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& elem) = 0;
    virtual bool append(const Elem& elem) = 0;
    virtual bool remove(const Elem& 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 i) = 0;
    virtual bool getValue(Elem& value) = 0;
    virtual bool 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.hasValue(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

- Time to insert:

Array-Based List Delete

- Time to delete:

Array-Based List Class

- The class header:

```
#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:

```
A1   A2   A3   A4   A5
```

- 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 A2 and A3:

```
A1   A2   X   A3   A4   A5
```

- Time to insert:

- Deleting A3:

```
A1   A2   A4   A5
```

- Time to delete:

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

```
header
A1   A2   A3   A4   A5
```
Use of Fence in Linked List

- Shaffer uses the "fence" instead of a "curr" pointer.
- Better approach:

```
   head
   /
   |
   v
20  23
   |
   |
   V
   |
12  15
   |
   |
```

- Naive approach:

```
   head
   /
   |
20  23
   |
   |
   V
   |
12  15
   |
   |
```

Use of Fence in Linked List

- 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.

```
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; }
};
```

CSC 375-Turner, Page 13

CSC 375-Turner, Page 14

CSC 375-Turner, Page 15

CSC 375-Turner, Page 16
Linked List Class

- Linked list header file:

```cpp
template <class Elem> class ListElem { public: 
  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
  List(int size=DefaultListSize); 
  ~List();
  void clear(); // Remove and reset the list
  bool insert(const Elem&); 
  bool append(const Elem&);
  bool remove(Elem&);
  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 List<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 List<Elem>::append(const Elem& item) {
  tail = tail->next = new Link<Elem>(item, NULL);
  rightcnt++;
  return true;
}
```

Remove

- Remove and return the first element (Elem) in the right partition:

```cpp
template <class Elem> bool List<Elem>::remove(Elem& it) {
  if (fence->next == NULL) return false; // Empty right
  it = fence->next->element; // Remember value
  fence->next = fence->next->next; // Remove from list
  if (tail == fence) tail = fence; // Reset tail
  delete it;
  return true;
}
```

Positioning

- Next and Prev:

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

```cpp
// Move fence one step left; no change if left is empty
template <class Elem> void List<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:

```cpp
// Set the size of left partition to pos
template <class Elem>
bool List<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 $O(n)$
  - Array must be pre-allocated
  - No overhead if the array is full
  - Inefficient use of space if list is almost empty

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

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

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

CSC 375-Turner, Page 21

Memory Reclamation

- Removeall:
  ```cpp
template <class Elem>
void List<Elem>::removeall() {
    while(head != NULL) {
        fence = head;
        head = head->next;
        delete fence;
    }
}
```

- Removeall makes the destructor very simple:
  ```cpp
template <class Object>
List<Object>::~List() {
    removeall();
}
```

CSC 375-Turner, Page 22

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

CSC 375-Turner, Page 23

Free List Link Class

- Major difference is static freelist variable plus overloaded operators.

  ```cpp
template <class Elem> class Link {
    private:
    static List<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
    }
```

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

CSC 375-Turner, Page 24
Overloaded Operators

- New and Delete:

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

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* prev = NULL, Link* next = NULL) {
        element = e;
        prev = prev;
        next = next;
    }

    Link(Link* prev = NULL, Link* next = NULL) {
        prev = prev;
        next = next;
    }

};
```

Comparator Class

- How can comparison be generalized?

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

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

  - Define a function with a standard name
    □ Problems:
      o Implied obligation
        o 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

Insert and Remove

- Doubly Linked Insert:

```cpp
template <class Elem>
bool LinkedList::insert(const Elem &item) {
    fence->next = new Link<Elem>(item, fence, fence->next);  // Append new Elem
    if (fence->next == NULL) return false;  // Empty right
    return true;
}
```

- Doubly Linked Remove:

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

```cpp
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;
    };
```

CSC 375-Turner, Page 29

Array-Based Stack

- Some implementation details:
  ```cpp
  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?

CSC 375-Turner, Page 30

Linked List Stack

- Some implementation details:
  ```cpp
  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?

CSC 375-Turner, Page 31

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

CSC 375-Turner, Page 32
Array-Based Queue

- Queue drift

```
front  rear
20 5 12 17
```

```
front  rear
12 17 3 30 4
```

Array-Based Queue

- Circular implementation issues

```
front  rear
20 5 12 17
```

```
front  rear
12 17 3 4 30
```

- 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