VE280 2021SU Final Review Part2

Contact: Yanjun Chen - Email: chen.yanjun@sjtu.edu.cn

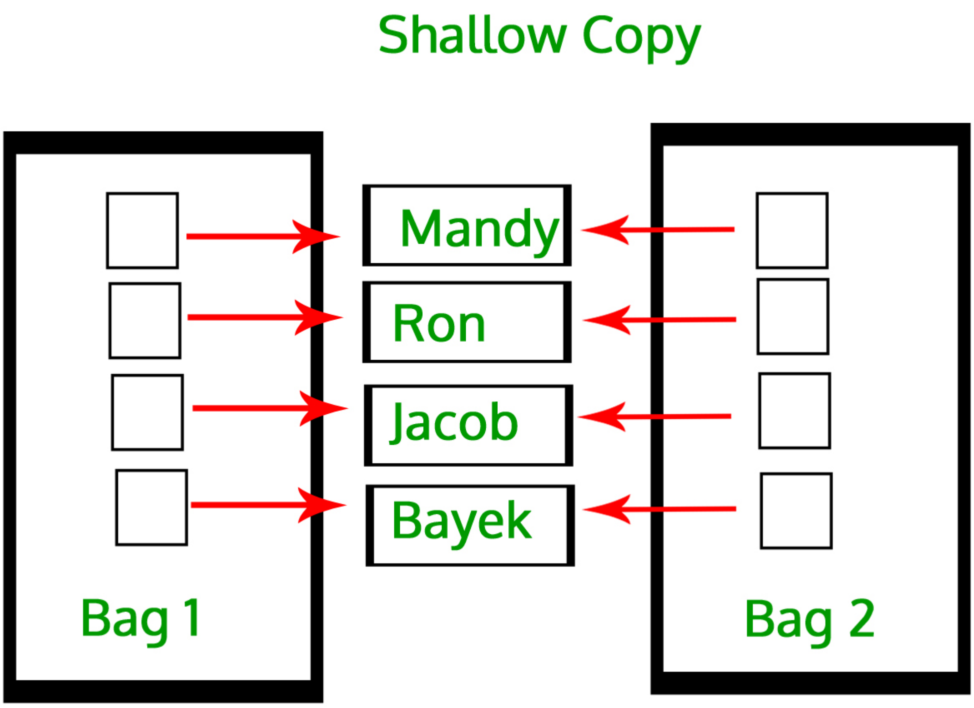
L18: Deep Copy

Shallow Copy & Deep Copy

Because C++ does not **know much about your class**, the default **copy constructor** and default **assignment operator** it provides use a copying method known as a member-wise copy, also known as a *shallow copy*.

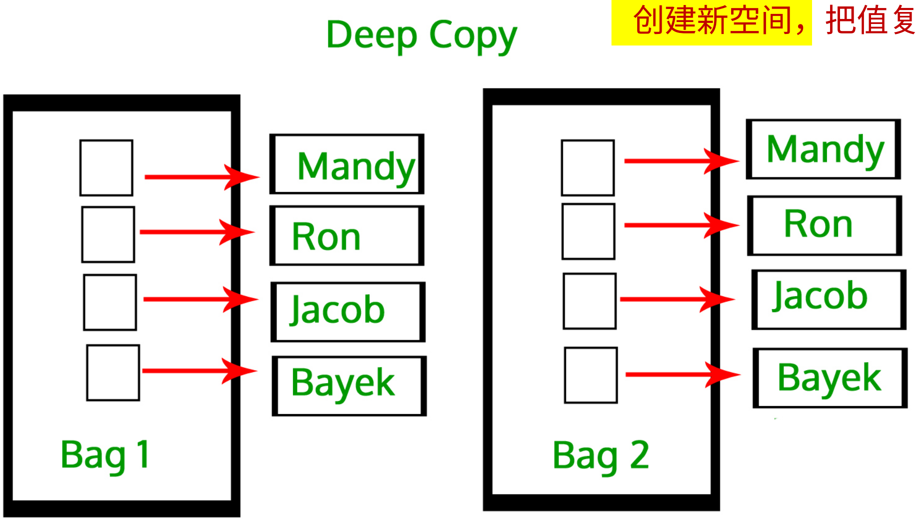
This works well if the fields are **values**, but may not be what you want for fields that point to **dynamically allocated memory**. **The pointer will be copied, but the memory it points to will not be copied:** the field in both the original object and the copy will then point to the same dynamically allocated memory, causing **dangling pointers**.

```cpp
#include <iostream>

using namespace std;
const int MAX_CAPACITY = 10;

class Bag
{
    string *items;

public:
    Bag();
    void insert(string str); // implementation omitted
};

Bag::Bag() : items(new string[MAX_CAPACITY])
{
}

int main()
{
    Bag bag1;
    bag1.insert("WOW");
    Bag bag2 = bag1;
}
```



Instead, a *deep copy* copies all fields, and makes copies of dynamically allocated memory pointed to by the fields.

