Linked Lists

Steven J. Zeil

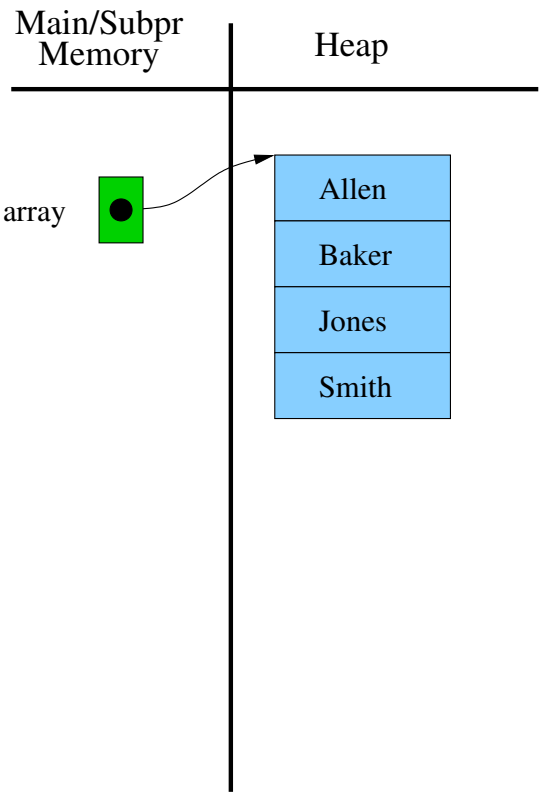
November 18, 2013

Contents

1 Linked Lists: the Basics

Sequences: arrays

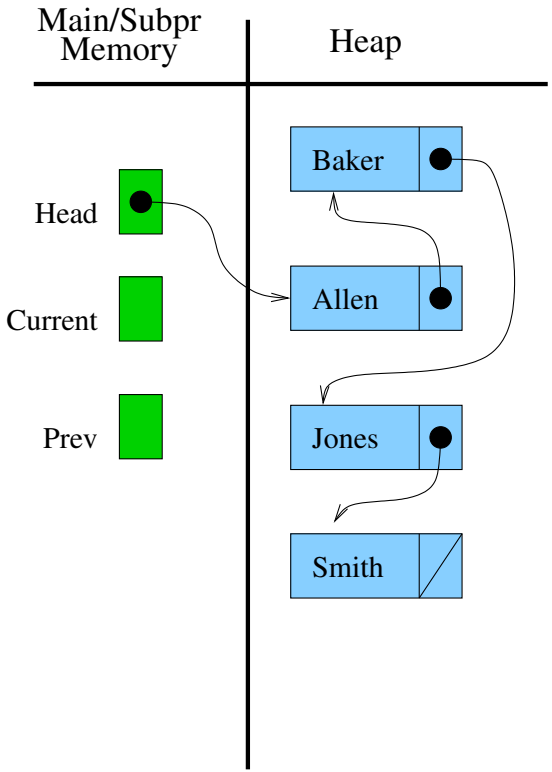
Consider the abstraction: a sequence of elements



Arrays (and vectors) work by storing elements of a sequence contiguously in memory

- Easy to access elements by number
- Inserting things into the middle is slow and awkward

Sequences: Linked Lists



Linked lists store each element in a distinct *node*. Nodes are *linked* by pointers.

- Accessing elements by number is slow and awkward
- Easy to insert things into the middle

.....

Linked List Nodes

A linked list node:

```
struct LinkedListNode {
  DataType data;
  LinkedListNode* next;
```


```
};
LinkedListNode* firstNode;
```

- A linked list consists of a number of *nodes*.
- Each node provides a *data* field and a *next* pointer.

. . . . . .

Basic List Operations

- We *traverse* (move from node to node) by tracing the pointers.
- We insert new nodes by working from the node *prior* to the insertion point.
- We remove nodes by moving the previous pointer "around" the unwanted node.

. . . . . .

2 Coding for Linked Lists

The Data Types

We need two data types to build a linked list:

- The nodes

```
template <typename Data>
struct LListNode
{
  Data data;
  LListNode<Data>* next;

  LListNode() {next = 0;}
```


```
  LListNode
    (const Data& d,
     LListNode<Data>* nxt = 0)
    : data(d), next(nxt)
  {}
};
```

  - I'm doing this as a template so it can be easily re-used
  - The actual kind of data stored is not particularly important

- And the header

```
template <typename Data>
struct LListHeader {

  LListNode<Data>* first;

  LListHeader();
   ⋮
};
```

  - provides the starting point for the list

.....

- I am treating these as utilities, not full-fledged ADTs
  - We would use these to build "proper" ADTs like `BidCollection`
  - So I'm not going to be as picky about encapsulation and providing operators as I would usually be

Example: Using the LListHeader

```
class BidderCollection {

  int size;
  LListHeader<Bidder> list;

public:

  typedef LListNode<Bidder>* Position;

  /**
   * Create a collection capable of holding any number of items
   */
  BidderCollection ();
    ⋮
```

.....

In the sections that follow, we will look both at how we would use the linked list (in `BidderCollection` and `BidCollection`) and how we would implement it.

2.1 Traversing a Linked List

Traversing a Linked List

```
class BidderCollection {
    ⋮
  // Print the collection
  void print (std::ostream& out) const;

  // Comparison operators
  bool operator== (const BidderCollection&) const;
  bool operator< (const BidderCollection&) const;
};
```



All of these require us to walk the list, one node at a time.

- We use a pointer to a node to denote our current position
- We advance to the next position by getting the link field out of that node.

.....

Printing a collection - while loop

```
// Print the collection
void BidderCollection::print (std::ostream& out) const
{
  out << size << "{";
  Position current = list.first;
  while (current != NULL)
    {
      out << "  ";
      current->data.print (out);
      out << "\n";

      current = current->next;
    }
  out << "}";
}
```

.....

Printing a collection - for loop

```
// Print the collection
void BidderCollection::print (std::ostream& out) const
{
  out << size << "{";
  for (Position current = list.first;
```


```
       current != NULL; current = current->next)
    {
      out << "  ";
      current->data.print (out);
      out << "\n";
    }
  out << "}";
}
```

.....

