Linked Lists

• Dynamic Memory Allocation
• Programming with Linked Lists
  – Various operations on linked list.
  – Pointer-Based implementation
  – Linked List Processing
• Variations on Linked Lists.
• Applications of Linked Lists.
Array-Based List

• What are the problems?
  – Fixed Size and hence can’t handle large lists.
  – Insert or Delete involves shifting which is time-consuming
  – Retrieving an item is however efficient.
  – It is also easy to implement.
  – The code is quite readable.
  – Not suitable if insert or delete are the operations that occur most often.
Pointers

- Compiler allocates memory cell for every variable in a function when the function gets executed.
- A pointer variable is capable of storing the address of a memory cell.
- Pointer are used to locate memory cells.
- Each memory cell has associated type.
- The memory cell for an int variable can store an integer.
Pointers (contd)

- To point to an int memory cell, we need an int pointer.
- To point to a double memory cell, we need a double pointer.
- A pointer variable itself needs a memory cell in which we store the address of another memory cell. (indirection)
- The address operator & can be used to get the address of any variable. (even pointer)
int x;

int* p = &x; // same as int *p;

*p = 75; // same as x = 75
Pointers (contd)

- The memory space allocated to variables are done at compile time and hence are known as *static* memory.
- You can also allocate memory dynamically using the operator `new`. You need to indicate what type of memory cell you need (required) and how much (optional).
- To free a memory allocated using `new`, you can use the operator `delete`. 
Another Pointer Example

```c
int* p;
int* q;
int  x;

p  = &x;
*p = 6;
p  = new int;
*p = 7;
q  = p;
q  = new int;
*q = 8;
p  = NULL; // Some programmers prefer 0.
delete q;
```
Pointer Facts

• When you declare a pointer variable, its initial value is garbage. Do not dereference.
• NULL is a special value & points to nothing
• A pointer variable can either point to static memory or dynamic memory.
• Every dynamic memory allocated should be deleted eventually.
• If you loose the address of a dynamic memory, there is no way to delete it.
Memory Leak

- If a program is buggy and allocates dynamic memory but does not free them, eventually there will be no more memory available.
- If you loose the address of a dynamically allocated memory, it is garbage.
- Some languages (lisp, java) collects garbage automatically, but C++ does not.
- Pointer variables should be handled extremely carefully. Programmer’s headache.
Pointer Usage

- It is possible to make a pointer variable point to deallocated storage. Avoid this.

```cpp
int* p;
int* q;

p = new int;
q = p;
delete p; // Where does q point to?
```
typedef int* IntPtr;
IntPtr p; // Another way to declare
int x = 0;

*p = 5; // invalid. why?
p = &x;
*p = *p + 1; // x is now 1.
delete p; // invalid. why?
p = new int;
delete p;
*p = 5; // invalid. Why?
Dynamic Array Allocation

• You can allocate more than one memory cell of any type with new.
• When you free such memory, you MUST indicate to delete that you are freeing array.

```cpp
int* p;

p = new int[10]; // p is array now.
p[0] = 4; p[1] = 7;
delete [] p;
```
Pointer-Based Linked List

• Linked List contains components that are liked together.
• Each component is called a node - containing both data item and a pointer to the next node.
• The last item will point to nothing. NULL
• You need a pointer variable to remember the address of the first node. Head of List.
• The head will be NULL if the list is empty.
inserting new item is easy.
Node Structure

typedef int ListItem;
struct ListNode; // defined later.
typedef ListNode* ListNodePtr;
struct ListNode
{
    ListItem    item;
    ListNodePtr next;
};

ListNodePtr head;
List Creation

ListNodePtr head = NULL;

head = new ListNode;
head->item = 7;
head->next = NULL; // one item list

ListNodePtr temp = new ListNode;
temp->item = 8;
temp->next = NULL;
head->next = temp; // 2 item list.
List Analysis

• What is List? Is it just a pointer?
• How does client create a list?
  – typedef ListNodePtr List; // in list.h
  – List myList; // in main program.
• Does not support Wall? Client owns the data and can access it.
• Need to call some function to initialize myList. Else, it is uninitialized.
• Solution: Use class.
Displaying a List

// pseudocode
curptr <- first node in list.
while (curptr is not NULL)
{
    display data in curr node.
    set curptr to next of curr node.
}

Display List in C++

// Display the data in a linked list pointed to by head.
// loop invariant: cur points to the next node to be displayed.
ListNodePtr cur;
for (cur = head; cur != NULL; cur = cur->next)
{
    cout << cur->item << endl;
}
Deleting a node in a list