创建新空间，把值复制进去



The Rule of the Big 3

These are 5 typical situations where resource management and ownership is critical. **You should never leave them undefined whenever dynamic allocation is involved.** Traditionally **constructor/destructor/copy assignment operator** forms **a rule of 3.** (NOT COVERED) Another 2 is about move semantics, a feature available in C++11, which is beyond the scope of this course. Learn more about them in EECS 381.

If you want to use the version synthesized by the compiler, you can use `default` :

```
Type(const Type& type) = default;
Type& operator=(Type&& type) = default;
```

Usually, we would need to implement some private helper functions `removeAll()` and `copyFrom()`, and use them in the big 3.

Consider the `Dlist` example which you may have encountered in *p5*.

- A destructor

```
template <class T>
Dlist<T>::~Dlist() {
    removeAll();
}
```

- A copy constructor

```
template <class T>
Dlist<T>::Dlist(const Dlist &l): first(nullptr), last(nullptr) { // DO NOT
FORGET about initialization
  copyFrom(l);
}
```

- An assignment operator

```
template <class T>
Dlist<T> &Dlist<T>::operator=(const Dlist &l) {
    if (this != &l) {
        removeAll();
        copyFrom(l);
    }
    return *this;
}
```

L19: Dynamic Resizing

In many applications, we do not know **the length of a container in advance**, and may need to grow the size of it at runtime. In this kind of situation, we may need dynamic resizing.

If the implementation of the list is a dynamically allocated array, we need the following steps to grow it:

- Make a new array with the desired size. For example,

  ```
  int *tmp = new int[newSize];
  ```

- Copy the elements from the original array to the new array iteratively. Suppose the original array is `arr` with size `size`.

```
for (int i = 0; i < size; i++){
    tmp[i] = arr[i];
}
```

- Delete the original array pointer(DO NOT FORGET THIS) and replace it with the new array.

```
delete [] arr;
arr = tmp;
```

- Make sure all invariants are fixed. In this example, we need to fix the `size = <Number of Elements>` invariant

```
size = newSize;
```

Different choice of newly allocated capacity for dynamic resizing can be:

- `size + 1`: This approach is memory-efficient, but is not time-efficient. Inserting `N` elements from capacity 1 needs `N(N-1)/2` number of copies, and `insert` is of amortized complexity `O(N)`.
- `2*size`: Much more efficient than `size+1`. The number of copies for inserting `N` elements becomes smaller than `2N`, and `insert` is of amortized complexity `O(1)`.

(OPTIONAL) Learn more about amortized complexity in VE 281.

L20: Linked List

Expandable arrays are only one way to implement container that can grow and shrink at runtime. To enlarge a list implemented by linked list, you can simply add a node at the end of the linked list.

Review what you have implemented for *p5* thoroughly.

Single-Ended & Double-Ended

Linked lists could be either single-ended or double-ended, depending on the the number of node pointers in the container.

In a single-ended list, we only need to maintain `first`.

```
class IntList {
    node *first;
    //...
};
```

In a double-ended list, we introduce `last`.

```
class IntList {
    node *first;
    node *last;
    //...
};
```

Especially, when handling a singly ended list, you need to be concerned about the **boundary situation** where

- size = 0: `first = nullptr`

In a double-ended list, the `last` makes it slightly more complicated:

- size = 0: `first = last = nullptr`
- size = 1: `first = last`.

Advantage - efficient insertion at last (O(n) -> O(1))

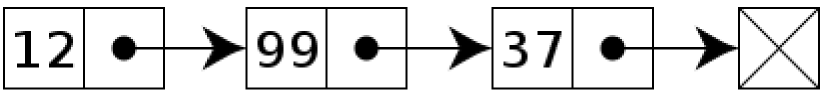
Disadvantage - takes up more memory

Singly-Linked & Doubly-Linked

Linked lists could be either single linked or doubly linked, depending on the the number of directional pointers in `node`.

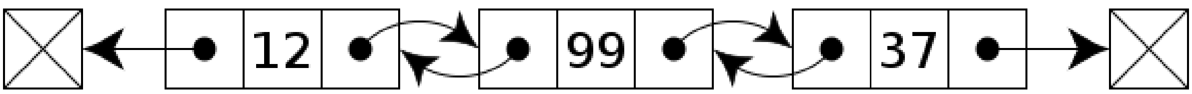
In a singly linked list, we only need a `next`.

```
struct node {
    node *next;
    int value;
};
```



In a doubly linked list, we need also a `prev`.

