DATA STRUCTURES AND ALGORITHMS

Lecture 05

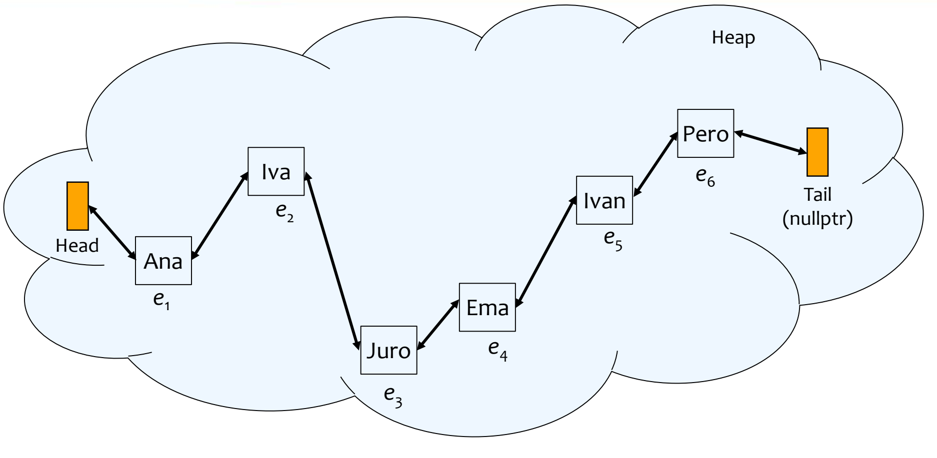
Learning outcome 2

LINKED LIST

Linked list as ADT

- An ADT linked list has the following features:
  - An element is usually called a node
  - The nodes in the memory are not placed one after the other
    - We must not use pointer arithmetic
    - We cannot directly access the *i*-th element
  - A node knows where the next (and maybe previous) node is
    - It is usually one or two pointers
    - Sometimes, due to compactness, the link to the previous node is dropped
  - Modifying the list anywhere is effective
    - The list efficiently grows and shrinks during program execution by adding or removing nodes

Graphical representation



- Each node is dynamically allocated from the heap and deallocated when it is no longer needed
- Each node contains one or two pointers in addition to the data

CONCRETE LINKED LISTS

Concrete linked lists

- C ++ comes with two implementations:
  - Generic class `list<T>`
  - Generic class `forward_list<T>`
- The classes are similar, with the following main differences:
  - `forward_list<T>` consumes less memory per node because it contains only a pointer to the next node
    - Important for resource-constrained environments (like Arduino)
    - Allows only iteration from start to end
  - `list<T>` contains a pointer to the previous element also
    - It also allows iteration from end to start
- Everything we say in the following slides applies to both lists (differences will be highlighted)

Creating and destroying linked lists (1/2)

- There are six basic ways to create a linked list:
  - `list<int> one;`
    - Creates an empty linked list (*default*)
  - `list<int> two(n);`
    - Creates a linked list of *n* elements initialized to the default value (*fill*)
  - `list<int> three(n, val);`
    - Creates a linked list of *n* elements, each a copy of *val* (*fill*)
  - `list<int> four(iter1, iter2);`
    - Creates a linked list by copying elements from a given range (*range*)

Creating and destroying linked lists (2/2)

  - `list<int> five(three);`
    - Creates a linked list by copying all items from the provided linked list (*copy*)
  - `list<int> six({ 11, 22, 33 });`
    - Creates a list by copying all elements from the initialization list (*initializer list*)
- The linked list is automatically destroyed by ending the function in which it was declared
  - If a linked list stores objects, a destructor is called on each
- `operator=` copies the contents of one linked list to another
  - Previous contents of left-side linked list are destroyed

Accessing linked list elements

- There is no `operator[]` nor method `at()` on linked list
- The linked list offers only the following ways to access the elements:
  - `l.front()` returns a reference to the first element
    - If the linked list is empty, the behavior is not defined
  - `l.back()` returns a reference to the last element
    - If the linked list is empty, the behavior is not defined
    - Doesn't exist on `forward_list<T>`
  - Using iterators

