Chapter 5: Linked Lists

Presenter: Mingyuan Yan
Preliminaries
Motivations

Options for implementing an ADT

- Array
  - Has a fixed size
  - Data must be shifted during insertions and deletions

- Linked list
  - Size can grow or shrink dynamically
  - Does not require the shifting of items during insertion and deletion
Array in C/C++ and Java

A collection of same type elements under the same variable identifier referenced by index number. (with contiguous physical memory)

- Array in C/C++
  - Declared using any of the data types available
  - Size must be declared using constant value before initialization
  - Initialize while declaration
  - Trying to access a value outside of range results in runtime errors. Compiler will not complain, but your program will crash when it executes

```c
<data-type> array-name[size-of-array]
int studentID[4];
int studentID[4]={1,2,3,4};
```
Motivations

Array in C/C++ and Java

- Dynamic array in C/C++
  
  ```c
  int *p=(int *)malloc(n*sizeof(int));
  ```
  
  - Dynamic allocation
  - Not an actual array because no reference represented as array
  - ask for continuous space and access the location by using points
Motivations

Array in C/C++ and Java

Array in Java

- Declare an array reference without specify its length
- Initialize (or instantiate) to fix length before access the array
- Initialize while declaration
- Trying to access a value outside of range results in complier error
- ADTs for array
  - Java.util.Arrays: [http://docs.oracle.com/javase/6/docs/api/java/util/Arrays.html](http://docs.oracle.com/javase/6/docs/api/java/util/Arrays.html)
  - java.util.ArrayList: [http://docs.oracle.com/javase/6/docs/api/java/util/ArrayList.html](http://docs.oracle.com/javase/6/docs/api/java/util/ArrayList.html)

```java
<data-type> array-name[];
arrayName=new type[arraySize];

int studentID[ ];
studentID=new int[4];
```
Motivations

- An Array-Based Implementation of the ADT List

  - **Arbitrary array (unordered)**
    - $M < \text{array.size}$ items are needed to be insert, waste $\text{array.size} - M$ spaces
    - Insert is convenient
    - Search by array traversal
    - Delete item
      - Keep the array unchanged (waste spaces, cost more for searching)
      - Shift (waste spaces and shift cost)
Motivations

An Array-Based Implementation of the ADT List

- *sorted array (ordered)*
  - $M < \text{array.size}$ items are needed to be inserted, waste $\text{array.size} - m$ spaces
  - Insert need shifts

<table>
<thead>
<tr>
<th>water</th>
<th>pencil</th>
<th>juice</th>
<th>milk</th>
<th>butter</th>
<th>apple</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>water</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pencil</td>
<td>water</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>juice</td>
<td>pencil</td>
<td>water</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>milk</td>
<td>juice</td>
<td>pencil</td>
<td>water</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

.....
Motivations

An Array-Based Implementation of the ADT List

- **sorted array (ordered)**
  - Worst case: shift all the time
  - Search is convenient (binary search)
  - Delete item
    - Keep the array unchanged (waste spaces, cost more for searching)
    - Shift (waste spaces and shift cost)
## Motivations

**Options for implementing an ADT**

- **Array**
  - Has a fixed size
  - Data must be shifted during insertions and deletions

<table>
<thead>
<tr>
<th>C/C++</th>
<th>memory</th>
<th>stack</th>
<th>heap</th>
<th>static</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1M (related to the compiler and operating system)</td>
<td>No more than the physical page size</td>
<td>2^{31}-1 if the operating system is 32 bit</td>
</tr>
</tbody>
</table>
**Motivations**

- **Options for implementing an ADT**
  - **Linked list**
    - Size can grow or shrink dynamically
    - Does not require the shifting of items during insertion and deletion
Object Reference

A reference variable

- Refers to an object of a given class
- Contains the location of an object
- Come into existence until apply the `new` operator

```java
Integer intRef;
intRef = new Integer(5);
```

Default value

- As a data field of a class: has the default value `null`
  - Java.lang.NullPointerException error at run time
- A local reference variable to a method: Does not have a default value
  - compiler gives “variable may not have been initialized” error
Object Reference

A reference variable

**Figure 5-2**
A reference to an Integer object

(a) This view shows the data members and methods for the object.