2.2 Searching a Linked List

Searching a Linked List

To support:

```
class BidderCollection {
    ⋮
  Position findBidder (std::string name) const;
  // Returns the position where a bidder matching
  // the given name can be found, or null if no
  // bidder with that name exists.
```

we do a traversal but stop early if we find what we are looking for.

.....

findBidder

```
/**
 * Find the index of the bidder with the given name. If no such bidder exists,
 * return null.
 */
BidderCollection::Position BidderCollection::findBidder
   (std::string name) const
{
```


```
  for (Position current = list.first; current != NULL;
       current = current->next)
    {
      if (name == current->data.getName())
        return current;
    }
  return NULL;
}
```

.....

Searching Linked Lists

As a general utility, we might try to provide:

```
template <typename Data>
struct LListHeader {

  LListNode<Data>* first;
    ⋮
  // Search for a value. Returns null if not found
  LListNode<Data>* find (const Data& value) const;

  // Search an ordered list for a value. Returns
     null if not found
  LListNode<Data>* findOrdered (const Data& value) const;
```

.....

Searching Unordered Linked Lists

```
  // Search for a value. Returns null if not found
template <typename Data>
LListNode<Data>* LListHeader<Data>::find
   (const Data& value) const
{
```



```
  LListNode<Data>* current = first;
  while (current != NULL && value != current->data)
    current = current->link;
  return current;
}
```

.....

Searching Ordered Linked Lists

```
  // Search an ordered list for a value. Returns null if not found
template <typename Data>
LListNode<Data>* LListHeader<Data>::findOrdered
  (const Data& value) const
{
  LListNode<Data>* current = first;
  while (current != NULL && value > current->data)
    current = current->link;
  if (current != NULL && value == current->data)
    return current;
  else
    return NULL;
}
```

.....

findBidder (alternate)

```
/**
 * Find the index of the bidder with the given name. If no such bidder exists,
 * return null.
 */
BidderCollection::Position BidderCollection::findBidder
   (std::string name) const
{
```



```
  Bidder searchFor (name, 0.0);
  return list.find (searchFor);
}
```

.....

(works only because

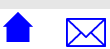
```
bool Bidder::operator== (const Bidder& b) const
{
  return name == b.name;
}


bool Bidder::operator< (const Bidder& b) const
{
  if (name < b.name)
    return true;
  else
    return false;
}
```

compare only by name).

Walking Two Lists at Once

Although not really a search, the relational operator code is similar, in that we do a traversal with a possible early exit. But now we have to walk two lists:

```
  // Comparison operators
bool BidderCollection::operator== (const BidderCollection& bc) const
{
  if (size == bc.size)
    {
      Position current = list.first;
      Position bcurrent = bc.list.first;
```

```
      while (current != NULL)
        {
          if (!(current->data == bcurrent->data))
            return false;
          current = current->next;
          bcurrent = bcurrent->next;
        }
      return true;
    }
  else
    return false;
}
```

.....

2.3 Adding to a Linked List

Adding to a Linked List

```
void BidderCollection::add (const Bidder& value)
```

could be implemented as

```
void BidderCollection::add (const Bidder& value)
//  Adds this bidder
//Pre: getSize() < getMaxSize()
{
  list.addToEnd (value);
  ++size;
}
```

.....

add functions

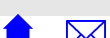
```
template <typename Data>
struct LListHeader {

  LListNode<Data>* first;


  LListHeader();

  void addToFront (const Data& value);
  void addToEnd (const Data& value);

  // Add value in sorted order.
  //Pre: all existing values are already ordered
  void addInOrder (const Data& value);
    ⋮
```

Let's look at how to do addToEnd first.

.....

addToEnd

```
template <typename Data>
void LListHeader<Data>::addToEnd (const Data& value)
{
  LListNode<Data>* newNode = new LListNode<Data>(value, NULL);
  if (first == NULL)
    {
      first = newNode;
    }
  else
    {
      // Move to last node
      LListNode<Data>* current = first;
```

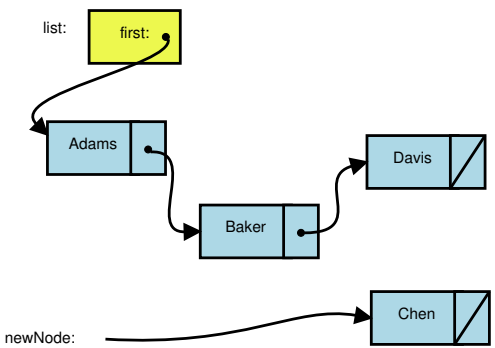



```
      while (current->next != NULL)
        current = current->next;

      // Link after that node
      current->next = newNode;
    }
}
```

.....

addToEnd Example

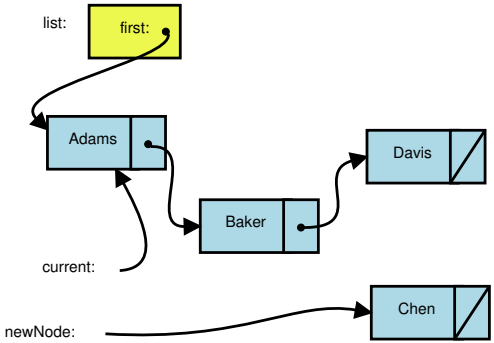
E.g. Suppose we have a list containing "Adams", "Baker", & "Davis", and we are going to add "Chen".
We create a new node.



