Lists, Stacks, and Queues

Many significant programs use lists, stacks, or queues in some form.

- Motivation:
  - Review ADTs
  - Review familiar list, stack, and queue data types
  - Introduce analysis with them
  - Discuss efficient implementations of lists, stacks, queues
  - Review some common applications of lists, stacks, queues

Lists

- list: a finite, ordered sequence of data items called elements

- Associated definitions/concepts:
  - Each list element has a data type
  - The empty list contains no elements
  - The list length is the number of elements currently stored
  - The beginning of the list is the head
  - The end of the list is the tail
  - Sorted lists have elements positioned in ascending order of value
  - Unsorted lists have no relationship between position and element value
  - Notation: $A_1, A_2, A_3, \ldots, A_n$
    or $(A_1, A_2, A_3, \ldots, A_n)$
  - Popular operations: print, makeEmpty, insert, remove, next, prev, etc.
List Implementation Concepts

- List defined in terms of left and right partitions
  - Either or both partitions may be empty
  - Each partition is separated by a fence.
  - Example: <20, 23 | 12, 15>

- List ADT:

  ```cpp
  template <class Elem> class List {
    public:
      virtual void clear() = 0;
      virtual bool insert(const Elem&) = 0;
      virtual bool append(const Elem&) = 0;
      virtual bool remove(const Elem&) = 0;
      virtual void setStart() = 0;
      virtual void setEnd() = 0;
      virtual void prev() = 0;
      virtual void next() = 0;
      virtual int leftLength() const = 0;
      virtual int rightLength() const = 0;
      virtual bool setPos(int pos) = 0;
      virtual bool getValue(Elem&) = 0;
      virtual void print() const = 0;
  };
  ```

List ADT Examples

- A list containing <12 | 32, 15>
  - Execute `MyList.insert(99);
  - Result: <12 | 99, 32, 15>

- List Iteration:

  ```cpp
  for (MyList.setStart(); MyList.getValue(it);
       MyList.next()) {
    (Do something with this list element.)
  }
  ```

- List Find Function

  ```cpp
  bool find(List<int>& L, int K) {
    int it;
    for (L.setStart(); L.getValue(it);
         L.next())
      if (K == it) return true;
    return false;
  }
  ```
Array-Based Lists

- A contiguous block of memory containing elements:

- Time estimates for:
  - print
  - find

- See web site for code examples

Array-Based List Insert

- Insert 42 here

(c)
Array-Based List Delete

- Time to delete:

<table>
<thead>
<tr>
<th>13</th>
<th>42</th>
<th>20</th>
<th>12</th>
<th>3</th>
<th>8</th>
</tr>
</thead>
</table>

delete 1st element

<table>
<thead>
<tr>
<th>13</th>
<th>42</th>
<th>20</th>
<th>12</th>
<th>3</th>
<th>8</th>
</tr>
</thead>
</table>

Array-Based List Class

- The class header:

```cpp
#include "list.h"

template <class Elem>
class AList : public List<Elem> {
private:
    int maxSize; // Maximum size of list
    int listSize; // Actual number of elements in list
    int fence; // Position of fence
    Elem* listArray; // Array holding list elements
public:
    AList(int size=DefaultListSize);
    ~AList();
    void clear();
    bool insert(const Elem&);
    bool append(const Elem&);
    bool remove(Elem&);
    void setStart();
    void setEnd();
    void prev();
    void next();
    int leftLength() const;
    int rightLength() const;
    bool setPos(int pos);
    bool getValue(Elem& it) const;
    void print() const;
};
```

- (See web site for remaining code.)
**Linked Lists**

- A series of memory blocks containing **nodes**:

  ![Linked List Diagram](image)

  - Nodes contain:
    - element (the data)
    - next link to another node containing the successor element
  - Time estimates for:
    - print
    - find

**linked List Insert/Delete**

- Inserting $X$ between $A_2$ and $A_3$:

  ![Insert Diagram](image)

  - Time to insert:

- Deleting $A_3$:

  ![Delete Diagram](image)

  - Time to delete:
Linked List Positioning

- How do we insert 10 before the 12?
  - Naive approach:
    - Better approach:

Use of a Header Node

- Several problems not yet solved:
  - There is no obvious way to insert at the head of the list
  - Removing from the front is a special case
  - Deletion requires finding the node before the one to be deleted

- Simple change solves all three: use a dummy header node
Use of Fence in Linked List

Shaffer uses the "fence" instead of a "curr" pointer.

- Again, how do we insert 10 before the 12?
- Naive approach:

```plaintext
head

20 -- 23 -- 12 -- 15

fence
```

Use of Fence in Linked List

- Better approach:

```plaintext
head

20 -- 23 -- 12 -- 15

fence
tail
```

CSC 375-Turner, Page 13

CSC 375-Turner, Page 14
Linked List Implementation

- One view: implement three separate classes:
  - ListNode, to implement the nodes themselves
  - ListIter, to implement the concept of position
  - List, to implement the list

- Shaffer uses two classes: Link nodes and the list itself
  - The link class stores the data and pointer to next node
  - The list class stores list functions and pointers to Link nodes.