(b) This view only shows the data members for simplicity. This is the view used throughout the text.
A reference variable

(a) `Integer p;`  
   `Integer q;`

(b) `p = new Integer(5);`

(c) `p = new Integer(6);`

(d) `q = p;`

(e) `q = new Integer(9);`

(f) `p = null;`

(g) `q = p;`
An array of objects

- Declare an array of objects and apply the `new` operator, actually creates references to the objects

- Example

```
Integer[] scores = new Integer[30];
```

- Instantiating Integer objects for each array reference

```
scores[0] = new Integer(7);
scores[1] = new Integer(9);  // and so on ...
```
Resizable arrays

- The number of references in a Java array is of fixed size
- Resizable array
  - An array that grows and shrinks as the program executes
  - An illusion that is created by using an allocate and copy strategy with fixed-size arrays
- java.util.Vector / java.util.ArrayList class
  - Uses a similar technique to implement a growable array of objects
Reference-Based Linked Lists

- **Linked list**
  - Contains nodes that are linked to one another
  - **node**
    - Data and “link” to the next item (Java reference variable)
    - Node of the list can be implemented as an object

- **Example**

```java
class IntegerList {
    class IntegerNode{
        int item;
        IntegerNode next;
    }

    IntegerNode n1 = new IntegerNode();
    IntegerNode n2 = new IntegerNode();
    n1.item = 5;
    n2.item = 9;
    n1.next = n2;
}
```
Reference-Based Linked Lists

Linked list

- How to distinguish the beginning and the end of the list?
  - How to refer the beginning of the linked list?
  - What’s value should be given to the next field of the last node?

Solution

- Data field next in the last node is set to null
- head reference variable
  - References the list’s first node
  - Always exists even when the list is empty
  - can be assigned null without first using new operator

```java
head = new Node(new Integer(5));
head = null;
```
Programming with linked list
Based on the linked list structure, how can we implement the following operations:

- How to insert a node into a specified position of a linked list?
  - Any where except the beginning
  - In the beginning

- How to display the contents of the linked list?

- How to delete a specified node from a linked list?
  - Any node except the first node
  - The first node
Inserting a node into a specified position of a linked list

1. Find the right position

```
prev = null;
curr = head;

while (curr != null && newValue > curr.item) {
    prev = curr;
    curr = curr.next
}
```

Insertion at the end of a linked list is not a special case
Inserting a node into a specified position of a linked list

2. create a node for the new item

   ```java
   newNode = new Node(item);
   ```

3. insert a node between two nodes

   ```java
   newNode.next = curr;
   Prev.next = newNode;
   ```

![Diagram of linked list insertion](image-url)
To insert a node at the beginning of a linked list

```java
newNode.next=head;
head=newNode;
```
Three steps to insert a new node into a linked list

1. Determine the point of insertion
2. Create a new node and store the new data in it
3. Connect the new node to the linked list by changing references
Display

Display a linked list

1. `curr` reference variable
   - References the current node
   - Initially references the first node

2. To display the data portion of the current node
   ```java
   System.out.println(curr.item);
   ```

3. To advance the current position to the next node
   ```java
   curr = curr.next;
   ```
Display a linked list

- To display all the data items in a linked list

```java
for (Node curr = head; curr != null; curr = curr.next)
{
    System.out.println(curr.item);
}
```
To delete node N which curr references

1. Locate the `prev` and `curr`

2. Set next in the node that precedes N to reference the node that follows N

\[ \text{prev.next} = \text{curr.next}; \]
Deleting the first node is a special case

```
head = head.next;
```

To return a node that is no longer needed to the system

```
curr.next = null;
Curr = null;
```
Three steps to delete a node from a linked list

1. Locate the node that you want to delete
2. Disconnect this node from the linked list by changing references
3. Return the node to the system
A reference-based implementation of the ADT List

1. Does not shift items during insertions and deletions
2. Does not impose a fixed maximum length on the list

Read your textbook, try to implement it as a practice
Comparison

➢ Size

- **Array-based:**
  - **Fixed size**
    
    » Can you predict the maximum number of items in the ADT?
    