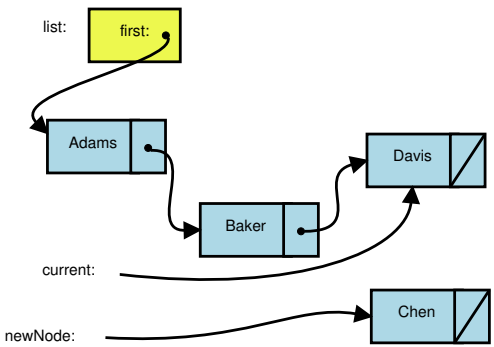

```
template <typename Data>
void LListHeader<Data>::addToEnd (const Data& value)
{
  LListNode<Data>* newNode
     = new LListNode<Data>(value, NULL);
```

If the list is not empty, we will have to find a pointer to the last node currently in the list.

```
⋮
if (first == NULL)
  {
    first = newNode;
  }
else
  {
    // Move to last node
    LListNode<Data>* current = first;
⋮
```



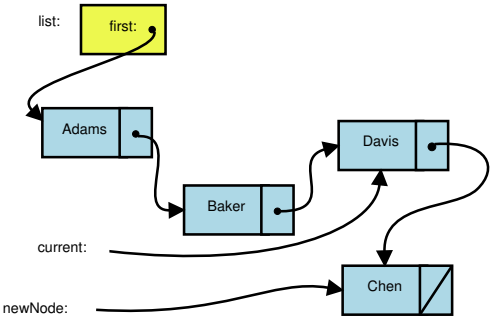

```
    ⋮
    // Move to last node
    LListNode<Data>* current = first;
    while (current->next != NULL)
      current = current->next;
    ⋮
```



Eventually we find it.

and then link in the new node.

```
    ⋮
     while (current->next != NULL)
       current = current->next;

     // Link after that node
     current->next = newNode;
}
```



Run the Demo

.....

2.3.1 addInOrder

addInOrder

```
template <typename Data>
void LListHeader<Data>::addInOrder (const Data& value)
{
  if (first == NULL)
    first = new LListNode<Data>(value, NULL);
  else
    {
      LListNode<Data>* current = first;
      LListNode<Data>* prev = NULL;
      while (current != NULL && value > current->data)
        {
          prev = current;
          current = current->next;
        }
      // Add between prev and current
```


```
      if (prev == NULL)
        addToFront (value);
      else
        addAfter (prev, value);
    }
}
```

.....

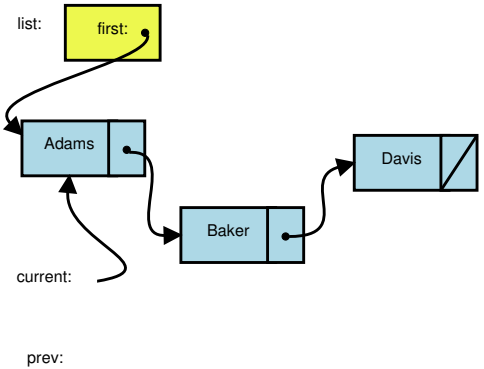
addInOrder Example

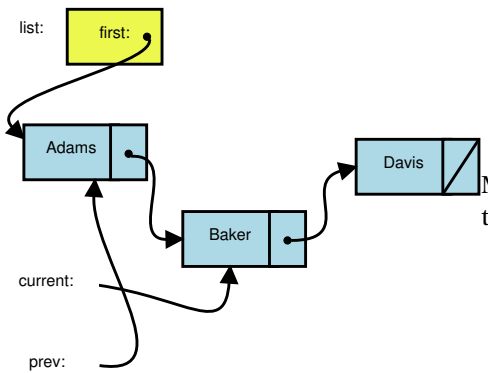
Again, suppose we have a list containing "Adams", "Baker", & "Davis", and we are going to add "Chen".

```
template <typename Data>
void LListHeader<Data>::addInOrder (const Data& value)
{
  if (first == NULL)
    first = new LListNode<Data>(value, NULL);
  else
    {
      LListNode<Data>* current = first;
      LListNode<Data>* prev = NULL;
       ⋮
```


Start by setting up two position pointers.

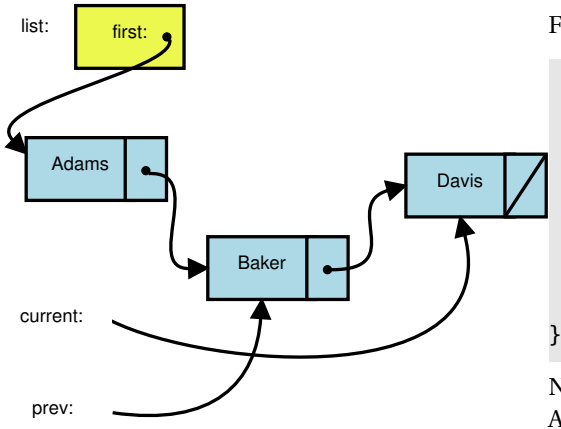
- *prev* will point one position in front of *current*.





Move forward till *prev* and *current* "bracket" the place where we want to insert the new data.

```
⋮
    while (current != NULL && value > current->data)
      {
        prev = current;
        current = current->next;
      }
```

Found it!

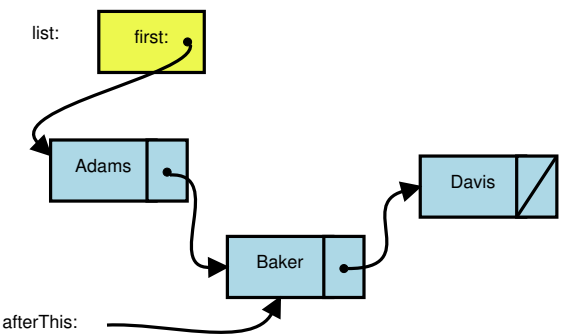
```
      :
        }
      // Add between prev and current
      if (prev == NULL)
        addToFront (value);
      else
        addAfter (prev, value);
    }
}
```

Now we just add the new data *after* the *prev* position.
And that's one of the functions we need to implement anyway.

2.3.2 addAfter

addAfter

Starting from here




```
template <typename Data>
void LListHeader<Data>::addAfter (LListNode<Data>* afterThis, const Data& value)
{
```


```
  LListNode<Data>* newNode = new LListNode<Data>(value, afterThis->next);
  afterThis->next = newNode;
}
```

.....

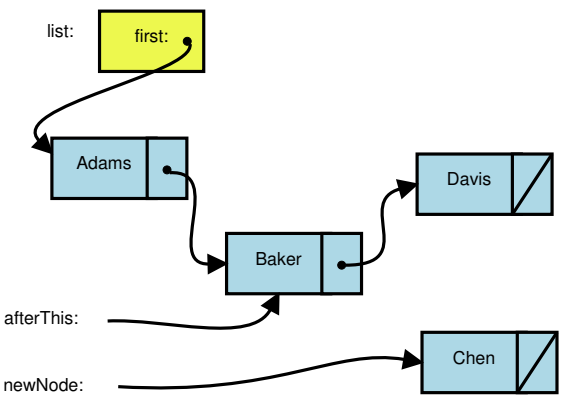
addAfter 2

Create the new node:

```
template <typename Data>
void LListHeader<Data>::addAfter (LListNode<Data>* afterThis,
                                  const Data& value)
{
  LListNode<Data>* newNode = new
        LListNode<Data>(value, afterThis->next);
  afterThis->next = newNode;
}
```

Notice that it is created with its link field already pointing to the correct place.

