CMSC 341

Lists 3

Doubly-Linked Lists

Option: add pointer to previous node

Issues

- doubles number of pointers
- allows immediate (O(1)) access to previous node

Circular Linked Lists

Configurations

- header in circle

- header before circle

- no header

Doubly-Linked Circular Linked List

May or may not have a header

Useful for applications requiring fast access to head and tail

Vector Implementation

```
template <class Object>
void List<Object>::
insert(const Object &x, const ListItr<Object> &p)
{
   if (!p.isPastEnd())
   {
        // shift all elements down
        for (i = this.last; i > p.current+1; i--)
              nodes[i + 1] = nodes[i];
        // put new element in hole
        nodes[p.current + 1] = x;
   }
}
```

Cursor Implementation

Linked list look & feel

- data stored as collection of nodes: data and next
- nodes allocated and deallocated

without dynamic memory allocation

Basic concepts

- have node pool of all nodes, keep list of free ones
- use pool indices in place of pointers; 0 == NULL

Cursor List Class

```
template <class Object>
class List {
   List();
   // same old list public interface
public:
   struct CursorNode {
        CursorNode() : next(0) {};
   private:
        CursorNode(const Object &theElement, int n)
               : element(theElement), next(n) {};
        Object element;
        int next;
        friend class List<Object>;
        friend class ListItr<Object>;
   };
```

Cursor List Class (cont.)

```
…
private:
   int header;
   static vector<CursorNode> cursorSpace;
   static void initializeCursorSpace( );
   static int alloc( );
   static void free (int p);
   friend class ListItr<Object>;
};
```

Cursor Initialization

```
template <class Object>
void List<Object>::initializeCursorSpace()
{
  static int cursorSpaceIsInitialized = false;
  if (!cursorSpaceIsInitialized)
  {
      cursorSpace.resize(100);
      for(int i = 0; i < cursorSpace.size(); i++)
            cursorSpace[i].next = i + 1;
      cursorSpace[cursorSpace.size()-1].next = 0;
      cursorSpaceIsInitialized = true;
  }
}
```

cursorSpace

Cursor Allocation

```
template <class Object>
void List<Object>::alloc()
{
  int p = cursorSpace[0].next;
  cursorSpace[0].next = cursorSpace[p].next;
  return p;
}

template <class Object>
void List<Object>::free(int p)
{
  cursorSpace[p].next = cursorSpace[0].next;
   cursorSpace[0].next = p;
}
```

Cursor Implementation (cont.)

```
template <class Object>
ListItr<Object> List<Object>::
find(const Object &x) const
{
  int itr = cursorSpace[header].next;
  while (itr!=0 && cursorSpace[itr].element != x)
       itr = cursorSpace[itr].next;

  return ListItr<Object>(itr);
}
```

Cursor Implementation (cont.)

```
template <class Object>
void List<Object>::
insert(const Object &x, const ListItr<Object> &p)
{
  if (!p.isPastEnd())
  {
      int pos = p.current;
      int tmp = alloc();
      cursorSpace[tmp] =
            CursorNode(x, cursorSpace[pos].next);
      cursorSpace[pos].next = tmp;
  }
}
```

Cursor Implementation (cont.)

```
template <class Object>
void List<Object>::
remove(const Object &x)
{
  ListItr<Object> p = findPrevious(x);
  int pos= p.current;

  if (cursorSpace[pos].next != 0)
  {
      int tmp = cursorSpace[pos].next;
      cursorSpace[pos].next =
                  cursorSpace[tmp].next;
      free (tmp);
  }
}
```

Comparing Performance

	Linear	S Linked	D Linked	Cursor
constructor	O(1)	O(1)	O(1)	O(1)
find	O(n)	O(n)	O(n)	O(n)
findPrev	O(n)	O(n)	O(n)	O(n)
insert	O(n)	O(1)	O(1)	O(1)
remove	O(n)	O(n)	O(n)	O(n)
makeEmpty	O(1)	O(n)	O(n)	O(n)