Link Class

Uses dynamic allocation of new list elements.

- Class header:

```cpp
template <class Elem> class Link {
public:
    Elem element; // Value for this node
    Link *next; // Pointer to next node in list

    Link(const Elem& elemval, Link* nextval = NULL) {
        element = elemval; next = nextval;
    }

    Link(Link* nextval = NULL) { next = nextval; }
};
```
**Linked List Class**

- Linked list header file:

```cpp
template <class Elem> class LList: public List<Elem> {
private:
    Link<Elem>* head; // Pointer to list header
    Link<Elem>* tail; // Pointer to last Elem in list
    Link<Elem>* fence; // Last element on left side
    int leftcnt; // Size of left partition
    int rightcnt; // Size of right partition
    void init(); // Initialization routine
    void removeall(); // Return link nodes to free store
public:
    LList(int size=DefaultListSize);
    ~List();
    void clear(); // Remove and reset the list
    bool insert(const Elem& item);
    bool append(const Elem& item);
    bool remove(Elem&); // Use of template
    void setStart(); // Move the fence to the far left
    void setEnd(); // Move the fence to the far right
    void prev(); // Move the fence one left
    void next(); // Move the fence one right
    int leftLength() const;
    int rightLength() const;
    bool setPos(int pos);
    bool getValue(Elem& it) const;
    void print() const;
};
```

**Insert and Append**

- Insert at front of right partition:

```cpp
template <class Elem>
bool LList<Elem>::insert(const Elem& item) {
    fence->next = new Link<Elem>(item, fence->next);
    if (tail == fence) tail = fence->next; // New tail
    rightcnt++;
    return true;
}
```

- Append Elem to the end of the list:

```cpp
template <class Elem>
bool LList<Elem>::append(const Elem& item) {
    tail = tail->next = new Link<Elem>(item, NULL);
    rightcnt++;
    return true;
}
```
• Remove and return the first element (Elem) in the right partition

    template <class Elem> bool LList<Elem>::remove(Elem& it) {
        if (fence->next == NULL) return false; // Empty right
        it = fence->next->element;       // Remember value
        Link<Elem>* ltemp = fence->next;  // Remember link node
        fence->next = ltemp->next;       // Remove from list
        if (tail == ltemp) tail = fence;  // Reset tail
        delete ltemp;                    // Reclaim space
        rightcnt--;
        return true;
    }

Positioning

• Next and Prev:

    // Move fence one step right; no change if at tail.
    template <class Elem> void LList<Elem>::next() {
        if (fence != tail) {
            fence = fence->next;
            rightcnt--; leftcnt++;
        }
    }

    // Move fence one step left; no change if left is empty
    template <class Elem> void LList<Elem>::prev() {
        Link<Elem>* temp = head;
        if (fence == head) return; // No previous Elem
        while (temp->next != fence) temp = temp->next;
        fence = temp;
        leftcnt--; rightcnt++;
    }

• SetPos:

    // Set the size of left partition to pos
    template <class Elem>
    bool LList<Elem>::setPos(int pos) {
        if ((pos < 0) || (pos > rightcnt+leftcnt))
            return false;
        rightcnt = rightcnt + leftcnt - pos; // Set counts
        leftcnt = pos;
        fence = head;
        for (int i=0; i<pos; i++)
            fence = fence->next;
        return true;
    }
Comparison of List Implementations

- Array-based lists:
  - Insert and delete are $\Theta(n)$
  - Array must be pre-allocated
  - No overhead if the array is full
  - Inefficient use of storage if list is almost empty

- Linked lists:
  - Insertion and deletion are $\Theta(1)$, but finding previous and direct access are $\Theta(n)$
  - Space grows with number of elements
  - Every element requires overhead

- Space break-even point:
  \[ DE = n(P + E) \]
  \[ n = \frac{DE}{P + E} \]

  $E$ is space for data value, $P$ is space for pointer, and $D$ is number of elements in the array

Memory Reclamation

- Removeall:

  ```cpp
  template <class Elem>
  void LList<Elem>::removeall() {
    while(head != NULL) {
      fence = head;
      head = head->next;
      delete fence;
    }
  }
  
  Removeall makes the destructor very simple:

  template <class Object>
  LList<Object>::~LList() {
    removeall();
  }
  ```
Freelists

- Some languages do not support dynamic memory allocation, and C++ can simulate it.

- Desirable features:
  - Data are stored in a collection of nodes, each of which also contains a link to the next node.
  - A new node can be obtained from system memory by a call to `new`.

- Motivations for simulation in any C++ program:
  - Calls to the system's `new` and `delete` can be expensive (slow).
  - You can improve performance by up to 30% by replacing `new` and `delete`.

- Methodology:
  - Create a large array of "Link nodes".
  - Initially, for all i, set A[i].next to point at A[i+1].
  - Use a header node to point at A[0].
  - Remove and return (`new` and `delete`) from/to the array.

- Method is also known as cursor implementation.