- That's not magic – we just passed the appropriate value to the node constructor.

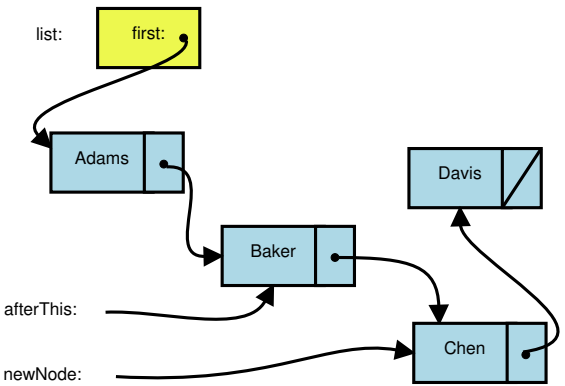


.....

addAfter 3

```
template <typename Data>
void LListHeader<Data>::addAfter (LListNode<Data>* afterThis,
                                  const Data& value)
{
  LListNode<Data>* newNode = new
        LListNode<Data>(value, afterThis->next);
```

Then change the link from the *afterThis* node.



And we're done!

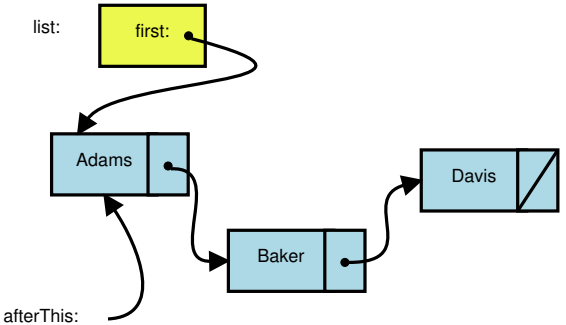
. . . . . .

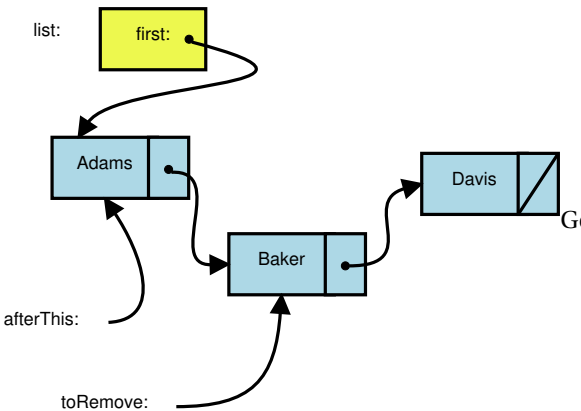
2.4 Removing from a Linked List

Removing from a Linked List

```
template <typename Data>
void LListHeader<Data>::removeAfter (LListNode<Data>* afterThis)
{
  LListNode<Data>* toRemove = afterThis->next;
  afterThis->next = toRemove->next;
  delete toRemove;
}
```

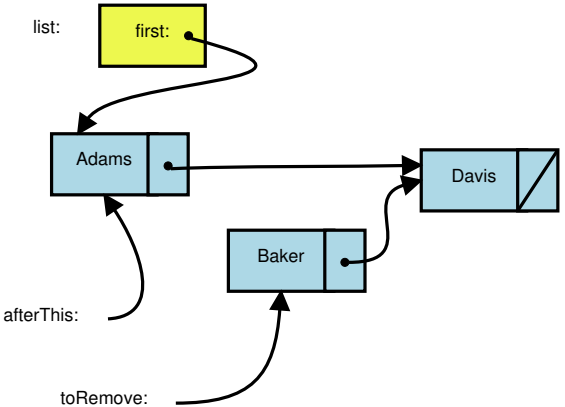
Starting here.

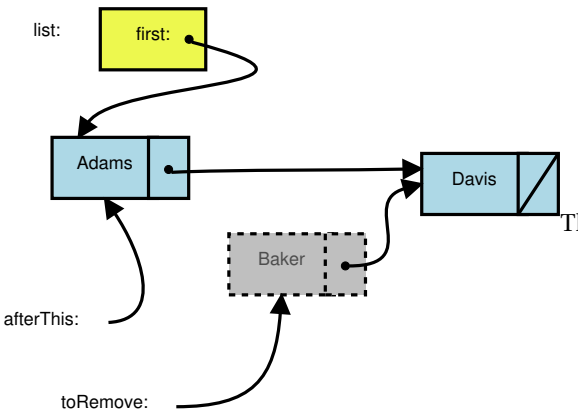




Get a pointer to the node we actually want to remove.

Make the earlier node point “around” the one we want to remove.





Then delete the unwanted node.

. . . . . .

2.4.1 remove

remove

```
template <typename Data>
void LListHeader<Data>::remove (LListNode<Data>* here)
{
  if (here == first)
    {
      LListNode<Data>* after = first->next;
      delete first;
      first = after;
    }
  else
    {
      LListNode<Data>* prev = first;
      while (prev->next != here)
        prev = prev->next;
      prev->next = here->next;
      delete here;
    }
}
```

Basically `removeAfter` preceded by a traversal.

.....

