Problem Set III

Huy Quang Lai
132000359

Texas A&M University

27 September 2022

List ADT

Problem 1

A. Pseudocode

```
printLots(L, P):
    for i = 0; i < size(P); ++i:
        let x = access(P, i)
        let y = access(L, x)
        print y
    end
end
```

B. Assuming both L and P are ArrayLists, the run time should be $O(N)$. Since accessing ArrayLists are $O(1)$, traversing through each element of P would contribute more to runtime than accessing the elements of L.

Problem 2

A. Pseudocode

```
intersection(A, B):
    let intersect = []
    for i = 0; i < size(A); ++i:
        let x = access(A, i)
        for j = 0; j < size(B); ++j:
            let y = access(B, j)
            if (x == y)
                insert(intersect, x)
            end
        end
    end
    return intersect
end
```

B. Assuming A and B are both ArrayLists, this algorithm should have a runtime of $O(N^2)$. Since accessing arrays are $O(1)$, traversing through each element of both arrays would contribute more to runtime. Additionally, inserting into an arraylist is also $O(N)$. Because of this, the double nested for loop will dominate the other algorithms.

Problem 3

A. Code

```
#include <numeric>
#include <vector>
#include <stdexcept>

using std::vector, std::begin, std::end;
using std::iota;
using std::invalid_argument;

int potato(size_t M, size_t N) {
    if (!N)
        throw std::invalid_argument("0 people \u2639");

    vector<int> people(N);
    iota(begin(people), end(people), 1);
    size_t index = 0;
    while (people.size() != 1) {
        index += M;
        index %= people.size();
        people.erase(people.begin() + index);
    }
    return people[0];
}
```

B. $O(\log_{M+1} N)$

C. If $M = 1$ then $O(\log_2 N)$. Run time graph is on the next page.

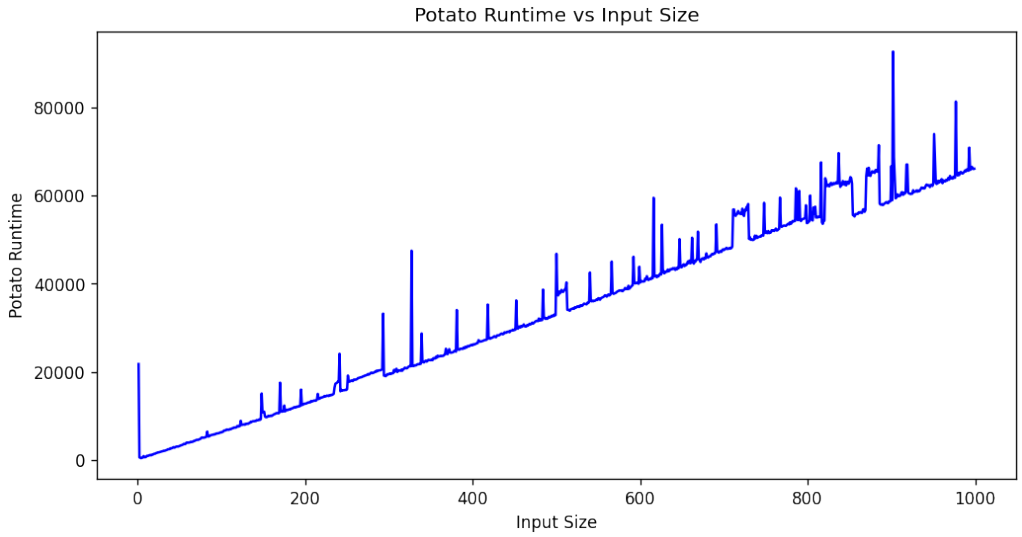

Figure 1: Potato Runtime

LinkedList Review

Problem 1

A. return the size of the linked list

```
template <typename T>
size_t List<T>::size() const {
    size_t size;
    Node* node = this->_head;
    while (node) {
        node = node->_next;
        ++size;
    }

    return size;
}
```

B. Print

```
#include <iostream>

using std::ostream, std::cout;

template <typename T>
void List<T>::print(ostream& os=cout) const {
    Node* node = this->_head;
    while (node) {
        os << node->_value << " ";
        node = node->_next;
    }
}
```

C. Contains

```
template <typename T>
bool List<T>::contains(const T& value) const {
    Node* node = this->_head;
    while (node) {
        if (node->_value == value) { return true; }
        node = node->_next;
    } return false;
}
```