Linked list iterators (1/2)

- The most important iterators are:
  - `list<T>::iterator` is an iterator whose ++ moves towards the end
  - `list<T>::reverse_iterator` is an iterator whose ++ moves towards the start
    - Doesn't exist on `forward_list<T>`

Linked list iterators (2/2)

- The following methods return iterators:
  - `l.begin()` – returns the iterator pointing to the first element
  - `l.end()` – returns the iterator pointing to the first element after the end
  - `l.rbegin()` – returns the reverse iterator to the last element
    - Doesn't exist on `forward_list<T>`
  - `l.rend()` – returns the reverse iterator to the element in front of the first
    - Doesn't exist on `forward_list<T>`
  - `l.before_begin()` – returns the iterator to the element before the first
    - Doesn't exist on `list<T>`
    - Used in `emplace_after()`, `insert_after()` and `erase_after()`

A little more about iterators (1/2)

- The following code works correctly:

```
vector<int> v1({ 11, 22, 33, 44, 55 });
vector<int> v2(v1.begin() + 3, v1.end());
```

- What then is the problem with the following code:

```
list<int> l1({ 11, 22, 33, 44, 55 });
list<int> l2(l1.begin() + 3, l1.end());
```

- Answer: The linked list elements are not placed one after the other in memory so we cannot write + 3
  - Remember: all the iterator provides us are operations `++it` and `*it`

A little more about iterators (2/2)

- One possible solution to a problem:

```
list<int> l1({ 11, 22, 33, 44, 55 });
auto it1 = l1.begin();
for (int i = 0; i < 3; i++) {
    ++it1;
}
list<int> l2(it1, l1.end());
```

- Another, a bit simpler solution:

```
list<int> l1({ 11, 22, 33, 44, 55 });
auto it1 = l1.begin();
advance(it1, 3);
list<int> l2(it1, l1.end());
```


Linked list size

- The linked list has no notion of capacity
  - Memory for a node is allocated when needed
- The linked list has its size
  - `l.size()` returns the number of elements placed in the linked list
    - Due to performance does not exist on `forward_list<T>`
    - „Not providing size() is more consistent with the goal of zero overhead ... Maintaining a count doubles the size of a forward_list object (one word for the list head and one for the count), and it slows down every operation that changes the number of nodes."

Taken from: www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2543.htm

Manually resizing the linked list

- We can also explicitly change the size:
  - `l.resize(n, val);`
    - Changes the size of the linked list to exactly *n* elements
    - If *n* is less than the current size, the end elements are discarded
      - Objects are destroyed
    - If *n* is larger than the current size, elements are added at the end
      - Optionally, we can tell what the value *val* of added elements is

Linked list modifiers (1/4)

- The linked list offers the following modifiers:
  - `l.assign()` is similar to constructors

```
l1.assign(7, 100);
l2.assign(it1, it2);
l3.assign({ 11, 22, 33 });
```

    - All elements previously contained in the linked list are destroyed
  - `l.push_front(val)`
    - Add a copy of *val* to the beginning
  - `l.emplace_front(arg1, arg2, …)`
    - Constructs the object at the beginning
  - `l.pop_front()`
    - Deletes and destroys the first element from the beginning

Linked list modifiers (2/4)

- `l.clear()` completely empties the linked list and destroys all elements
- `list<T>` additionally offers the following six exclusive methods:
  - `l.insert()` inserts one or more elements into a given position:
    - First parameter is a position, other are like in constructors:

```
l.insert(it, 99);
l.insert(it, 10, 99);
l.insert(it, { 11, 22, 33 });
```

  - `l.emplace(it, arg1, arg2, …)`
    - Constructs an object at a given position
  - `l.erase()` removes one or more elements:

```
l.erase(it);
l.erase(it1, it2);
```


Linked list modifiers (3/4)

- `l.push_back(val)`
  - Adds a copy of the val to the end
- `l.emplace_back(arg1, arg2, …)`
  - Constructs an object at the end
- `l.pop_back()`
  - Deletes and destroys the first element from the end