2.5 Copying and Clean-up

Copying

```
BidderCollection::BidderCollection
  (const BidderCollection& bc)
  : size(bc.size)
{
  list.append(bc.list);
}

BidderCollection& BidderCollection::operator=
```


```
  (const BidderCollection& bc)
{
  if (this != &bc)
    {
      list.clear();
      size = bc.size;
      list.append(bc.list);
    }
  return *this;
}
```

.....

append

```
// Add all values from another list onto the end of this one
template <typename Data>
void LListHeader<Data>::append (const LListHeader<Data>& list)
{
  // Move to last node
  LListNode<Data>* last = first;
  while (last->next != NULL)
    last = last->next;

  // Append new nodes onto end of list
  const LListNode<Data>* current = list.first;
  while (current != NULL)
    {
      LListNode<Data>* newNode = new LListNode<Data>(current->data, NULL);
      if (last != NULL)
          last->next = newNode;
      last = newNode;
```


```
      current = current->next;
    }
}
template <typename Data>
void LListHeader<Data>::append (const LListHeader<Data>& list)
{
  // Move to last node
  LListNode<Data>* last = first;
  if (last != NULL)
    while (last->next != NULL)
      last = last->next;

  // Append new nodes onto end of list
  const LListNode<Data>* current = list.first;
  while (current != NULL)
    {
      LListNode<Data>* newNode = new LListNode<Data>(current->data, NULL);
      if (last != NULL)
    last->next = newNode;
      else
    first = newNode;
      last = newNode;
      current = current->next;
    }

}
```

A traversal to the end of the current list, followed by repeated "addToEnd" equivalents.

. . . . . .

Collection Destructor

```
BidderCollection::~BidderCollection ()
{
  list.clear();
}
```

.....

clear

```
template <typename Data>
void LListHeader<Data>::clear()
{
  LListNode<Data>* current = first;
  LListNode<Data>* nxt = NULL;
  while (current != NULL)
    {
      nxt = current->next;
      delete current;
      current = nxt;
    }
  first = NULL;
}
```

.....

Be careful when deleting

Only "trick" here is that we cant do the usual

```
current = current->next;
```

at the end of the loop, because we will have already deleted the node that contains `next`.

.....

Example: The List-based Collection

```
#ifndef BIDDERCOLLECTION_H
#define BIDDERCOLLECTION_H

#include "bidders.h"
#include <iostream>
#include "sllistUtils.h"

//
//  Bidders Registered for auction
//


class BidderCollection {

  int size;
  LListHeader<Bidder> list;

public:

  typedef LListNode<Bidder>* Position;

  /**
   * Create a collection capable of holding any number of items
   */
  BidderCollection ();

  // Big 3
  BidderCollection (const BidderCollection&);
  BidderCollection& operator= (const BidderCollection&);
```

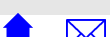


```
~BidderCollection ();



// Access to attributes
int getSize() const {return size;}



// Access to individual elements

const Bidder& get(Position index) const;
Bidder& get(Position index);

Position getFirst() const;
bool more (Position afterThis) const;
Position getNext(Position afterThis) const;

// Collection operations


void add (const Bidder& value);
//  Adds this bidder to the collection at an unspecified position.

//Pre: getSize() < getMaxSize()



void remove (Position);
// Remove the item at the indicated position
```


```
  Position findBidder (std::string name) const;
  // Returns the position where a bidde mathcing the given
  // name can be found, or null if no bidder with that name exists.



  /**
   * Read all items from the indicated file
   */
  void readBidders (std::string fileName);

  // Print the collection
  void print (std::ostream& out) const;


  // Comparison operators
  bool operator== (const BidderCollection&) const;
  bool operator< (const BidderCollection&) const;


};

inline
std::ostream& operator<< (std::ostream& out, const BidderCollection& b)
{
  b.print(out);
  return out;
}

#endif
```


The above code makes use of some utility functions for singly-linked lists:

```
#ifndef SLLISTUTILS_H
#define SLLISTUTILS_H

#include <cstdlib>  // for NULL


/**
 *  Utility operations for singly linked lists
 *
 */




template <typename Data>
struct LListNode
{
  Data data;
  LListNode<Data>* next;

  LListNode() {next = 0;}
  LListNode (const Data& d, LListNode<Data>* nxt = 0)
    : data(d), next(nxt)
  {}
};
```

```
template <typename Data>
struct LListHeader {

  LListNode<Data>* first;

  LListHeader();

  void addToFront (const Data& value);
  void addToEnd (const Data& value);

  // Add value after the indicated position
  void addAfter (LListNode<Data>* afterThis, const Data& value);

  // Add value before the indicated position
  void addBefore (LListNode<Data>* beforeThis, const Data& value);

  // Add value in sorted order.
  //Pre: all existing values are already ordered
  void addInOrder (const Data& value);

  // Remove value at the indicated position
  void remove (LListNode<Data>* here);

  // Add value after the indicated position
  void removeAfter (LListNode<Data>* afterThis);


  // Search for a value. Returns null if not found
  LListNode<Data>* find (const Data& value) const;
```


```
  // Search an ordered list for a value. Returns null if not found
  LListNode<Data>* findOrdered (const Data& value) const;


  // Empty the list
  void clear();

};



template <typename Data>
LListHeader<Data>::LListHeader()
  : first(NULL)
{}

template <typename Data>
void LListHeader<Data>::addToFront (const Data& value)
{
  LListNode<Data>* newNode = new LListNode<Data>(value, first);
  first = newNode;
}

template <typename Data>
void LListHeader<Data>::addToEnd (const Data& value)
{
  LListNode<Data>* newNode = new LListNode<Data>(value, NULL);
  if (first == NULL)
    {
      first = newNode;
    }
```


```
  else
    {
      // Move to last node
      LListNode<Data>* current = first;
      while (current->next != NULL)
    current = current->next;

      // Link after that node
      current->next = newNode;
    }
}

// Add value after the indicated position
template <typename Data>
void LListHeader<Data>::addAfter (LListNode<Data>* afterThis, const Data& value)
{
  LListNode<Data>* newNode = new LListNode<Data>(value, afterThis->next);
  afterThis->next = newNode;
}

  // Add value before the indicated position
template <typename Data>
void LListHeader<Data>::addBefore (LListNode<Data>* beforeThis, const Data& value)
{
  if (beforeThis == first)
    addToFront (value);
  else
    {
      // Move to front of beforeThis
      LListNode<Data>* current = first;
      while (current->next != beforeThis)
```


```
    current = current->next;

      // Link after that node
      addAfter (current, value);
    }
}

// Add value in sorted order.
//Pre: all existing values are already ordered
template <typename Data>
void LListHeader<Data>::addInOrder (const Data& value)
{
  if (first == NULL)
    first = new LListNode<Data>(value, NULL);
  else
    {
      LListNode<Data>* current = first;
      LListNode<Data>* prev = NULL;
      while (current != NULL && value < current->data)
    {
      prev = current;
      current = current->next;
    }
      // Add between prev and current
      if (prev == NULL)
    addToFront (value);
      else
    addAfter (prev, value);
    }
}
```


```
// Add all values from another list onto the end of this one
template <typename Data>
void LListHeader<Data>::append (const LListHeader<Data>& list)
{
  // Move to last node
  LListNode<Data>* last = first;
  while (last->next != NULL)
    last = last->next;

  // Append new nodes onto end of list
  const LListNode<Data>* current = list.first;
  while (current != NULL)
    {
      LListNode<Data>* newNode = new LListNode<Data>(current->data, NULL);
      if (last != NULL)
      last->next = newNode;
      last = newNode;
    }
}

// Remove value at the indicated position
template <typename Data>
void LListHeader<Data>::remove (LListNode<Data>* here)
{
  if (here == first)
    {
      LListNode<Data>* after = first->next;
      delete first;
      first = after;
    }
```


```
  else
    {
      LListNode<Data>* prev = first;
      while (prev->next != here)
    prev = prev->next;
      prev->next = here->next;
      delete here;
    }
}



  // Remove value after the indicated position
template <typename Data>
void LListHeader<Data>::removeAfter (LListNode<Data>* afterThis)
{
  LListNode<Data>* toRemove = afterThis->next;
  afterThis->next = toRemove->next;
  delete toRemove;
}


  // Search for a value. Returns null if not found
template <typename Data>
LListNode<Data>* LListHeader<Data>::find (const Data& value) const
{
  LListNode<Data>* current = first;
  while (current != NULL && value != current->data)
    current = current->next;
  return current;
}
```


```
  // Search an ordered list for a value. Returns null if not found
template <typename Data>
LListNode<Data>* LListHeader<Data>::findOrdered (const Data& value) const
{
  LListNode<Data>* current = first;
  while (current != NULL && value < current->data)
    current = current->next;
  if (current != NULL && value == current->data)
    return current;
  else
    return NULL;
}

template <typename Data>
void LListHeader<Data>::clear()
{
  LListNode<Data>* current = first;
  LListNode<Data>* nxt = NULL;
  while (current != NULL)
    {
      nxt = current->next;
      delete current;
      current = nxt;
    }
  first = NULL;
}

#endif
```


