List Implementations

Sandra Batista, Mark Redekopp, and David Kempe

List ADT Operations

Operation	Description	Input(s)	Output(s
insert	Add a new value at a particular location shifting others back	Index	
remove	Remove value at the given location	Index	Value at location
get	Get value at given location	Index	Value at location
set	Changes the value at a given location	Index & Value	
empty	Returns true if there are no values in the list		bool
size	Returns the number of values in the list		Size_t
push_back	Add a new value to the end of the list	Value	
find	Return the location of a given value	Value	Index

Linear Data Structures to Implement List ADT

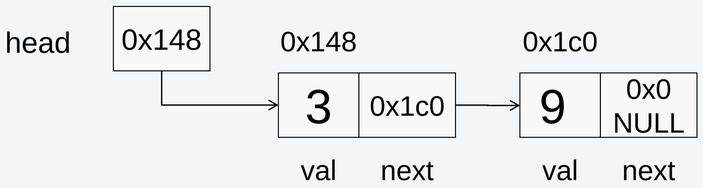
Linked Implementations

Allocate each item separately

Random access (get the i-th element) is O(___)

Adding new items never requires others to move

Memory overhead due to pointers



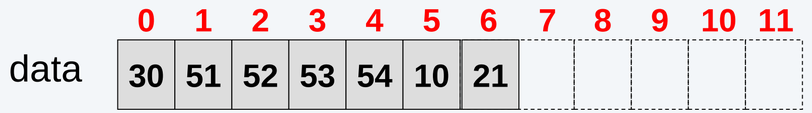
Array-based Implementations

Allocate a block of memory to hold many items

Random access (get the i-th element) is O(___)