• You need to locate the node to be deleted and the previous node (if it exists).
• Deleting the first node is a special case that should be handled differently.
• You need prev so that we can unlink the node pointed to by cur.
• You should remember to delete the memory space occupied by cur.
Deleting an interior node

Delete this node

Prev
Cur

NULL

head

5
item
next

1
item
next

8
item
next

Delete this node

COP3530 Data Structures
420
Deleting the first node

Delete this node
Deletion Code (Method 1)

// First find prev and cur.
if (prev == NULL)
{
    head = cur->next;
    delete cur;
}
else
{
    prev->next = cur->next;
    delete cur;
}
Deleteion Code (Method 2)

// Instead of prev being NULL, make it
// point to the ptr to be changed.
ListNodePtr *prev = &head;
ListNodePtr cur = head;
// Move prev and cur as you find node to
// be deleted. prev points to next field.
// e.g. prev = &cur->next; cur=cur->next;
*prev = cur->next;
delete cur;
// Note that there is no special case
// to consider in this method.
Inserting at front

a, b, c: Shows the order.
Inserting in middle

a,b,c: Shows the order.
Insertion Code (Method 1)

// Insert before cur, after prev.
// cur can be NULL (Inserting at end).
newPtr = new ListNode;
newPtr->item = newItem;
newPtr->next = cur;
if (prev == NULL)
{
    head = newPtr;
}
else
{
    prev->next = newPtr;
}

Insertion Code (Method 2)

// First find prev and cur.
// Note that prev should point to the
// pointer to be changed.
// e.g. prev = &cur->next;
//      cur = cur->next;

newPtr = new ListNode;
// You should check for newPtr == NULL.
newPtr->item = newItem;
newPtr->next = cur;
*prev        = newPtr;
Determining Prev and Cur

• Depends on type of insertion or deletion.
• For sorted list insert, we need to traverse the list until new item is <= cur->item.
• For position based list, traverse the list required number of times.
• Initialize prev to NULL and cur to head for method 1.
• Initialize prev to &head and cur to head for method 2.
Determining Prev & Cur (contd)

• If cur becomes NULL at any point, the traversal should stop no matter what.

// An example of code with fault.
prev = NULL;
cur   = head;
while (newValue > cur->item)
{
    prev = cur;
    cur   = cur->next;
}

// What happens if cur becomes NULL at some point?
Change to (cur != NULL && newValue > cur->item).

- The code works for deletion too but outside the loop, you need to check whether cur node contains the item to be deleted.
- Inserting at the end is NOT a special case.
- Empty list is taken care of by insertion at front of the list.
- Insertion at front or deletion of first node is a special case for method 1 only.
Another Example

• Determine prev and cur for insertion at a given position (1..listlength+1).

```c
prev = NULL;
cur = head;
for (i = 1; i < pos && cur; i++)
{
    prev = cur;
    cur = cur->next;
}
if (i != pos)
{
    // List is shorter. Handle this case.
}
```
Shallow and Deep Copy

- Shallow Copy simply copies data members
- Deep copy copies the whole structure.
Array vs pointer based lists

- Arrays are easy to use but have fixed size.
- Need to predict max # of items in ADT.
- Even dynamic array needs fixed size.
- Expanding array can be expensive.
- Array is good for small list.
- Pointer based list has no size restriction.
- Array has contiguous memory.
- Pointer based list uses non-contiguous mem
Array Vs Pointer based List (contd)

- Array based list uses less memory
- Array provides random access to items.
- With ptr based list, traversal is needed.
- Time to access i-th item depends on i.
- Insertion & Deletion needs shifting with Array but not with linked list.
Saving a List using a file.

- Pointers are useful only in RAM.
- Do not save pointers in file.
- Write only the data in the nodes.
- Writing of data needs knowledge of the data type of the item.
- Unless user of ADT has overloaded << operator, we cannot use << on the data item.
- Should saving be done in Client Program?
Saving a List (Code)

// Client Program to save the list
void SaveList(const List& list,
              const string& fileName)
{
    ofstream outFile(fileName.c_str());
    ListItem item;
    unsigned int pos;
    bool success;
}
for (pos = 1;
    pos <= list.Length();
    pos++)
{
    item = list.Retrieve(pos, item,
        success);
    outFile << item << endl;
}

outFile.close();
Reading a List from a file

void ReadList(List& list,
              const string& fileName)
{
    ifstream inFile(fileName.c_str());
    ListItem item;
    bool success;
    unsigned int pos = 1;
}
Reading a List from a file (contd)

while (inFile >> item)
{
    list.Insert(pos++, item, success);
}
list.close();

void List::Save(const string& fileName)
{
    ofstream outFile(fileName.c_str());
    ListItem item;
    ListNodePtr cur;
for (cur = head;
    cur != NULL;
    cur = cur->next)
{
    outFile << cur->item << endl;
}
outFile.close();