3 Variations: Headers with First and Last

Adding on Either End

Adding to (either) end of a list is very common, but compare the amount of work required:

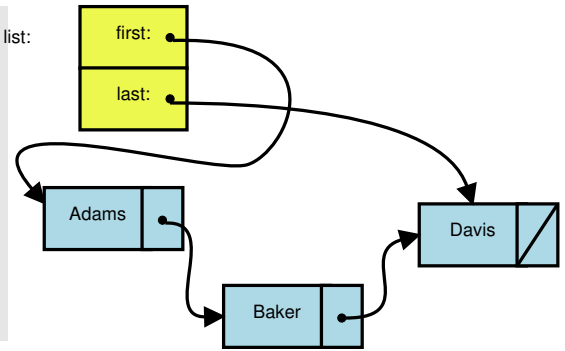
```
template <typename Data>
void LListHeader<Data>::addToFront (const Data& value)
{
  LListNode<Data>* newNode = new LListNode<Data>(value, first);
  first = newNode;
}

template <typename Data>
void LListHeader<Data>::addToEnd (const Data& value)
{
  LListNode<Data>* newNode = new LListNode<Data>(value, NULL);
  if (first == NULL)
    {
      first = newNode;
    }
  else
    {
      // Move to last node
      LListNode<Data>* current = first;
      while (current->next != NULL)
        current = current->next;

      // Link after that node
      current->next = newNode;
    }
}
```

.....

Adding a Last Pointer

We can simplify by adding a second pointer in the header:

```
template <typename Data>
struct LListHeader {

  LListNode<Data>* first;
  LListNode<Data>* last;

  LListHeader();
    ⋮
```



Look at the Change

```
template <typename Data>
void LListHeader<Data>::addToFront (const Data& value)First−Last Header Algorithms
{
  LListNode<Data>* newNode = new LListNode<Data>(value, first);
  first = newNode;
  if (last == NULL)
    last = first;
}

template <typename Data>
void LListHeader<Data>::addToEnd (const Data& value)
{
  LListNode<Data>* newNode = new LListNode<Data>(value, NULL);
  if (last == NULL)
    {
      first = last = newNode;
```


```
    }
  else
    {
      last->next = newNode;
      last = newNode;
    }
}
```

.....

3.0.1 First-Last Header Algorithms

- ```
  #ifndef SLLISTUTILS_H
  #define SLLISTUTILS_H

  #include <cstdlib>  // for NULL


  /**
   *  Utility operations for singly linked lists
   *
   *  Variant: header has first and last pointers
   *
   */




  template <typename Data>
  struct LListNode
  {
    Data data;
  ```



  ```

```
  LListNode<Data>* next;

  LListNode() {next = 0;}
  LListNode (const Data& d, LListNode<Data>* nxt = 0)
    : data(d), next(nxt)
  {}
};


template <typename Data>
struct LListHeader {

  LListNode<Data>* first;
  LListNode<Data>* last;

  LListHeader();

  void addToFront (const Data& value);
  void addToEnd (const Data& value);

  // Add value after the indicated position
  void addAfter (LListNode<Data>* afterThis, const Data& value);

  // Add value before the indicated position
  void addBefore (LListNode<Data>* beforeThis, const Data& value);

  // Add value in sorted order.
  //Pre: all existing values are already ordered
  void addInOrder (const Data& value);

  // Append all nodes from another list onto the end of this one
```


```
  void append (const LListHeader<Data>& list);

  // Remove value at the indicated position
  void remove (LListNode<Data>* here);

  // Add value after the indicated position
  void removeAfter (LListNode<Data>* afterThis);


  // Search for a value. Returns null if not found
  LListNode<Data>* find (const Data& value) const;

  // Search an ordered list for a value. Returns null if not found
  LListNode<Data>* findOrdered (const Data& value) const;