    » Will an array waste storage?
  
  - **Resizable array**
    
    » Increasing the size of a resizable array can waste storage and time

- **Reference-based**
  
  - **Do not have a fixed size**
    
    » Do not need to predict the maximum size of the list
    
    » Will not waste storage
Storage requirements

- **Array-based:**
  - Requires less memory than a reference-based implementation
    - There is no need to store explicitly information about where to find the next data item

- **Reference-based**
  - Requires more storage
    - An item explicitly references the next item in the list
Access time

- **Array-based:**
  - **Constant access time**
    - There is no need to store explicitly information about where to find the next data item

- **Reference-based**
  - The time to access the ith node depends on i
Comparison

- Insertion and deletions
  - **Array-based:**
    - Require you to shift the data
  - **Reference-based**
    - Do not require you to shift the data
    - Require a list traversal
A method with access to a linked list’s `head` reference has access to the entire list

When `head` is an actual argument to a method, its value is copied into the corresponding formal parameter

Use `return` to modify `head`’s value
Traversal

- **display a list**
  - Write the first node of the list
  - Write the list minus its first node

- **display a list backward**
  - **writeListBackward strategy**
    - Write the last node of the list
    - Write the list minus its last node backward
  - **writeListBackward2 strategy**
    - Write the list minus its first node backward
    - Write the first node of the list
Insertion

```java
private static Node insertRecursive(Node headNode, java.lang.Comparable newItem) {
    if ((headNode == null) || (newItem.compareTo(headNode.item) < 0)) {
        Node newNode = new Node(newItem, headNode);
        headNode = newNode;
    } else {
        Node nextNode = insertRecursive(headNode.next, newItem);
        headNode.next = nextNode;
    }
    return headNode;
}

head = insertRecursive(head, newItem);
```
Variations of the linked list
Variations of the linked list

- Tail references
  - Remembers where the end of the linked list is
  - To add a node to the end of a linked list

```java
tail.next = new Node(request, null);
tail = tail.next;
```

```
head  a  b  c  d  ...  tail
```
Variations of the linked list

- **Circular linked list**
  - Last node references the first node
  - Every node has a successor
  - Access the list through any node

```java
if(list!=null){
    Node first=list.next;
    Node curr=first;
    do{
        System.out.println(curr.item);
        curr=curr.next;
    }while(curr!=first);
}
```
Variations of the linked list

- **Dummy head nodes**
  - Always present, even when the linked list is empty
  - Insertion and deletion algorithms initialize `prev` to reference the *dummy* head node, rather than null

```javascript
Prev.next = curr.next;
```

![Diagram of a linked list with a dummy head node]
Variations of the linked list

- **Doubly linked list**
  - Each node references both its predecessor and its successor
  - Dummy head nodes are useful in doubly linked lists

  ```
  prev = curr.previous;
  ```

- **Circular Doubly linked list**

```
Maintaining an inventory
Maintaining an inventory

- **Insertion and deletions**
  - **Problem-solving process**
    - Design of a solution
    - Implementation of the solution
    - Final set of refinements to the program
  - **Linked list related contents**
    - Add new people to the end of the wait list when they want a DVD that is sold out
    - Delete people from the beginning of the wait list when new DVD are delivered
    - Display the names on a wait list for a particular title
The java collections framework
Java provides classes that implement many of the more commonly used ADTs

- **Java collections framework**
  - A unified architecture for representing and manipulating collections
  - Include
    - Interfaces or ADTs representing collections
    - Implementations or concrete implementation of collection interfaces
    - Algorithms or methods that perform useful computations, e.g. sorting searching

- **JCF List interface**
  - `Java.util.List`
  - Classes implement List interface
    - `LinkedList`, `ArrayList`, and `Vector`
Summary

- Preliminaries
  - The advantage of LinkedList and disadvantage of Array
  - The preliminaries of java references
  - How to get a reference-based linked list

- Programming on Linked List
  - Insertion, display and deletion
  - The comparison between array-based and linked list based implementation

- Variations of Linked List
  - Tail references
  - Circular linked list
  - Dummy head nodes
  - Doubly linked list

- Application and JCF