D. Insert

```
template <typename T>
void List<T>::insert(const T& value) {
    if (this->contains(value)) { return; }

    Node* new_node = new Node(value);
    if (value < this->_head) {
        Node* temp = this->_head;
        this->_head = new_node;
        this->_head->_next = temp;
        return;
    }

    Node* node = this->_head;
    while (node->_value < value) {
        new_node->_next = node;
        node = node->_next;
    }

    new_node->_next->_next = new_node;
    new_node->_next = node;
}
```

E. Remove

```
void remove(const Object& value) {
    if (this->_head->_value == value) {
        Node* temp = this->_head;
        this->_head = this->_head->_next;
        delete temp;
        return;
    }


    Node* node = this->_head, * prev = nullptr;
    while (node) {
        prev = node;
        node = node->_next;
        if (node->_value == value) {
            prev->_next = node->_next->_next;
            delete node;
            return;
        }
    }
}
```

Problem 2

A. Print in reverse

```
#include <stack>
#include <iostream>

using std::cout, std::ostream;
using std::stack;

template <typename T>
void List<T>::printReverse(ostream& os=cout) const {
    stack<T> st;
    Node* node = this->_head;
    while (node) {
        st.push(node->_value);
        node = node->_next;
    }

    while (!st.empty()) {
        os << st.top() << " ";
        st.pop();
    }
}
```

B. This algorithm can be as a `const` since this method does not modify any attributes of the Singly Linked List.

Problem 3

```
// We assume that index < this->_size - 1;
void remove(size_t index) {
    Node* node = this->nodeAt(index);
    Node* temp = node->_next;
    node->_next = temp->_next;
    delete temp;
}
```

Stack ADT

Problem 1

Assuming the underlining storage type for the Stack is a Doubly Linked List with a `_size`, `_head`, and `_tail`.

With these assumptions, `push`, `pop`, and `top` can be written as follows.

```
template <typename Object>
void Stack<Object>::push(const Object& value) {
    Node* node = new Node(value);
    if (!this->_head) {
        this->_head = node;
        this->_tail = node;
    } else {
        this->_head->_prev = node;
        node->_next = this->_head;
        this->_head = this->_head->_prev;
    }
    ++this->_size;
}

template <typename Object>
void Stack<Object>::pop() {
    if (!this->_head)
        throw out_of_range("Stack underflow");
    if (this->_size == 1)
        delete this->_head;
        this->_head = nullptr;
    else {
        this->_head = this->_head->_next;
        delete this->_head->_prev;
    }
    --this->_size;
}
```

```
template <typename Object>
const Object& Stack<Object>::top() {
    if (!this->_head)
        throw out_of_range("Stack is empty");
    return this->_head->_value;
}
```

With these algorithms, `push`, `pop`, and `top` are all $O(1)$ operations for the `Stack`

Problem 2

Stack A will grow from the beginning of the array while Stack B will grow from the end of the array.

```
template <typename T>
class DoubleStack {
public:
    enum Type {A, B};
    static size_t max = static_cast<size_t>(-1);
private:
    T* _data;
    size_t _capacity, _top0, _top1;

    // Assume Rule of Five is implemented correctly

    bool isEmpty(const Type& type) {
        switch (type) {
        case A:
            return this->_top0 == max;
        default:
            return this->_top1 = this->_capacity;
        }
    }

    bool isFull() { return this->_top0 == this->_top1; }
```

```
void push(const T& value, const Type& type) {
    if (this->isFull())
        throw overflow_error("Stack overflow");
    switch (type) {
    case A:
        this->_data[++this->_top0] = value;
        break;
    default:
        this->_data[--this->_top1] = value;
        break;
    }
}

const T& top(const Type& type) const {
    if (this->isEmpty())
        throw out_of_range("Stack is empty");
    switch (type) {
    case A:
        return this->_data[this->_top0];
    default:
        return this->_data[this->_top1];
    }
}
```

```
    void pop(const Type& type) {
        if (this->isEmpty())
            throw underflow_error("Stack Underflow")
        switch (type) {
        case A:
            if (!this->_top0)
                throw underflow_error("Stack A empty");
            --this->_top0;
            break;
        default:
            if (this->_top1 + 1 == this->_capacity)
                throw underflow_error("Stack B empty");
            ++this->_top1;
            break;
        }
    }
}
```

Problem 3

The worst case of `deleteMin` will be if the minimum value of the Stack is at the bottom. In this case, the algorithm will have to move all elements of the stack into another temporary stack, pop from the temp stack. and then push all the elements back into the original stack. This algorithm will have a runtime of $O(N)$.