  // Empty the list
  void clear();

};



template <typename Data>
LListHeader<Data>::LListHeader()
  : first(NULL), last(NULL)
{}

template <typename Data>
void LListHeader<Data>::addToFront (const Data& value)
{
```


```
  LListNode<Data>* newNode = new LListNode<Data>(value, first);
  first = newNode;
  if (last == NULL)
    last = first;
}

template <typename Data>
void LListHeader<Data>::addToEnd (const Data& value)
{
  LListNode<Data>* newNode = new LListNode<Data>(value, NULL);
  if (last == NULL)
    {
      first = last = newNode;
    }
  else
    {
      last->next = newNode;
      last = newNode;
    }
}

// Add value after the indicated position
template <typename Data>
void LListHeader<Data>::addAfter (LListNode<Data>* afterThis,
       const Data& value)
{
  LListNode<Data>* newNode = new LListNode<Data>(value, afterThis->next);
  afterThis->next = newNode;
  if (afterThis == last)
    last = newNode;
}
```


```
  // Add value before the indicated position
template <typename Data>
void LListHeader<Data>::addBefore (LListNode<Data>* beforeThis,
       const Data& value)
{
  if (beforeThis == first)
    addToFront (value);
  else
    {
      // Move to front of beforeThis
      LListNode<Data>* current = first;
      while (current->next != beforeThis)
    current = current->next;

      // Link after that node
      addAfter (current, value);
    }
}

// Add value in sorted order.
//Pre: all existing values are already ordered
template <typename Data>
void LListHeader<Data>::addInOrder (const Data& value)
{
  if (first == NULL)
    first = last = new LListNode<Data>(value, NULL);
  else
    {
      LListNode<Data>* current = first;
      LListNode<Data>* prev = NULL;
```


```
      while (current != NULL && value > current->data)
    {
      prev = current;
      current = current->next;
    }
      // Add between prev and current
      if (prev == NULL)
    addToFront (value);
      else
    addAfter (prev, value);
    }
}

// Add all values from another list onto the end of this one
template <typename Data>
void LListHeader<Data>::append (const LListHeader<Data>& list)
{
  const LListNode<Data>* current = list.first;
  while (current != NULL)
    {
      addToEnd (current->data);
      current = current->next;
    }
}


// Remove value at the indicated position
template <typename Data>
void LListHeader<Data>::remove (LListNode<Data>* here)
{
  if (here == first)
```


```
    {
      LListNode<Data>* after = first->next;
      delete first;
      first = after;
    }
  else
    {
      LListNode<Data>* prev = first;
      while (prev->next != here)
    prev = prev->next;
      prev->next = here->next;
      if (here == last)
    last = prev;
      delete here;
    }
}

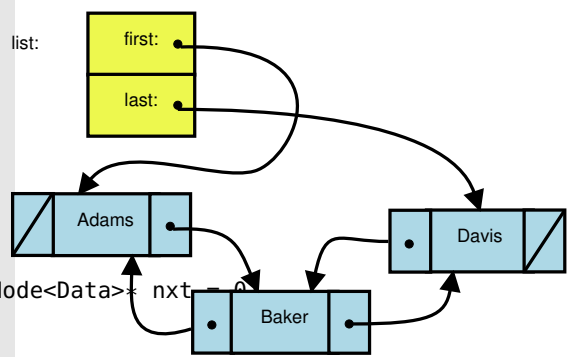


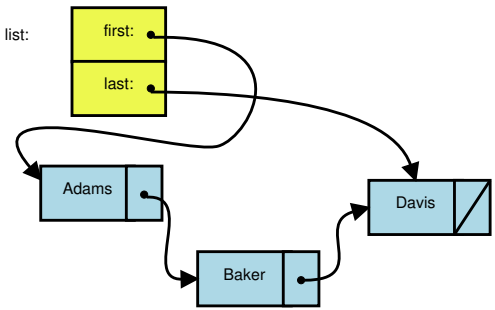
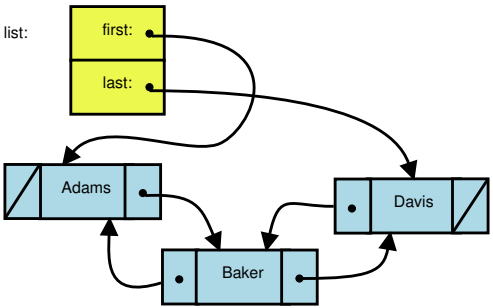
  // Add value after the indicated position
template <typename Data>
void LListHeader<Data>::removeAfter (LListNode<Data>* afterThis)
{
  LListNode<Data>* toRemove = afterThis->next;
  afterThis->next = toRemove->next;
  if (toRemove == last)
    last = afterThis;
  delete toRemove;
}
```


```
  // Search for a value. Returns null if not found
template <typename Data>
LListNode<Data>* LListHeader<Data>::find (const Data& value) const
{
  LListNode<Data>* current = first;
  while (current != NULL && value != current->data)
    current = current->next;
  return current;
}

  // Search an ordered list for a value. Returns null if not found
template <typename Data>
LListNode<Data>* LListHeader<Data>::findOrdered (const Data& value) const
{
  LListNode<Data>* current = first;
  while (current != NULL && value > current->data)
    current = current->next;
  if (current != NULL && value == current->data)
    return current;
  else
    return NULL;
}

template <typename Data>
void LListHeader<Data>::clear()
{
  LListNode<Data>* current = first;
  LListNode<Data>* nxt = NULL;
  while (current != NULL)
    {
      nxt = current->next;
```


```
      delete current;
      current = nxt;
    }
  first = last = NULL;
}

#endif
```

4 Variations: Doubly-Linked Lists

Doubly-Linked Lists

By modifying the node structure:

```
template <typename Data>
struct DListNode
{
  Data data;
  DListNode<Data>* prev;
  DListNode<Data>* next;

  DListNode() {next = prev = 0;}
  DListNode (const Data& d, DListNode<Data>* prv = 0, DListNode<Data>* nxt = 0
    : data(d), next(nxt), prev(prv)
  {}
};
```



we can

- Move backwards as well as forward in the list
  - by following the `prev` pointers

- Easily add in front of a node

addBefore: Singly Linked