Free List Link Class

- Major difference is static freelist variable plus overloaded operators.

  ```
  template <class Elem> class Link {
  private:
    static Link<Elem>* freelist; // Head of the freelist
  public:
    Elem element; // Value for this node
    Link* next; // Point to next node in list

    Link(const Elem& elemval, Link* nextval = NULL) {
      element = elemval; next = nextval; }
    Link(Link* nextval = NULL) { next = nextval; }

    void* operator new(size_t); // Overloaded new operator
    void operator delete(void*); // Overloaded delete operator
  }

  template <class Elem>
  Link<Elem>* Link<Elem>::freelist = NULL;
  ```

CSC 375-Turner, Page 23
Overloaded Operators

- New and Delete:

  ```cpp
template <class Elem>
void* Link<Elem>::operator new(size_t) {
    if (freelist == NULL)
        return ::new Link;    // Create space
    Link<Elem>* temp = freelist; // Can take from freelist
    freelist = freelist->next; // Return the link
    return temp;
}

template <class Elem>
void Link<Elem>::operator delete(void* ptr) {
    ((Link<Elem>*)ptr)->next = freelist; // Put on freelist
    freelist = (Link<Elem>*)ptr;
}
```

Doubly Linked lists

- Simplifies insertion/deletion by adding an extra pointer.

- Doubly-linked list class header:

  ```cpp
template <class Elem> class Link {
public:
    Elem element;    // Value for this node
    Link* next;      // Pointer to next node in list
    Link* prev;      // Pointer to previous node

    Link(const Elem& e, Link* prevp = NULL, 
         Link* nextp = NULL) {
        element = e;
        prev = prevp;
        next = nextp;
    }

    Link(Link* prevp = NULL, Link* nextp = NULL) 
    { prev = prevp; next = nextp; }
};
```
**Insert and Remove**

- **Doubly Linked Insert:**
  ```cpp
template <class Elem>
bool LLList<Elem>::insert(const Elem& item) {
    fence->next = new Link<Elem>(item, fence, fence->next);
    if (fence->next->next != NULL) // If not at end
        fence->next->next->prev = fence->next;
    if (tail == fence) // Appending new Elem
        tail = fence->next; // so set tail
    rightcnt++; // Added to right
    return true;
}
```

- **Doubly Linked Remove:**
  ```cpp
template <class Elem> bool LLList<Elem>::remove(Elem& it)
if (fence->next == NULL) return false; // Empty right
if (fence->next == fence) // Empty right
    return true;
if (ltmp = fence->next) // Remember link node
    if (ltmp->next != NULL) ltmp->next->prev = fence;
else tail = fence; // Reset tail
fence->next = ltmp->next; // Remove from left
delete ltmp; // Reclaim space
rightcnt--; // Reduced from right
return true;
}
```

**Comparator Class**

How can comparison be generalized?

- **Use ==, <=, >= with no modification.**
  - Problems?

- **Overload ==, <=, >=, etc.**
  - Problems?

- **Define a function with a standard name**
  - Problems:
    - Implied obligation
    - Breaks down if multiple key fields or indices are used for the same object

- **Pass in a function**
  - Requires an explicit obligation
  - Can pass in as a function parameter in the template parameter
  - Shaffer uses his Dictionary ADT to illustrate this
The Stack ADT

Also known as a LIFO (Last-In, First-Out) list

- A stack is a list with access restrictions:
  - Insertion and deletions may only be performed at one end of the list, the top
  - Implementation may determine which physical end of the list is actually used

- Notation
  - Insert: push
  - Delete: pop
  - Only accessible element: top

- Stack Class Header

  template <class Elem> class Stack {
  public:
    virtual void clear() = 0;
    virtual bool push(const Elem&) = 0;
    virtual bool pop(Elem&) = 0;
    virtual bool topValue(Elem&) const = 0;
    virtual int length() const = 0;
  };

Array-Based Stack

- Some implementation details:
  - private:
    - int size;
    - int top;
    - Elem *listArray;

- Issues:
  - Which end of the array is the top?
  - Where does top point to?
  - What is the cost of operations?
Linked List Stack

- Some implementation details:
  
  private:
   Link<Elem>* top;
   int size;

- Issues:
  - What is the cost of operations?
  - How do space requirements compare to that of the array-based implementation?

The Queue ADT

Also known as a **FIFO** (First-In, First-Out) list

- A queue is also a list with access restrictions:
  - insertion and deletions are performed at opposite ends of the list.

- Notation
  - Insert: **enqueue**
  - Delete: **dequeue**
  - First element: **front**
  - First element: **rear**

- Array-based queue implementation issues:
  - What to do with “drift” of front and rear indices?
  - When array is “circular”, how to distinguish full and empty?

- Applications:
  - Operating Systems
  - Real-life lines
  - Computer networking
  - Computer simulation
Array-Based Queue

- Queue drift

Array-Based Queue

- Circular implementation issues

- Use of mod function gives effect of circular queue

- Questions:
  - Where do front/rear pointers point?
  - How do we distinguish full from empty?
    - Leave an empty slot
    - Use external variable

CSC 375-Turner, Page 33