Adding new items may require others to shift positions

Memory overhead due to potentially larger block of memory with unused locations



Implementation Options

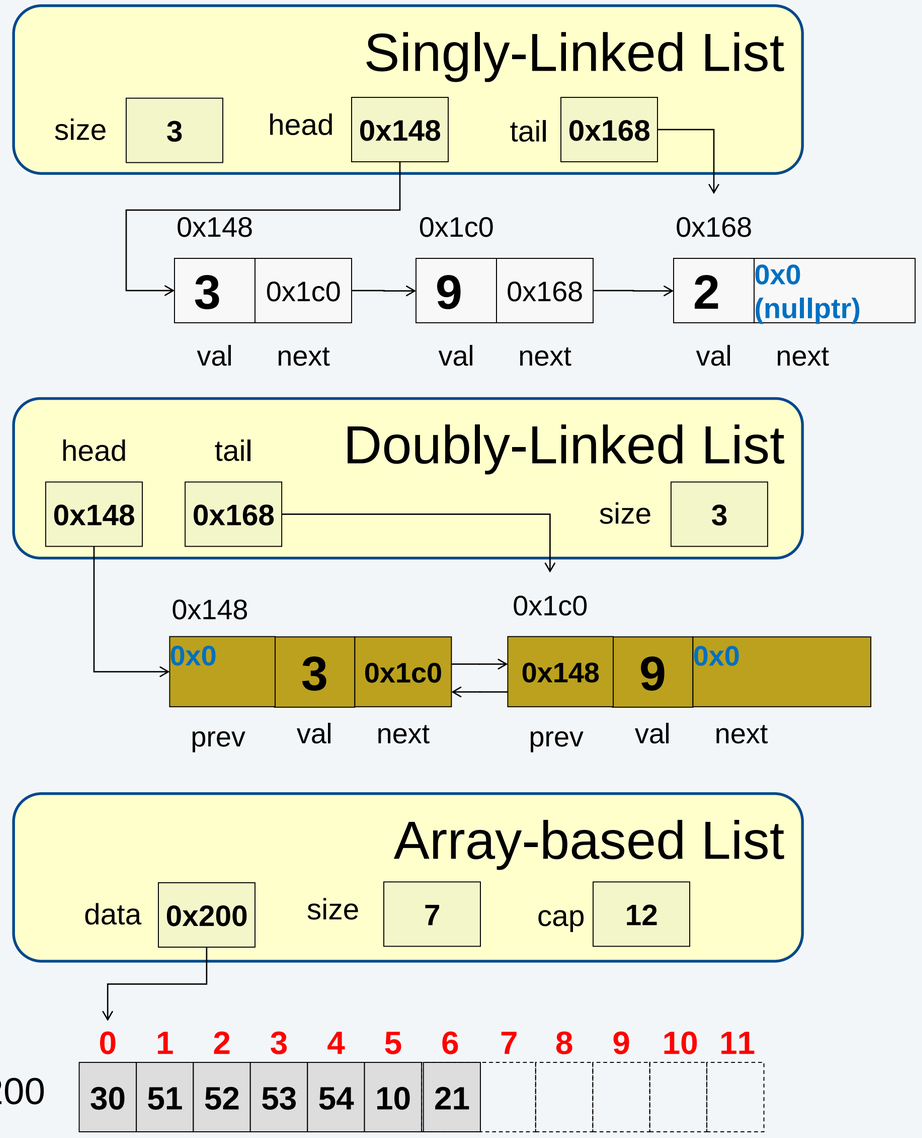
Singly-Linked List

- With or without tail pointer

Doubly-Linked List

- With or without tail pointer

Array-based List



Summary of Linked List Implementations

Operation vs Implementation for Edges	Push_front	Pop_front	Push_back	Pop_back	Memory Overhead Per Item
Singly linked-list w/ head ptr ONLY	Θ(1)	Θ(1)	Θ(n)	Θ(n)	1 pointer (next)
Singly linked-list w/ head and tail ptr	Θ(1)	Θ(1)	Θ(1)	Θ(n)	1 pointer (next)
Doubly linked-list w/ head and tail ptr	Θ(1)	Θ(1)	Θ(1)	Θ(1)	2 pointers (prev + next)

- What is worst-case runtime of get(i)? Θ(i)
- What is worst-case runtime of insert(i, value)? Θ(i)
- What is worst-case runtime of remove(i)? Θ(i)

Array List Runtime Analysis

What is worst-case runtime of set(i, value)?

What is worst-case runtime of get(i)?

What is worst-case runtime of pushback(value)?

What is worst-case runtime of insert(i, value)?

What is worst-case runtime of remove(i)?

Shared_ptr

- **std::shared_ptr<type> is for *shared ownership* of memory address.**
- **A reference counts keeps track of how many std::shared_ptr own the same memory address**

- Shared_ptrs *can be copied and assigned .*

- **When a shared_ptr is destroyed, its reference count is decremented.**
- **Once reference count is zero,** shared_ptr automatically destroys object contained in the raw pointer using delete by default

Reference Count = 4

Shared_ptr Declaration

Preferably instantiate shared_ptr<type> using make_shared

Use the shared_ptr as you would a raw pointer on the object.

Use function use_count to check the number of references

Use as you would raw pointer to the type

```
#include <iostream>
#include <string>
#include <memory>
using namespace std;
int main(){
     shared_ptr<string> sp1;  // null smart pointer
     shared_ptr<string> sp2 = make_shared<string>("Hello ");
     shared_ptr<string> sp3(new string("world!"))
     shared_ptr<string> sp4(sp3);         // use to initialize shared pointer increments reference count;

     cout << sp1.use_count() << endl; // 0
     cout << sp2.use_count() << endl; // 1
     cout << sp3.use_count() << endl; // 2
     cout << sp4.use_count() << endl; // 2

     cout << *sp2 << *sp4 << endl;  // prints "Hello world"
     sp3 = sp2;    // Smart pointers can be assigned!! increments RHS reference count and decrements LHS reference count
     cout << sp2.use_count() << endl; // 2
     cout << sp3.use_count() << endl; // 2
     cout << sp4.use_count() << endl; // 1
     cout << *sp2 << *sp3 << endl;  // prints "Hello Hello"
     cout << *sp2 << *sp4<< endl;  // prints "Hello world!"
}    //When smart pointers go out of scope, they are destroyed and reference count decremented.
```

Incrementing and decrementing reference counts

Reference counts are incremented when 1) initializes another shared pointer 2) RHS of assignment and 3) pass to or returned from function by value

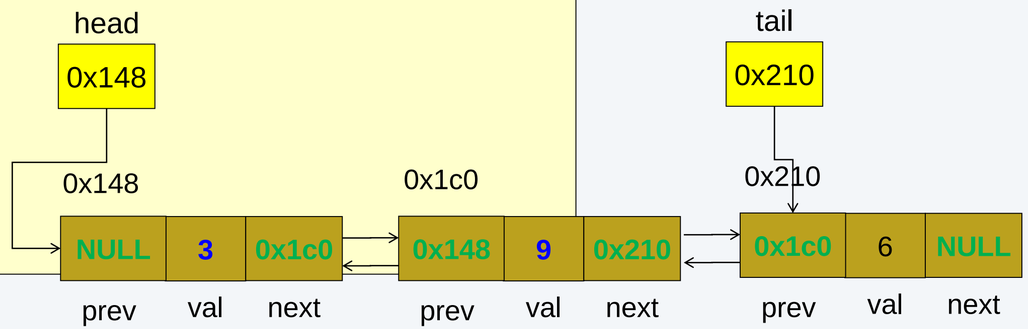
Reference counts are decremented 1) LHS of assignment 2) when destroyed such as when goes out of scope