Its best case will be if the minimum is at the top of the Stack. In this case, the algorithm needs to move the top of the stack into a temporary stack and then pop from the temporary stack. This algorithm will have a runtime of $\Omega(1)$.

With these two conditions,
$1 \leq N$ Taking log of both sides $\log_2 1 = \log_2 N$. $\therefore$ `deleteMin` is $\Omega \log_2(N)$

Queue ADT

Problem 1

Assuming the underlining storage type for the Queue is a Doubly Linked List with a `_size`, `_head`, and `_tail`.

With these assumptions, `enqueue`, `dequeue`, and `front` can be written as follows.

```
template <typename Object>
void Queue<Object>::enqueue(const Object& value) {
    Node* node = new Node(value);
    if (!this->_tail) {
        this->_head = node;
        this->_tail = node;
    }

    this->_tail->_next = node;
    node->_prev = this->_tail;
    this->_tail = this->_tail->_next;

    ++this->_size;
}

template <typename Object>
void Queue<Object>::dequeue() {
    if (!this->_head)
        throw underflow_error("Queue is empty");
    if (this->_size == 1) {
        delete this->_head;
        this->_head = this->_tail = nullptr;
    } else {
        this->_head = this->_head->_next;
        delete this->_head->_prev;
        this->_head->_prev = nullptr;
    }
    --this->_size;
}
```

```
template <typename Object>
const Object& Queue<Object>::front() const {
    if (!this->_size)
        throw out_of_range("Queue is empty");
    return this->_head->_value;
}
```

With these algorithms, `enqueue`, `dequeue`, and `front` are all $O(1)$ operations for the `Queue`.

Problem 2

Both iterators are worse case constant time. This is because since the class maintains a pointer to the head and tail, the iterators can simply query this value.

LSQ Redux

Problem 1

A. Drawing

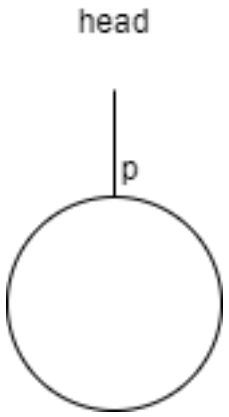

Figure 2: Linked List with Cycle

B. Algorithm

```
bool containsCycle() const {
    iterator it0(this->_head), it1(this->_head);

    while (it1 != this->end()) {
        ++it0;
        ++it1;
        ++it1;
        if (it0 == it1)
            return true;
    }
    return false;
}
```

Problem 2

A. `find` for an Array List

```
template <typename T>
void List<T>::find(size_t index) {
    this->remove(index);
    this->insert(index);
}

template <typename T>
void List<T>::remove(size_t index) {
    if (index >= this->_size)
        throw std::out_of_range("Index out of bounds");

    if (index == this->_size - 1)
        this->_data[index] = Object();
    else
        for (size_t i = index; i < this->_size; ++i)
            this->_data[i] = this->_data[i + 1];

    --this->_size;
}
```

B. `find` for a Linked List

```
template <typename T>
void List<T>::find(size_t index) {
    this->remove(index);
    this->insert(index);
}

template <typename T>
void List<T>::remove(size_t index) {
    if (index >= this->_size)
        throw out_of_range("Index out of bounds");

    if (this->_size == 1) {
        delete this->_head;
        this->_head = nullptr;
        this->_tail = nullptr;
    } else if (!index) {
        this->_head = this->_head->_next;
        delete this->_head->_prev;
        this->_head->_prev = nullptr;
    } else if (index == this->_size - 1) {
        this->_tail = this->_tail->_prev;
        delete this->_tail->_next;
        this->_tail->_next = nullptr;
    } else {
        Node* node = this->_head;
        for (size_t i = 0; i < index; ++i)
            node = node->_next;
        node->_prev->_next = node->_next;
        node->_next->_prev = node->_prev;
        delete node;
    }

    --this->_size;
}
```

C. Time complexity. Since the array list has to shift all the elements every time `find` is called, its time complexity if $O(N)$. Linked List also requires $O(N)$. Although updating pointers is $O(1)$, traversing the list to get to whatever element is accessed by `find` will be $O(N)$.

D. Elements with a higher probability will be accessed by `find` more frequently. Due to `find` behavior of placing elements it accesses at the beginning of the list, these elements will be found near the beginning.