```
template <typename Data>
void LListHeader<Data>::addBefore (LListNode<Data>* beforeThis,
                                   const Data& value)
{
  if (beforeThis == first)
    addToFront (value);
  else
    {
      // Move to front of beforeThis
      LListNode<Data>* current = first;
      while (current->next != beforeThis)
        current = current->next;

      // Link after that node
      addAfter (current, value);
    }
}
```



addBefore: Doubly Linked

```
template <typename Data>
void DListHeader<Data>::addBefore (DListNode<Data>* beforeThis,
                                   const Data& value)
{
  if (beforeThis == first)
    addToFront (value);
  else
    {
      // Move to front of beforeThis
      DListNode<Data>* current
          = beforeThis->prev;
      // Link after that node
      addAfter (current, value);
    }
}
```



4.1 Doubliy-Linked List Algorithms

- ```
  #ifndef DLLISTUTILS_H
  #define DLLISTUTILS_H

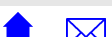
  #include <cstdlib>  // for NULL


  /**
   *  Utility operations for doubly linked lists
   *
   *  Variant: header has first and last pointers
   *
   */
  ```
  ```

```
template <typename Data>
struct DListNode
{
  Data data;
  DListNode<Data>* prev;
  DListNode<Data>* next;

  DListNode() {next = prev = 0;}
  DListNode (const Data& d, DListNode<Data>* prv = 0, DListNode<Data>* nxt = 0)
    : data(d), next(nxt), prev(prv)
  {}
};


template <typename Data>
struct DListHeader {

  DListNode<Data>* first;
  DListNode<Data>* last;

  DListHeader();

  void addToFront (const Data& value);
  void addToEnd (const Data& value);

  // Add value after the indicated position
  void addAfter (DListNode<Data>* afterThis, const Data& value);
```


```
  // Add value before the indicated position
  void addBefore (DListNode<Data>* beforeThis, const Data& value);

  // Add value in sorted order.
  //Pre: all existing values are already ordered
  void addInOrder (const Data& value);

  // Remove value at the indicated position
  void remove (DListNode<Data>* here);

  // Add value after the indicated position
  void removeAfter (DListNode<Data>* afterThis);


  // Search for a value. Returns null if not found
  DListNode<Data>* find (const Data& value) const;

  // Search an ordered list for a value. Returns null if not found
  DListNode<Data>* findOrdered (const Data& value) const;


  // Empty the list
  void clear();

};



template <typename Data>
DListHeader<Data>::DListHeader()
  : first(NULL), last(NULL)
```


```
{}

template <typename Data>
void DListHeader<Data>::addToFront (const Data& value)
{
  DListNode<Data>* newNode = new DListNode<Data>(value, NULL, first);
  if (first == NULL)
    first = last = newNode;
  else
    {
      first->prev = newNode;
      first = newNode;
    }
}

template <typename Data>
void DListHeader<Data>::addToEnd (const Data& value)
{
  DListNode<Data>* newNode = new DListNode<Data>(value, last, NULL);
  if (last == NULL)
    {
      first = last = newNode;
    }
  else
    {
      last->next = newNode;
      last = newNode;
    }
}

// Add value after the indicated position
```

```
template <typename Data>
void DListHeader<Data>::addAfter (DListNode<Data>* afterThis,
       const Data& value)
{
  DListNode<Data>* newNode
    = new DListNode<Data>(value, afterThis, afterThis->next);
  if (afterThis == last)
    {
      last = newNode;
    }
  else
    {
      afterThis->next->prev = newNode;
      afterThis->next = newNode;
    }
}

  // Add value before the indicated position
template <typename Data>
void DListHeader<Data>::addBefore (DListNode<Data>* beforeThis,
        const Data& value)
{
  if (beforeThis == first)
    addToFront (value);
  else
    {
      // Move to front of beforeThis
      DListNode<Data>* current = beforeThis->prev;
      // Link after that node
      addAfter (current, value);
    }
```


```
}

// Add value in sorted order.
//Pre: all existing values are already ordered
template <typename Data>
void DListHeader<Data>::addInOrder (const Data& value)
{
  if (first == NULL)
    first = last = new DListNode<Data>(value, NULL, NULL);
  else
    {
      DListNode<Data>* current = first;
      DListNode<Data>* prev = NULL;
      while (current != NULL && value < current->data)
    {
      prev = current;
      current = current->next;
    }
      // Add between prev and current
      if (prev == NULL)
    addToFront (value);
      else
    addAfter (prev, value);
    }
}

// Remove value at the indicated position
template <typename Data>
void DListHeader<Data>::remove (DListNode<Data>* here)
{
  if (here == first)
```


```
    {
      DListNode<Data>* after = first->next;
      delete first;
      first = after;
    }
  else
    {
      DListNode<Data>* prev = here->prev;
      prev->next = here->next;
      here->prev = prev;
      if (here == last)
    last = prev;
      delete here;
    }
}



  // Add value after the indicated position
template <typename Data>
void DListHeader<Data>::removeAfter (DListNode<Data>* afterThis)
{
  DListNode<Data>* toRemove = afterThis->next;
  afterThis->next = toRemove->next;
  if (afterThis->next != NULL)
    afterThis->next->prev = afterThis;
  if (toRemove == last)
    last = afterThis;
  delete toRemove;
}
```


```
  // Search for a value. Returns null if not found
template <typename Data>
DListNode<Data>* DListHeader<Data>::find (const Data& value) const
{
  DListNode<Data>* current = first;
  while (current != NULL && value != current->data)
    current = current->next;
  return current;
}

  // Search an ordered list for a value. Returns null if not found
template <typename Data>
DListNode<Data>* DListHeader<Data>::findOrdered (const Data& value) const
{
  DListNode<Data>* current = first;
  while (current != NULL && value < current->data)
    current = current->next;
  if (current != NULL && value == current->data)
    return current;
  else
    return NULL;
}

template <typename Data>
void DListHeader<Data>::clear()
{
  DListNode<Data>* current = first;
  DListNode<Data>* nxt = NULL;
  while (current != NULL)
    {
```


```
      nxt = current->next;
      delete current;
      current = nxt;
    }
  first = last = NULL;
}

#endif
```