object to the tail.

We need to (i) link Sam to Lin, (ii) link Lin to Sam, (iii) update tail.
Example: (Sam, Lin), (Lin, Sam), (Lin, Tail)
Example: (Sam, Lin), (Lin, Sam), (Lin, Tail)
Example: (Sam, Lin), (Lin, Sam), (Lin, Tail)

```
head
  →
001
  ↓
Tom
  ↓
null
  →
007
```

```
007
  ↓
Jim
  ↓
001
  ↓
005
```

```
005
  ↓
Sam
  ↓
023
  ↓
null
```

Lin.prev = tail→next
Example: (Sam, Lin), (Lin, Sam), (Lin, Tail)
[5] Delete an object from the doubly linked list.

Suppose we want to delete Jim from the list.
Theoretically, we are supposed to...

We need to (i) link Tom to Sam, (ii) link Sam to Tom. Then Jim will be automatically removed.
Example: (Tom, Sam), (Sam, Tom)
Example: (Tom, Sam), (Sam, Tom)

* link (Tom, Jim) is overwritten/removed.
Example: (Tom, Sam), (Sam, Tom)

* link (Sam, Jim) is overwritten/removed.

head
↓
001
↓
Tom null 005
↓
Jim 001 005
↓
tail
005
↓
Sam 001
↓
null

tail→prev = head
Although Jim remains in memory, we lost track of it and its pointers to Tom and Sam become useless.
Example: (Tom, Sam), (Sam, Tom)

Jim is removed from the list.
[5+] Delete head from the doubly linked list.

Suppose we want to delete Tom, and make Jim the head.
Theoretically, we are supposed to ...

We need to (i) assign head to Jim, (ii) set Jim.prev = null (no previous object).
Example: (i) (Jim, Head), (ii) Jim.prev=NULL.
Example: (i) *(Jim, Head)*, (ii) Jim.prev=NULL.

```
Tom
null
007

Jim
001
005

Sam
null
null
```

head = head→next
Example: (i) (Jim, Head), (ii) Jim.prev=NULL.

Tom is now lost in the memory…
But his service in this example will be remembered…

head→prev = null.  * Link back to Tom is removed.
Delete tail from the doubly linked list.

Suppose we want to delete Sam, and make Jim the tail.
Theoretically, we are supposed to …

We need to (i) assign tail to Jim, (ii) set Jim.next = null (no next object).
Example: (i) (Jim, Tail), (ii) Jim.next=NULL.
Example: (i) (Jim, Tail), (ii) Jim.next=NULL.

tail = tail→prev or, tail = head→next
Example: (i) (Jim, Tail), (ii) Jim.next=001.

001

Tom
null
007

007

Jim
null

001

head

001

tail

007

Sam
null
null

005

* Link to Sam is removed.

Sam is now lost in the memory…

tail→nextprev = 001.
Other tasks on doubly linked list.

Search: look for Jim in the list and return its position or failure.

Size: get the size of the list.

Attach: attach a list to another.
Attach a doubly linked list to another.

We need to build two links to attach the doubly lists.
Outline

Background: dynamic list of objects

Linked list: concepts, singly list, doubly list, circular list

Singly linked list: creation, traverse, insertion, deletion, other operations

Doubly linked list: creation, traverse, insertion, deletion, other operations

Discussions
Singly Linked List vs Doubly Linked List

Singly Linked List

head -> Tom -> null -> 007 -> Jim -> null -> 005 -> Sam -> null

Doubly Linked List

head -> Tom -> null -> 007 -> null -> 005 -> null

tail -> null -> 005
Generalized Linked List

In a generalized linked list, each object may contain a sublist.
Data Structure of a Generalized Linked List
Using generalized linked list to model a tree.