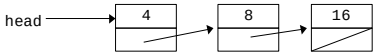
CSE 143

Linked Lists

[Chapter 4; Chapter 6, pp. 265-271]

Linked Lists

- A linked list is a collection of dynamically allocated nodes
- Each node contains at least one member (field) that points to another node in the list.
- In the simplest case, each node has two members: a **data** item and a **link** field which points to the successor node.
- Example: a list of 3 integers:



Creating Nodes

- First we'll declare a struct to represent a node:

```
struct Node {
  int data;
  Node* next;
};
```

- Now we can create new nodes:

```
Node* p;
p = new Node;
p->data = 100;    // shorthand for: (*p).data = 100
p->next = NULL;   // shorthand for: (*p).next = NULL
```


Manipulating Nodes (1)

- Draw the picture that results from the following code:

```
Node* front;
Node* temp;

front = new Node;
front->data = 1;
```


Manipulating Nodes (2)

```
front->next = new Node;  // add a node
front->next->data = 2;
front->next->next = NULL;
```


Manipulating Nodes (3)

```
temp = new Node;    // add node between two others
temp->data = 17;
temp->next = front->next;
front->next = temp;

temp = front;         // delete a node
front = front->next;
delete temp;
```

Example: Set of Ints

- Re-implement class IntSet (set of integers with no duplicates) -- no change to interface(!)
- Use linked list to represent set
- Same operations as before
  - Construct empty set
  - Add/remove numbers from set
  - Search for number in set, report size of set
  - Stream output
- Add other operations needed for full C++ class
  - Copy constructor, destructor, assignment

IntSet Representation

- In intset.h

```
class IntSet{
  ...
private:
  // Representation: Single-linked list

  Struct Node {      // one node in the list
    int val;         //   integer value
    Node * next;     //   pointer to next
  };                 //     node or NULL

  Node * first;      // pointer to first node
                     //   in the list, or
                     //   NULL if this IntSet
  ...                //   is empty.
```


Constructor

- In intset.cpp

```
// Construct empty IntSet
IntSet::IntSet( ) {
   first = NULL;
}
```


List Traversal - size

- Several operations require a list traversal -- sequence through the nodes until reaching NULL at the end. Example:

```
// Yield the size of this IntSet
int IntSet::size( ) const {
   Node * p;  // current element
   int n;     // # nodes counted

   p = first;
   n = 0;
   while (p != NULL) {
      n++;
      p = p->next;
   }
   return n;
}
```


size (cont)

- Example:

- This takes O(n) time. Could be O(1) if we kept track of set size as items added/removed.

Membership test

```
// = "n is in this IntSet"
bool IntSet::isMember(int n) const {
   Node * p;  // current element
   p = first;
   while (p != NULL && p->val != n)
      p = p->next;
   return (p != NULL);
}
```

- Notice how this depends on && not evaluating its second argument if the first one is false
- **WARNING:** *Never* dereference a pointer p (p->x) unless you are *absolutely sure* p!=NULL (p!=0).

Stream Output (1)

```
// Print IntSet t on s as {e1,e2,e3,…,en}
ostream& operator<< (ostream &s,
                     const IntSet & t) {

  IntSet::Node * p;// current element

   s << '{'
   p = t.first;
   if (p != NULL) {
      s << p->val;
      p = p->next;
   }
   ...
```


Stream Output (2)

```
   ...
   while (p!=NULL){
      s << ',' << p->val;
      p = p->next;
   }
   return (s << '}');
}
```

- operator<< is a friend of IntSet. It can access type Node, but it must explicitly say *which* Node: IntSet::Node

Add New Element

- Since the list is unordered, we can add new elements onto the front of the list

```
// Add n to this IntSet if not already in
void IntSet::add(int n) {
   if (!isMember(n)){
      Node *p = new Node;
      p->val = n;
      p->next = first;
      first = p;
   }
}
```

- Cost? O(1)? O(n)?
- If the list were sorted (i.e., list position mattered), what would have to be different?