• Note that ListItem should have << overloaded or else it will not work!!!
Passing a List to a function

- Always pass the list by reference.
- Use const if the list is not modified by fn.
- Passing the list by value involves creation of a new object of type List (formal param)
- Passing the list by value involves the use of copy constructor.
- Depending on shallow or deep copy, the behavior of the function will be different.
Objects as list data

• You are not restricted to any type of item to be stored in list.
• ListItem can be a class type.
• If you allow both simple type (such as int) as well as class type (e.g. Person), then you can use only those operations on the item that are common to both type of objects.
• If you restrict ListItem to be a class type, then you can have a better design. Need wrapper classes for int, double etc.
Variations on Linked List

• Circular Linked List
  – Last Node points to the first node.
  – No first or last node concept.
  – We still need an entry point.
  – We normally a pointer to the last (?) node.
  – Thus, we can access first node using last node.
  – NULL value in head indicates empty list.

• When traversing a circular list, we need to be careful not to loop forever.
Doubly Linked List

- Each node has two pointers: prev, next.
- We can traverse either side from the middle.
- We need only one pointer for insert/delete.
- Needs more memory space.
- Dummy node at the beginning is useful.
- Circular Double Linked List? Why not?
- Circular doubly linked list handles special cases nicely.
Closer Look at C++

• Constructors are called whenever an object is created. (How do we know that a class got initialized properly? Use a flag in class)

• Destructors are called whenever an objects is deleted (i.e., destroyed).

• An object is created when you declare it in a function.

• An objects can also be created by the compiler to store temporary values.
Closer Look at C++ (contd)

- If a class object is created and it has member objects which are also class objects, they get created too.
- C++ calls the constructor or destructor functions for the member objects before calling the constructor or destructor for the object itself.
- If a formal is a class object that is passed by value, then copy constructor is called.
Sharing Dynamic Data

• If you have two objects that share the same dynamically allocated data values, problem arises when these objects gets destroyed. The destructor functions for both objects are called and this may create problem.

• If one object is destroyed before the other, it may end up deleting data that are still used by the other object.

• Lesson: Do not share dynamic data.
Example

List Tail()
{
    List tempList;

    if (size > 0)
    {
        tempList.size = size - 1;
        tempList.head = head->next;
    }
    return (tempList);
}
Example Analysis

main:
    List myList, tailList; bool success;

    myList.Insert(1,4, success);
    myList.Insert(1,6, success);// List is 6 4 now.
    tailList = myList.Tail(); // Not allowed. Why?
        // This needs operator=
    myList.Tail().Display(); // Display Tail List.
    myList.Display(); // Crashes, Why?

Inside Tail, tempList is created first. Empty. Then, tempList.head points to the node with 4. The return statment creates a temporary list object and invokes copy constructor. Fine. What happens when tempList was destroyed? Node 4 is gone, but node 6 still points to it!!!
Object Assignment

• Copy constructor is used when a new object is created and is assigned another object that has already been created and initialized.

• If we want to assign one object to another after both are created, we need to overload = operator. (read page 377 of book).

• What is the difference? The object being assigned needs to be cleaned up first before the actual assignment. Else, get garbage!!
Object Assignment (contd)

List firstList; bool success;
firstList.Insert(1,5, success);
firstList.Insert(1,6, success);

List secondList(firstList); // copy const
List thirdList; // Default constructor

thirdList.Insert(1,3,success);
thirdList.Insert(1,6,success);
thirdList = firstList; // Needs operator=
    // Need to cleanup thirdList first.
Object Assignment (contd)

- The function operator= needs to perform the job of destructor function followed by the job of copy constructor.
- Use an auxiliary function Destroy that can be called from the destructor function as well as operator=.
- Use an auxiliary function Copy that can be called from Copy constructor as well as from operator=.
- operator= should always check for self-assignment. Example: a = a;
- What is meant by two objects are the same? Same value? or Same memory? or some other mechanism?
Operator=

- One way of implementing operator=.

```cpp
List& List::operator=(const List& rhs)
{
    if (this == &rhs)
    {
        return (*this);
    }
    Destroy(); // destroy lhs list.
    Copy(rhs); // make lhs just like rhs.
    return (*this);
}
More later!!
```