Linked list modifiers (4/4)

- `forward_list<T>` additionally offers the following three exclusive methods:
  - `l.insert_after(it, val)`
    - Adds a copy of the *val* to the position after *it*
  - `l.emplace_after(it, arg1, arg2, …)`
    - Constructs an object in the position after *it*
  - `l.erase_after(it)`
    - Deletes and destroys the element after *it*

ALGEBRA

Example

```
list<int> l = { 11, 22, 33, 44, 55 };
auto it1 = l.begin();
advance(it1, 3);
l.insert(it1, 999);

// Display…

forward_list<int> fl = { 11, 22, 33, 44, 55 };
auto it2 = fl.begin();
advance(it2, 3);
fl.insert_after(it2, 999);

// Display…
```


ALGEBRA

Other important methods

- `l.empty()` returns whether the linked list is empty or not
- `l.remove(val)` removes and destroys all elements that have a value equal to *val*
- `l.remove_if(predicate)` removes and destroys all elements for which the function `predicate` returns `true`
    - The function receives a value from the list and returns `true/false`
- `l.reverse()` rearranges the elements from the end to the beginning

Problem

- Let’s write our own implementation of a single-linked list of integer that allows:
    - Making an empty list
    - Inserting at the beginning of the list
    - * Creating iterators that allow access to elements

How it should work

```
int main() {
    MyList l;
    l.push_front(11);
    l.push_front(22);
    l.push_front(33);

    for (MyList::iterator it = l.begin(); it != l.end(); ++it) {
        cout << *it << endl;
    }

    return 0;
}
```


MyList.h

```
struct Node {
    int data;
    Node* next;
};

class MyList {
private:
    Node* front;
public:
    MyList();
    ~MyList();
    void push_front(int number);

    class iterator {
    private:
        Node* curr;
    public:
        iterator(Node* c);
        iterator& operator++();
        bool operator!=(const iterator& rhs) const;
        int& operator*() const;
    };

    iterator begin();
    iterator end();
};
```

MyList.cpp (1/2)

```
MyList::MyList() {
    front = nullptr;
}

MyList::~MyList() {
    Node* curr = front;
    while (curr != nullptr) {
        Node* tmp = curr->next;
        delete curr;
        curr = tmp;
    }
}

void MyList::push_front(int number) {
    Node* c = new Node;
    c->data = number;

    c->next = front;

    front = c;
}
```


MyList.cpp (2/2)

```
MyList::iterator::iterator(Node* c) {
    curr = c;
}

MyList::iterator& MyList::iterator::operator++() {
    curr = curr->next;
    return *this;
}

int& MyList::iterator::operator*() const {
    return curr->data;
}

bool MyList::iterator::operator!=(const iterator& rhs) const {
    return curr != rhs.curr;
}

MyList::iterator MyList::begin() {
    return MyList::iterator(front);
}

MyList::iterator MyList::end() {
    return MyList::iterator(nullptr);
}
```

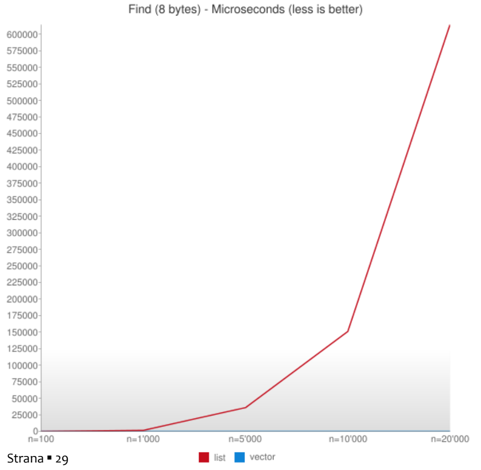
LINKED LIST PERFORMANCE

The complexity of some operations

Method	Complexity	Method	Complexity
list<T> l;	O(1)	l.erase(iterator);	O(1)
list<T> l(it1, it2);	O(n)	l.erase(begin, end);	O(1)
l.size();	O(1)	l.remove(value);	O(n)
l.empty();	O(1)	l.remove_if(test);	O(n)
l.begin();	O(1)	l.reverse();	O(n)
l.end();	O(1)	l.sort();	O(n log n)
l.front();	O(1)	l.sort(comparison);	O(n log n)
l.back();	O(1)	l.merge(l2);	O(n)
l.push_front(value);	O(1)		
l.push_back(value);	O(1)		
l.insert(iterator, value);	O(1)		
l.pop_front();	O(1)		
l.pop_back();	O(1)		