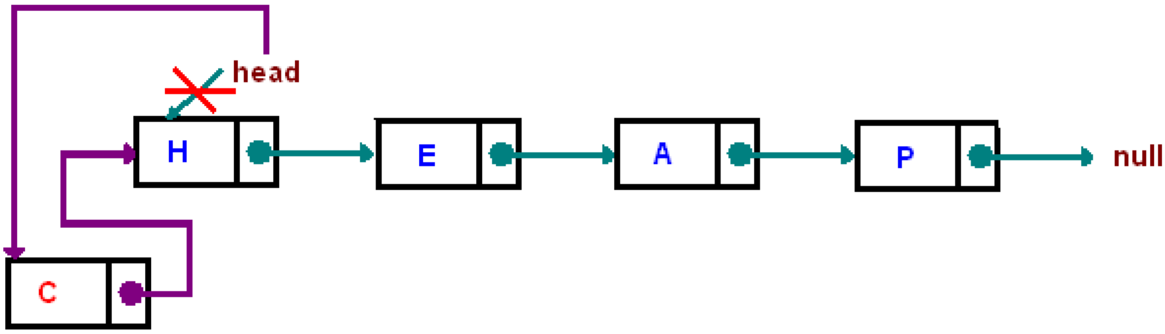
```
struct node {
    node *next;
    node *prev;
    int value;
};
```



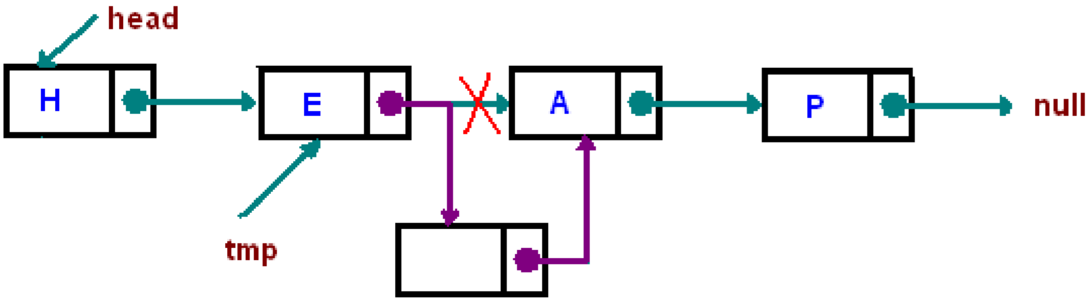
Linked List Methods

Insertion (at any position)

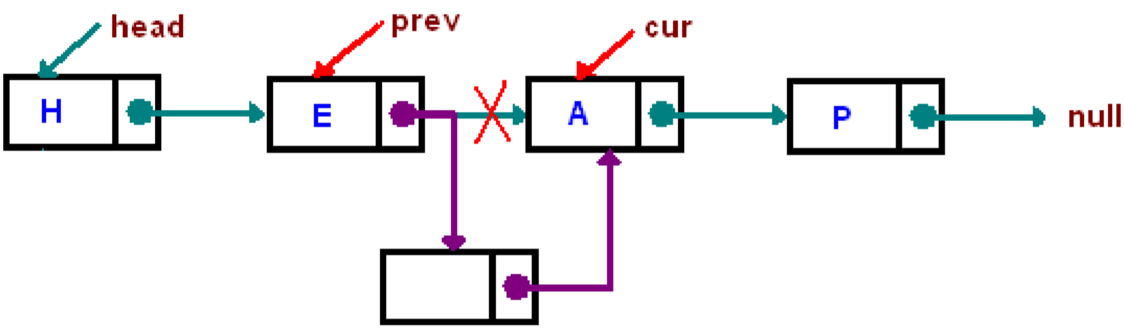
Insertion at ends (either first or last):



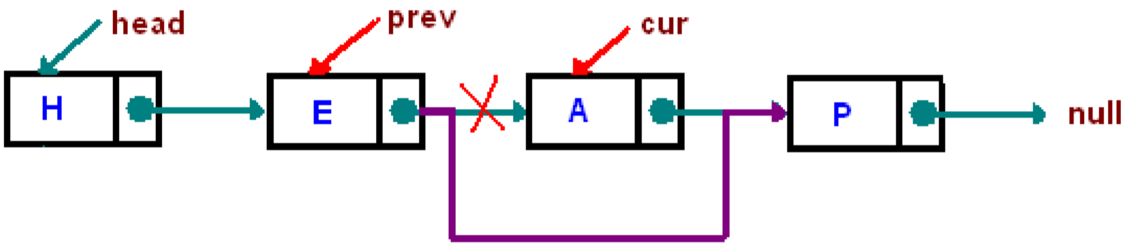
Insertion after:



Insertion before:



Deletion (at any position)



Advantage - efficient deletion at any position (not only last) & insertBefore (O(n) -> O(1))

Disadvantage - takes up more memory