When reference count is 0, the shared_ptr will destroy object to which it points freeing memory (using delete)

```
// include <iostream>, <string>, <memory> and using namespace std;
int main(){
    shared_ptr<string> p1 = make_shared<string>(“Hello!”);
    cout << p1.use_count() << endl; // 1
    pass_to_function1(p1);          // 2
    cout << p1.use_count() << endl; // 1

    shared_ptr<string> p2 = pass_to_function2(p1);
    cout << p2.use_count() << endl; // 2
    cout << p2.use_count() << endl; // 2
}

void pass_to_function1(shared_ptr<string> p){
          cout << p.use_count() << endl;
          return;
}

shared_ptr<string> pass_to_function2(shared_ptr<string> p){
          cout << p.use_count << endl;  // 2
          shared_ptr<string> temp = p;
          cout << p.use_count() << endl; // 3
          return p;
}
```

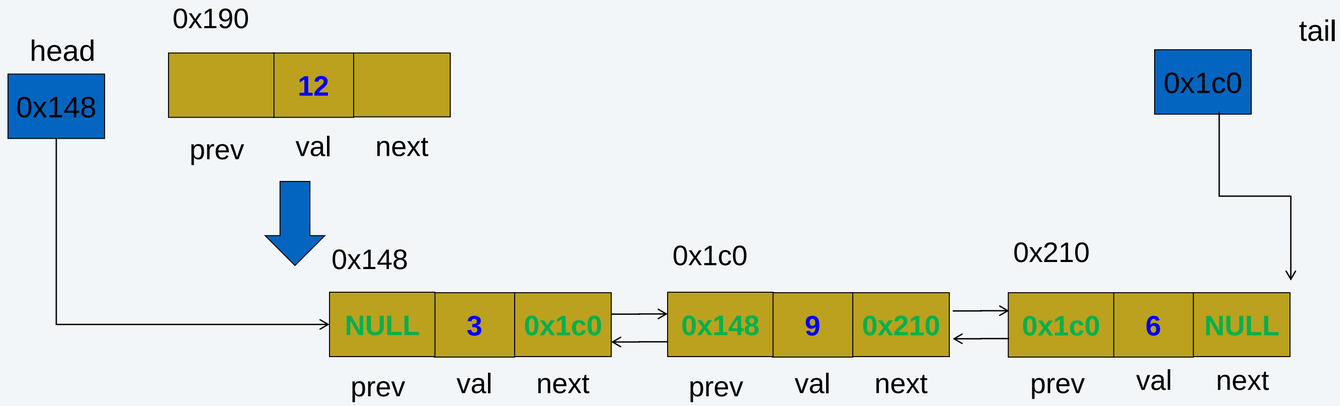
Doubly-Linked Lists

```
#include <memory>

class DoubleLL{
       struct Item {
            int value;
            std::shared_ptr<Item>  next;
            std::shared_ptr<Item>  prev;
            Item():next(nullptr), prev(nullptr) {}
            Item(Item const & other) = default;
             Item(int itemValue): value(itemValue), next(nullptr),prev(nullptr) {}
            ~Item(){}
       };
        size_t count = 0;
        std::shared_ptr<Item> head;
        std::shared_ptr<Item> tail;
        void remove(std::shared_ptr<Item> p);

        public:
        DoubleLL(): head(nullptr), tail(nullptr){}
        DoubleLL(DoubleLL const & other);
        DoubleLL& operator=(DoubleLL const & other);
        ~DoubleLL();
        int get(size_t index) const;
        size_t size() const;
        bool empty() const;
        void push_back(int value);
        void set(size_t index, int value);
        void remove(size_t index);
        void push_front(int value);
};
```



Doubly-Linked Push Front

```
void DoubleLL::push_front(int value) {
     if (count == 0){
          head = std::make_shared<Item>(value);
          tail = head;
     } else {
         std::shared_ptr<Item> tmp = head;
         head = std::make_shared<Item>(value);
         head->next = tmp;
         if (head->next) head->next->prev = head;
     }
     count++;

}
```



Doubly-Linked List Remove

Given pointer to an item, what may need to be updated check list?

- Previous item's next field?
- Next item's previous field ?
- head?
- tail?
- Use reference count (use_count) to help you and valgrind to check