Searching for a random value

- Each container contains unsorted numbers [0, N]
  - After that, each of the numbers is searched linearly

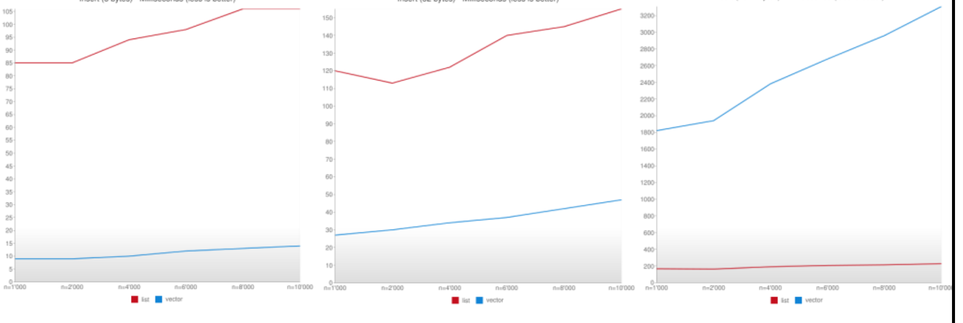


- Reason: cache!
  - Cache is a few orders of magnitude faster than RAM
  - Because vector elements are places one after another, retrieving the first one retrieves a huge amount of other elements as well
  - With list, CPU most of the time waits for the RAM => cache transfer

Taken from: dzone.com/articles/c-benchmark-%E2%80%93-stdvector-vs

Inserting to a random position

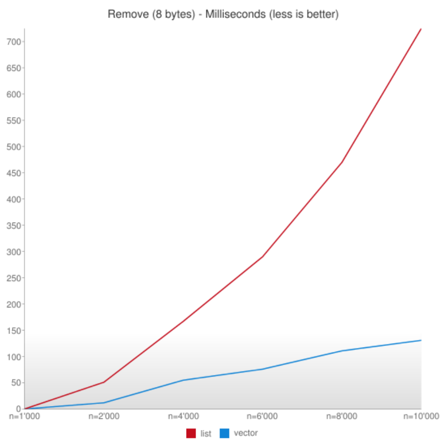
- Inserting 1000 values to random positions

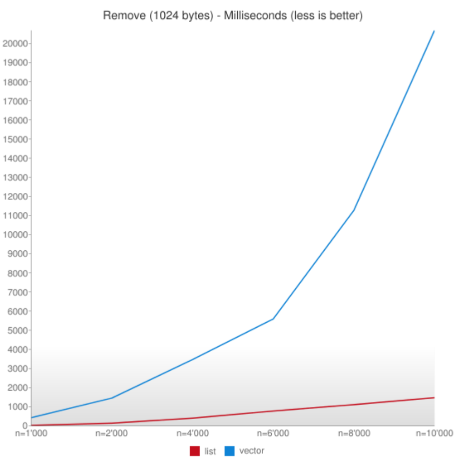


- Finding a position in a list is slower than moving a lot of small elements to the right (if we know the position, the list is faster)
- As the element sizes increase, the performance of the vector decreases drastically, and list's remain approximately the same

Deleting from a random position

- In theory equal to insertion





- Equal to the explanation for inserting

Strana ▪ 31

When to use which container

- Theoretical advices:
  - For linear search => vector
  - To insert / delete small data => vector
  - To insert / delete large data at end => vector
  - To insert / delete large data at the beginning => list
- Practical advices:
  - Start with the vector
  - Test your performance
  - If they are insufficient, try a list
- Bjarne Stroustrup: „Vectors are always better than lists”

Strana ▪ 33