Delete (1)

- Need to find the number, then remove its node from the list

```
// Remove n from this IntSet if present
void IntSet::remove(int n) {
   Node * p = first;
   while (p != NULL && p->val != n)
      p = p->next;
   if (p != NULL) {
      // p->val==n; remove node p from the
      // list and then delete p


      ----------------------------
}
```

- What goes on the blank line?

Delete (2)

- To remove node p, we need to update the **`next`** pointer in the previous node.
- Use another pointer **`prev`** with **`prev->next=p,`** where **`p`** is the node we want to delete. So,

```
   Node * prev = first;
   Node * p = first->next;
   while (p != NULL && p->val != n){
      prev = p; p = p->next;
   }
   if (p != NULL) {
      prev->next = p->next;
      delete p;
   }
}
```

- Everything ok?

Delete (3)

- Bugs
  - If the list is empty, **`first->next`** is an error (Why?)
  - If the number to be deleted is in the first list node, we need to update **`first`**, not the **`next`** field of some node.

Delete (4)

- Final version

```
// Remove n from this IntSet if present
void IntSet::remove(int n) {
  Node * p;    // current node
  Node * prev; // prev. node (prev->next=p)

  // special case: empty list
  if (first == NULL) return;

  // special case: delete 1st node
  if (first->val == n) {
    p = first->next;
    delete first;
    first = p;
    return;
  }
```


Delete (5)

- Final version (cont)

```
  // Search non-empty list for n
  prev = first; p = first->next;
  while (p != null && p->val != n) {
    prev = p; p = p->next;
  }
  // post (p==NULL) or (p->val==n)
  if (p != NULL) {
    prev->next = p->next;
    delete p;
  }
}
```


Standard Operations

- For a complete class we need to add a copy constructor, destructor, and assignment. These require different combinations of two operations
  - Destroy -- ensure that all nodes on the current list have been deleted properly (destructor, assignment)
  - Copy -- make a deep copy of the nodes belonging to another IntSet (copy constructor, assignment)
- Implement Destroy and Copy as private member functions -- avoid duplicating code.

Copy (1)

- Store a deep copy of another IntSet in this one
- Makes no assumption about previous contents of this IntSet
  - Caller must ensure any previously allocated nodes have already been reclaimed first.

```
// Store a copy of s in this IntSet
void IntSet::Copy(const IntSet &s) {
  Node * last;     // last node added to
                   //   this IntSet so far
  Node * p         // first node in s not
                   //   yet copied
  Node * q;
```


Copy (2)

```
  // If s is empty, set this IntSet empty
  if (s.first == NULL){
    first = NULL;
    return;
  }

  // copy first node of non-empty IntSet s
  first = new Node;
  first->val  = s.first->val;
  first->next = NULL;
  ...
```

- Note difference between . to select fields and -> to follow pointer then select field

Copy (3)

```
  // copy nodes after the first
  p = s.first->next;
  last = first;
  while (p != NULL) {
    // create new node
    q = new Node;
    q->val  = p->val;
    q->next = NULL;
    // add new node to end of copied list
    last->next = q;
    last = q;
    p = p->next;
  }
}
```

Destroy

- Only trick here is to watch assignment order

```
// Deallocate the nodes in this IntSet
void IntSet::Destroy( ) {
  Node * p;   // first node not yet deleted
  Node * temp;

  p = first;
  while (p != NULL) {
    temp = p->next;
    delete p;
    p = temp;
  }

  first = NULL;
}
```


Standard Class Members

- Given those operations, the rest is simple

```
// Destructor
IntSet::~IntSet( ) { Destroy( ); }

// Copy constructor
IntSet::IntSet(const IntSet & val)
  { Copy(val); }

// Assignment
IntSet & IntSet::operator=
                (const IntSet & val) {
  if (this != &val) {
    Destroy( );
    Copy(val);
  }
  return *this
}
```